Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

TCA-DARE: A Task-Complexity-Aware Dynamic Agent Routing Framework for Multi-Agent LLM Systems

Moksha Arora
Department of Computer Science and Engineering, Bharati Vidyapeeth's College of Engineering, A-4 Paschim Vihar, New Delhi 110063, India https://orcid.org/0009-0001-0016-6057

Bhawna Mittal
Department of Computer Science and Engineering, Bharati Vidyapeeth's College of Engineering, A-4 Paschim Vihar, New Delhi 110063, India https://orcid.org/0009-0004-7962-4648

Prof. Mohit Tiwari
mohit.tiwari@bharatividyapeeth.edu

Department of Computer Science and Engineering, Bharati Vidyapeeth's College of Engineering, A-4 Paschim Vihar, New Delhi 110063, India https://orcid.org/0000-0003-1836-3451

Research Article

Keywords: Large language models, Multi-agent systems, Agent orchestration, Task complexity estimation, Dynamic agent routing, Token-efficient inference

Posted Date: November 23rd, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11814023/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: The authors declare no competing interests.

TCA-DARE: A Task-Complexity-Aware Dynamic Agent Routing Framework for Multi-Agent LLM Systems

Moksha Arora[1], Bhawna Mittal[1], Mohit Tiwari[1,*]

[1]Department of Computer Science and Engineering, Bharati Vidyapeeth's College of Engineering, A-4 Paschim Vihar, New Delhi 110063, India

*Corresponding author: Mohit Tiwari (mohit.tiwari@bharatividyapeeth.edu)

ORCID: Moksha Arora 0009-0001-0016-6057; Bhawna Mittal 0009-0004-7962-4648; Mohit Tiwari 0000-0003-1836-3451

Emails: moksha.arora10@gmail.com; bhawnamittal9a@gmail.com; mohit.tiwari@bharatividyapeeth.edu

Abstract

Autonomous Large Language Model (LLM) agents have demonstrated significant capability in complex problem-solving. However, existing multi-agent systems predominantly rely on fixed, static execution topologies where a predefined suite of specialist agents is invoked sequentially regardless of query complexity. This rigid paradigm imposes a heavy computational tax, leading to inflated token consumption, high execution latency, and elevated operational costs on simple to moderately complex tasks. In this paper, we propose TCA-DARE (Task-Complexity-Aware Dynamic Agent Routing and Execution), an adaptive multi-agent framework that evaluates incoming academic requests prior to agent instantiation and allocates a minimal viable agent topology. By combining a lightweight four-feature mathematical complexity-scoring router (Layer 1), a dynamic specialist agent pool (Layer 2), a source-grounding verification guardrail (Layer 3, specified for future integration), and a telemetry logger (Layer 4), the framework routes queries across three distinct execution topologies: single-agent extraction (C_1), dual-pipeline synthesis (C_2), and full multi-agent orchestration (C_3). We present the complete system architecture, formal routing algorithm, and an implemented (not yet experimentally executed) local Python prototype built on Ollama (llama3) and tiktoken, alongside a structured multi-tier evaluation protocol designed for upcoming empirical validation against static multi-agent baselines.

Keywords: Large language models; Multi-agent systems; Agent orchestration; Task complexity estimation; Dynamic agent routing; Token-efficient inference

1. Introduction

The emergence of Large Language Models (LLMs) has catalyzed a paradigm shift from simple zero-shot prompting toward autonomous agentic workflows. While single-agent frameworks utilizing reasoning-acting loops (such as ReAct) excel at focused information retrieval, tackling open-ended, multi-step workflows requires collaborative multi-agent architectures. Frameworks like MetaGPT, ChatDev, and AutoGen organize agents into specialized roles (e.g., Product Manager, Coder, Reviewer) connected via structured communication graphs.

Despite their qualitative success, these multi-agent systems suffer from a structural inefficiency: static topology enforcement. In traditional multi-agent systems, every incoming query—regardless of whether it requires a simple factual lookup or a comprehensive systematic review—triggers the entire cohort of specialized agents. This static execution results in:

- **Redundant Token Inflation:** Context tokens can be consumed by unnecessary inter-agent messaging and prompt overhead even for simple tasks.
- **Latency & Cost Overhead:** Sequential agent handoffs add wall-clock delay and increase commercial API expenditure.
- **Error Cascade Risk:** Long agent interaction chains introduce unnecessary points of failure where hallucination can propagate.

