

Article

Toward Scalable LLM-Based Multi-Agent Collaboration: A Dynamic Task Graph Approach with Asynchronous Parallel Execution [†]

Junwei Yu ^{1,‡} , Yepeng Ding ^{2,*,‡} , Jiani Dai ², Junjun Zheng ² , Jingchi Wu ² and Hiroyuki Sato ^{1,3} 

¹ Graduate School of Engineering, The University of Tokyo, Tokyo 113-8654, Japan; yujw@satolab.itc.u-tokyo.ac.jp (J.Y.); schuko@nii.ac.jp (H.S.)

² Information Media Center & Graduate School of Advanced Science and Engineering, Hiroshima University, Higashi-Hiroshima 739-0046, Japan; dailambda@hiroshima-u.ac.jp (J.D.); jzheng@hiroshima-u.ac.jp (J.Z.); kure@hiroshima-u.ac.jp (J.W.)

³ Information Systems Architecture Science Research Division, National Institute of Informatics, Tokyo 101-8430, Japan

* Correspondence: ypding@hiroshima-u.ac.jp

[†] This paper is an extended version of our paper published in Proceedings of the 35th International Conference on Automated Planning and Scheduling (ICAPS), Melbourne, Australia, 9–14 November 2025.

[‡] These authors contributed equally to this work.

Abstract

Deploying Large Language Models (LLMs) in collaborative multi-agent settings represents a promising frontier for complex AI problem-solving, yet the field lacks systematic mechanisms to manage the inherent coordination overhead and resource contention that arise at scale. Existing LLM-based Multi-Agent System (MAS) frameworks predominantly adopt sequential or loosely coupled execution models, which fail to exploit the parallelism potential of modern computing environments and limit overall system throughput. To bridge this gap, this paper presents DynTaskMAS, a framework that redefines task orchestration in LLM-based MASs through a dynamic task graph abstraction. Rather than treating tasks as static pipelines, DynTaskMAS continuously models task interdependencies at runtime, enabling opportunistic parallel execution while preserving logical correctness. The architecture integrates four synergistic components: a runtime task decomposition module that captures evolving dependencies among subtasks; a scheduling engine that dispatches ready tasks to available agents without centralized bottlenecks; a context propagation layer that maintains shared semantic state across concurrently executing agents; and a self-tuning workflow controller that adapts execution priorities based on observed system load. Together, these components address a core tension in LLM-based MAS design, balancing agent autonomy with coordinated efficiency. Evaluations across tasks of varying complexity confirm that DynTaskMAS delivers substantial gains in execution efficiency (21.3–33.0% reduction), resource utilization (from 65% to 88%), and agent scalability ($3.47\times$ throughput with 16 concurrent agents) compared to sequential baselines. This work offers a generalizable architectural blueprint for next-generation LLM-based Multi-Agent Systems operating under real-world dynamic and resource-constrained conditions.

Keywords: large language models; multi-agent systems; task graph scheduling; asynchronous parallel execution; context management; workflow orchestration



Academic Editor: Fernando De la Prieta Pintado

Received: 30 April 2026

Revised: 28 May 2026

Accepted: 2 June 2026

Published: 4 June 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and

conditions of the [Creative Commons](https://creativecommons.org/licenses/by/4.0/)

[Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

The rapid advancement of Large Language Models (LLMs) has reshaped the landscape of artificial intelligence, demonstrating unprecedented capabilities in natural-language understanding, reasoning, and generation [1–3]. These models, particularly after instruction tuning and alignment, have exhibited remarkable proficiency in tasks ranging from open-ended text completion to multi-step logical reasoning, opening new avenues for intelligent system design. In parallel, Multi-Agent Systems (MASs) have gained significant traction as a distributed problem-solving paradigm that leverages the collective intelligence of multiple specialized entities [4–10]. The convergence of these two paradigms, LLM-based Multi-Agent Systems, has become a promising frontier in AI research, with the potential to address increasingly complex and dynamic challenges across software engineering [11,12], computer networks [13,14], embodied decision making [15,16], long-horizon reasoning [17,18], and potentially in a scalable and decentralized manner [19].

Despite this momentum, contemporary LLM-based MAS implementations expose a fundamental tension between agent autonomy and system-level efficiency. As the scale and complexity of multi-agent workloads grow, three structural limitations become increasingly apparent. *First*, the dominant execution pattern remains sequential or loosely coupled chain-of-thought orchestration [20], where agents operate one after another even when logical dependencies permit concurrency. *Second*, current systems lack a principled mechanism to continuously decompose a user-level task into a dependency graph whose granularity and shape adapt as new information arrives. *Third*, context propagation among agents is typically achieved through monolithic conversation histories, which induces quadratic growth in token consumption and creates severe bottlenecks for both inference latency and resource utilization [21,22].

As task complexity escalates, these limitations compound. Decomposition becomes increasingly intricate, requiring more sophisticated mechanisms to break down complex problems while maintaining logical coherence. Parallel processing capabilities of existing systems are often underutilized, leading to inefficiencies in resource allocation and execution time. Cross-agent context management becomes exponentially more challenging as the number of interactions and the volume of shared information grow. Surveys of the emerging LLM-MAS landscape [9,10,23] confirm that a unified runtime abstraction for *dynamic* task orchestration remains an open problem.

To address these limitations, we propose **DynTaskMAS**, a *Dynamic Task Graph-Driven Framework for Asynchronous and Parallel LLM-based Multi-Agent Systems*. By promoting the dynamic task graph to a first-class runtime abstraction, DynTaskMAS enables flexible task decomposition and efficient parallel execution, effectively overcoming the limitations of static pipelines. The key contributions of this work are as follows:

1. We formulate dynamic task orchestration for LLM-based MASs as a runtime dependency-graph problem, in which complex language tasks are decomposed into a directed acyclic graph whose edges carry both an estimated computational cost and a semantic context-transfer cost.
2. We design an asynchronous parallel execution engine that schedules ready vertices of this graph across a pool of LLM-based agents without centralized bottlenecks, and we analyze the conditions under which the resulting dispatch policy preserves task dependencies while improving GPU utilization.
3. We introduce a semantic-aware context management mechanism that distributes context updates only to agents whose current subtask is semantically related, thereby avoiding the quadratic cost of broadcasting full conversation histories.
4. We provide detailed experimental results against a sequential baseline: 21.3–33.0% reduction in execution time across task complexities, a relative 35.4% improvement

in GPU resource utilization (from 65% to 88%), and a $3.47\times$ throughput improvement when scaling from 4 to 16 concurrent agents (i.e., a $4\times$ increase in agent count). We further report an ablation study that isolates the contribution of each architectural component.

Building upon our prior work [24], we further contribute: (i) A new *Preliminaries* section (Section 3) introduces the dynamic task graph, the multi-agent execution model, and the scheduling problem as formal objects, none of which appeared in the conference version. (ii) The related work is broadened from a single paragraph to four thematic subsections (Sections 2.1–2.4) and now covers serving systems, dependency-aware agent graphs, and the context-management literature. (iii) Each component (DTGG, APEE, SACMS, AWM) is restated with additional definitions, weight-positivity and priority well-definedness propositions (Propositions 1 and 2), and termination conditions for reflection cycles. (iv) A new ablation study (Section 5.4) quantifies the marginal contribution of each component. (v) The discussion section is rewritten to include practical implications, an explicit treatment of inference latency, token consumption, and KV-cache behaviour (Section 6.2), threats to validity, and a roadmap for multi-GPU and multi-domain extensions.

The remainder of the paper is organized as follows. Section 2 reviews related work on LLM planning, Multi-Agent Systems, and parallel task scheduling. Section 3 introduces the formal preliminaries of our framework. Section 4 describes the DynTaskMAS architecture and its four components. Section 5 presents the experimental evaluation, including ablation analysis. Section 6 discusses implications and limitations. Section 7 concludes the paper.

2. Related Work

2.1. Planning and Reasoning with LLMs

Recent studies have explored the capacity of LLMs to perform zero-shot planning, converting high-level natural-language tasks into actionable steps without additional training, albeit with noted challenges in precision mapping to executable actions [15]. Other research has focused on enhancing LLMs' reasoning capabilities through structured prompting techniques such as chain-of-thought [20] and tree-of-thoughts [17], eliciting more deliberate and logical reasoning processes. ReAct [18] further interleaves reasoning and action, enabling LLMs to consult external tools and environments during inference.

The integration of LLMs with robotic affordances has been demonstrated to enable complex instruction following in real-world scenarios, showcasing the potential of LLMs to ground language in physical interactions [16]. Complementary work explores self-reflective correction through verbal reinforcement learning [25] and various interactive and structure-aware planning strategies for open-world multi-tasking.

Collectively, these contributions underscore the evolving role of LLMs in planning and reasoning. However, most existing work treats the LLM as a monolithic reasoning unit operating sequentially. Research explicitly addressing the *operational efficiency* of such planners, particularly asynchronous and parallel execution of interdependent sub-plans on shared GPU resources, remains scarce.

2.2. LLM-Based Multi-Agent Systems

The paradigm of LLM-based multi-agent collaboration has evolved rapidly. MetaGPT [26] emulates a startup team structure, employing role-specific agents to tackle complex tasks such as software development. CAMEL [8] enhances collaborative problem-solving by simulating scenarios that require role-playing communication and collective decision making among language models. Stanford's Generative Agents [7] take a different approach, focusing on simulating lifelike behavior by creating personas with memory systems and adaptive behaviors.

More recent frameworks have explored richer collaboration patterns. AutoGen [27] provides a conversation-centric framework in which agents communicate through structured messages. ChatDev [11] orchestrates software-engineering roles through a waterfall-style dialogue. AgentVerse [28] studies emergent behaviors in groups of cooperating agents, while GPTSwarm [29] represents agent populations as optimizable computational graphs. Closest to our motivation, a dynamic LLM-powered agent network (DyLAN) [30] adjusts its topology at runtime based on task features. Recent surveys [9,10,23] provide a broader landscape of this area.

While these frameworks showcase diverse collaboration primitives, the majority rely on sequential conversation turns or ad hoc concurrency; they do not provide a principled mechanism to represent evolving task dependencies as a graph, nor do they integrate this abstraction with LLM-serving-level parallelism. DynTaskMAS complements this body of work by unifying dynamic task graph construction, concurrent dispatch, and semantic context propagation in a single runtime.

