AGENTTAXO: DISSECTING AND BENCHMARKING TOKEN DISTRIBUTION OF LLM MULTI-AGENT SYSTEMS

Qian Wang[1]*, Zhenheng Tang[2]*, Zichen Jiang[1] Nuo Chen[1] Tianyu Wang[3] Bingsheng He[1]

[1]National University of Singapore [2]The Hong Kong University of Science and Technology
[3]Shanghai Jiao Tong University

ABSTRACT

LLM-based multi-agent (LLM-MA) systems have demonstrated potential in complex tasks such as reasoning and code generation. However, compared to single-agent systems, LLM-MA systems incur significantly higher inference latency and token costs due to repeated LLM calls. In this work, we identify duplicated tokens as a major contributor to these inefficiencies, acting as a "communication tax" that hinders scalability. To systematically analyze token duplication patterns, we propose AgentTaxo, a taxonomy that categorizes agent roles into Planner, Reasoner, and Verifier across various applications. AgentTaxo dissects inter-agent communication and identifies redundant reasoning results frequently reused for validation. We benchmark and analyze token costs in popular LLM-MA systems, quantifying the impact of this communication tax through experimental evaluation. Our findings provide insights into optimizing efficiency and scalability in LLM-MA architectures.

1 INTRODUCTION

LLM-MA systems (Wu et al., 2023b; Chen et al., 2023; Hong et al., 2024) leverage collaborative interactions among multiple LLM agents to address many tasks. While these systems demonstrate strong performance in domains such as code and math (Chen et al., 2023), their effectiveness is hampered by the significantly higher token consumption (Singh & Strouse, 2024) and long latency caused by dependency (Lin et al., 2024; Tan et al., 2024) between LLM agents as shown in Figure 1. The significant latency hinders real-world deployment of LLM-MA systems especially in some emergency scenarios, such as hospitals and natural disasters.

To understand where is the source of higher token consumption, we empirically measure three real-world LLM-MA frameworks (Hong et al., 2024; Li et al., 2023b; Chen et al., 2023) in completing three tasks. Table 1 shows token statistics, where the duplication rate reaches up to 86%. We call these duplicated tokens as the *Communication Tax*, referring to the costs of scaling up the number of agents to conduct tasks. Observing such duplicates, researchers have proposed application-aware optimizations, including KV cache reuse via RadixAttention (Zheng et al., 2023), semantic variable tracking for prompt reuse (Lin et al., 2024), predictive resource allocation (Wu et al., 2024), and parallelized agent execution pipelines (Tan et al., 2024). From an algorithmic perspective, the prevalence of repeated tokens suggests consolidating agent functions to reduce redundancy (Xue et al., 2024; Wang et al., 2024b).

*Equal contribution

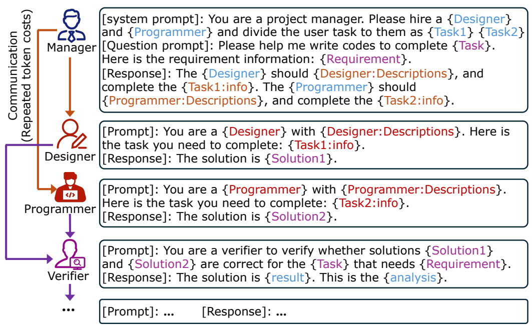

Figure 1: An example of LLM-MA application (Hong et al., 2023). Different colors indicate the communicated messages towards different types of roles (real-world case studies in Appendix B).

However, these advanced LLM-MA optimizations require a deeper understanding of token distribution patterns across agents: an aspect largely overlooked in current literature. This gap motivates us to explore the following fundamental research question:

How can we classify sequences in LLM-MA and analyze their occurrence patterns? What are the sources of token duplication?

Table 1: Duplicate Token Statistics. *Tokens as duplicate if they appear in at least two LLM calls.*

Framework	Task Type	# Tokens	Duplicate(%)
MetaGPT (Hong et al., 2023)	Code	30k $\sim$ 90k	72
CAMEL (Li et al., 2023b)	Math	2k	86
AgentVerse (Chen et al., 2023)	Reasoning	10k	53

Current LLM-based multi-agent frameworks employ diverse role definitions tailored to specific applications like code generation (Hong et al., 2023), academic projects (Wang et al., 2024c) and business management (Xu et al., 2024). This complicates direct comparisons and obscuring underlying patterns in token distributions. To unify these variations, we propose a general benchmark framework AgentTaxo to help answer above questions.

First, inspiring from philosophy of the divide-and-conquer problem solving, AgentTaxo generalizes various agent roles into three common agents: the planner for task decomposition; the reasoner for direct solving sub-tasks; and the verifier checks results and provides reflection (Section 3.1). This generalization enables systematic analysis of how role specialization impacts token allocation across different LLM-MA frameworks.

Second, with these general role definitions, AgentTaxo taxonomizes different sequences in the inputs and outputs of these agents into system problems, context, questions, plan, results, verification analysis (Section 3.3). Built upon them, we can dissect how these sequences are communicated between different agents to enable the divide-and-conquer problem solving. These categories also align with the semantic variables in (Lin et al., 2024) or the primitives in (Zheng et al., 2023), providing opportunities for future system optimization.

Based on AgentTaxo, we analyze token distribution across three LLM-MA systems: MetaGPT (linear), CAMEL (flat), and AgentVerse (hierarchical). Our evaluation includes extensive tasks including code generation, reasoning, and role-playing, following established experimental settings Chang et al. (2024); Hong et al. (2024); Li et al. (2023a); Chen et al. (2023). We categorize tokens into planner, reasoner, and verifier phases to systematically assess token distribution and communication overhead in their interactions.

Results show that reasoning and verification dominate token consumption. Input tokens consistently outnumber output tokens by a 2:1 to 3:1 ratio, indicating inefficiencies in prompt design. Verification phases disproportionately consume input tokens, as seen in MetaGPT's 2048 development experiment, where 72% of tokens are used for verification. Communication overhead mainly arises from repeated reasoning results appearing in verification, leading to duplicate tokens. Our contributions are:

- **First Taxonomy of LLM-MA Token Distribution:** We propose AgentTaxo that pioneers a systematic dissection of token distribution across planner, reasoner, and verifier phases.
- **Communication Tax Quantification:** With AgentTaxo and comprehensive benchmarking, we rigorously quantify redundant token costs from inter-agent interactions, introducing "communication tax" as a metric.
- **Empirical Optimization Guidelines:** We conduct empirical benchmarking across diverse tasks in MetaGPT, CAMEL, and AgentVerse, providing quantitative evidence of token inefficiencies and offering insights for optimizing LLM-MA frameworks.

2 Related Work

LLM-MA Systems. Recent LLM advances (Achiam et al., 2023; Team et al., 2023) have spurred LLM-MA systems that coordinate specialized agents for complex tasks like code generation and planning (Hong et al., 2023; Chen et al., 2023). Enhanced by persona design (Chen et al., 2024; Chan et al., 2024), strategic planning (Yuan et al., 2023), and memory architectures (Zhang et al., 2023a; Hatalis et al., 2023), these systems outperform single-agent approaches. Frameworks like MetaGPT (Hong et al., 2023), AutoGen (Wu et al., 2023b), AgentVerse (Chen et al., 2023), and MegaAgent (Wang et al., 2024a) exemplify effective multi-agent coordination.

LLM Token Costs. LLM-MA systems amplify high token consumption of LLMs (Liu et al., 2024b; Keith et al., 2024; Han et al., 2024c), with added overhead from role definitions (Cheng et al., 2024), system prompts and inter-agent communication (Zhang et al., 2024a). Recent work challenges multi-agent necessity (Wang et al., 2024b) by showing single agents with optimized prompts can match multi-agent performance while saving tokens.

Divide and Conquer. Multi-step reasoning requires decomposing problems into sequential sub-tasks. LLMs act as meta-agents to autonomously decompose and schedule tasks (Hong et al., 2024; Wu et al.), augmented by symbolic reasoning for optimized planning (Zhang et al., 2023b). Many approaches leverage multiple calls for answer aggregation (Brown et al., 2024), including majority voting in CoT-SC (Wang et al., b), structured reasoning frameworks like Tree or Graph-of-Thought (Yao et al., 2024; Besta et al., 2024) and knowledge graph integration (LUO et al.; Sun et al.). Triplet-formatted prompts (Jiang et al., 2023) and decentralized LLM collaborations (Li et al., 2023a; Hong et al., 2024; Du et al.) further enhance reasoning.

LLM-MA Serving. Recently, because of the population of LLM-MA frameworks, researchers begin to focus on their serving system. Different from single LLM call, LLM-MA frameworks introduce multiple calls dependent on each other and can be formulated as directed acyclic graphs (DAGs). Thus, the serving latency of one complete MA is significantly larger than the one LLM call. Many works have noticed this and propose the application-aware optimization including KV cache reuse via RadixAttention (Zheng et al., 2023), semantic variable tracking (Lin et al., 2024) for prompt reuse, predictive resource allocation (Wu et al., 2024), parallelization and pipelining of different agent calls (Tan et al., 2024).

Despite these literature, no benchmark systematically evaluates token distribution across task stages, limiting the efficiency of LLM-MA system design. To address this, we establish a comprehensive benchmark for evaluating token distribution in LLM-MA systems. We leave more detailed discussion in Appendix A due to limited space.

3 Methodology

As different LLM-MA agents have many different role definitions based on different downstream applications like coding (Hong et al., 2023), academic research (Wang et al., 2024c), business project (Xu et al., 2024), it is difficult to generally analyze how tokens are costed of different stages. To this end, we present AgentTaxo as a systematic framework for analyzing token LLM-based Multi-Agent systems. Our methodology consists of three key components: (1) A general formalization of individual LLM-based agents as shown in Figure 2; (2) Unified categories of different input and output sequences as shown in Figure 4; (3) different topologies of multi-agent system architecture.

3.1 LLM-BASED AGENTS

Language Modeling. Current widely used language modeling (Achiam et al., 2023) is the next word prediction, which predicts the next token x_t given the previous tokens $x_{1:t-1}$ for all $t = 1, \ldots, T$. Formally, a LLM parameterized by θ is a language modeling distribution $f_\theta(x_t|x_{1:t-1})$. Each token x_t is sampled from a vocabulary $\mathbb{O}$. In this work, we temporarily assume that all agents utilize the same language modeling distribution f. In other words, we assume that all agents utilize the same backbone LLM, which is a common practice in current works Zhao et al. (2023); Zhou et al. (2024). Thus, we write f_θ as f for simplicity.