To address these limitations, we introduce TCA-DARE, an architectural framework designed to bridge the gap between static orchestration and adaptive execution. Our core design hypothesis is that pre-filtering task complexity prior to agent instantiation can prune unnecessary computational overhead while preserving source-grounded verification on complex queries. In its current stage, this work delivers the complete formal architecture, routing algorithm, and a local prototype implementation, establishing the foundation for systematic multi-dataset benchmarking in future validation.

2. Related Work

2.1 Single-Agent Reasoning and Tool Augmentation

Early agent architectures focused on enhancing single-LLM trajectory execution. ReAct (Yao et al., 2022) established the interleaved Thought-Action-Observation framework, grounding LLM reasoning with search API calls. Toolformer (Schick et al., 2023) demonstrated self-supervised API usage learning, while Reflexion (Shinn et al., 2023) utilized verbal reinforcement memory buffers for iterative self-correction. SWE-agent (Yang et al., 2024) emphasized the impact of Agent-Computer Interfaces (ACIs) on coding performance.

2.2 Role-Based Multi-Agent Workflows

To solve multi-phase problems, multi-agent cooperation emerged. CAMEL (Li et al., 2023) introduced inception prompting for autonomous role-playing pairs. ChatDev (Qian et al., 2023) structured software development into corporate roles over a waterfall chat chain. MetaGPT (Hong et al., 2023) introduced Standard Operating Procedures (SOPs) into agent workflows, utilizing structured document outputs to prevent context drifting. AutoGen (Wu et al., 2023) provided a generalizable conversation-centric framework.

2.3 Dynamic Recruitment and Scaling Topologies

Recent efforts explore flexible agent topologies. AgentVerse (Chen et al., 2023) introduced stage-based expert recruitment, while Mixture-of-Agents (MoA) (Wang et al., 2024) highlighted accuracy gains from

layered aggregation. Magentic-One (Fourney et al., 2024) deployed a central Orchestrator pattern for web/file execution, and MacNet (Qian et al., 2024) studied DAG-based topological scaling.

2.4 Diagnostics, Scaling Laws, and Protocols

Empirical analyses have identified key limits of multi-agent expansion. The MAST taxonomy (Cemri et al., 2025) categorized 14 failure modes in multi-agent execution traces. Kim et al. (2025) derived quantitative scaling principles for agent systems, reporting that multi-agent performance is highly task- and architecture-dependent rather than universally beneficial. Standardized communication protocols, such as MCP, ACP, A2A, and ANP (Ehtesham et al., 2025), have emerged to unify interoperability.

3. Research Gap and Contributions

3.1 Detailed Research Gap

Existing multi-agent LLM frameworks operate on fixed topology assignments or stage-based recruitment that lacks pre-execution cost awareness. When a simple factual or metadata query is submitted to a static system, it executes the entire agent sub-graph. This can lead to unnecessary token consumption, added wall-clock latency, and higher operational API costs.

Existing adaptive frameworks evaluate dynamic recruitment after stage execution begins rather than evaluating task complexity before agent allocation. There is a critical need for a pre-execution task-complexity-aware routing mechanism that predicts minimal sub-agent topologies to balance efficiency and accuracy.

3.2 Novelty Comparison Matrix

Architectural Dimension	Static Multi-Agent (MetaGPT/ChatDev)	Stage-Dynamic (AgentVerse)	Orchestrated Loop (Magentic-One)	Proposed TCA-DARE Framework
Pre-Execution Complexity Scoring	None (Static)	None (Stage-based)	None (Continuous)	Explicit (C_score via Layer 1)
Topology Allocation	Fixed Sequential Graph	Dynamic Recruitment per Phase	Centralized Orchestrator Dispatch	Dynamic Sub-Graph Allocation (C_1, C_2, C_3)
Cost & Token Optimization	Not Optimized	Indirect	Acknowledged Limitation	Design Target via Minimal Sub-Graph Routing
Latency Optimization	Not Optimized	Not Optimized	High Re-planning Latency	Structural Pruning of Unnecessary Handoffs
Verification Guardrail	Role-based QA	Self-Evaluation Step	Outer Loop Monitor	Layer 3 Grounding Verification (0.95 Target)

Table 1. Architectural comparison between existing multi-agent frameworks and the proposed TCA-DARE design.