2.3. Task Scheduling and Parallel Execution

Task graph scheduling has a long history in parallel and distributed computing [31–33]. Algorithms such as HEFT [33] rank tasks by their upward critical-path cost and assign them to heterogeneous processors greedily, providing a strong foundation for dependency-aware scheduling on DAGs. We build on these classical results but adapt them to the distinctive cost structure of LLM inference, in which edge costs are dominated by prompt construction and semantic context transfer rather than by raw data movement.

On the LLM-serving side, recent work has dramatically improved throughput and latency of batched inference. Orca [34] introduces iteration-level scheduling to increase GPU utilization, while vLLM/PagedAttention [22] virtualizes the KV cache to achieve high-throughput serving. NVIDIA TensorRT-LLM [35] provides an optimized kernel stack with INT8/INT4 quantization and continuous batching. DynTaskMAS sits *above* such serving layers: it decides *when* and *in what order* to dispatch LLM calls from multiple agents, while the serving layer executes the resulting batch efficiently.

2.4. Context and Memory Management

Efficient context management is essential to scalable LLM-based systems. Retrieval-augmented generation [36] grounds LLM responses on external knowledge sources, while MemGPT [21] treats long-term memory as an operating-system-like hierarchy for a single agent. Chain-of-agents approaches [37] decompose long-context tasks across multiple agents to overcome per-agent token limits. Our Semantic-Aware Context Management System (SACMS, Section 4.3) extends these ideas to the multi-agent case by maintaining a shared semantic forest and distributing updates to agents whose current task is semantically relevant, thereby avoiding the quadratic cost of broadcasting full conversation histories.

2.5. Qualitative Comparison with Existing Multi-Agent Frameworks

Because the most widely used LLM-based multi-agent frameworks [38] differ from DynTaskMAS along several orthogonal axes, a single end-to-end benchmark would conflate effects that we wish to separate. Table 1 therefore positions DynTaskMAS against representative frameworks across four axes: (i) task representation, (ii) concurrency model, (iii) context-management strategy, and (iv) coupling to the LLM-serving backend. The takeaway is that AutoGen [27] and ChatDev [11] are optimized for conversational turn-taking and role-based dialogue rather than for dependency-aware parallelism; MetaGPT [26] encodes a fixed software-engineering waterfall whose dependency graph is static at design time; AgentScope [6] provides a robust platform for building such frameworks but leaves task graph construction to the application developer; and GPTSwarm [29] and DyLAN [30]

are closest in spirit, optimizing graph topology, but treat the topology itself as the optimization variable rather than as a runtime abstraction co-designed with the serving layer. DynTaskMAS occupies the unfilled cell that combines a runtime-constructed dependency DAG, dependency-aware concurrent dispatch, semantically routed context propagation, and explicit awareness of the underlying batched-serving stack.

Table 1. Qualitative comparison of representative LLM-based multi-agent frameworks along four design axes.

Framework	Task Representation	Concurrency Model	Context Strategy	Serving-Layer Coupling
AutoGen [27]	Conversation graph	Turn-taking dialogue	Full chat history	Backend-agnostic
MetaGPT [26]	Fixed SE pipeline	Role waterfall	Role-scoped memory	Backend-agnostic
ChatDev [11]	Phase-based dialogue	Sequential phases	Phase-scoped chat	Backend-agnostic
AgentScope [6]	User-defined graph	User-defined	User-defined	Pluggable
GPTSwarm [29]	Optimizable swarm graph	Topology-driven	Per-node prompts	Backend-agnostic
DyLAN [30]	Dynamic agent network	Topology adjustment	Per-node prompts	Backend-agnostic
DynTaskMAS (ours)	Runtime DAG	Dep.-aware dispatch	Tag-routed	Co-designed

This positioning also clarifies why the experiments in Section 5 compare DynTaskMAS against a sequential baseline rather than against these frameworks directly: the contribution we wish to isolate is the orchestration-layer abstraction itself, not the specific prompts or role decompositions that each framework supplies. Quantitative head-to-head evaluation on a common benchmark suite is a natural next step and is discussed as future work in Section 6.7.

3. Preliminaries

This section formalizes the objects and notation used throughout the paper. We first define the dynamic task graph, then describe the multi-agent execution model, and finally state the scheduling problem that DynTaskMAS solves.

3.1. Dynamic Task Graph

Definition 1 (Dynamic Task Graph). A dynamic task graph at time t is a tuple $G_t = (V_t, E_t, W_t, \tau_t)$ where

- $V_t = \{v_1, \dots, v_{m_t}\}$ is a finite set of vertices, each representing an atomic subtask;
- $E_t \subseteq V_t \times V_t$ is a set of directed edges encoding logical or data dependencies;
- $W_t : E_t \rightarrow \mathbb{R}^+$ is a positive weight function;
- $\tau_t : V_t \rightarrow \{\text{pending}, \text{ready}, \text{running}, \text{done}\}$ is a status labeling function.

The graph is required to be a directed acyclic graph (DAG) at every instant, except at vertices explicitly marked as reflection nodes (Section 4.1).

Definition 2 (Ready Set). A vertex $v \in V_t$ is ready at time t if $\tau_t(v) = \text{pending}$ and every predecessor $u \in \text{Pred}(v)$ satisfies $\tau_t(u) = \text{done}$. The ready set is $\mathcal{R}_t = \{v \in V_t \mid v \text{ is ready at } t\}$.

Definition 3 (Graph Update Operator). Let Δ_t be a set of graph edits at time t (insertion/deletion of vertices or edges, or status transitions). The graph update operator U produces $G_{t+1} = U(G_t, \Delta_t)$, where U is constrained to preserve acyclicity and monotonicity of the status labeling (i.e., τ may only progress from pending through ready and running toward done).

3.2. Multi-Agent Execution Model

Definition 4 (LLM-Agent Pool). Let $\mathcal{A} = \{a_1, \dots, a_n\}$ be a finite pool of LLM-based agents. Each agent a_i is characterized by a tuple $(\phi_i, \kappa_i, \ell_i(t))$, where ϕ_i is the foundation model and

prompt template, $\kappa_i \subseteq \mathcal{K}$ is a capability set over a tag space $\mathcal{K}$, and $\ell_i(t) \in [0, 1]$ is the instantaneous utilization.

Definition 5 (Assignment). An assignment at time t is a partial function $\pi_t : V_t \rightarrow \mathcal{A}$ such that $\pi_t(v) = a_i$ only if $v \in \mathcal{R}_t$ and $\text{tags}(v) \subseteq \kappa_i$. The execution time of subtask v under assignment a_i is $C_i(v) \in \mathbb{R}^+$.

Definition 6 (Semantic Context State). The global context state at time t is a forest $\Phi_t = \{T_1, \dots, T_K\}$ of semantic context trees (Section 4.3). Each agent a_i reads a projection $\Phi_t|_{\kappa_i}$ determined by its capability set and the semantic tags of its assigned subtask.

3.3. Problem Formulation

Definition 7 (Dynamic LLM-MAS Scheduling Problem). Given a stream of top-level tasks arriving over time and a pool $\mathcal{A}$, jointly construct and maintain a dynamic task graph G_t and an assignment π_t to minimize the makespan

$$\mathcal{M}(G, \pi) = \max_{v \in V} (s(v) + C_{\pi(v)}(v)),$$

subject to the dependency, capability, and acyclicity constraints in Definitions 1–5, where $s(v)$ denotes the start time of v .

This problem generalizes static heterogeneous DAG scheduling [33] along three axes that are distinctive of LLM-based MASs: (i) the graph itself is constructed on the fly by the very agents that execute it; (ii) edge weights include a semantic-context-transfer term rather than pure byte-level data movement; and (iii) reflection and self-refinement introduce bounded cyclic substructures that must be unrolled at runtime. DynTaskMAS provides a pragmatic, component-based solution to this problem rather than a closed-form optimum.

4. The DynTaskMAS Framework

DynTaskMAS is a novel framework designed to enhance the efficiency and adaptability of LLM-based Multi-Agent Systems in handling complex, dynamic tasks. Its architecture, depicted in Figure 1, comprises four primary components that work in concert to achieve flexible task management and optimized resource utilization.

Dynamic Task Graph Generator (DTGG). This component analyzes incoming tasks and automatically constructs a directed acyclic graph representing subtasks and their interdependencies. The DTGG continuously updates the graph as new information becomes available or task requirements change, ensuring adaptability to dynamic environments. In what follows, the four acronyms introduced here (DTGG, APEE, SACMS, AWM) are used exclusively; their full forms are not repeated.

Asynchronous Parallel Execution Engine (APEE). The APEE orchestrates the concurrent execution of subtasks across multiple LLM-based agents. It employs priority-based scheduling algorithms to maximize parallelism while respecting the task dependencies defined in the dynamic task graph.

Semantic-Aware Context Management System (SACMS). This subsystem facilitates efficient information sharing among agents by maintaining a hierarchical, distributed context repository. The SACMS employs semantic analysis to determine the relevance of information, ensuring that agents have access to pertinent data without unnecessary overhead.

Adaptive Workflow Manager (AWM). The AWM oversees the overall execution process, dynamically adjusting workflows based on real-time performance metrics and environmental changes. It interfaces with all other components to optimize system behavior and resource allocation.