Definition 3.1 (LLM Call)**.** An LLM call is to sample a sequence from a language modeling distribution as $R \sim f(R|S)$, in which the S is usually called the prompt or say input sequence, while the R is called the response or say output sequence. The maximum length of the input sequence is $L_{inp}^{\max}$ while the maximum length of the output sequence is $L_{out}^{\max}$.

We write $S = S_1 \circ S_2 \circ \cdots \circ S_n$ as the concatenation of n sequences. For simplicity, we also directly write a concated sequence as $S = S_1 S_2 \cdots S_n$.

Definition 3.2 (LLM-based Agent)**.** An LLM-based agent is a language modeling distribution $f(R|S_{\text{sys}} \circ S)$ with a fixed prefix system prompt S_{sys}, and input sequence S with the maximum length $L_S^{\max} < L_{inp}^{\max} - L(S_{\text{sys}})$. With input sequence S, the agent can generate a sequence $R \sim f(R|S_{\text{sys}} \circ S)$.

Calling an LLM-based agent means sampling a sequence from the language modeling distribution defined by the agent.

Agent Roles. Current multi-agent systems usually define agents with different roles like product manager, code architect, programmer in a code writing project (Hong et al., 2023), or PhD student, Postdoc, Professor in a paper writing project (Wang et al., 2024c), or CEO, CFO, CTO, sales, marketing, supply chain in a business project (Xu et al., 2024). As shown in Figure 2, we generalize these different roles into the planner f_{plan}, reasoner f_{reason}, and verifier f_{verify}. To solve a task, these agents collaboratively work together to output the final result R_{result}. According to different role definitions and the concrete application requirements, different agents will be assigned different system prompts S_{sys}.

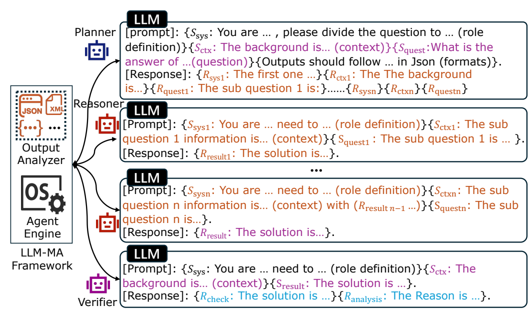

Figure 2: An example of unified linear communication topology and sequence categories as in MetaGPT. Different colors indicate the communicated messages towards different types of roles.

3.2 MULTI-AGENT SYSTEM

Agent Runtime Engine. Because LLM based agents themselves cannot call or activate another agents, or manage and construct new messages, there should be some external programs as an engine to call LLMs and manage messages between them (Chen et al., 2023; Lin et al., 2024). As shown in Figure 2, an agent is actually a LLM equipped with different system prompt including role definitions. The agent engine calls LLMs with different inputs, and receives their outputs, then process messages with given formats. Then, the extracted information with some other texts are constructed as new prompts to other agents.

Standard Operating Procedures (SOPs). SOP serves as foundational guidelines for orchestrating agent behaviors and communication protocols in LLM-MA systems. By defining step-by-step workflows, SOPs ensure consistency, coordination, and scalability across agents. Critically, these procedures rely on the agent engine to mediate interactions between LLMs, enabling agents to exchange information, resolve dependencies, and align their outputs with system-wide objectives.

Predefined SOPs are manually crafted protocols that govern agent roles, task decomposition, and communication patterns. Pioneering works like (Hong et al., 2023; Huang et al., 2023; Park et al.,

2023) demonstrate their effectiveness in structuring specialized agents for code generation, collaborative problem-solving, and workflow automation. *Automated SOPs.* dynamically generate and refine agent workflows using real-time system feedback or learned policies. These protocols leverage the external control system to analyze agent capabilities, task requirements, and interaction histories, enabling on-the-fly adjustments to role assignments and communication rules (Piatti et al., 2024; Mou et al., 2024; Pan et al., 2024).

In the overall agent workflow, agents interact by exchanging information to accomplish tasks collaboratively. The planner f_{plan} decomposes the task and assigns responsibilities to different agents. The reasoner f_{reason} processes the task description and generates a solution. The f_{verify} then evaluates both the question and the generated result to ensure correctness. We detail how different agents are equipped with these components and how their inputs and outputs interact within the system in the following:

Definition 3.3 (Planner)**.** A planner f_{plan} is an LLM-based agent $f(R_{\text{plan}}|S_{\text{sys}}S_{\text{ctx}}S_{\text{quest}})$ with a fixed system prompt S_{sys}, and varied context S_{ctx} and query S_{quest}. The $R_{\text{plan}} = R_{\text{sys1}}R_{\text{ctx1}}R_{\text{quest1}} \cdots R_{\text{sysn}}R_{\text{ctxn}}R_{\text{questn}}$ is the partition of the task.

Definition 3.4 (Reasoner)**.** A reasoner f_{reason} is an LLM-based agent $f(R_{\text{result}}|S_{\text{sys}}S_{\text{ctx}}S_{\text{quest}})$ with a fixed system prompt S_{sys}, and varied context S_{ctx} and query S_{quest}.

Definition 3.5 (Verifier)**.** A verifier f_{verify} is an LLM-based agent $f(R_{\text{verify}}|S_{\text{sys}}S_{\text{ctx}}S_{\text{quest}}S_{\text{result}})$ with a fixed system prompt S_{sys}, and varied context S_{ctx} and query S_{quest}.

Definition 3.6 (Composite Agent)**.** A composite agent is an LLM-based agent that both conduct multiple functions include planning, reasoning and verification. Given the system prompt S_{sys}, and varied context S_{ctx} and query S_{quest}, it can be defined as distribution $f(R_{\text{plan}}R_{\text{result}}R_{\text{verify}}|S_{\text{sys}}S_{\text{ctx}}S_{\text{quest}}S_{\text{result}})$.

3.3 DISSECTING TOKEN COST DISTRIBUTION

Communication Tax. Regardless of the agent framework design, agents must exchange token sequences to collaboratively solve tasks. This shared information often appears redundantly in both input and output stages, as illustrated in Figure 2. For instance, the planner's output $R_{\text{ctx}}R_{\text{quest}}$ is divided and reformulated into sub-tasks, which are then structured as context S_{ctx} and question S_{quest} for the reasoner f_{reason}. While this divide-and-conquer approach facilitates complex problem-solving, it also incurs significant token costs for *communication between agents and the agent engine*. We define these additional token costs as the *Communication Tax* in LLM-MA systems.

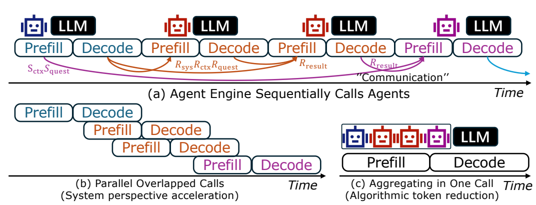

Figure 3: (a) An example of timeline of linear communication topology (Hong et al., 2023); (b) The system optimization of multiple agent calls (Lin et al., 2024; Zheng et al., 2023); (c) Solving the problem within one call.

LLM inference requires processing input (prefilling) generating output (decoding) and save previous Key Value attention tensors. As shown in Figure 3(a), repeated sequences in different agents increase redundant computation and memory costs, and enlarge the LLM serving latency due to launching different agent calls sequentially.

Optimization Opportunities of Communication Tax. Figure 3(b) shows that some recent works propose the application-aware optimization including KV cache reuse (Zheng et al., 2023) and parallel calls (Lin et al., 2024). Figure 3(c) shows it is possible to aggregate different agent calls into one thus avoiding redundant computation and storage such as Definition 3.6. To provide further support on the both systematic and algorithmic for LLM-MA systems, understanding the token cost distribution is critical, which lacks discussion in current literature.

Agent-level Token Cost Comparison. As shown in Figure 2, different agent roles can be generalized as the planner f_{plan}, reasoner f_{reason}, and verifier f_{verify} of Definition 3.3, 3.4 and 3.5 in Section 3.2. With this definition, we can compare the token cost distribution with agent-level in different LLM-MA frameworks uniformly.

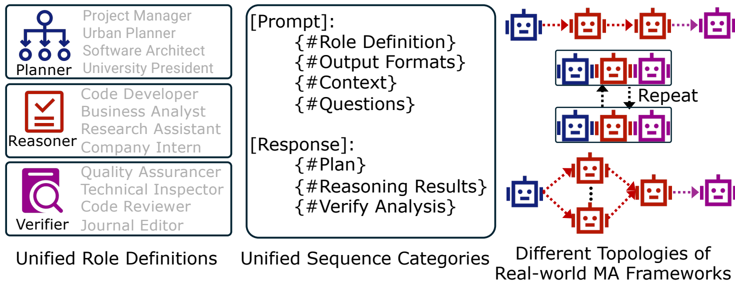

Figure 4: Three different dimensions of our benchmark, which unifies different agent roles and sequence categories, identifies typical topologies of current LLM-MA frameworks (Hong et al., 2023), and various tasks.

Sequence-level Token Cost Comparison. To enable fine-grained optimization of LLM-MA systems, one needs to know how different sequences* appear in prompts and responses of agents. As shown in Figure 4, we divide the prompts into `Role Definition`, `Output Formats`, `Context`, `Plan`, `Question`, `Reasoning Results` and `Verify Analysis`. All agents receive the `Role Definition`, `Output Formats` as the system prompt S_{sys}. Input and output sequences of the planner, reasoner and verifier are categorized as follows.

- *The planner* f_{plan} receives `Context` S_{ctx} as the background or descriptions, and `Question` S_{quest} and output $R_{\text{plan}} = R_{\text{sys}1} R_{\text{ctx}1} R_{\text{quest}1} \cdots R_{\text{sys}n} R_{\text{ctx}n} R_{\text{quest}n}$ to divide original question into sub-questions and define a group of reasoners to solve them. For convenience of comparison, we also regard the `Plan` R_{plan} as a kind of `Reasoning Results` of the f_{plan} in Section 4.
- *The reasoner* receives inputs $S_{\text{ctx}} S_{\text{quest}}$ which might be generated by the planner f_{plan}, or predefined by the framework, and it returns `Reasoning Results` R_{result}. After reasoning, the engine will receive and identify the result R_{result} for further processing.
- *The verifier* receives both $S_{\text{ctx}} S_{\text{quest}}$ and the result S_{result} from the reasoner for verification. It outputs the verification results `Verify Analysis` $R_{\text{verify}} = R_{\text{ckc}} R_{\text{analysis}}$. In which the R_{ckc} is the check operation that indicates whether the result is correct, and the R_{analysis} is the reflection analysis of the verifier. If the R_{ckc} is identified as true, the question S_{quest} is seen as solved. Otherwise, the original question and the analysis R_{analysis} might be input to the planner or reasoner to solve the problem again, in which the process depends on the concrete SOPs.