3.3 Core Contributions

- **Mathematical Complexity Scoring Engine:** Formulated a lightweight, pre-execution scoring mechanism analyzing token length, academic entity density, multi-hop operator cues, and synthesis intent.
- **Dynamic Sub-Graph Router (TCA-DARE):** Designed and implemented an adaptive routing algorithm supporting three distinct execution topologies (C_1 Single-Agent, C_2 Dual-Pipeline, C_3 Full Orchestration).
- **Prototype & Validation Protocol:** Implemented a local Python prototype (not yet experimentally executed) using Ollama (llama3) and tiktoken with built-in telemetry capture, accompanied by a structured multi-tier benchmark protocol for upcoming empirical validation.

4. Proposed Methodology

4.1 Feature Extraction and Mathematical Formalization

When an academic query Q is received, Layer 1 extracts four feature dimensions:

- **Token Length (F_len):** Normalized prompt length:

$$\text{F_len} = \min(\ |\text{T_Q}|\ /\ 100\ ,\ 1.0\ ) \times 100$$

- **Academic Entity Density (F_ent):** Density of scientific entities (model names, datasets, author citations):

$$\text{F_ent} = \min(\ \text{N_entities} \times 20.0\ ,\ 100.0\ )$$

- **Multi-Hop Operator Indicator (F_hop):** Binary/scaled indicator for multi-step relational logic:

$$\text{F_hop} = 100.0 \quad \text{if relational keywords present;} \quad 10.0 \quad \text{otherwise}$$

- **Synthesis Demand Indicator (F_syn):** Measure of multi-document review or survey intent:

$$\text{F_syn} = 100.0 \quad \text{if synthesis keywords present;} \quad 10.0 \quad \text{otherwise}$$

The continuous complexity score C_score ∈ [0, 100] is computed as:

$$\text{C_score} = w_1 \cdot \text{F_len} + w_2 \cdot \text{F_ent} + w_3 \cdot \text{F_hop} + w_4 \cdot \text{F_syn}$$

where $w_1 = w_2 = w_3 = w_4 = 0.25$.

4.2 Agent Selection Rules and Topology Thresholds

Thresholds $\tau_1 = 30.0$ and $\tau_2 = 70.0$ govern execution routing:

$$\text{Selected Topology} =$$
$$\text{Route } C_1 \quad \text{if C_score} < \tau_1 \qquad \text{(Single Agent: A_ext)}$$
$$\text{Route } C_2 \quad \text{if } \tau_1 \le \text{C_score} < \tau_2 \qquad \text{(Dual Pipeline: A_src} \rightarrow \text{A_syn} \rightarrow \text{A_chk)}$$
$$\text{Route } C_3 \quad \text{if C_score} \ge \tau_2 \qquad \text{(Full Orchestration: A_src, A_ext, A_aud, A_syn, A_chk)}$$