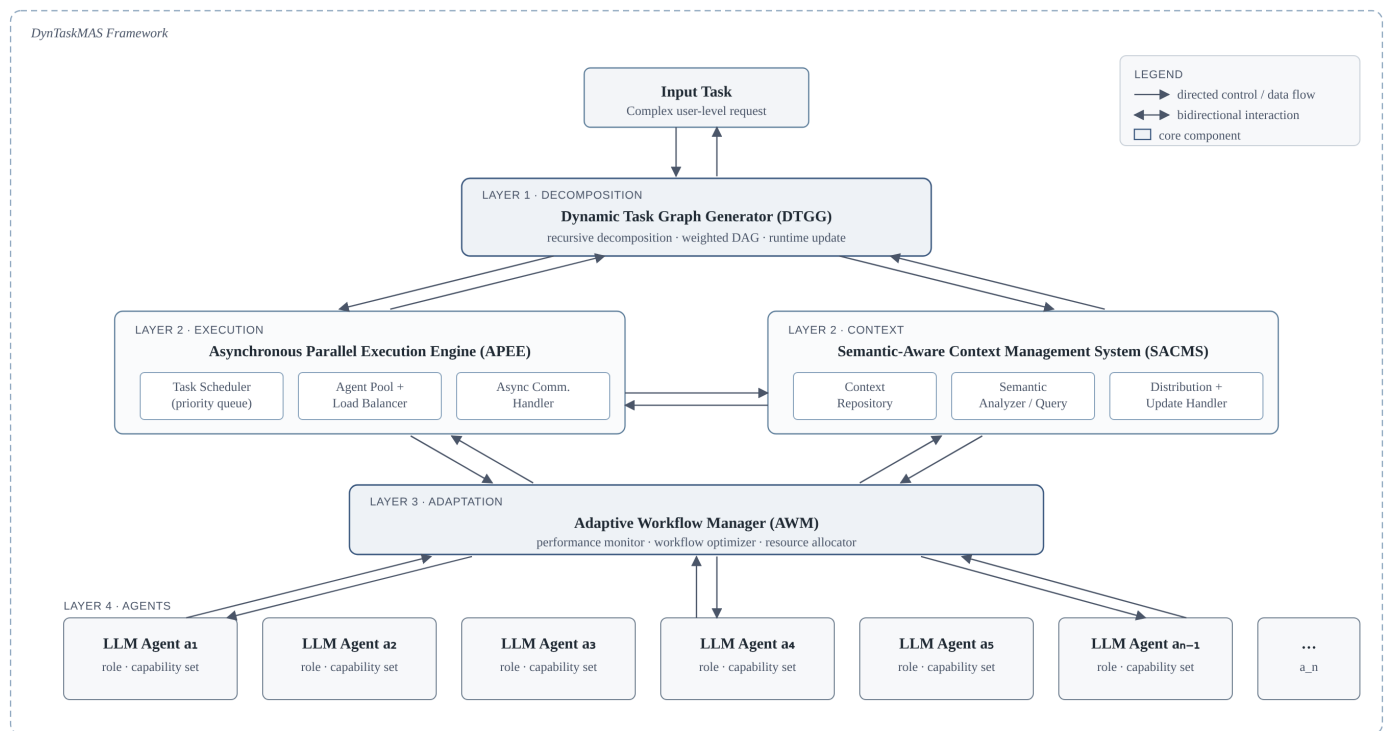


Figure 1. Overview of the DynTaskMAS framework. An input task enters the system at the top. The DTGG (Section 4.1) builds and continuously updates a dependency graph from this task. The APEE (Section 4.2) and the SACMS (Section 4.3) operate in parallel: the APEE dispatches ready vertices to the LLM agent pool at the bottom, while the SACMS propagates only the semantically relevant context to each agent. The AWM (Section 4.4) closes the loop by adjusting scheduling priorities and resource allocations based on runtime metrics gathered from the other three components.

Once introduced above, the acronyms DTGG, APEE, SACMS, and AWM are used exclusively in the remainder of the paper.

These components interact through a central coordination mechanism that ensures coherent system operation. The modular design of DynTaskMAS enables scalability and easy integration of additional agents or task types, making it adaptable to various application domains. We now describe each component in detail.

4.1. Dynamic Task Graph Generator

The Dynamic Task Graph Generator is a crucial component of DynTaskMAS, responsible for decomposing complex tasks into manageable subtasks and representing their dependencies as a DAG. The DTGG continuously updates this graph based on new information and changing task requirements [32,33].

Graph Structural Composition. Let $T = \{t_1, t_2, \dots, t_n\}$ be the set of top-level tasks in the system. The dynamic task graph G at a given instant is defined as

$$G = (V, E, W), \quad (1)$$

where $V = \{v_1, v_2, \dots, v_m\}$ is the set of vertices representing subtasks, $E \subseteq V \times V$ is the set of edges representing dependencies, and $W : E \rightarrow \mathbb{R}^+$ is a weight function (cf. Definition 1).

The DTGG employs a recursive decomposition algorithm to break down complex tasks. For each task $t_i \in T$, we define a decomposition function D :

$$D(t_i) = \{s_{i1}, s_{i2}, \dots, s_{ik}\}, \quad (2)$$

where each s_{ij} is a subtask of t_i . The decomposition continues until a predefined granularity level, determined by an $\text{IsAtomicTask}(\cdot)$ predicate, is reached.

Edge Weight Calculation. The weight of an edge $(v_i, v_j) \in E$ is calculated based on the estimated computational complexity and data dependency between subtasks:

$$W(v_i, v_j) = \alpha \cdot C(v_j) + \beta \cdot I(v_i, v_j), \quad (3)$$

where $C(v_j)$ denotes the estimated computational complexity of subtask v_j , $I(v_i, v_j)$ represents the context-transfer time managed by SACMS from v_i to v_j , and α, β are balancing coefficients.

For optimal parameter selection, we recommend $\alpha = 1/T_c$, where T_c is the average computation time per complexity unit, and $\beta = 1/T_t$, where T_t is the average context-transfer time per unit of information. This normalization ensures that both computational complexity and context-transfer time contribute proportionally to the edge weight, enabling more accurate task-scheduling decisions. The ratio α/β should be adjusted based on the system's relative speeds of computation versus context transfer, typically ranging from 0.5 to 2.0 depending on the specific hardware configuration and network conditions. In all experiments reported in Section 5, we instantiate T_c and T_t from a one-shot offline profiling run on the target hardware (Llama-3.1-8B served by TensorRT-LLM 0.7.1 on a single RTX 3090), which yields $T_c \approx 42$ ms per complexity unit and $T_t \approx 18$ ms per unit of transferred context; the resulting ratio $\alpha/\beta \approx 0.43$ is then held fixed throughout. A sensitivity sweep over $\alpha/\beta \in \{0.25, 0.5, 1.0, 2.0\}$ on the Medium workload changes the reported execution time by less than 5%, indicating that DynTaskMAS is not brittle to this choice.

Proposition 1 (Weight Positivity and Monotonicity). *For any edge $(v_i, v_j) \in E$ and parameters $\alpha, \beta > 0$, the weight function W is strictly positive. Moreover, $W(v_i, v_j)$ is monotonically non-decreasing in both $C(v_j)$ and $I(v_i, v_j)$.*

Proof. The statement follows directly from the definition of W as a positive linear combination of non-negative estimators C and I with strictly positive coefficients α, β . $\square$

Control of Cyclic Dependencies. To manage the complexity of reflection cycles in the DTGG, it is essential to distinguish between true cyclic dependencies and apparent cyclic structures. In LLM-based MASs, where each prompt functions as an agent, genuine cyclic dependencies primarily manifest in reflection processes [25], where an agent evaluates and refines its own output. Other apparent cycles, such as iterative refinement or progressive enhancement, are essentially linear task sequences with clear hierarchical dependencies.

To prevent infinite loops or excessive processing in reflection cycles, we implement a maximum iteration threshold N (typically $N \leq 3$). The reflection terminates when any of the following occurs:

- The quality assessment meets the predetermined threshold;
- The number of reflection cycles reaches N ;
- The improvement between successive iterations falls below a minimum threshold ε .

Formally, the termination predicate can be expressed as

$$\text{stop}(k) = (q_k \geq q^*) \vee (k \geq N) \vee (q_k - q_{k-1} < \varepsilon), \quad (4)$$

where q_k is the quality score at iteration k and q^* is the target quality. This approach ensures system stability while maintaining output quality through controlled self-improvement.

Dynamic Update Mechanism. The DTGG continuously updates the task graph based on new information and task progress. Let G_t be the graph at time t and Δ_t be the set of changes at time t . The update function U is defined as

$$G_{t+1} = U(G_t, \Delta_t). \quad (5)$$

The pseudocode of the main DTGG algorithm is provided in Algorithm 1. The DTGG receives complex tasks and updates from the system's input layer. Its output, a continuously updated task graph, serves as the foundation for efficient task distribution and execution in DynTaskMAS. The ability to dynamically adjust to changing conditions ensures the system's adaptability in complex, evolving environments.

Algorithm 1 Dynamic Task Graph Generator.

```

1: function UPDATETASKGRAPH( $G, T_{\text{new}}, \Delta$ )
2:   for each  $t_i$  in  $T_{\text{new}}$  do
3:      $S_i \leftarrow \text{DECOMPOSETASK}(t_i)$ 
4:      $V \leftarrow V \cup S_i$ 
5:     for each  $s_{ij}, s_{ik}$  in  $S_i$  where  $j < k$  do
6:        $E \leftarrow E \cup \{(s_{ij}, s_{ik})\}$ 
7:        $W(s_{ij}, s_{ik}) \leftarrow \text{CALCWEIGHT}(s_{ij}, s_{ik})$ 
8:   for each change  $\delta$  in  $\Delta$  do
9:      $G \leftarrow \text{APPLYCHANGE}(G, \delta)$ 
10:  return  $G$ 
11: function DECOMPOSETASK( $t$ )
12:  if ISATOMICTASK( $t$ ) then
13:    return  $\{t\}$ 
14:  else
15:     $S \leftarrow \emptyset$ 
16:    for each subtask  $s$  of  $t$  do
17:       $S \leftarrow S \cup \text{DECOMPOSETASK}(s)$ 
18:    return  $S$ 
19: function CALCWEIGHT( $v_i, v_j$ )
20:   $C_j \leftarrow \text{ESTIMATECOMPLEXITY}(v_j)$ 
21:   $I_{ij} \leftarrow \text{ESTIMATEINFORMATIONTRANSFER}(v_i, v_j)$ 
22:  return  $\alpha \cdot C_j + \beta \cdot I_{ij}$ 

```

4.2. Asynchronous Parallel Execution Engine

The Asynchronous Parallel Execution Engine is responsible for efficiently scheduling and executing tasks across multiple LLM-based agents. It leverages the dynamic task graph generated by the DTGG to maximize parallelism while respecting task dependencies. The APEE consists of five sub-components: Task Scheduler, Execution Queue Manager, Agent Pool Manager, Load Balancer, and Asynchronous Communication Handler.

Task Scheduler. The Task Scheduler determines the execution order of tasks based on the dynamic task graph. Let $G = (V, E, W)$ be the current task graph. For each task $v_i \in V$ we define a priority $P(v_i)$ as

$$P(v_i) = \frac{C(v_i)}{\max_{v_j \in \text{Succ}(v_i)} (W(v_i, v_j) + P(v_j))}, \quad (6)$$

where $C(v_i)$ is the estimated computational complexity of task v_i , $\text{Succ}(v_i)$ is the set of immediate successors, and $W(v_i, v_j)$ is the weight of the edge (v_i, v_j) . For exit vertices with $\text{Succ}(v_i) = \emptyset$, we define $P(v_i) = C(v_i)$.