Communication Topology. The communication flow decides which sequences are repeatedly shared between agents. Figure 2 shows a linear communication topology, where the planner divides the problem to be solved by reasoners sequentially as a chain structure. Thus, the intermediate reasoning results are sent to the next reasoner. As it is difficult to define a topology (or say DAG) that can generally cover different frameworks, we choose to choose three typical existing LLM-MA framework with different topologies to benchmark including *Linear*, *Flat* and *Hierarchical* in Section 4.

4 Experiments

We analyze the token distribution characteristics in LLM-MA systems through three research questions (RQs). **RQ1:** What are the token distributions across planner, reasoner, and verifier in LLM-MA systems? **RQ2:** How does communication tax influence token consumption in LLM-MA systems? **RQ3:** How does LLM-MA system topology influence token distribution and communication tax?

4.1 Preliminary

General Tasks in LLM Evaluation. To evaluate the performance of LLMs, researchers have curated a diverse range of datasets spanning multiple domains. These datasets are broadly categorized into several key areas, including code generation, mathematical reasoning, logical reasoning, general understanding, and role-playing tasks, each designed to assess specific capabilities of LLMs (Chang et al., 2024).

*The sequences can be scheduled as semantic variables in (Lin et al., 2024) or the primitives in (Zheng et al., 2023).

Choices of LLM-MA Systems. Building on the message-passing mechanisms of LLM-MA systems (Huang et al., 2024), we systematically analyze three fundamental topologies, chosen as they represent the core frameworks in this domain: (1) *Linear* (A→B→C), featuring sequential one-way communication for task specialization; (2) *Flat* (A↔B↔C), enabling fully bidirectional peer coordination; and (3) *Hierarchical* (A→(B↔C)), combining vertical command flow with lateral collaboration to transcend purely linear models.

In alignment with this classification, we select the following LLM-MA systems:

Linear: **MetaGPT** (Hong et al., 2023) employs Standard Operating Procedures (SOPs) to establish an efficient workflow within a simulated software company, utilizing five agents for code generation.

Flat: **CAMEL** (Li et al., 2023b) introduces a framework where a “User” agent iteratively refines outputs provided by an “Assistant” agent. It is applicable for many tasks.

Hierarchical: **AgentVerse** (Chen et al., 2023) implements a dynamic recruitment process, selecting agents for multi-round collaboration as required, and the agent number is flexible for each task.

Token Categories in LLM-MA Systems Tokens exchanged in LLM-MA can be systematically categorized based on their functional roles in LLM calls. These categories reflect the distinct purposes of agents and their interactions, enabling a clearer understanding of communication patterns and computational overhead. The primary token categories include:

Role Definition. specify the responsibilities and constraints of agents within a system call. They define how an agent should behave in a given context, ensuring alignment with the task. For example, an instruction like *"You are responsible for verifying calculations and providing feedback"* explicitly assigns a role to an agent.

Questions. represent actionable items or requests assigned to agents, such as generating code, solving problems, or completing specific tasks. These tokens drive the agent’s primary functionality and output. Examples include *"Generate a Python function to calculate Fibonacci numbers"* or *"Summarize the key points from this document."*

Output Formats. dictate the structure of responses in LLM calls. They ensure consistency across agent outputs by enforcing predefined formats. For instance, an instruction like *"Format the answer in JSON with a reasoning explanation"* guides the agent to produce structured and standardized results.

Reasoning Result. encapsulate intermediate or final reasoning outputs generated during an LLM call. These tokens reflect the agent’s thought process as it derives solutions. An example might be *"If A implies B and B is true, then A must also be true."*

Verification Analysis. is used to evaluate and validate reasoning results within LLM calls. They ensure logical consistency and correctness, enhancing the reliability of outputs. For example, an agent might state, *"Disagree. Rechecking the derivation: Step 2 contradicts Step 5, suggesting an inconsistency."*

Other/Metadata. include auxiliary information necessary for system-level coordination, such as session management or role assignment. These tokens often contain system logs or metadata, like *"Session ID: 12345"*.

4.2 Experiment Settings

Downstream Tasks. We evaluate the task-solving abilities of LLM-MA systems using a diverse set of general tasks:

General Reasoning: MGSM (Cobbe et al., 2021) is a subset of GSM-8k designed to evaluate agents’ mathematical reasoning capabilities. This dataset contains grade-school-level math problems to test multi-step problem-solving skills.

Code Generation: HumanEval (Chen et al., 2021) includes 164 hand-crafted programming problems to assess LLMs’ ability to synthesize correct and functional Python code.

Software Development: 2048 (Hong et al., 2024) involves developing a fully functional 2048 game. Similarly, Blackjack (Hong et al., 2024) focuses on creating a casino banking game based on standard blackjack rules.

Logical Reasoning: Logic Grid (Srivastava et al., 2022) contains multi-step logic problems to evaluate agents' logical reasoning capabilities.

General Understanding: FED (Mehri & Eskenazi, 2020) is a dialogue response dataset where agents or agent groups are required to generate responses based on multi-round chat histories.

Role Playing: AI Society (Li et al., 2023b) requires developing a Python trading bot that uses machine learning to analyze historical stock data, predict future price movements, and automatically execute buy/sell orders. Additionally, Code (Li et al., 2023b) involves creating a function that enables users to generate survey questions.

Please note that not all systems support all the tasks listed here. Therefore, we follow the experimental settings described in their respective papers to conduct the corresponding experiments.

We use GPT-4-Turbo* as the backbone for all experiments, setting the temperature to zero to ensure consistent and robust results.

4.3 RQ1: Token Distribution

Table 2: Token Distribution across Models and Tasks. The most dominant phase is highlighted in red.

Model	Task	# Agents	# Tokens	# Input	# Output	# Planner	# Reasoner	# Verifier	Max Input Phase	Max Output Phase
MetaGPT	2048	5	31,852	24,664 (77%)	7,188 (23%)	1,237 (4%)	7,526 (24%)	23,089 (72%)	Verifier	Reasoner
	Blackjack	5	90,112	63,628 (71%)	26,483 (29%)	2,633 (3%)	17,607 (19%)	69,872 (78%)	Verifier	Reasoner
CAMEL	Code	2	2,250	1,051 (47%)	1,199 (53%)	1,048 (47%)	1,202 (53%)	0 (0.00%)	Planner	Reasoner
	AI Society	2	1,325	689 (52%)	636 (48%)	601 (45%)	724 (55%)	0 (0.00%)	Planner	Reasoner
AgentVerse	HumanEval	4	15,423	10,582 (69%)	4,840 (31%)	2,761 (18%)	8,416 (55%)	4,246 (29%)	Verifier	Reasoner
	MGSM	2	9,871	7,649 (78%)	2,222 (23%)	1,280 (13%)	2,150 (22%)	6,461 (65%)	Verifier	Reasoner
	LogicGrid	2	7,657	4,834 (63%)	2,823 (37%)	860 (11%)	4,714 (62%)	2,083 (27%)	Reasoner	Reasoner
	FED	2	11,230	9,193 (82%)	2,037 (18%)	1,166 (10%)	2,973 (26%)	7,091 (63%)	Verifier	Reasoner

To analyze token distributions across planning, reasoning, and verification, we examine two key aspects: (1) phase-specific token distribution and (2) input-output token distribution. Based on the results in Table 2, we derive the following key findings:

Finding 1: Reasoning or verification dominates token consumption. Across all frameworks and datasets, reasoning or verification accounts for the largest share of token consumption. This finding highlights the varying priorities across frameworks, with reasoning or verification driving resource usage depending on the task, while planning consistently accounts for a smaller share.

Finding 2: Verification introduces significant token overhead. Frameworks incorporating explicit verification mechanisms, such as MetaGPT and AgentVerse, incur significant token costs. For instance, AgentVerse (MGSM) dedicates nearly two-thirds of its tokens to verification, while MetaGPT (2048) similarly exhibits verification as the dominant token consumer. In contrast, frameworks that deprioritize verification, such as CAMEL, allocate negligible tokens to this phase. These findings indicate that verification introduces substantial overhead.

Finding 3: Input tokens dominate. Input tokens consistently exceed output tokens, often by a ratio of 2:1 to 3:1 across various tasks and frameworks. This emphasizes the heavy reliance of LLM-MA systems on extensive prompts, including role definitions, instructions, and task contexts, to guide agent behavior. These findings highlight potential inefficiencies in input tokens.

Finding 4: Verification phases disproportionately consume input tokens compared to output tokens. In most cases, verification phases account for a substantial share of input tokens, likely due to their design, which involves format checking and performing repetitive validations for each reasoning output. This highlights the importance of analyzing the input tokens consumed during verification to identify opportunities for optimization.

Finding 5: Reasoner occupies the most output tokens. Across all frameworks, the reasoning phase consistently generates the highest proportion of output tokens. This reflects the role of the reasoner in synthesizing information and generating responses. This is also intuitive as the reasoner would give the detailed responses to the questions.

[*] https://platform.openai.com/docs/models/gpt-4

4.4 RQ2: Communication Tax

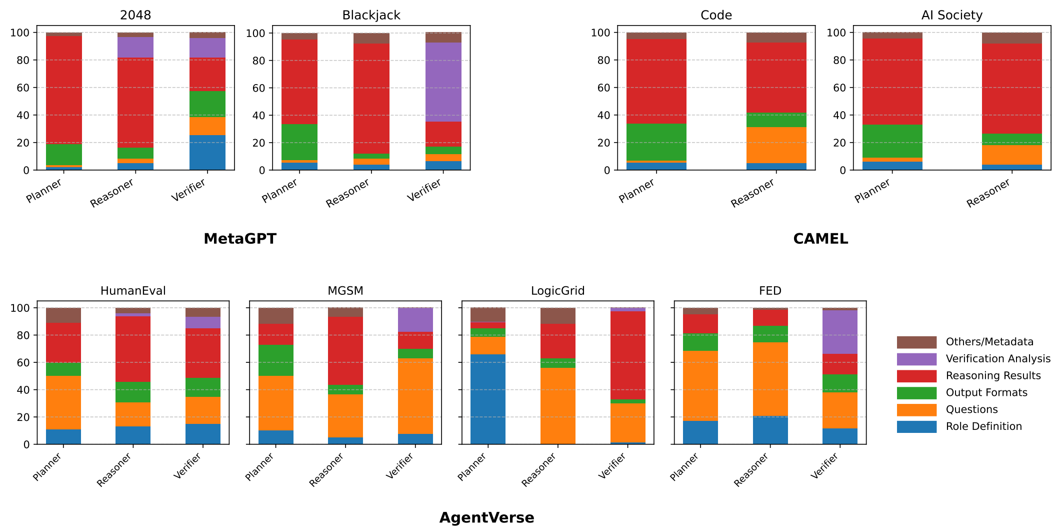

Figure 5: Analysis of token categories within each phase.