5. System Architecture and Algorithm

5.1 Layer Breakdown

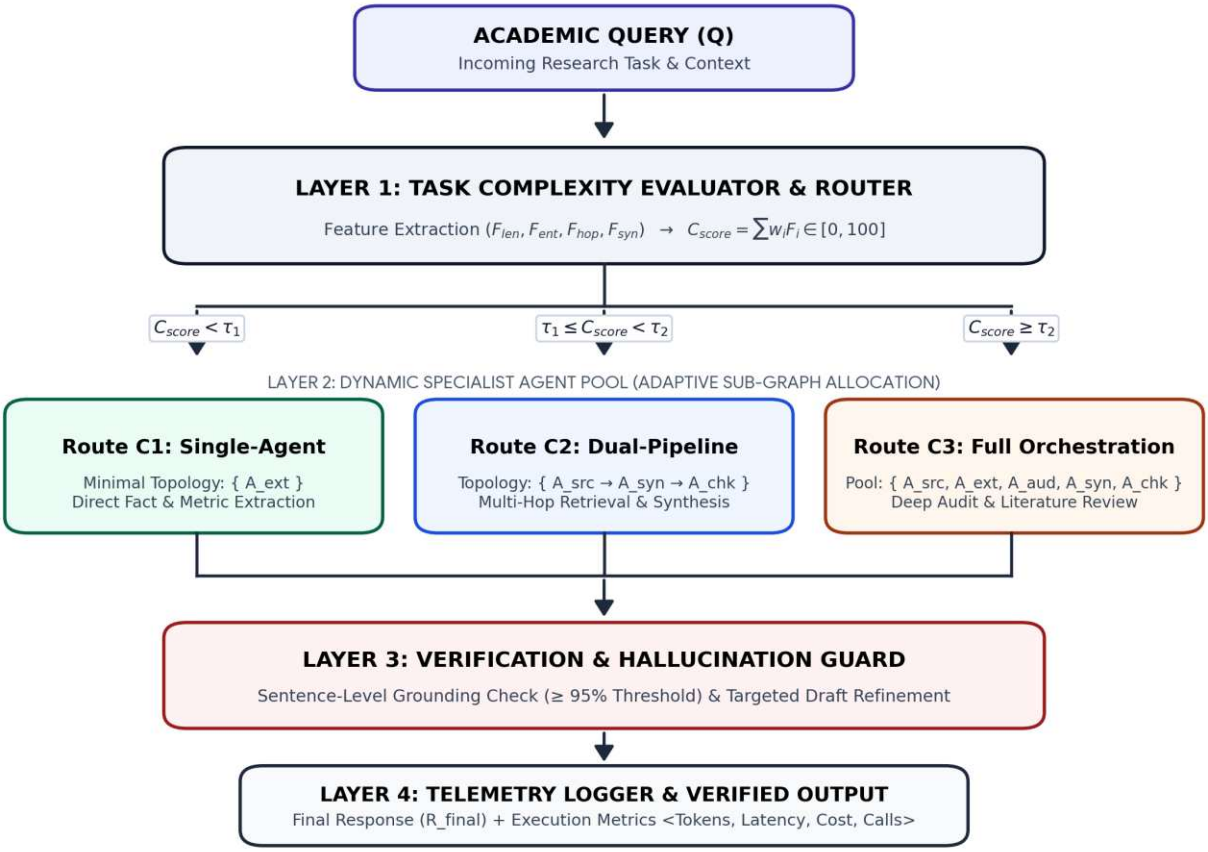


Figure 1. End-to-end layered architecture and dynamic sub-graph routing workflow of TCA-DARE. Layer 3 is specified in Algorithm 1 but is not yet implemented in the current prototype.

5.2 Detailed Breakdown of Architecture Layers

- **Layer 1: Task Complexity Evaluator & Router:** Intercepts incoming academic queries (Q) before heavy agent allocation occurs. It extracts key structural, semantic, and syntactic features to compute a continuous complexity score (C_score) based on token length (F_len), academic entities (F_ent), multi-hop operators (F_hop), and synthesis keywords (F_syn). Based on thresholds τ_1 and τ_2, it dispatches the query to Route C_1, C_2, or C_3.
- **Layer 2: Dynamic Specialist Agent Pool:** Modular, specialized agent roles instantiated only when requested by the router:
  - **Literature Searcher (A_src):** Intended to query academic APIs/databases (arXiv, Semantic Scholar) to retrieve primary texts and abstracts; retrieval integration is planned (see Section 6).
  - **Data Extractor (A_ext):** Parses specific metrics, baseline figures, and tabular data directly from paper sources.

    - **Methodology Auditor (A_aud):** Evaluates academic methodology rigor, dataset validity, and reported limitations.
    - **Fact Checker (A_chk):** Intended to compute sentence-level cross-verification between generated agent outputs and primary source texts; planned together with Layer 3 (see Section 6).
    - **Synthesis Writer (A_syn):** Aggregates findings from worker agents, organizing them into structured text blocks.
- **Layer 3: Verification & Hallucination Guard:** Specifies an automated self-correction loop for multi-agent routes (C_2 and C_3) that evaluates the groundedness ratio against primary source text and triggers targeted refinements when grounding falls below the 0.95 design threshold.
- **Layer 4: Output Generator & Telemetry Logger:** Compiles the final response and logs execution telemetry (total tokens, latency, cost, and API call count). In the current local prototype, cost is recorded as zero because inference runs locally.