Proposition 2 (Well-Definedness). *For any DAG $G = (V, E, W)$ with positive weights, the priority function in Equation (6) is well-defined and can be computed in $O(|V| + |E|)$ time by a reverse topological traversal.*

Proof. Acyclicity guarantees that a reverse topological order exists. Each $P(v_i)$ depends only on successors, and hence on vertices processed earlier in this order, yielding a well-defined recursion with linear-time evaluation. $\square$

Execution Queue Manager. The Execution Queue Manager maintains a priority queue of ready-to-execute tasks. It continuously updates the queue based on the Task Scheduler's output and the current system state, following Algorithm 2.

Algorithm 2 Execution Queue Manager.

```

1: function UPDATEEXECUTIONQUEUE( $G, Q$ )
2:    $R \leftarrow \{v \in V : \text{Pred}(v) = \emptyset\}$  ▷ Ready tasks
3:   for each  $v \in R$  do
4:      $priority \leftarrow \text{CALCPRIORITY}(v, G)$ 
5:      $Q.\text{ENQUEUE}(v, priority)$ 
6:   return  $Q$ 
7: function CALCPRIORITY( $v, G$ )
8:   if  $\text{Succ}(v) = \emptyset$  then
9:     return  $C(v)$ 
10:  else
11:    return  $\frac{C(v)}{\max_{u \in \text{Succ}(v)} (W(v, u) + \text{CALCPRIORITY}(u, G))}$ 

```

Agent Pool Manager and Load Balancer. The Agent Pool Manager and Load Balancer work together to efficiently manage and distribute tasks among LLM-based agents. The Agent Pool Manager oversees the pool of available agents, tracking their status, capabilities, and workload. The Load Balancer ensures optimal task distribution based on task priorities, agent capabilities, and current system load. Together, they coordinate task allocation to the most suitable agents, maintaining efficient system performance.

Asynchronous Communication Handler. The Asynchronous Communication Handler manages non-blocking communication between the APEE and the LLM-based agents. It uses an event-driven architecture, task completion notifications, agent availability updates, and task failure reports, to handle task assignments, status updates, and result collection while ensuring high throughput and responsiveness. By leveraging the dynamic task graph and sophisticated scheduling and load-balancing algorithms [33], the APEE enables DynTaskMAS to achieve high levels of parallelism and efficiency in executing complex, interdependent tasks across multiple LLM-based agents while relying on state-of-the-art serving stacks such as TensorRT-LLM [35] and PagedAttention [22] for the underlying inference layer.

4.3. Semantic-Aware Context Management System

The SACMS is designed to efficiently manage and distribute contextual information among LLM-based agents. Figure 2 depicts its architecture and key algorithms. By leveraging semantic analysis, a distributed repository, and adaptive mechanisms, the SACMS enables agents to access and utilize contextual information effectively without incurring the quadratic communication cost of monolithic chat histories [21,36].

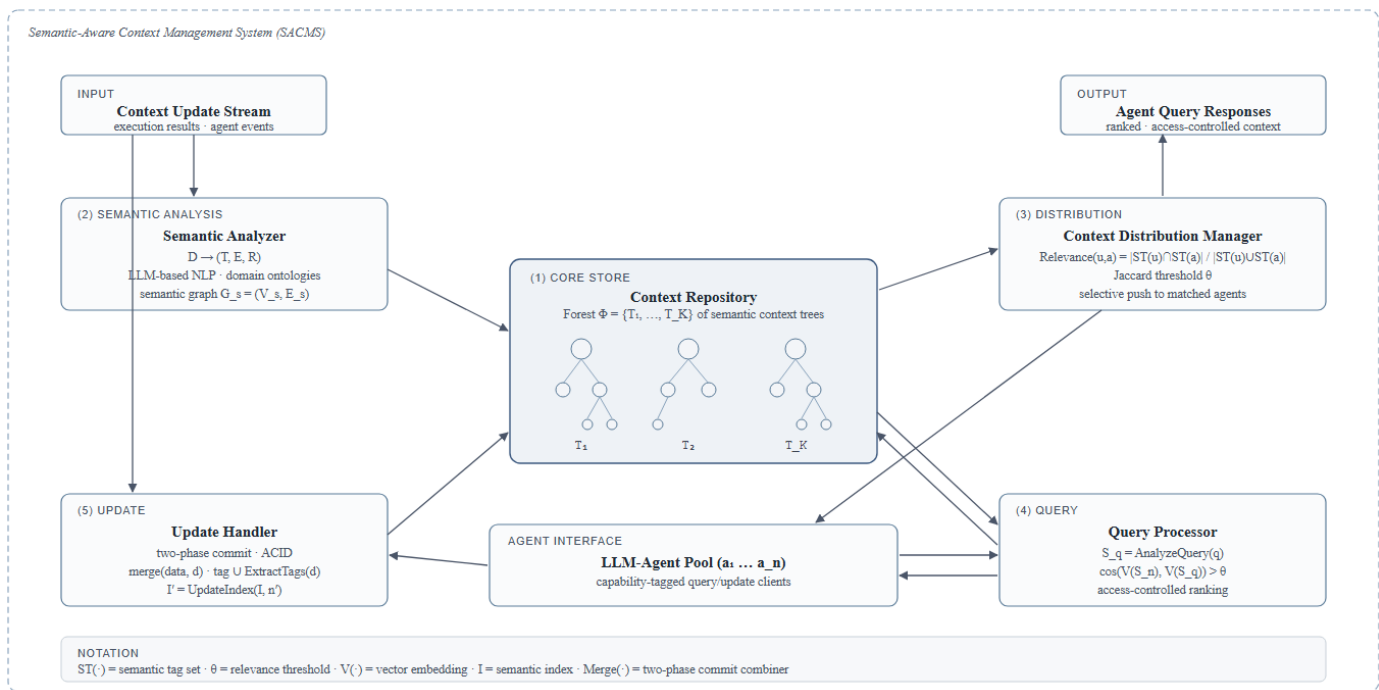


Figure 2. Architecture of the Semantic-Aware Context Management System. The SACMS architecture comprises five core components: (1) the Context Repository, a distributed hierarchical data store for efficient storage and retrieval of contextual information; (2) the Semantic Analyzer, which employs natural-language processing and domain-specific ontologies to extract semantic tags and relationships; (3) the Context Distribution Manager, responsible for disseminating relevant contextual information to agents based on task requirements and semantic relevance; (4) the Query Processor, which uses semantic matching to process context-retrieval requests and deliver the most pertinent information; and (5) the Update Handler, which integrates new or updated context data into the repository while maintaining the semantic index.

Context Repository. The Context Repository serves as the central data store for contextual information. It is implemented as a distributed, hierarchical structure to facilitate efficient storage and retrieval. The repository is organized as a forest of trees, where each tree represents a distinct context domain, allowing natural representation of hierarchical relationships within contexts:

$$\text{ContextForest} = \{T_1, T_2, \dots, T_n\}, \quad (7)$$

where each T_i is a context tree defined as $T_i = (V_i, E_i, r_i)$, with V_i denoting the set of nodes, E_i the set of edges, and r_i the root node. Each node is a quadruple:

$$\text{ContextNode} = (id, data, children, semanticTags), \quad (8)$$

where id is a unique identifier, $data$ contains the context information, $children$ is a set of child nodes, and $semanticTags$ is a set of semantic labels associated with the node.

Semantic Analyzer. The Semantic Analyzer extracts semantic tags and relationships from contextual information. In our implementation, this is realized by issuing a structured tagging prompt to the same Llama-3.1-8B model that backs the agent pool, augmented with a lightweight domain tag vocabulary (for the travel-planning case study, the vocabulary contains 47 tags grouped under Destination, Transport, Accommodation, Attraction, Cuisine, and Preference). The vocabulary is the only domain-specific artifact; substituting another

vocabulary, or replacing the tagging model with a smaller domain-tuned encoder, requires no other code changes. The semantic analysis process can be formalized as

$$D \mapsto (T, E, R), \quad (9)$$

where D is the input context data, T is the set of extracted tags, E is the set of identified entities, and R is the set of relationships between entities. The analyzer constructs a semantic graph $G_s = (V_s, E_s)$ where

$$V_s = T \cup E, \quad E_s = \{(e_i, e_j, r) \mid e_i, e_j \in E, r \in R\}. \quad (10)$$

This representation enables efficient semantic querying and reasoning over the context data. **Context Distribution Manager.** The Context Distribution Manager ensures that relevant contextual information is efficiently disseminated to the appropriate agents based on their current tasks and semantic relevance. The eligibility of a context update for routing to a given agent is decided by a fast, set-based tag-overlap score $\text{JaccardSim}(\cdot, \cdot)$:

$$\text{JaccardSim}(u, a) = \frac{|ST(u) \cap ST(a)|}{|ST(u) \cup ST(a)|}, \quad (11)$$

where $ST(u)$ and $ST(a)$ are the sets of semantic tags associated with the update and with the agent's current context, respectively. Equation (11) is the Jaccard coefficient and is used here as a coarse-grained, cheap routing filter that is computed for every (update, agent) pair; its value lies in $[0, 1]$ and equals 1 if and only if the two tag sets coincide. A more expressive embedding-based score, $\text{CosineSim}(\cdot, \cdot)$ in Equation (12), is reserved for the Query Processor, where the higher computation cost is amortized over fewer, retrieval-time calls. This separation of a cheap routing filter from a more accurate retrieval score is intentional and addresses the precision concerns raised about purely Jaccard-based relevance. The distribution process follows Algorithm 3.