To analyze communication tax, we investigate the sources of exchanged tokens, including role definitions, context, formatting, reasoning, verification, and other components. Additionally, we identify communication-induced overhead, such as reasoning results reappearing in the verifier phase, leading to repeated token consumption. Based on Figure 5, we present the following findings.

Finding 6: Verification Overhead Mirrors Reasoning Redundancy. Verification phases consume tokens not through novel validation but by reprocessing reasoning outputs. In AgentVerse (MGSM), about 5-% of verification tokens fund recursive checks of prior reasoning steps, such as re-evaluating arithmetic in multi-step calculations. MetaGPT (2048) exhibits similar inefficiencies, with 20% of verification tokens reinvested in reasoning results. This redundancy aligns with Finding 5's observation that verification dominates input tokens—not because it introduces new logic, but because it recycles reasoning.

Finding 7: Output Formatting Decreases Efficiency. Token allocation to formatting instructions reveals a systemic tension between usability and efficiency. MetaGPT dedicates 15% of Planner tokens to structuring code outputs, while CAMEL spends about 25% of Planner tokens on dialogue organization. These investments, though critical for human interpretability, expose a trade-off: strict formatting requirements anchor tokens to syntactic compliance rather than semantic problem-solving.

4.5 RQ3: Topology Influences

To analyze the impact of topology on token distribution and overhead, we examine Table 2 alongside Figure 5. From this analysis, we derive the following findings:

Finding 8: Linear systems reduce verifier communication tax compared to hierarchical systems. A comparison of verifier token usage between MetaGPT and AgentVerse reveals that MetaGPT incurs lower percentage reasoning token costs, which is the communication tax. This reduction is attributed to the linear system's lack of iterative communication loops, which are prevalent in hierarchical systems and contribute to increased token consumption.

Finding 9: Code tasks favor linear topologies for minimizing communication tax. For structured programming tasks, linear topologies optimize token efficiency by enforcing a strict one-way pipeline. In contrast, AgentVerse incurs significant overhead due to the repeated use of reasoning results in verification and the propagation of verification results back into reasoning, as shown in HumanEval chart in Figure 5. However, when linear structures reduce redundant token consumption, they trade off adaptability.

Finding 10: Topologies exhibit distinct reasoning and verification patterns. Linear systems enforce a structured workflow, reducing redundant verification but limiting adaptability. Flat systems enhance reasoning flexibility through iterative exchanges but risk higher token costs due to extended reasoning interactions. Hierarchical structures distribute reasoning across specialized agents yet accumulate verification overhead through multi-stage validation. These differences highlight the need for topology-specific optimizations to balance efficiency and adaptability.

5 Conclusion

AgentTaxo is the first taxonomy for analyzing token distribution in LLM-based multi-agent systems, providing a structured understanding of how tokens are allocated across planner, reasoner, and verifier roles. Through communication tax quantification, we identify and measure the inefficiencies arising from inter-agent interactions, revealing that a substantial fraction of verification tokens is consumed by redundant revalidation rather than novel insights. Finally, our findings derived from extensive experiments provide actionable strategies to enhance token efficiency in LLM-MA frameworks. By systematically dissecting and quantifying token usage, our work paves the way for more cost-effective and scalable multi-agent LLM systems.

References

Tensorrt-llm: A tensorrt toolbox for optimized large language model inference. URL https://github.com/NVIDIA/TensorRT-LLM.

Lightllm. https://github.com/ModelTC/lightllm, 2023. Commit: 84671a7, Accessed on: 2023-11-25.

Promptflow, 2023. URL https://https://github.com/microsoft/promptflow.

Huggingface text generation inference. https://github.com/huggingface/text-generation-inference, 2023. Commit: 3c02262, Accessed on: 2023-11-25.

Langchain, 2024. URL https://github.com/langchain-ai/langchain.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15. IEEE, 2020.

Apache. Tez. https://tez.apache.org/, November 2019.

Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 17682–17690, 2024.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Baobao Chang, Junjie Hu, et al. Pyramidkv: Dynamic kv cache compression based on pyramidal information funneling. *arXiv preprint arXiv:2406.02069*, 2024.

Souradip Chakraborty, Soumya Suvra Ghosal, Ming Yin, Dinesh Manocha, Mengdi Wang, Amrit Bedi, and Furong Huang. Transfer q-star : Principled decoding for LLM alignment. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=5PrShrKxoX.

Xin Chan, Xiaoyang Wang, Dian Yu, Haitao Mi, and Dong Yu. Scaling synthetic data creation with 1,000,000,000 personas. *arXiv preprint arXiv:2406.20094*, 2024.

Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology*, 15(3):1–45, 2024.

Jiangjie Chen, Xintao Wang, Rui Xu, Siyu Yuan, Yikai Zhang, Wei Shi, Jian Xie, Shuang Li, Ruihan Yang, Tinghui Zhu, et al. From persona to personalization: A survey on role-playing language agents. *arXiv preprint arXiv:2404.18231*, 2024.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chen Qian, Chi-Min Chan, Yujia Qin, Yaxi Lu, Ruobing Xie, et al. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors in agents. *arXiv preprint arXiv:2308.10848*, 2023.

Yuheng Cheng, Ceyao Zhang, Zhengwen Zhang, Xiangrui Meng, Sirui Hong, Wenhao Li, Zihao Wang, Zekai Wang, Feng Yin, Junhua Zhao, et al. Exploring large language model based intelligent agents: Definitions, methods, and prospects. *arXiv preprint arXiv:2401.03428*, 2024.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Databricks. Llm inference performance engineering: Best practices. https://www.databricks.com/blog/llm-inference-performance-engineering-best-practices, 2023. Accessed on: 2023-11-25.

Peijie Dong, Lujun Li, Zhenheng Tang, Xiang Liu, Xinglin Pan, Qiang Wang, and Xiaowen Chu. Pruner-zero: Evolving symbolic pruning metric from scratch for large language models. In *Forty-first International Conference on Machine Learning*.

Peijie Dong, Lujun Li, Yuedong Zhong, Dayou Du, Ruibo Fan, Yuhan Chen, Zhenheng Tang, Qiang Wang, Wei Xue, Yike Guo, and Xiaowen Chu. Stbllm: Breaking the 1-bit barrier with structured binary llms, 2024. URL https://arxiv.org/abs/2408.01803.

Zhen Dong, Zhewei Yao, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. HAWQ: hessian aware quantization of neural networks with mixed-precision. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*, pp. 293–302. IEEE, 2019. doi: 10.1109/ICCV.2019.00038. URL https://doi.org/10.1109/ICCV.2019.00038.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. In *Forty-first International Conference on Machine Learning*.

Elias Frantar and Dan Alistarh. Optimal brain compression: A framework for accurate post-training quantization and pruning. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=ksVGCOlOEba.

Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (eds.), *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pp. 10323–10337. PMLR, 2023. URL https://proceedings.mlr.press/v202/frantar23a.html.

Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. Break the sequential dependency of llm inference using lookahead decoding. In *Forty-first International Conference on Machine Learning*.

Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *Proc. USENIX OSDI*, 2020.

Chi Han, Qifan Wang, Hao Peng, Wenhan Xiong, Yu Chen, Heng Ji, and Sinong Wang. LM-infinite: Zero-shot extreme length generalization for large language models. In Kevin Duh, Helena Gomez, and Steven Bethard (eds.), *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 3991–4008, Mexico City, Mexico, 2024a. Association for Computational Linguistics. URL https://aclanthology.org/2024.naacl-long.222.

Shanshan Han, Qifan Zhang, Yuhang Yao, Weizhao Jin, Zhaozhuo Xu, and Chaoyang He. Llm multi-agent systems: Challenges and open problems. *arXiv preprint arXiv:2402.03578*, 2024b.

Tingxu Han, Chunrong Fang, Shiyu Zhao, Shiqing Ma, Zhenyu Chen, and Zhenting Wang. Token-budget-aware llm reasoning. *arXiv preprint arXiv:2412.18547*, 2024c.

Kostas Hatalis, Despina Christou, Joshua Myers, Steven Jones, Keith Lambert, Adam Amos-Binks, Zohreh Dannenhauer, and Dustin Dannenhauer. Memory matters: The need to improve long-term memory in llm-agents. In *Proceedings of the AAAI Symposium Series*, volume 2, pp. 277–280, 2023.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VtmBAGCN7o.

Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, et al. Inference without interference: Disaggregate llm inference for mixed downstream workloads. *arXiv preprint arXiv:2401.11181*, 2024.

Dong Huang, Qingwen Bu, Jie M Zhang, Michael Luck, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2023.

Jen-tse Huang, Jiaxu Zhou, Tailin Jin, Xuhui Zhou, Zixi Chen, Wenxuan Wang, Youliang Yuan, Maarten Sap, and Michael R Lyu. On the resilience of multi-agent systems with malicious agents. *arXiv preprint arXiv:2408.00989*, 2024.

Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pp. 59–72, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936363. doi: 10.1145/1272996.1273005.

Jinhao Jiang, Kun Zhou, Zican Dong, Keming Ye, Wayne Xin Zhao, and Ji-Rong Wen. Structgpt: A general framework for large language model to reason over structured data. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 9237–9251, 2023.

Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. S3: Increasing gpu utilization during generative inference for higher throughput. *Advances in Neural Information Processing Systems*, 36:18015–18027, 2023.

Hao Kang, Qingru Zhang, Souvik Kundu, Geonhwa Jeong, Zaoxing Liu, Tushar Krishna, and Tuo Zhao. Gear: An efficient kv cache compression recipefor near-lossless generative inference of llm. *arXiv preprint arXiv:2403.05527*, 2024.

Christopher Keith, Michael Robinson, Francis Duncan, Allan Worthington, Joseph Wilson, and Sofia Harris. Optimizing large language models: A novel approach through dynamic token pruning. 2024.

Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W. Mahoney, and Kurt Keutzer. Squeezellm: Dense-and-sparse quantization. *CoRR*, abs/2306.07629, 2023. doi: 10.48550/ARXIV.2306.07629. URL https://doi.org/10.48550/arXiv.2306.07629.

Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *International Conference on Learning Representations*, 2019.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023a.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proc. ACM SOSP*, 2023b.