5.3 Algorithm Specification (TCA-DARE)

```
Algorithm 1: Task-Complexity-Aware Dynamic Agent Routing and Execution (TCA-DARE)
Require: Academic Query Q, Thresholds tau_1, tau_2, Specialist Agent Pool A, Weight Vector w
Ensure: Final Response R_final, Telemetry T = <Tokens, Latency, Cost, Calls>
 1: procedure TCA_DARE_EXECUTE(Q, tau_1, tau_2, A, w)
 2:     Initialize T <- <Tokens:0, Latency:0.0, Cost:0.0, Calls:0>
 3:     t_start <- GetCurrentTime()
 4:
 5:     // Layer 1: Feature Extraction and Scoring
 6:     F_len <- ExtractTokenLength(Q)
 7:     F_ent <- CountAcademicEntities(Q)
 8:     F_hop <- DetectMultiHopOperators(Q)
 9:     F_syn <- DetectSynthesisKeywords(Q)
10:     C_score <- (w[0]*F_len) + (w[1]*F_ent) + (w[2]*F_hop) + (w[3]*F_syn)
11:
12:     // Layer 2 & 3 Execution Routing
13:     if C_score < tau_1 then
14:         Topology M <- {A_ext}
15:         R_raw, T_sub <- ExecuteAgent(A_ext, Q, Context=None)
16:         T <- Accumulate(T, T_sub)
17:         R_final <- R_raw
18:     else if tau_1 <= C_score < tau_2 then
19:         Topology M <- {A_src, A_syn, A_chk}
20:         Docs, T1 <- ExecuteAgent(A_src, Q)
21:         Draft, T2 <- ExecuteAgent(A_syn, Q, Context=Docs)
22:         V_score, T3 <- ExecuteAgent(A_chk, Draft=Draft, Context=Docs)
23:         T <- Accumulate(T, T1 + T2 + T3)
24:         if V_score.GroundednessRatio < 0.95 then
25:             R_final, T_ref <- RefineDraft(A_syn, Draft, V_score.Feedback)
26:             T <- Accumulate(T, T_ref)
27:         else
28:             R_final <- Draft
29:         end if
30:     else
31:         Topology M <- {A_src, A_ext, A_aud, A_syn, A_chk}
```

```
        Docs, T1 <- ExecuteAgent(A_src, Q)
        ExtData, T2 <- ExecuteAgent(A_ext, Q, Context=Docs)
        Audit, T3 <- ExecuteAgent(A_aud, Context=ExtData)
        Draft, T4 <- ExecuteAgent(A_syn, Query=Q, Context={ExtData, Audit})
        V_score, T5 <- ExecuteAgent(A_chk, Draft=Draft, Context=Docs)
        T <- Accumulate(T, T1 + T2 + T3 + T4 + T5)
        if V_score.GroundednessRatio < 0.95 then
            R_final, T_ref <- RefineDraft(A_syn, Draft, V_score.Feedback)
            T <- Accumulate(T, T_ref)
        else
            R_final <- Draft
        end if
    end if

    T.Latency <- GetCurrentTime() - t_start
    return R_final, T
end procedure
```

6. Prototype Implementation

A software prototype (tca_dare_runner.py) has been implemented in Python using local LLM execution (llama3 via the Ollama REST API) and tiktoken (cl100k_base) for token counting and telemetry capture. The implemented codebase realizes the Layer 1 feature-extraction and complexity-scoring router (TCADARERouter), the role-parameterized local execution engine (OllamaAgentExecutionEngine), and the conditional three-route orchestration pipeline (run_pipeline_experiment). The full source code is reproduced in Appendix A (Listing 1).