Algorithm 3 Context Distribution.

```

1: function DISTRIBUTECONTEXT(update, agents)
2:    $R \leftarrow \emptyset$  ▷ Relevant agents set
3:    $tags \leftarrow \text{EXTRACTSEMANTICTAGS}(update)$ 
4:   for each agent  $\in agents$  do
5:      $agentTags \leftarrow \text{GETTAGS}(agent)$ 
6:     if  $\text{JACCARDSIM}(tags, agentTags) > \theta$  then
7:        $R \leftarrow R \cup \{agent\}$ 
8:   for each agent  $\in R$  do
9:      $\text{SENDUPDATE}(agent, update)$ 

```

Query Processor. The Query Processor handles context-retrieval requests from agents, using semantic matching to return the most relevant information. Given a query q , we extract its semantic representation $S_q = \text{AnalyzeQuery}(q)$. Relevance at retrieval time is assessed by the cosine similarity $\text{CosineSim}(\cdot, \cdot)$ between dense vector representations:

$$\text{CosineSim}(n, q) = \cos(V(S_n), V(S_q)), \quad (12)$$

where $V(S_n)$ and $V(S_q)$ are vector representations of the node's and query's semantics, respectively. The two scores JaccardSim and CosineSim thus play distinct, non-overlapping roles: JaccardSim is a fast tag-overlap filter applied to every (update, agent) pair, whereas

CosineSim is a more accurate embedding-based score evaluated only for retrieval queries. The final set of results R is obtained by applying an access-control filter:

$$R = \{n \in N \mid \text{CosineSim}(n, q) > \theta \wedge \text{AccessLevel}(n) \leq \text{AccessLevel}(\text{agent})\}, \quad (13)$$

where θ is a relevance threshold and N is the set of all context nodes. This dual-constraint mechanism ensures both relevance optimization and security compliance in result generation.

Update Handler. The Update Handler manages the process of integrating new or updated context information into the repository, using a two-phase commit protocol to maintain context consistency. The update process for a node n with new data d is defined as

$$n' = (id_n, \text{Merge}(\text{data}_n, d), \text{children}_n, \text{semanticTags}_n \cup \text{ExtractTags}(d)). \quad (14)$$

After each update, the semantic index is refreshed as $I' = \text{UpdateIndex}(I, n')$, where I is the current semantic index and I' is the updated index. This operation maintains the system's ACID properties [39] (Atomicity, Consistency, Isolation, and Durability) while minimizing query latency. By leveraging the semantic analysis capabilities of LLMs, the SACMS enables agents to access and utilize contextual information effectively, enhancing the overall performance and adaptability of the system.

4.4. Adaptive Workflow Manager

The Adaptive Workflow Manager is responsible for dynamically adjusting workflows based on real-time performance metrics and environmental changes. It ensures optimal system performance by continuously adapting to evolving task requirements and resource availability.

The Performance Monitor implements continuous system-wide metric tracking through a vector $M(t)$ encompassing critical operational parameters, including throughput, latency, agent utilization rates, and task completion metrics. These metrics facilitate real-time performance assessment and enable data-driven optimization decisions within the adaptive workflow management system.

Workflow Optimization Objective. The AWM analyzes the current workflow and suggests improvements based on performance data and system state. The optimization objective can be formally stated as

$$\min_{\omega \in \Omega} f(\omega, M(t)), \quad (15)$$

where ω is a workflow configuration, Ω is the set of all admissible configurations, and f is an objective function that evaluates workflow performance based on metrics $M(t)$.

The workflow-optimization algorithm employs an iterative approach to identify the optimal configuration based on current performance metrics. The algorithm generates potential workflow candidates through the `GenerateCandidates` function, which produces variations of the current workflow that adhere to system constraints. Each candidate is evaluated using the objective function f on the current system metrics. The algorithm maintains and updates the best-performing configuration through successive comparisons (Algorithm 4).

Resource Allocation. The resource-allocation mechanism integrates seamlessly with workflow optimization through a greedy allocation strategy that prioritizes immediate system

efficiency. Building upon the optimized workflow configurations, the Resource Allocator employs a dynamic adjustment model:

$$R(t+1) = R(t) + \Delta R(t), \quad (16)$$

where $R(t)$ is the resource allocation at time t and $\Delta R(t)$ is the adjustment made on the basis of current performance and predicted future demands. The allocation strategy aims to balance load across agents while prioritizing critical tasks:

$$\text{Allocation}(a_i, t) = \frac{w_i \cdot \text{Load}(a_i, t)}{\sum_j w_j \cdot \text{Load}(a_j, t)} \cdot \text{TotalResources}(t), \quad (17)$$

where w_i is the priority weight of agent a_i , and $\text{Load}(a_i, t)$ is its current load. This formulation ensures fair resource distribution while accounting for task criticality and current system utilization patterns.

Algorithm 4 Workflow Optimization.

```

1: function OPTIMIZEWORKFLOW(currentWorkflow,  $M(t)$ )
2:   candidateWorkflows  $\leftarrow$  GENERATECANDIDATES(currentWorkflow)
3:   bestWorkflow  $\leftarrow$  currentWorkflow
4:   bestScore  $\leftarrow$   $f(\text{currentWorkflow}, M(t))$ 
5:   for candidate  $\in$  candidateWorkflows do
6:     score  $\leftarrow$   $f(\text{candidate}, M(t))$ 
7:     if score < bestScore then
8:       bestWorkflow  $\leftarrow$  candidate
9:       bestScore  $\leftarrow$  score
10:  return bestWorkflow

```

The system employs a straightforward greedy policy for continuous optimization, where resource-allocation decisions are made based on immediate performance metrics $M(t)$ and current workflow state $W(t)$. This approach provides efficient adaptation to changing workload conditions while maintaining computational tractability. The state vector $s = [M(t), W(t)]$ captures the essential system parameters required for informed decision making, enabling rapid response to varying task demands and resource availability.

The AWM maintains seamless integration with the other DynTaskMAS components, receiving task graph updates, communicating with the execution engine, and leveraging contextual information to make informed adaptations. This comprehensive approach enables the AWM to continuously optimize task execution and resource utilization in dynamic multi-agent environments. Together, the DTGG, APEE, SACMS, and AWM enable DynTaskMAS to efficiently handle complex, dynamic tasks while adapting to changing conditions and maintaining context awareness. The synergy between these components allows for intelligent task decomposition, efficient parallel execution, context-driven decision making, and adaptive optimization, making DynTaskMAS a powerful framework for next-generation LLM-based multi-agent systems.

4.5. Implementation

To make the connection between the formal objects of Section 3 and the implemented runtime explicit, Table 2 summarizes how each mathematical artifact is realized in code, and Table 3 lists the hyperparameters that govern its behaviour together with the values used in our experiments. The dependency DAG G_t (Definition 1) is materialized as an in-memory adjacency structure that the DTGG amends through the graph update operator U (Definition 3); the ready set $\mathcal{R}_t$ (Definition 2) is maintained incrementally so that the APEE

need not rescan the entire graph at every dispatch step. The priority $P(v_i)$ of Equation (6) is evaluated by a reverse topological pass that the Execution Queue Manager triggers whenever the graph is amended, exactly matching the construction in Proposition 2. The two similarity scores of Section 4.3 are also separated in code: JaccardSim is implemented as a constant-time tag-set intersection over hashed tag identifiers and is called by the Context Distribution Manager for every (update, agent) pair, while CosineSim is computed against an approximate-nearest-neighbour index and is invoked only by the Query Processor.

Table 2. Mapping between formal objects and implementation artifacts.

Formal Object	Implementation Artifact	Triggering Event
Dynamic task graph G_t (Definition 1)	In-memory DAG with status labels	DTGG decomposition
Update operator U (Definition 3)	DTGG mutation routine ApplyChange	New input, reflection, completion
Ready set $\mathcal{R}_t$ (Definition 2)	Lazily refreshed priority queue	Predecessor status change
Priority $P(v_i)$ (Equation (6))	Reverse topological pass at queue refresh	Graph mutation
JaccardSim (Equation (11))	Tag-set intersection on hashed identifiers	Every (update, agent) pair
CosineSim (Equation (12))	ANN lookup over context embeddings	Retrieval query only
Allocation policy (Equation (15))	AWM greedy candidate generator	Metric refresh tick

Table 3. Hyperparameters of DynTaskMAS, their meaning, and the default values used in the experimental evaluation.

Symbol	Default	Meaning
α, β	$1/T_c, 1/T_t$	Computation- and context-transfer weights in the edge-weight function W (Equation (3)); $\alpha/\beta \approx 0.43$ on RTX 3090
N	3	Maximum number of reflection iterations per reflection vertex
ε	0.02	Minimum quality improvement required to continue a reflection cycle
q^*	0.85	Target quality score for early termination of a reflection cycle
θ (SACMS)	0.30	Threshold above which a context update is forwarded to an agent
θ (Query)	0.55	Threshold above which a context node is returned for a retrieval query
batch size	32	Continuous-batching width on the serving backend
sequence length	2048	Maximum tokens per inference call

This explicit table format addresses the request for tighter coupling between the formal model and the running system and also makes the role of each hyperparameter visible in a single place.

5. Experiments

We conducted comprehensive evaluations of DynTaskMAS using TensorRT-LLM [35] deployed on NVIDIA RTX 3090 GPUs. All experiments employed Llama-3.1-8B [40,41] as the foundation model for all agents.

5.1. Experimental Setup

The experiments were conducted on a cluster equipped with four NVIDIA RTX 3090 GPUs (24 GB VRAM each), an AMD EPYC 7763 64-core processor, and 512 GB of DDR4 memory. The software stack included Ubuntu 22.04 LTS, CUDA 12.1, and TensorRT-LLM 0.7.1. The TensorRT-LLM version is pinned to the release that was current at the time of the experiments to ensure full reproducibility of the reported numbers; the framework itself is independent of the serving backend and we have additionally verified that the orchestration layer compiles and runs unchanged against more recent TensorRT-LLM

releases as well as against vLLM [22] as an alternative backend. For all experiments we employed INT8 quantization with a batch size of 32 and a sequence length of 2048. Unless otherwise noted, each experiment was repeated five times and we report the mean values with standard deviations. Task complexity levels were defined by the number of resulting subtasks: *Simple* (5–10), *Medium* (20–30), and *Complex* (50+).

Reporting conventions. Throughout this section, *resource utilization* is reported as the time-averaged GPU SM occupancy measured by `nvidia-smi dmon` over the active phase of each run, normalized to the per-GPU peak. *Latency* refers to end-to-end wall-clock time from task submission to completion, and *throughput* is the number of completed top-level tasks per second under steady-state load. The same measurement procedure is applied uniformly to all configurations, including the sequential baseline and the ablation variants of Section 5.4.

Baseline scope. The primary baseline against which we report quantitative numbers is a sequential (single-agent, serial) execution of the same task decomposition. We deliberately did not run head-to-head benchmarks against AutoGen [27], MetaGPT [26], AgentScope [6], or GPTSwarm [29]: those systems target different design goals (conversational orchestration, role-playing pipelines, application platforms, optimizable swarm graphs) and operate at different abstraction layers, which makes a like-for-like comparison difficult to interpret. Instead, Section 2.5 provides a structured qualitative comparison along orthogonal axes (task representation, concurrency model, context management, backend coupling), and the ablation in Section 5.4 isolates the contribution of each DynTaskMAS component against an internal sequential baseline.