Rémi Leblond, Jean-Baptiste Alayrac, L. Sifre, Miruna Pislar, Jean-Baptiste Lespiau, Ioannis Antonoglou, K. Simonyan, and O. Vinyals. Machine translation decoding beyond beam search. In *Conference on Empirical Methods in Natural Language Processing*, 2021. URL https://www.semanticscholar.org/paper/e8cc5b6204970a88cd1b2df491aa10c4333e083e.

Changhun Lee, Jungyu Jin, Taesu Kim, Hyungjun Kim, and Eunhyeok Park. OWQ: outlier-aware weight quantization for efficient fine-tuning and inference of large language models. In Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan (eds.), *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, pp. 13355–13364. AAAI Press, 2024a. doi: 10.1609/AAAI.V38I12.29237. URL https://doi.org/10.1609/aaai.v38i12.29237.

Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. Infinigen: Efficient generative inference of large language models with dynamic kv cache management. *arXiv preprint arXiv:2406.19707*, 2024b.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in Neural Information Processing Systems*, 36:51991–52008, 2023a.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in Neural Information Processing Systems*, 36:51991–52008, 2023b.

Yuan Li, Bingqiao Luo, Qian Wang, Nuo Chen, Xu Liu, and Bingsheng He. Cryptotrade: A reflective llm-based agent to guide zero-shot cryptocurrency trading. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 1094–1106, 2024a.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: Llm knows what you are looking for before generation. *ArXiv preprint*, abs/2404.14469, 2024b. URL https://arxiv.org/abs/2404.14469.

Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, and et al. Alpaserve: Statistical multiplexing with model parallelism for deep learning serving. In *Proc. USENIX OSDI*, 2023c.

Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Shuming Shi, and Zhaopeng Tu. Encouraging divergent thinking in large language models through multi-agent debate. *arXiv preprint arXiv:2305.19118*, 2023.

Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. Parrot: Efficient serving of llm-based applications with semantic variable. In *Proc. USENIX OSDI*, 2024.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. AWQ: activation-aware weight quantization for LLM compression and acceleration. *CoRR*, abs/2306.00978, 2023. doi: 10.48550/arXiv.2306.00978. URL https://doi.org/10.48550/arXiv.2306.00978.

Akide Liu, Jing Liu, Zizheng Pan, Yefei He, Gholamreza Haffari, and Bohan Zhuang. Minicache: Kv cache compression in depth dimension for large language models. *arXiv preprint arXiv:2405.14366*, 2024a.

Yiheng Liu, Hao He, Tianle Han, Xu Zhang, Mengyuan Liu, Jiaming Tian, Yutong Zhang, Jiaqi Wang, Xiaohui Gao, Tianyang Zhong, et al. Understanding llms: A comprehensive overview from training to inference. *arXiv preprint arXiv:2401.02038*, 2024b.

Yuhan Liu, Hanchen Li, Kuntai Du, Jiayi Yao, Yihua Cheng, Yuyang Huang, Shan Lu, Michael Maire, Henry Hoffmann, Ari Holtzman, et al. Cachegen: Fast context loading for language model applications. In *arXiv preprint arXiv:2310.07240*, 2023.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750*, 2024c.

LINHAO LUO, Yuan-Fang Li, Reza Haf, and Shirui Pan. Reasoning on graphs: Faithful and interpretable large language model reasoning. In *The Twelfth International Conference on Learning Representations*.

Xinyin Ma, Gongfan Fang, and Xinchao Wang. Llm-pruner: On the structural pruning of large language models. *arXiv preprint arXiv:2305.11627*, 2023.

Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. SONIC: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 285–301. USENIX Association, July 2021. ISBN 978-1-939133-23-6.

Shikib Mehri and Maxine Eskenazi. Unsupervised evaluation of interactive dialog with dialogpt. *arXiv preprint arXiv:2006.12719*, 2020.

Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. Spotserve: Serving generative large language models on preemptible instances. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 1112–1127, 2024.

Xinyi Mou, Xuanwen Ding, Qi He, Liang Wang, Jingcong Liang, Xinnong Zhang, Libo Sun, Jiayu Lin, Jie Zhou, Xuanjing Huang, et al. From individual to society: A survey on social simulation driven by large language model-based agents. *arXiv preprint arXiv:2412.03563*, 2024.

Xuchen Pan, Dawei Gao, Yuexiang Xie, Zhewei Wei, Yaliang Li, Bolin Ding, Ji-Rong Wen, and Jingren Zhou. Very large-scale multi-agent simulation in agentscope. *arXiv preprint arXiv:2407.17789*, 2024.

Gunho Park, Baeseong park, Minsub Kim, Sungjae Lee, Jeonghoon Kim, Beomseok Kwon, Se Jung Kwon, Byeongwook Kim, Youngjoo Lee, and Dongsoo Lee. LUT-GEMM: Quantized matrix multiplication based on LUTs for efficient inference in large-scale generative language models. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=gLARhFLE0F.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pp. 1–22, 2023.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. *arXiv*, 2023. URL https://arxiv.org/abs/2311.18677.

Giorgio Piatti, Zhijing Jin, Max Kleiman-Weiner, Bernhard Schölkopf, Mrinmaya Sachan, and Rada Mihalcea. Cooperate or collapse: Emergence of sustainable cooperation in a society of llm agents. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5:606–624, 2023.

Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with" gradient descent" and beam search. *arXiv preprint arXiv:2305.03495*, 2023.

Corban Rivera, Grayson Byrd, William Paul, Tyler Feldman, Meghan Booker, Emma Holmes, David Handelman, Bethany Kemp, Andrew Badger, Aurora Schmidt, et al. Conceptagent: Llm-driven precondition grounding and tree search for robust task planning and execution. *arXiv preprint arXiv:2410.06108*, 2024.

Hang Shao, Bei Liu, and Yanmin Qian. One-shot sensitivity-aware mixed sparsity pruning for large language models. In *ICASSP 2024 - 2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 11296–11300, 2024. doi: 10.1109/ICASSP48485.2024.10445737.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pp. 31094–31116. PMLR, 2023.

Zhengliang Shi, Shen Gao, Xiuyi Chen, Yue Feng, Lingyong Yan, Haibo Shi, Dawei Yin, Pengjie Ren, Suzan Verberne, and Zhaochun Ren. Learning to use tools via cooperative and interactive agents. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 10642–10657, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-emnlp.624. URL https://aclanthology.org/2024.findings-emnlp.624.

Aaditya K Singh and DJ Strouse. Tokenization counts: the impact of tokenization on arithmetic in frontier llms. *arXiv preprint arXiv:2402.14903*, 2024.

Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615*, 2022.

Jiashuo Sun, Chengjin Xu, Lumingyuan Tang, Saizhuo Wang, Chen Lin, Yeyun Gong, Lionel Ni, Heung-Yeung Shum, and Jian Guo. Think-on-graph: Deep and responsible reasoning of large language model on knowledge graph. In *The Twelfth International Conference on Learning Representations*.

Mingjie Sun, Zhuang Liu, Anna Bair, and J Zico Kolter. A simple and effective pruning approach for large language models. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=PxoFut3dWW.

Chris Sypherd and Vaishak Belle. Practical considerations for agentic llm systems. *arXiv preprint arXiv:2412.04093*, 2024.

Xin Tan, Yimin Jiang, Yitao Yang, and Hong Xu. Teola: Towards end-to-end optimization of llm-based applications. *arXiv preprint arXiv:2407.00326*, 2024.

Zhenheng Tang, Shaohuai Shi, Xiaowen Chu, Wei Wang, and Bo Li. Communication-efficient distributed deep learning: A comprehensive survey. *arXiv preprint arXiv:2003.06307*, 2020.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

MLC team. Mlc-llm. *GitHub*. URL https://github.com/mlc-ai/mlc-llm. Commit: 3358029, Accessed on: 2023-11-25.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, a.

Qian Wang, Zhen Zhang, Zemin Liu, Shengliang Lu, Bingqiao Luo, and Bingsheng He. Ex-graph: A pioneering dataset bridging ethereum and x. *arXiv preprint arXiv:2310.01015*, 2023.

Qian Wang, Tianyu Wang, Qinbin Li, Jingsheng Liang, and Bingsheng He. Megaagent: A practical framework for autonomous cooperation in large-scale llm agent systems. *arXiv preprint arXiv:2408.09955*, 2024a.

Qineng Wang, Zihao Wang, Ying Su, Hanghang Tong, and Yangqiu Song. Rethinking the bounds of llm reasoning: Are multi-agent discussions the key? *arXiv preprint arXiv:2402.18272*, 2024b.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, b.

Yidong Wang, Qi Guo, Wenjin Yao, Hongbo Zhang, Xin Zhang, Zhen Wu, Meishan Zhang, Xinyu Dai, Min zhang, Qingsong Wen, Wei Ye, Shikun Zhang, and Yue Zhang. Autosurvey: Large language models can automatically write surveys. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024c. URL https://openreview.net/forum?id=FExX8pMrdT.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. In *arXiv preprint arXiv:2305.05920*, 2023a.

Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism. *arXiv*, 2024. URL https://arxiv.org/abs/2404.09526.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversation. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents*.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*, 2023b.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=NG7sS51zVF.

Frank F Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Z Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, et al. Theagentcompany: benchmarking llm agents on consequential real world tasks. *arXiv preprint arXiv:2412.14161*, 2024.

Fuzhao Xue, Yao Fu, Wangchunshu Zhou, Zangwei Zheng, and Yang You. To repeat or not to repeat: Insights from scaling llm under token-crisis. *Advances in Neural Information Processing Systems*, 36, 2024.

Dongjie Yang, Xiaodong Han, Yan Gao, Yao Hu, Shilin Zhang, and Hai Zhao. PyramidInfer: Pyramid KV cache compression for high-throughput LLM inference. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association for Computational Linguistics ACL 2024*, pp. 3258–3270, Bangkok, Thailand and virtual meeting, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.195. URL https://aclanthology.org/2024.findings-acl.195.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, Carlsbad, CA, July 2022a. USENIX Association. ISBN 978-1-939133-28-1.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for transformer-based generative models. In *Proc. USENIX OSDI*, 2022b.

Siyu Yuan, Jiangjie Chen, Ziquan Fu, Xuyang Ge, Soham Shah, Charles Robert Jankowski, Yanghua Xiao, and Deqing Yang. Distilling script knowledge from large language models for constrained language planning. *arXiv preprint arXiv:2305.05252*, 2023.

Guibin Zhang, Yanwei Yue, Zhixun Li, Sukwon Yun, Guancheng Wan, Kun Wang, Dawei Cheng, Jeffrey Xu Yu, and Tianlong Chen. Cut the crap: An economical communication pipeline for llm-based multi-agent systems. *arXiv preprint arXiv:2410.02506*, 2024a.