In the current prototype, each specialist agent is realized as a role-parameterized prompt to the same local llama3 model. External literature retrieval (A_src), structured data extraction (A_ext), and sentence-level verification with iterative refinement (A_chk, Layer 3) are not yet implemented; these agents therefore act as role-labelled language-model calls rather than tool-using specialists. Cost is recorded as zero because inference runs locally. If the Ollama endpoint is unreachable, the prototype raises an error instead of recording telemetry, so that no execution log can be produced without a genuine model call. The prototype also simplifies the context passing of Algorithm 1: in Route C_3, each agent receives the concatenated outputs of all preceding agents rather than the role-specific inputs shown in Algorithm 1.

Under the current thresholds, Route C_3 (C_score ≥ τ_2 = 70.0) is reached only when both the multi-hop and synthesis indicators are active and F_len + F_ent ≥ 80. Consequently, the third sample query in Listing 1, although intended as a full-orchestration example, is routed to C_2. Calibration of τ_2 and of the feature weights is deferred to the planned validation phase (Section 7).

7. Implementation Status and Planned Validation Protocol

7.1 Completed Implementation vs. Planned Empirical Validation

To maintain strict transparency between written code artifacts and upcoming empirical benchmarks, Table 2 delineates the current status across all framework modules. While the core routing logic, dynamic sub-graph dispatcher, and telemetry instrumentation are implemented in the prototype source code (tca_dare_runner.py), live execution testing on local queries, large-scale dataset runs, baseline benchmarking, and ablation experiments have not yet been executed and are scheduled for the next validation phase.

Framework Component / Phase	Current Status	Implementation / Validation Scope
Layer 1: Complexity Scoring Router	Code Written (Untested)	Implemented in TCADARERouter (F_len, F_ent, F_hop, F_syn, thresholds τ_1=30, τ_2=70); live execution verification pending.
Layer 2: Dynamic Sub-Graph Dispatch	Code Written (Untested)	Implemented in run_pipeline_experiment across C_1, C_2, and C_3 execution routes; live execution verification pending.
Layer 4: Token & Latency Telemetry	Code Written (Untested)	Implemented via tiktoken (cl100k_base) and wall-clock timers in OllamaAgentExecutionEngine; live execution logs pending.
Layer 3: Automated Refinement Loop	Designed (Planned)	Specified in Algorithm 1 (0.95 groundedness threshold); automated NLI/LLM critic integration planned.
Multi-Dataset Empirical Benchmark	Planned Validation	Full execution logs across SciQ, HotpotQA, GAIA, and baseline systems pending execution.

Table 2. Separation of completed prototype implementation from planned experimental validation.

7.2 Planned Dataset Suite & Task Categorization Protocol

For upcoming empirical evaluation, a 120-task multi-tier benchmark suite has been defined to test routing accuracy and resource efficiency across three complexity tiers:

Target Benchmark	Target Tier	Planned Sample	Categorization Rule / Criteria	Resource Link
SciQ Benchmark	Tier 1 (C_1)	40 Tasks	Direct single-entity lookup, single factual answer from context.	https://allenai.org/data/sciq
HotpotQA Academic	Tier 2 (C_2)	40 Tasks	Multi-hop reasoning requiring joining information across 2 abstracts.	https://hotpotqa.github.io/
GAIA Benchmark (L1/L2)	Tier 3 (C_3)	40 Tasks	Complex multi-modal tool use, file navigation, multi-step synthesis.	https://huggingface.co/datasets/gaia-benchmark/GAIA
Custom Academic Corpus	Tier 3 (C_3)	50 Papers (Planned)	Comprehensive systematic literature synthesis across 25+ recent papers.	Private arXiv Corpus (Planned)

Table 3. Planned academic benchmark suite and tier categorization criteria for future validation.

7.3 Planned Baseline Comparisons and Ablation Design

Once full-scale execution logs are collected on the benchmark suite above, TCA-DARE will be evaluated against three reference configurations:

- **Baseline 1 (Single-Agent ReAct):** Single LLM reasoning-acting loop with search tool integration to establish lower-bound token usage.
- **Baseline 2 (Static Multi-Agent):** Full 5-agent sequential pipeline executed unconditionally for every query to establish upper-bound token and latency cost.
- **Baseline 3 (Heuristic Length Router):** Rule-based router allocating agents using raw character length without semantic entity or multi-hop indicators.
- **Planned Ablation Variants:** Two controlled ablations are planned—Variant A (disabling Layer 1 routing via random route assignment) and Variant B (disabling Layer 3 verification)—to quantify the individual impact of pre-execution routing and post-generation grounding checks.