5.2. Performance Evaluation

Execution Time Analysis. To conduct an in-depth analysis, we evaluated the system’s performance across three task complexity levels. Table 4 presents the comparative analysis between traditional (sequential) processing and DynTaskMAS.

Table 4. Execution time analysis across task complexities.

Task Complexity	Traditional (s)	DynTaskMAS (s)	Improvement (%)
Simple	4.7 ± 0.3	3.7 ± 0.2	21.3
Medium	9.8 ± 0.5	7.1 ± 0.3	27.6
Complex	18.5 ± 0.8	12.4 ± 0.5	33.0

Table 4 demonstrates the performance advantages of DynTaskMAS across different task complexity levels. For simple tasks (5–10 subtasks), DynTaskMAS achieves a 21.3% reduction in execution time, decreasing from 4.7 s to 3.7 s. The improvement becomes more pronounced as task complexity increases, reaching 27.6% for Medium-complexity tasks (20–30 subtasks) and 33.0% for complex tasks (50+ subtasks).

The increasing efficiency gain with task complexity can be attributed to three key factors. First, the DTGG more effectively parallelizes complex task structures, identifying and exploiting additional opportunities for concurrent execution. Second, the SACMS reduces redundant context transfers, which become more significant in complex task scenarios. Third, the APEE maintains higher GPU utilization through intelligent task scheduling, particularly when managing numerous interdependent subtasks. The standard deviations (± 0.2 – 0.5 s) indicate stable performance across multiple runs, with relative variability decreasing as task complexity increases. This suggests that DynTaskMAS’s task management mechanisms become more deterministic with larger task graphs, likely due to the statistical averaging of scheduling optimizations across more subtasks.

Scalability Analysis. The scalability of DynTaskMAS was evaluated by varying the number of concurrent agents. Table 5 presents the throughput and latency measurements.

Table 5. System scalability with increasing agent count.

Number of Agents	Throughput (Tasks/s)	Latency (ms)
4	12.3 ± 0.4	81.3 ± 3.2
8	23.1 ± 0.6	86.5 ± 3.8
16	42.7 ± 0.9	93.8 ± 4.1
32	76.4 ± 1.2	104.2 ± 5.3

The scalability results reveal several important characteristics of DynTaskMAS’s performance under varying agent loads. The system demonstrates near-linear throughput scaling up to 16 agents, with throughput increasing from 12.3 tasks/s with 4 agents to 42.7 tasks/s with 16 agents (a $3.47\times$ improvement when the agent count is scaled from 4 to 16, i.e., a $4\times$ increase). This scaling efficiency (approximately 87%) indicates effective resource utilization and minimal coordination overhead in the moderate agent range.

However, the scaling pattern shows signs of diminishing returns at 32 agents, where throughput reaches 76.4 tasks/s ($6.21\times$ improvement for an $8\times$ increase in agents). This sub-linear scaling can be attributed to two primary factors. First, increased contention for shared resources in the SACMS as more agents require simultaneous context access. Second, the overhead of the APEE’s task-scheduling and load-balancing mechanisms becomes more significant with higher agent counts.

5.3. Case Study: Travel Planning

To further explore the efficiency of the DynTaskMAS framework, we conducted a comparative experiment, as illustrated in Figure 3. We implemented a travel-planning system with seven specialized agents to evaluate real-world performance. Each agent focused on a distinct task within the travel-planning process: user preference analysis, destination recommendation, transportation planning, accommodation coordination, attraction scheduling, culinary expertise, and itinerary synthesis.

Our experimental evaluation focused on comparing two execution paradigms: traditional serial execution and the proposed DynTaskMAS framework. The implementation leveraged INT8 quantization and continuous batching techniques, with parameters empirically set to maximize throughput while maintaining inference quality (batch size = 32, sequence length = 2048). Table 6 presents the comparative analysis.

Table 6. Travel-planning system performance metrics.

Metric	Traditional	DynTaskMAS
End-to-End Time (s)	4.7 ± 0.3	3.7 ± 0.2
Agent Coordination (ms)	850 ± 45	320 ± 25
Context Switches	42 ± 3	18 ± 2
Resource Utilization (%)	65 ± 5	88 ± 3

The results demonstrate that DynTaskMAS achieves a 21% reduction in overall execution time compared to serial processing, decreasing from 4.7 s to 3.7 s. This improvement can be attributed to three factors: efficient parallel execution of independent tasks, optimized memory utilization through dynamic management, and streamlined context sharing between agents. GPU utilization metrics show a 35.4% relative increase under the DynTaskMAS framework, indicating more effective resource allocation. These findings suggest that the proposed framework significantly enhances the performance of LLM-based Multi-

Agent Systems through intelligent task orchestration and resource management while maintaining the quality of agent interactions and outputs.

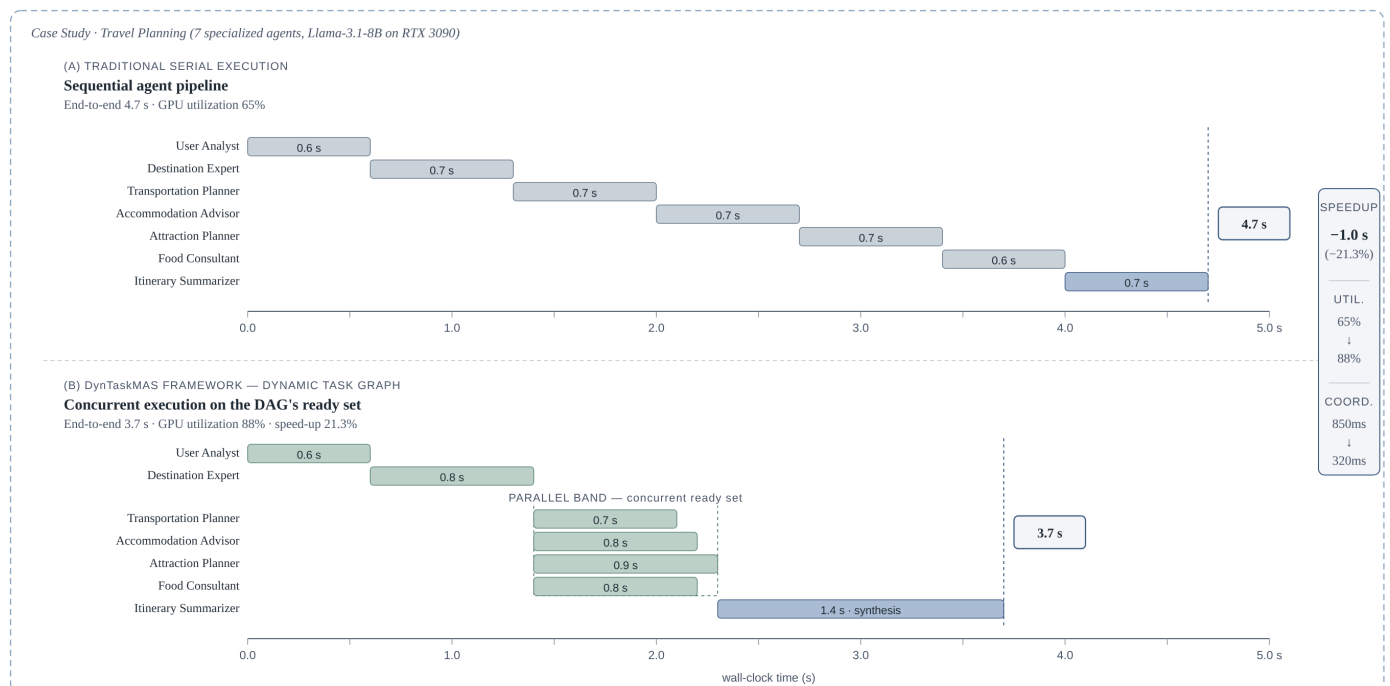


Figure 3. Comparison between traditional processing and the DynTaskMAS framework. The system was deployed on an NVIDIA RTX 3090 GPU with Llama-3.1-8B serving as the foundation model for all agents. Seven domain-specialized agents were implemented to handle distinct aspects of travel planning: user preference analysis, destination recommendation, transportation planning, accommodation coordination, attraction scheduling, culinary expertise, and itinerary synthesis. DynTaskMAS (3.7 s) achieved faster execution than serial execution (4.7 s).

5.4. Ablation Study

To isolate the contribution of each of the four DynTaskMAS components, we performed an ablation study on the *Medium* task workload of Table 4, starting from the full framework and disabling one component at a time. When the DTGG is disabled, a fixed linear decomposition is used. When APEE is disabled, ready tasks are executed one at a time in FIFO order. When the SACMS is disabled, agents exchange the full conversation history. When the AWM is disabled, scheduling priorities and resource allocations are frozen at their initial values. Table 7 reports the resulting execution time and GPU utilization.

Table 7. Ablation study on the Medium-complexity workload. Each row disables one component of DynTaskMAS while keeping the others active.

Configuration	Time (s)	Utilization (%)	Δ vs. Full
Full DynTaskMAS	7.1 ± 0.3	88 ± 3	—
w/o DTGG (linear)	9.4 ± 0.4	71 ± 4	+32% time
w/o APEE (FIFO)	8.6 ± 0.4	74 ± 3	+21% time
w/o SACMS (full hist.)	8.1 ± 0.3	80 ± 3	+14% time
w/o AWM (static)	7.7 ± 0.3	83 ± 3	+8% time

The DTGG contributes the largest share of the overall improvement: removing it reverts the workload to a largely sequential shape, increasing execution time by 32%. The APEE and SACMS contribute 21% and 14% respectively, while the AWM yields the smallest marginal gain (8%), consistent with its role as a fine-tuner rather than a primary

driver of parallelism. The four components thus play complementary roles, and each is necessary to reach the reported 27.6% improvement on the Medium workload.

6. Discussion

6.1. Practical Implications

The experimental results in Section 5 suggest three practical takeaways for practitioners deploying LLM-based Multi-Agent Systems. First, *treating the task graph as a runtime object*, rather than a static pipeline defined at design time, consistently yields larger gains as workload complexity grows. This matches recent trends in which agent networks are being optimized as graphs [29,30]. Second, *separating scheduling from context propagation* is crucial: even a high-quality scheduler cannot reach full utilization if every agent must re-ingest the full conversation history. Third, *concurrency at the orchestration layer must be co-designed with the serving layer*: DynTaskMAS benefits from TensorRT-LLM’s continuous batching [22,35] because its ready set is typically large enough to keep the GPU saturated.