Kai Zhang, Fubang Zhao, Yangyang Kang, and Xiaozhong Liu. Memory-augmented llm personalization with short-and long-term memory coordination. *arXiv preprint arXiv:2309.11696*, 2023a.

Shizhuo Zhang, Curt Tigges, Stella Biderman, M. Raginsky, and T. Ringer. Can transformers learn to solve problems recursively? *arXiv.org*, 2023b. URL https://www.semanticscholar.org/paper/45c196d28d16b2a8c0a078e8b79bcb39887a8a9f.

Yuxin Zhang, Lirui Zhao, Mingbao Lin, Sun Yunyun, Yiwu Yao, Xingjia Han, Jared Tanner, Shiwei Liu, and Rongrong Ji. Dynamic sparse no training: Training-free fine-tuning for sparse LLMs. In *The Twelfth International Conference on Learning Representations*, 2024b. URL https://openreview.net/forum?id=1ndDmZdT4g.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36:34661–34710, 2023c.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, and et al. Efficiently programming large language models using sglang. *arXiv*, 2023. URL https://arxiv.org/abs/2312.07104.

Zangwei Zheng, Xiaozhe Ren, Fuzhao Xue, Yang Luo, Xin Jiang, and Yang You. Response length perception and sequence scheduling: An llm-empowered llm inference pipeline. *Advances in Neural Information Processing Systems*, 36, 2024.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *Forty-first International Conference on Machine Learning*.

Zixuan Zhou, Xuefei Ning, Ke Hong, Tianyu Fu, Jiaming Xu, Shiyao Li, Yuming Lou, Luning Wang, Zhihang Yuan, Xiuhong Li, et al. A survey on efficient inference for large language models. *arXiv preprint arXiv:2404.14294*, 2024.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. Language agents as optimizable graphs. *arXiv preprint arXiv:2402.16823*, 2024.

APPENDIX

A MORE RELATED WORKS

A.1 LLM-MA SYSTEMS

Recent breakthroughs in LLMs (Achiam et al., 2023; Team et al., 2023) have catalyzed the development of LLM-MA systems. These systems orchestrate multiple specialized agents to tackle complex tasks ranging from code generation to mathematical reasoning and real-world planning (Hong et al., 2023; Chen et al., 2023). By integrating advanced components such as persona design (Chen et al., 2024; Chan et al., 2024), strategic planning mechanisms (Yuan et al., 2023), and memory architectures (Zhang et al., 2023a; Hatalis et al., 2023), LLM-MA systems have demonstrated superior problem-solving capabilities compared to single-agent approaches. Representative frameworks like MetaGPT (Hong et al., 2023), AutoGen (Wu et al., 2023b), AgentVerse (Chen et al., 2023), and MegaAgent (Wang et al., 2024a) showcase how specialized agent roles can be effectively coordinated for complex task completion.

However, current LLM-MA systems lack theoretical foundations in their design approaches. While existing frameworks typically assign agent profiles and establish connection patterns (flat, hierarchical, or nested structures) (Han et al., 2024b; Huang et al., 2024), this design is based on human common practices (Hong et al., 2023; Li et al., 2023a), and lacks systematic guidance. The current classification of agent structures remains superficial, and there exists no comprehensive methodology to inform structural design decisions. This gap between practice and theory highlights the need for a more rigorous analytical framework in LLM-MA system design.

LLM Token Costs. LLMs have much token consumption during inference (Xue et al., 2024; Liu et al., 2024b; Keith et al., 2024; Han et al., 2024c), a challenge that becomes even more severe in LLM-MA systems. This is due to additional token overhead such as agent role definitions (Cheng et al., 2024) and inter-agent communication (Zhang et al., 2024a). Recent research has even questioned the necessity of multi-agent architectures; for example, (Wang et al., 2024b) demonstrates that single agents, when equipped with well-crafted prompts, can not only match the performance of multi-agent systems but also achieve significant token savings.

Despite these findings, a critical gap persists in understanding the token costs associated with LLM-MA systems. Currently, no comprehensive analysis benchmarks token consumption across different frameworks, leaving several fundamental questions unanswered: How are tokens distributed across various stages of task execution? What are the comparative token efficiencies of different LLM-MA architectures? Addressing these questions is essential for designing more efficient and scalable LLM-MA systems.

SOPs in LLM-MA Systems. Allocating SOPs is a common approach in designing agent profiles and tasks within LLM-based multi-agent (LLM-MA) systems (Hong et al., 2023; Huang et al., 2023; Park et al., 2023; Zhuge et al., 2024; Shi et al., 2024). These systems define SOPs for both individual agents and their communication protocols. While this method has proven effective in previous works, it has two major limitations: (1) Agents may possess unforeseen capabilities that cannot be anticipated during the human design stage but become relevant during task execution (Rivera et al., 2024; Sypherd & Belle, 2024; Piatti et al., 2024); (2) As the scale of LLM-MA systems grows—potentially involving thousands or even billions of agents—designing SOPs manually for each agent becomes infeasible (Mou et al., 2024; Pan et al., 2024).

A.2 LLM REASONING

The Chain-of-Thought (CoT) reasoning framework illustrates that detailed reasoning conducted in multiple stages is notably beneficial for the effectiveness of Large Language Models (LLMs), particularly when compared to the limitations of single-step reasoning (Wei et al., 2022). The advantage stems from the tendency of single-step methods to overlook essential intermediate stages that are vital for effective problem-solving (Wei et al., 2022; Wang et al., 2023; Li et al., 2024a). By emulating the cognitive processes observed in humans, the multi-step reasoning approach significantly boosts LLM performance (Wei et al., 2022).

Single LLM Call. The CoT method serves as a prime example of a single invocation of an LLM, where the model is utilized once. Recent investigations go beyond merely prompting LLMs for detailed reasoning and suggest methods for enabling advanced search strategies, such as Monte-Carlo Tree Search (MCTS) (Leblond et al., 2021) or Q-star search (Chakraborty et al., 2024), during the decoding process. Moreover, there is research advocating for the use of backtracking algorithms, which permit LLMs to revisit and reassess prior decisions, thus refining their overall outcomes (Fu et al.).

Multiple LLM Calls. In contrast, certain methodologies favor the use of multiple independent LLM calls, each potentially arriving at accurate answers (Brown et al., 2024). In addition to the solitary CoT application, the CoT-SC variant introduces the concept of multiple CoT-based LLM calls, which select the best answer from a range of options to enhance final results (Wang et al., b). Nevertheless, answers from these calls can exhibit direct dependencies. To improve the organization and execution of the reasoning process, techniques such as Tree-of-Thought (ToT) reasoning (Yao et al., 2024) and Graph-of-Thought (GoT) reasoning (Besta et al., 2024) have been proposed, which arrange reasoning steps into tree or graph formats. Additionally, some research indicates that integrating knowledge graphs can empower LLMs to reason within graph-based systems, further improving their reasoning capabilities (LUO et al.; Sun et al.). Prompt structuring into triplet formats using LLMs has also been shown to enhance reasoning performance (Jiang et al., 2023). In scenarios lacking a centralized controller, other studies suggest simulating collaborations between multiple LLM agents to jointly tackle problems (Li et al., 2023a; Hong et al., 2024; Liang et al., 2023; Du et al.).

Planning and Scheduling. Central to the concept of multi-step reasoning is the breakdown of a complex problem into smaller, manageable sub-problems, which are then resolved sequentially. This necessitates effective planning and scheduling. Recent research indicates a trend towards utilizing LLMs as meta-agents, which can independently orchestrate planning and scheduling by decomposing the original issue and allocating sub-problems to various LLMs according to a defined schedule (Hong et al., 2024; Wu et al.; Zhou et al.; Wang et al., a). Additionally, by leveraging external symbolic reasoning, LLMs can engage in planning and scheduling processes to tackle problems more effectively (Zhang et al., 2023b).

A.3 Inference Costs of LLMs.

LLM Compression. Model Pruning (Frantar & Alistarh, 2023; Sun et al., 2024; Shao et al., 2024; Zhang et al., 2024b; Dong et al.; Tang et al., 2020) refers to the systematic elimination of less significant parameters. Pruning enhances the model's efficiency by removing redundancies, all while maintaining its performance. Overall, pruning methods will evaluate the importance of each parameter using some specific metrics like Tarloy expansion (Sun et al., 2024; Shao et al., 2024; Ma et al., 2023) or Fisher information (Dong et al., 2019) and remove the unimportant ones. *Model Quantization* (Park et al., 2024; Frantar & Alistarh, 2022; Lee et al., 2024a; Kim et al., 2023; Lin et al., 2023; Dong et al., 2024) involves lowering the precision of model weights and activations from high-precision formats, such as 32-bit floating-point representations, to 8-bit integers. LLM compression can save the computational and memory costs, and it is orthogonal to the agent systems. The upper-level plannig and scheduling and prompt engineering are decoupled from LLM compression.

Prefilling and decoding. Prefilling and decoding are crucial steps in the process of deploying large language models (LLMs) for various applications. Prefilling refers to the preparatory phase during which initial context or input data is provided to the model to compute the kay-value attention states cache (Pope et al., 2023). This phase is essential in applications where context and coherence are paramount, such as in conversational agents or document completion systems. It ensures that the model can conduct next word prediction without recomputing the KV vectors of all previous tokens.

The decoding phase follows prefilling and involves the generation of output sequences based on the prepared context. Various decoding strategies, including greedy decoding, beam search (Pryzant et al., 2023), and sampling (Brown et al., 2024) methods, can be employed to balance the trade-off between diversity and quality of the generated text.

Computation Efficiency. The computation efficiency of large language models (LLMs) has garnered substantial attention in recent literature. Various studies have examined how optimizations at both the architectural and algorithmic levels can reduce the computational burden during inference. Techniques such as model pruning, quantization, and knowledge distillation stand out as prominent strategies that help reduce the number of operations required for inference without significantly compromising performance (Dong et al.; Lin et al., 2023). Additionally, researchers have explored the use of more efficient transformer architectures, such as the Longformer (Beltagy et al., 2020) and Reformer (Kitaev et al., 2019), which are designed to mitigate the quadratic computational complexity inherent in traditional transformer models, thus improving computation efficiency.

KV Cache Management. The management of the KV cache, which dynamically adjusts in size as new tokens are produced, is highlighted by PagedAttention (Kwon et al., 2023a). It treats the KV cache as a series of non-contiguous memory blocks according to the duration and length of request generations. Current LLM serving frameworks, such as TGI (tgi, 2023), vLLM (Kwon et al., 2023a), and TensorRT-LLM (ten), have adopted this approach.