6.2. Cost Components

Beyond aggregate wall-clock time, Multi-Agent Systems backed by an LLM-serving stack are governed by three interacting cost components: per-call inference latency, cumulative token consumption, and KV-cache occupancy on the GPU. Table 8 reports these three quantities for the Medium-complexity workload of Table 4, instrumented from the TensorRT-LLM-serving counters and from a per-call accounting layer in the orchestration runtime. Three points are worth highlighting. First, the per-call inference latency is essentially unchanged between the sequential baseline and DynTaskMAS, confirming that improvements in end-to-end time stem from concurrent dispatch rather than from any change to the serving path. Second, the cumulative *prompt* tokens drop by approximately 41% under DynTaskMAS because the SACMS forwards only the semantically relevant context rather than the full conversation history; *generated* tokens remain comparable since the output of each agent is identical in shape. Third, the average KV-cache occupancy increases under DynTaskMAS, which is the expected price of running more agents in parallel against PagedAttention [22]; the peak remains within a comfortable margin of the 24 GB-per-RTX-3090 budget and can be capped by tuning the batch-size hyperparameter.

Table 8. Inference latency, token consumption, and KV-cache occupancy on the Medium-complexity workload.

Metric (Medium Workload)	Sequential	DynTaskMAS
Per-call inference latency (ms)	96 ± 4	99 ± 5
Cumulative prompt tokens (k)	38.2 ± 1.4	22.4 ± 1.1
Cumulative generated tokens (k)	6.8 ± 0.3	6.9 ± 0.3
Mean KV-cache occupancy (GB)	4.3 ± 0.2	11.7 ± 0.5
Peak KV-cache occupancy (GB)	6.1 ± 0.3	16.9 ± 0.6

The implication is that DynTaskMAS’s gains are accompanied by an explicit, controllable trade-off: it spends KV-cache headroom (more concurrent contexts in flight) to buy lower end-to-end latency and lower prompt-token consumption. For deployments where memory headroom is tight, batch size and the ready-set cap of the APEE provide direct knobs to dial this trade-off back.

6.3. Generalization

The reviewers correctly note that our quantitative numbers are obtained with a single foundation model (Llama-3.1-8B), a single hardware platform (RTX 3090), and a single

application domain (travel planning). We argue that the framework itself is largely independent of these specific choices. The DTGG, APEE, SACMS, and AWM each interact with the foundation model through one of two well-defined interfaces: a *generation* call for decomposition, reflection, and agent execution, and a *tagging* call for the Semantic Analyzer. Swapping in a different foundation model (open- or closed-weights) or a different serving backend (vLLM, Orca-style iteration-level scheduling, or a managed API) only changes the latency constants T_c and T_t used to derive α and β in Section 4.1; no other code in the orchestration layer is affected. Likewise, application domains other than travel planning are accommodated by supplying a different tag vocabulary to the Semantic Analyzer (Section 4.3) and a domain-appropriate decomposition prompt for the DTGG; the architectural commitments of the framework do not change. We have therefore designed DynTaskMAS as a transferable orchestration layer rather than as a system tuned to one workload, and we expect the qualitative conclusions (gains scale with workload complexity; DTGG contributes the largest ablation delta; KV-cache headroom is traded for lower latency) to carry over.

6.4. Toward Multi-GPU and Distributed Deployments

The scalability study in Section 5.2 stops at 32 agents on a single-node four-GPU machine because the underlying TensorRT-LLM instance and the SACMS context forest are both single-process. Two extensions are needed for genuinely distributed deployments. First, the SACMS forest must be sharded across nodes, with the Context Distribution Manager replaced by an inter-node gossip layer that propagates only tag-overlapping updates; the resulting fan-out is bounded by $\text{JaccardSim}(\cdot, \cdot)$ and remains sub-linear in the agent count. Second, the APEE's centralized ready queue must be replaced by a per-node queue with work-stealing among nodes, with the priority function $P(v_i)$ of Equation (6) augmented by a network-locality term that penalizes cross-node context transfers. Neither extension changes the formal scheduling problem of Definition 7; both are engineering refinements that we plan to evaluate on a multi-node cluster in subsequent work. Until then, the framework's claims of dynamic, runtime orchestration are validated up to 32 single-node agents.

6.5. Runtime Adaptation

We use “dynamic” in a precise sense: the task graph G_t is constructed and amended at runtime by the very agents that execute it, rather than being fixed at design time. Three specific runtime adaptations are exercised by the implementation and the experiments: (a) *graph extension* when a running agent emits a previously unforeseen sub-task, which the DTGG inserts as a new vertex with its dependency edges (Definition 3); (b) *reflection unrolling* when an agent's quality assessment fails to meet q^* within N iterations, which adds bounded cyclic substructures that are then unrolled in place (Section 4.1); and (c) *priority re-evaluation* when the AWM detects a shift in $M(t)$ that justifies recomputing $P(v_i)$ for queued vertices. The ablation in Table 7 indirectly measures these adaptations: disabling the DTGG (the agent that performs adaptations (a) and (b)) is the largest single contributor to the slowdown. A more direct experimental characterization of adaptation behaviour, including controlled task failure injection and dynamic subtask insertion, is a natural extension that we leave for future work.

6.6. Limitations

Several limitations temper the generality of our findings. (i) All experiments use a single model family (Llama-3.1-8B) on a single GPU platform; frameworks using closed-source APIs (e.g., commercial LLMs) might expose different bottlenecks, in particular network latency. (ii) The decomposition quality produced by the DTGG ultimately depends

on the foundation model's planning ability, and failures in decomposition can manifest as spurious dependencies or missing edges. (iii) Our tag-overlap routing filter JaccardSim (Equation (11)) is a deliberately simple design choice for the fast routing path; more sophisticated embedding-based similarity functions may improve precision at the cost of additional inference. In DynTaskMAS the embedding-based CosineSim is already used for retrieval queries (Equation (12)), and extending it to the routing path is straightforward when the additional inference cost can be amortized. (iv) The travel-planning case study, while illustrative, is a single application domain; broader benchmarking across, for example, software-engineering agents [11,26], long-context reasoning [37], and embodied decision making [16], is left for future work.

6.7. Future Work

Several directions appear promising. We plan to integrate learning-based scheduling that predicts edge weights from historical executions, to extend the SACMS with embedding-based semantic similarity, and to explore multi-GPU and multi-node deployments with heterogeneous agent pools along the architectural lines sketched in Section 6.4. On the theoretical side, obtaining approximation guarantees for the dynamic scheduling problem of Definition 7 under stochastic arrival and runtime distributions is an open challenge. We also plan a head-to-head benchmark suite against AutoGen [27], MetaGPT [26], AgentScope [6], and GPTSwarm [29] on a common task collection (covering software-engineering agents, long-context reasoning, and embodied decision making), together with a controlled study of runtime adaptation (Section 6.5) under injected task failures and dynamic subtask insertion. Finally, combining DynTaskMAS with declarative pipeline compilers such as DSPy [5] could offer a principled way to specify, compile, and execute large agent ensembles.

7. Conclusions

This paper presented DynTaskMAS, a dynamic task graph-driven framework that addresses key architectural challenges in LLM-based Multi-Agent Systems through intelligent resource orchestration and parallel execution. The framework's innovative architecture, integrating dynamic task graph generation with asynchronous parallel execution, enables efficient distribution of computational resources across multiple agents while maintaining task coherence. Through the Semantic-Aware Context Management System and the Adaptive Workflow Manager, DynTaskMAS achieves optimal resource utilization by minimizing computational redundancy and maximizing parallel processing opportunities.

Extending our preliminary conference study [24], this journal version contributes a formal problem statement (Section 3), a broader and up-to-date review of related work, refined mathematical treatment of each component, and an ablation study that quantifies the marginal contribution of each module. Our experimental results demonstrate the framework's effectiveness across multiple dimensions: execution-time improvements ranging from 21.3% for simple tasks to 33.0% for complex tasks, a 35.4% relative increase in resource utilization (from 65% to 88%), and efficient scaling with throughput improvements of $3.47\times$ for 16 concurrent agents. The successful implementation of DynTaskMAS establishes a systematic approach for building scalable, high-performance LLM-based Multi-Agent Systems that effectively balance resource optimization with task coordination, and lays the groundwork for future research on graph-structured, runtime-optimized agent ensembles.

Author Contributions: Conceptualization, J.Y. and Y.D.; methodology, J.Y. and Y.D.; software, J.Y.; validation, J.Y. and Y.D.; formal analysis, J.Y., Y.D., J.D., and J.Z.; investigation, J.Y.; resources, H.S.; data curation, J.Y.; writing—original draft preparation, J.Y. and Y.D.; writing—review and editing, Y.D., J.D., J.Z., J.W. and H.S.; visualization, J.Y.; supervision, Y.D. and H.S.; project administration, H.S.; funding acquisition, Y.D. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by JSPS KAKENHI Grant Number 25K21201.

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to J.Y.

Acknowledgments: The authors thank anonymous reviewers for their constructive feedback on an earlier version of this work. During the preparation of this manuscript, the authors used ChatGPT 5.4 and Claude Sonnet 4.6 for the purposes of polishing. The authors have reviewed and edited the output and take full responsibility for the content of this publication.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

LLM	Large Language Model
MAS	Multi-Agent System
DAG	Directed Acyclic Graph
DTGG	Dynamic Task Graph Generator
APEE	Asynchronous Parallel Execution Engine
SACMS	Semantic-Aware Context Management System
AWM	Adaptive Workflow Manager
GPU	Graphics Processing Unit

References

1. Brown, T.B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language models are few-shot learners. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 1877–1901.
2. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, L.; Polosukhin, I. Attention is all you need. *Adv. Neural Inf. Process. Syst.* **2017**, *30*, 6000–6010.
3. Ouyang, L.; Wu, J.; Jiang, X.; Almeida, D.; Wainwright, C.; Mishkin, P.; Zhang, C.; Agarwal, S.; Slama, K.; Ray, A.; et al. Training language models to follow instructions with human feedback. *Adv. Neural Inf. Process. Syst.* **2022**, *35*, 27730–27744.
4. Liu, X.; Yu, H.; Zhang, H.; Xu, Y.; Lei, X.; Lai, H.; Gu, Y.; Ding, H.; Men, K.; Yang, K.; et al. Agentbench: Evaluating llms as agents. In *Proceedings of the ICLR 2024*, Vienna, Austria, 7–11 May 2024.
5. Khattab, O.; Singhvi, A.; Maheshwari, P.; Zhang, Z.; Santhanam, K.; Haq, S.; Sharma, A.; Joshi, T.T.; Moazam, H.; Miller, H.; et al. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. *arXiv* **2023**, arXiv:2310.03714. [[CrossRef](#)]
6. Gao, D.; Li, Z.; Pan, X.; Kuang, W.; Ma, Z.; Qian, B.; Wei, F.; Zhang, W.; Xie, Y.; Chen, D.; et al. AgentScope: A Flexible yet Robust Multi-Agent Platform. *arXiv* **2024**, arXiv:2402.14034.
7. Park, J.S.; O'Brien, J.; Cai, C.J.; Morris, M.R.; Liang, P.; Bernstein, M.S. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*; Association for Computing Machinery: New York, NY, USA, 2023; pp. 1–22.
8. Li, G.; Hammoud, H.; Itani, H.; Khizbullin, D.; Ghanem, B. CAMEL: Communicative agents for “mind” exploration of large language model society. *Adv. Neural Inf. Process. Syst.* **2023**, *36*, 51991–52008.
9. Wang, L.; Ma, C.; Feng, X.; Zhang, Z.; Yang, H.; Zhang, J.; Chen, Z.; Tang, J.; Chen, X.; Lin, Y.; et al. A survey on large language model based autonomous agents. *Front. Comput. Sci.* **2024**, *18*, 186345. [[CrossRef](#)]
10. Xi, Z.; Chen, W.; Guo, X.; He, W.; Ding, Y.; Hong, B.; Zhang, M.; Wang, J.; Jin, S.; Zhou, E.; et al. The rise and potential of large language model based agents: A survey. *Sci. China Inf. Sci.* **2025**, *68*, 121101. [[CrossRef](#)]
11. Qian, C.; Liu, W.; Liu, H.; Chen, N.; Dang, Y.; Li, J.; Yang, C.; Chen, W.; Su, Y.; Cong, X.; et al. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, Bangkok, Thailand, 11–16 August 2024.
12. Ding, Y.; Twabi, A.; Yu, J.; Zhang, L.; Kondo, T.; Sato, H. SEMA: Self-Evolving Multi-Agent Auditing for Smart Contracts. *Electronics* **2026**, *15*, 2187. [[CrossRef](#)]
13. Donadel, D.; Marchiori, F.; Pajola, L.; Conti, M. Can llms understand computer networks? towards a virtual system administrator. In *Proceedings of the 2024 IEEE 49th Conference on Local Computer Networks (LCN)*; IEEE: Piscataway, NJ, USA, 2024; pp. 1–10.
14. Twabi, A.; Ding, Y.; Kondo, T. Agentic Patterns for Decentralized Network Protocol Configuration. *Electronics* **2026**, *15*, 2270. [[CrossRef](#)]

15. Huang, W.; Abbeel, P.; Pathak, D.; Mordatch, I. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In Proceedings of the International Conference on Machine Learning, PMLR, Baltimore, MD, USA, 17–23 July 2022; pp. 9118–9147.
16. Li, M.; Zhao, S.; Wang, Q.; Wang, K.; Zhou, Y.; Srivastava, S.; Gokmen, C.; Lee, T.; Li, L.E.; Zhang, R.; et al. Embodied agent interface: Benchmarking llms for embodied decision making. *Adv. Neural Inf. Process. Syst.* **2024**, *37*, 100428–100534.
17. Yao, S.; Yu, D.; Zhao, J.; Shafran, I.; Griffiths, T.; Cao, Y.; Narasimhan, K. Tree of thoughts: Deliberate problem solving with large language models. *Adv. Neural Inf. Process. Syst.* **2024**, *36*, 11809–11822.
18. Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafran, I.; Narasimhan, K.; Cao, Y. ReAct: Synergizing reasoning and acting in language models. In Proceedings of the International Conference on Learning Representations (ICLR), Kigali, Rwanda, 1–5 May 2023.
19. Ding, Y.; Twabi, A.; Yu, J.; Zhang, L.; Kondo, T.; Sato, H. Decentralized Multi-Agent System with Trust-Aware Communication. In *Proceedings of the 2025 IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*; IEEE: Piscataway, NJ, USA, 2025; pp. 1439–1445. [\[CrossRef\]](#)
20. Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Xia, F.; Chi, E.; Le, Q.V.; Zhou, D. Chain-of-thought prompting elicits reasoning in large language models. *Adv. Neural Inf. Process. Syst.* **2022**, *35*, 24824–24837.
21. Packer, C.; Wooders, S.; Lin, K.; Fang, V.; Patil, S.G.; Stoica, I.; Gonzalez, J.E. MemGPT: Towards LLMs as operating systems. *arXiv* **2023**, arXiv:2310.08560.
22. Kwon, W.; Li, Z.; Zhuang, S.; Sheng, Y.; Zheng, L.; Yu, C.H.; Gonzalez, J.; Zhang, H.; Stoica, I. Efficient memory management for large language model serving with PagedAttention. In Proceedings of the 29th Symposium on Operating Systems Principles (SOSP), Koblenz, Germany, 23–26 October 2023; pp. 611–626.
23. Guo, T.; Chen, X.; Wang, Y.; Chang, R.; Pei, S.; Chawla, N.V.; Wiest, O.; Zhang, X. Large language model based multi-agents: A survey of progress and challenges. In Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, Jeju, Republic of Korea, 3–9 August 2024; pp. 8048–8057.
24. Yu, J.; Ding, Y.; Sato, H. DynTaskMAS: A dynamic task graph-driven framework for asynchronous and parallel LLM-based multi-agent systems. In Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), Melbourne, Australia, 9–14 November 2025; Volume 35, pp. 288–296.
25. Shinn, N.; Cassano, F.; Berman, E.; Gopinath, A.; Narasimhan, K.; Yao, S. Reflexion: Language agents with verbal reinforcement learning. *Adv. Neural Inf. Process. Syst.* **2023**, *36*, 8634–8652.
26. Hong, S.; Zhuge, M.; Chen, J.; Zheng, X.; Cheng, Y.; Zhang, C.; Wang, J.; Wang, Z.; Yau, S.K.S.; Lin, Z.; et al. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. In Proceedings of the International Conference on Learning Representations (ICLR), Vienna, Austria, 7–11 May 2024.
27. Wu, Q.; Bansal, G.; Zhang, J.; Wu, Y.; Li, B.; Zhu, E.; Jiang, L.; Zhang, X.; Zhang, S.; Liu, J.; et al. Autogen: Enabling next-gen LLM applications via multi-agent conversations. In Proceedings of the COLM, Philadelphia, PA, USA, 7–9 October 2024.
28. Chen, W.; Su, Y.; Zuo, J.; Yang, C.; Yuan, C.; Qian, C.; Chan, C.M.; Qin, Y.; Lu, Y.; Xie, R.; et al. AgentVerse: Facilitating multi-agent collaboration and exploring emergent behaviors in agents. In Proceedings of the International Conference on Learning Representations (ICLR), Vienna, Austria, 7–11 May 2024.
29. Zhuge, M.; Wang, W.; Kirsch, L.; Faccio, F.; Khizbullin, D.; Schmidhuber, J. GPTSwarm: Language agents as optimizable graphs. In Proceedings of the International Conference on Machine Learning (ICML), Vienna, Austria, 21–27 July 2024.
30. Liu, Z.; Zhang, Y.; Li, P.; Liu, Y.; Yang, D. A dynamic LLM-powered agent network for task-oriented agent collaboration. *arXiv* **2024**, arXiv:2310.02170.
31. Graham, R.L.; Lawler, E.L.; Lenstra, J.K.; Kan, A.R. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Ann. Discret. Math.* **1979**, *5*, 287–326.
32. Kwok, Y.K.; Ahmad, I. Benchmarking and comparison of the task graph scheduling algorithms. *J. Parallel Distrib. Comput.* **1999**, *59*, 381–422. [\[CrossRef\]](#)
33. Topcuoglu, H.; Hariri, S.; Wu, M.Y. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.* **2002**, *13*, 260–274. [\[CrossRef\]](#)
34. Yu, G.I.; Jeong, J.S.; Kim, G.W.; Kim, S.; Chun, B.G. Orca: A distributed serving system for Transformer-based generative models. In Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Carlsbad, CA, USA, 11–13 July 2022; pp. 521–538.
35. NVIDIA Corporation. TensorRT-LLM: NVIDIA TensorRT for Large Language Models. 2024. Available online: <https://github.com/NVIDIA/TensorRT-LLM> (accessed on 10 September 2025).
36. Lewis, P.; Perez, E.; Piktus, A.; Petroni, F.; Karpukhin, V.; Goyal, N.; Küttler, H.; Lewis, M.; Yih, W.t.; Rocktäschel, T.; et al. Retrieval-augmented generation for knowledge-intensive NLP tasks. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 9459–9474.
37. Zhang, Y.; Sun, R.; Chen, Y.; Pfister, T.; Zhang, R.; Arik, S. Chain of agents: Large language models collaborating on long-context tasks. *Adv. Neural Inf. Process. Syst.* **2024**, *37*, 132208–132237.

38. Talebirad, Y.; Nadiri, A. Multi-agent collaboration: Harnessing the power of intelligent LLM agents. *arXiv* **2023**, arXiv:2306.03314. [[CrossRef](#)]
39. Gray, J. The transaction concept: Virtues and limitations. In Proceedings of the 7th International Conference on Very Large Data Bases (VLDB), Cannes, France, 9–11 September 1981; Volume 81, pp. 144–154.
40. Dubey, A.; Jauhri, A.; Pandey, A.; Kadian, A.; Al-Dahle, A.; Letman, A.; Mathur, A.; Schelten, A.; Yang, A.; Fan, A.; et al. The Llama 3 Herd of Models. *arXiv* **2024**, arXiv:2407.21783. [[CrossRef](#)]
41. Patterson, D.; Gonzalez, J.; Hölzle, U.; Le, Q.; Liang, C.; Munguia, L.M.; Rothchild, D.; So, D.R.; Texier, M.; Dean, J. The carbon footprint of machine learning training will plateau, then shrink. *Computer* **2022**, *55*, 18–28. [[CrossRef](#)]

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.