KV Compression. To address the substantial memory requirements of LLM serving, many works propose to compress the KV cache. Some works utilize the quantization techniques to compress the KV cache (Kang et al., 2024; Liu et al., 2024c; Sheng et al., 2023; Liu et al., 2024a). Different works estimate the KV cache quantization value based on different clusters, including groupwise quantization (Sheng et al., 2023), asymmetric quantization techniques for the Key and Value caches (Liu et al., 2024c), per-channel basis (Liu et al., 2024c). KV cache pruning and sparsification aims to remove unimportant tokens from the KV cache, which can be further compressed (Xiao et al., 2024; Han et al., 2024a; Yang et al., 2024; Cai et al., 2024). Some works observe that the initial and recent tokens consistently have high attention scores between different layers and attention heads (Xiao et al., 2024; Han et al., 2024a) which can be attained during inference. Some works propose dynamic KV cache pruning based on evaluating the attention scores (Zhang et al., 2023c; Li et al., 2024b). And it is observed that the long-context scenarios show different KV cache compression patterns (Yang et al., 2024; Cai et al., 2024).

Long-context KV Cache. The long context KV cache is a critical component in the serving of LLMs, especially in scenarios where the context exceeds the GPU's memory capacity. Most systems choose to transfer the KV cache to the CPU to handle this issue. InfiniGen (Lee et al., 2024b) proposes anticipating the crucial KV cache entries by simulating the attention computation. LoongServe (Wu et al., 2024) dynamically adjusts the varying resource usage across different requests and phases (including pre-filling and decoding), minimizing KV cache migration overhead and the fragmentation of the KV cache when processing lengthy sequences.

Longer sequence will result to the long context KV cache, which will lead to the KV cache migration overhead. This significantly reduces the serving througputs because the cloud server has limited GPU memory. Besides, the latency will increase due to the longer sequence. Thus, it is important to understand the token costs in the multi-agent system.

A.4 Optimization of Serving LLMs

LLM serving has seen a surge of research activity in recent years, with many systems developed to address the different challenges. The systems include Clockwork (Gujarati et al., 2020), AlpaServe (Li et al., 2023c), Orca (Yu et al., 2022a), vllm (Kwon et al., 2023b), SGLang (Zheng et al., 2023) and others. These serving systems explore many aspects including batching, caching, placement, scheduling, model parallelism for the serving of single or multiple models.

Serving Metrics. Key metrics in LLM serving include SLO, Latency, Throughput, TTFT (Time To First Token), and TPOT (Time Per Output Token) (Databricks, 2023). Different systems implement attention kernels uniquely, influencing their respective performances (Liu et al., 2023). For instance, TGI (tgi, 2023) directly imports Flash and Paged attention (Kwon et al., 2023a) libraries, while other systems like LightLLM (lig, 2023) and MLC-LLM (team) use customized implementations or optimizations. Balancing low latency and high throughput is crucial, as these dual objectives often conflict and require a strategic approach to achieve optimal performance (Miao et al., 2024).

Scheduling Requests. Effective request scheduling in LLM serving aims to maximize resource utilization while ensuring responsiveness within latency service level objectives (SLO) (Ali et al., 2020). Unique challenges arise from the iterative nature of LLMs, necessitating innovative solutions such as *selective batching* and *iteration-level scheduling* (Yu et al., 2022b). Techniques such as FastServe prioritize shorter input lengths to enhance job completion times, addressing the complexities of varying request loads (Wu et al., 2023a). Batching multiple requests can improve GPU utilization, but variable response lengths may lead to inefficiencies during processing (Zheng et al., 2024). Techniques like Response Length Perception and continuous batching at the token level help mitigate these issues by grouping similar request lengths together (Jin et al., 2023). Continuous batching, now an industry standard, allows for the dynamic joining of new requests to active batches without halting ongoing computations. This adaptability ensures that computational resources are used more effectively, leading to reduced waiting times and better overall throughput.

Prefilling-Decoding Decoupling. Decoupling the prefill and decoding stages allows for independent processing, minimizing interference and optimizing resource allocation (Hu et al., 2024). Systems like TetriInfer employ scheduling algorithms that predict resource usage, further enhancing efficiency during the decoding phase. By partitioning these tasks, distinct hardware can be utilized for each stage, as demonstrated by Splitwise approach to separating machines based on phase requirements (Patel et al., 2023). This specialization not only improves performance but also significantly reduces costs and energy consumption.

A.5 Optimization of Serving Agents

The previous serving frameworks discussed in Section A.4 mainly focus on optimizing single LLM call as the prompt-response pair. However, the real-world agent applications usually have multiple LLM calls and interactions with external resources and tools.

Multi-Agent Workflow Orchestrator. Multi-agent systems utilizing large language model (LLM) calls can be effectively represented as directed acyclic graphs (DAGs), with each node symbolizing an LLM call (or agent) and the edges indicating the interdependencies between these calls (lan, 2024). This graphical representation helps in systematically estimating system latency and throughput, allowing for more efficient processing. Unlike traditional single LLM call architectures, which handle requests in isolation, multi-agent workflows involve chains of requests that are interdependent. PromptFlow pro (2023) supports chains of native and semantic functions and visualizes them as a graph

Traditional DAG-Aware Serving Systems. Prior DAG-aware serving systems have implemented various optimization techniques to enhance the efficiency of request processing. These systems(Apache, 2019; Isard et al., 2007; Mahgoub et al., 2021) leverage task dependency to improve scheduling and execution, optimizing the use of parallel processing and minimizing communication overheads. However, they typically lack the understanding of application-specific request correlations, yielding suboptimal performance.

DAG-aware LLM Serving. Recent advancements in LLM orchestration have introduced novel frameworks to optimize efficiency and programmability. Parrot (Lin et al., 2024) introduces semantic variables (SVs), a mechanism to encapsulate task-specific context (e.g., instructions, examples) and expose dependencies and commonalities across LLM requests. By explicitly modeling these relationships, Parrot enables intelligent scheduling, reduces redundant computations, and facilitates shared prompt reuse. Traditional systems often overlook such correlations, leading to latency and resource inefficiencies.

Complementing this, SGLang (Zheng et al., 2023) addresses runtime inefficiencies in executing Language Model (LM) programs through RadixAttention, a technique for reusing Key-Value (KV) caches across LLM calls. Unlike conventional inference engines that discard KV caches post-execution, SGLang maintains an LRU-managed radix tree to cache and reuse these tensors. This approach efficiently identifies shared prefixes across requests, reducing redundant computations and memory overhead. Combined with cache-aware scheduling, RadixAttention enables significant throughput improvements, particularly in multi-call LM programs with overlapping contexts.

B Prompt-Response Examples

We provide prompt response examples in Table 3, Table 4, and Table 5.

C IMPACT STATEMENTS

Societal Impacts and Ethical Concerns. Our research did not involve any human subjects, data collection from individuals, or experiments on protected groups. We conducted all experiments using publicly available resources. The models and basic datasets utilized in this study are widely recognized and publicly accessible within the research community. All experiments were performed using resources that are publicly available. We would like to state that there are no conflicts of interest regarding this work. The datasets and models employed in this research are not only publicly available but also commonly used by others in the field.

Broader Impacts. Our analysis of token distribution in LLM-MA systems uncovers inefficiencies in inter-agent communication, exposing redundant tokens across Planner, Reasoner, and Verifier roles. By identifying token duplication, we provide insights for reducing token waste, enhancing agent coordination, and optimizing resource usage. This work advances multi-agent system development, benefiting dialogue systems, collaborative AI, and autonomous decision-making in large-scale LLM applications.

Table 3: Examples of using MetaGPT to program blackjack game.

Planner
Prompts: You are a Product Manager, named Alice, your goal is efficiently create a successful product that meets market demands and user expectations. the constraint is utilize the same language as the user requirements for seamless communication. ### Original Requirements Create a 2048 game "Language": "en_us", "Programming Language": "Python", "Original Requirements": "Create a 2048 game", "Product Goals": ["Create an engaging user experience", "Improve accessibility, be responsive", "More beautiful UI"], "User Stories": ["As a player, I want to be able to choose difficulty levels",...], "Competitive Analysis": ["2048 Game A: Simple interface, lacks responsive features", "play2048.co: Beautiful and responsive UI with my best score shown", "2048game.com: Responsive UI with my best score shown, but many ads"], other requirements... ## constraint Language: Please use the same language as Human INPUT. Format: output wrapped inside [CONTENT][/CONTENT] like format example...... ## action Follow instructions of nodes, generate output and make sure it follows the format example. **Response:** ## Design {"Implementation approach":"We will develop the 2048 game in Python utilizing the popular library 'pygame' for UI rendering and handling user input efficiently. The game will focus on delivering responsive design compatible with different screen sizes and devices, ..."} ## Task {"Required packages":["pygame","kivy"],"Required Other language third-party packages":["No third-party dependencies required"],"Logic Analysis":[["main.py","Contains Main class with run() method to start the game. Imports Game and UI."],["game.py","Contains Game class with methods for game logic, including move_tiles, generate_tile, check_game_over. Others .....}
Reasoner
Prompts: You are a Engineer, named Alex, your goal is write elegant, readable, extensible, efficient code. the constraint is the code should conform to standards like google-style and be modular and maintainable. Use same language as user requirement. You are a professional engineer; the main goal is to write google-style, elegant, modular, easy to read and maintain code Language: Please use the same language as the user requirement, but the title and code should be still in English. For example, if the user speaks Chinese, the specific text of your answer should also be in Chinese. ATTENTION: Use '##' to SPLIT SECTIONS, not '#'. Output format carefully referenced "Format example". ## Design {"Implementation approach":"We will develop the 2048 game in Python utilizing the popular library 'pygame' for UI rendering and handling user input efficiently. The game will focus on delivering responsive design compatible with different screen sizes and devices, ..."} ## Task {"Required packages":["pygame","kivy"],"Required Other language third-party packages":["No third-party dependencies required"],"Logic Analysis":[["main.py","Contains Main class with run() method to start the game. Imports Game and UI."],["game.py","Contains Game class with methods for game logic, including move_tiles, generate_tile, check_game_over. Others .....} **Response:** ## Legacy Code ‘‘‘Code —— main.py ‘‘‘## main.py import pygame from game import Game from ui import UI... {Other Code Snippets....}
Verifier

Prompts: You are a professional software engineer, and your main task is to review and revise the code. You need to ensure that the code conforms to the google-style standards, is elegantly designed and modularized, easy to read and maintain. Language: Please use the same language as the user requirement, but the title and code should be still in English. For example, if the user speaks Chinese, the specific text of your answer should also be in Chinese. ATTENTION: Use '##' to SPLIT SECTIONS, not '#'. Output format carefully referenced "Format example"
## Design {"Implementation approach":"We will develop the 2048 game in Python utilizing the popular library 'pygame' for UI rendering and handling user input efficiently. The game will focus on delivering responsive design compatible with different screen sizes and devices, ..."} 	
## Task {"Required packages":["pygame","kivy"],"Required Other language third-party packages":["No third-party dependencies required"],"Logic Analysis":[["main.py","Contains Main class with run() method to start the game. Imports Game and UI."],["game.py","Contains Game class with methods for game logic, including move_tiles, generate_tile, check_game_over. Others ...... } 	
## Code to be Reviewed: main.py 	
import pygame
from game import Game
from ui import UI...
{Other Code Snippets....}
# Instruction: Based on the actual code, follow one of the "Code Review Format example". Note the code filename should be 'main.py'. Return the only ONE file 'main.py' under review. 	
## Code Review: Ordered List. Based on the "Code to be Reviewed", provide key, clear, concise, and specific answer. If any answer is no, explain how to fix it step by step. 1. Is the code implemented as per the requirements? If not, how to achieve it? Analyse it step by step. 2. Is the code logic completely correct? If there are errors, please indicate how to correct them. 3. Does the existing code follow the "Data structures and interfaces"? ... 	
Response: ## Code Review: main.py: 1. No, we should fix the logic of class A due to ... 2. ... 3. ...
## Actions 1. Fix the 'handle_events' method to update the game state only if a move is successful. 	
"'python
def handle_events(self):
  for event in pygame.event.get():
  if event.type == pygame.QUIT:
Other code revision...

Table 4: Examples of using MetaGPT to program blackjack game.

Planner

Prompts: You are a Product Manager, named Alice, your goal is efficiently create a successful product that meets market demands and user expectations. the constraint is utilize the same language as the user requirements for seamless communication.
Original Requirements Write a cli Blackjack Game
"Language": "en_us", "Programming Language": "Python", "Original Requirements": "Create a 2048 game", "Product Goals": ["Create an engaging user experience", "Improve accessibility, be responsive", "More beautiful UI"], "User Stories": ["As a player, I want to be able to choose difficulty levels",...], "Competitive Analysis": ["2048 Game A: Simple interface, lacks responsive features", "play2048.co: Beautiful and responsive UI with my best score shown", "2048game.com: Responsive UI with my best score shown, but many ads"], other requirements...
constraint
Language: Please use the same language as Human INPUT. Format: output wrapped inside [CONTENT][/CONTENT] like format example......
action
Follow instructions of nodes, generate output and make sure it follows the format example.
Response: ## Design { "Implementation approach": "We will utilize Python to create a command-line Blackjack game. Our focus will be on implementing a user-friendly interface and the core Blackjack rules, such as hit, stand, double down, and split. }
"Required packages": ["argparse", "pickle", "rich"], "Required Other language third-party packages": ["No third-party dependencies required"], "Logic Analysis": [["main.py", "Contains the entry point of the program, utilizes other modules to manage game flow. Entry function start_game() is defined here."], Others...

Reasoner

Prompts: You are a Engineer, named Alex, your goal is write elegant, readable, extensible, efficient code. the constraint is the code should conform to standards like google-style and be modular and maintainable. Use same language as user requirement. You are a professional engineer; the main goal is to write google-style, elegant, modular, easy to read and maintain code Language: Please use the same language as the user requirement, but the title and code should be still in English. For example, if the user speaks Chinese, the specific text of your answer should also be in Chinese. ATTENTION: Use '##' to SPLIT SECTIONS, not '#'. Output format carefully referenced "Format example".
Design { "Implementation approach": "We will utilize Python to create a command-line Blackjack game. Our focus will be on implementing a user-friendly interface and the core Blackjack rules, such as hit, stand, double down, and split. }
"Required packages": ["argparse", "pickle", "rich"], "Required Other language third-party packages": ["No third-party dependencies required"], "Logic Analysis": [["main.py", "Contains the entry point of the program, utilizes other modules to manage game flow. Entry function start_game() is defined here."], Others...
Response:
Code: models.py
```python
## models.py
class Card:
  """Represents a single playing card, identified by its suit and value."""
  def __init__(self, suit: str, value: str):
    """
    Initialize a new card.
{Other Code Snippets....}
```

Verifier

Prompts: You are a professional software engineer, and your main task is to review and revise the code. You need to ensure that the code conforms to the google-style standards, is elegantly designed and modularized, easy to read and maintain.
Language: Please use the same language as the user requirement, but the title and code should be still in English. For example, if the user speaks Chinese, the specific text of your answer should also be in Chinese. ATTENTION: Use '##' to SPLIT SECTIONS, not '#'. Output format carefully referenced "Format example"
Design { "Implementation approach": "We will utilize Python to create a command-line Blackjack game. Our focus will be on implementing a user-friendly interface and the core Blackjack rules, such as hit, stand, double down, and split. }
"Required packages": ["argparse", "pickle", "rich"], "Required Other language third-party packages": ["No third-party dependencies required"], "Logic Analysis": [["main.py", "Contains the entry point of the program, utilizes other modules to manage game flow. Entry function start_game() is defined here."], Others...
Code: models.py
```python
## models.py
class Card:
  """Represents a single playing card, identified by its suit and value."""
  def __init__(self, suit: str, value: str):
    """
    Initialize a new card.
{Other Code Snippets....}
```
Instruction: Based on the actual code, follow one of the "Code Review Format example". Note the code filename should be 'main.py'. Return the only ONE file 'main.py' under review.
Code Review: Ordered List. Based on the "Code to be Reviewed", provide key, clear, concise, and specific answer. If any answer is no, explain how to fix it step by step. 1. Is the code implemented as per the requirements? If not, how to achieve it? Analyse it step by step. 2. Is the code logic completely correct? If there are errors, please indicate how to correct them. 3. Does the existing code follow the "Data structures and interfaces"? ...
Response: ## Code Review: main.py: 1. No, we should fix the logic of class A due to ... 2. ... 3. ...
Code Review: models.py
1. Yes, the code is implemented as per the requirements. It defines the classes 'Card' and 'Hand' which encapsulate the necessary attributes and methods as per the design.
2. Yes, the code logic is correct, however, there is a slight optimization...
3. Yes, the existing code correctly follows...

Table 5: Examples of using AgentVerse to generate code of HumanEval.

Planner

Prompts: You are the leader of a group of experts, now you need to recruit a small group of experts with diverse identity to correctly write the code to solve the given problems:

```
from typing import List
def below_zero(operations: List[int]) -> bool:
  """ You're given a list of deposit and withdrawal operations on a bank account that starts with zero balance. Your task is to detect if at any point the balance of account fallls below zero, and at that point function should return True. Otherwise it should return False. »> below_zero([1, 2, 3]) False »> below_zero([1, 2, -4, 5]) True """
```

You can recruit 4 expert in different fields. What experts will you recruit to better generate an accurate solution? Here are some suggestion: No advice yet." }, { "role": "user", "content": "# Response Format Guidance You should respond with a list of expert description. For example: 1. an electrical engineer specified in the filed of xxx. 2. an economist who is good at xxx. 3. a lawyer with a good knowledge of xxx. ... Only respond with the description of each role. Do not include your reason."

Response:

1. A software engineer experienced in algorithm design and complexity analysis to provide an efficient solution. 2. A financial expert specializing in banking operations who can understand the application of the problem in real-world scenarios and propose appropriate assumptions. 3. A cybersecurity expert who can guide the team in implementing the solution securely, especially if the problem has applications in sensitive financial systems. 4. A test engineer with expertise in writing unit and system tests to ensure the final system is robust and passes all edge cases.

Reasoner

Prompts: Can you complete the following code?

```
from typing import List
def below_zero(operations: List[int]) -> bool:
  """ You're given a list of deposit and withdrawal operations on a bank account that starts with zero balance. Your task is to detect if at any point the balance of account falls below zero, and at that point function should return True. Otherwise it should return False. »> below_zero([1, 2, 3]) False »> below_zero([1, 2, -4, 5]) True """
```

{ "role": "user", "content": "You are A software engineer experienced in algorithm design and complexity analysis to provide an efficient solution. Provide a correct completion of the code. Explain your reasoning. Your response should contain only Python code. Do not give any additional information. Use "'python to put the completed Python code in markdown quotes. When responding, please include the given code and the completion." }

Response: Here is the code with a simple modification. The function below_zero keeps track of the running balance. If the balance ever turns negative, it returns True. If it has processed all operations and the balance is still non-negative, it returns False.

```
from typing import List
def below_zero(operations: List[int]) -> bool:
  """ You're given a list of deposit and withdrawal operations on a bank account that starts with zero balance. Your task is to detect if at any point the balance of account fallls below zero, and at that point function should return True. Otherwise it should return False. »> below_zero([1, 2, 3]) False »> below_zero([1, 2, -4, 5]) True """
  balance = 0
  for operation in operations:
    balance += operation
    if balance < 0:
      return True
  return False
```

This solution scans the list once, meaning its time complexity is O(n).

Verifier

Prompts: Problem Complete the following function

```
from typing import List
def below_zero(operations: List[int]) -> bool:
  """ You're given a list of deposit and withdrawal operations on a bank account that starts with zero balance. Your task is to detect if at any point the balance of account fallls below zero, and at that point function should return True. Otherwise it should return False. »> below_zero([1, 2, 3]) False »> below_zero([1, 2, -4, 5]) True """
```

Experts The experts recruited in this turn includes: A software engineer experienced in algorithm design and complexity analysis to provide an efficient solution A financial expert specializing in banking operations who can understand the application of the problem in real-world scenarios and propose appropriate assumptions A cybersecurity expert who can guide the team in implementing the solution securely, especially if the problem has applications in sensitive financial systems A test engineer with expertise in writing unit and system tests to ensure the final system is robust and passes all edge cases

Writer's Solution:

```
from typing import List
def below_zero(operations: List[int]) -> bool:
  """ You're given a list of deposit and withdrawal operations on a bank account that starts with zero balance. Your task is to detect if at any point the balance of account fallls below zero, and at that point function should return True. Otherwise it should return False. »> below_zero([1, 2, 3]) False »> below_zero([1, 2, -4, 5]) True """
  balance = 0
  for operation in operations:
    balance += operation
    if balance < 0:
      return True
  return False
```

Tester's Feedback: Execution timed out."

You must respond in the following format: Score: (0 or 1, 0 for incorrect and 1 for correct) Response: (give your advice on how to correct the solution, and your suggestion on what experts should recruit in the next round)

Response: Score: 1 Response: The solution provided by the writer is correct. It iterates over the operations array, adding each operation to the balance. If at any point the balance falls below zero, it returns True.