8. Limitations and Future Scope

- **Absence of Executed Benchmark Logs:** The current manuscript reports the formal architecture, mathematical scoring model, and prototype source code; empirical execution logs across the 120-task benchmark suite and statistical comparisons against static baselines remain to be executed.
- **Heuristic Feature & Threshold Sensitivity:** The current Layer 1 feature extractor relies on lexical keyword lists and fixed routing thresholds ($\tau_1 = 30.0$, $\tau_2 = 70.0$), which may require calibration across different LLM backbones and academic sub-domains. Keywords are matched as substrings (e.g., “reagent” matches “agent”), and with $\tau_2 = 70.0$ Route C_3 is reachable only by long, entity-dense queries that contain both multi-hop and synthesis cues.
- **Layer 3 Verification Integration:** While specified in Algorithm 1, automated sentence-level entailment scoring and iterative refinement loops need to be integrated and stress-tested in the Python prototype.
- **Agent Specialization:** Specialist agents are currently role-labelled prompts to a single local model; integrating retrieval APIs, structured extraction, and tool use is required before the C_2 and C_3 routes can be compared meaningfully against static baselines.

9. Conclusion

We presented TCA-DARE, a task-complexity-aware multi-agent architectural framework designed to address token inflation and latency bottlenecks in static LLM orchestration pipelines. By formalizing a pre-execution four-feature complexity scoring model (Layer 1) coupled with dynamic sub-graph allocation across three execution modes (C_1, C_2, and C_3), a verification guardrail specification (Layer 3), and telemetry logging (Layer 4), TCA-DARE provides a structured blueprint for adaptive agent routing. We

have implemented, but not yet experimentally executed, the core routing and local Ollama execution prototype in Python (tca_dare_runner.py) and established a multi-tier evaluation protocol. Future work will execute full-scale empirical benchmarks across SciQ, HotpotQA, and GAIA to generate verifiable execution logs, compare performance against static multi-agent baselines, and explore automated threshold tuning.

Declarations

Conflicts of Interest

The authors declare no conflicts of interest.

AI and Large Language Model Disclosure

AI-assisted writing tools were used to assist in drafting and editing portions of this manuscript. The authors take full responsibility for the accuracy, integrity, and originality of the content.

Funding

The authors received no specific funding for this work.

Ethics Approval

Not applicable. This study did not involve human participants, human data, or animals.

Data Availability

No datasets were generated or analysed during the current study.

Code Availability

The source code of the prototype (tca_dare_runner.py) is provided as Supplementary File 1 and is reproduced in Appendix A.

Author Contributions

M.A. (Moksha Arora) contributed to the study design, methodology, framework design, prototype implementation, and preparation of the original manuscript. B.M. (Bhawna Mittal) contributed to manuscript review and revision. M.T. (Mohit Tiwari) provided research supervision, methodological guidance, critical review, and manuscript revision, and serves as the corresponding author. All authors reviewed and approved the final manuscript.

10. References

1. Yao, S., et al. (2022). ReAct: Synergizing Reasoning and Acting in Language Models. ICLR 2023.
2. Schick, T., et al. (2023). Toolformer: Language Models Can Teach Themselves to Use Tools. NeurIPS 2023.
3. Shinn, N., et al. (2023). Reflexion: Language Agents with Verbal Reinforcement Learning. NeurIPS 2023.

4. Hong, S., et al. (2023). MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework. ICLR 2024.
5. Wu, Q., et al. (2023). AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. arXiv:2308.08155.
6. Chen, W., et al. (2023). AgentVerse: Facilitating Multi-Agent Collaboration and Exploring Emergent Behaviors. ICLR 2024.
7. Wang, X., et al. (2024). Mixture-of-Agents Enhances Large Language Model Capabilities. arXiv:2406.04692.
8. Fourney, A., et al. (2024). Magentic-One: A Generalist Multi-Agent System for Solving Complex Tasks. arXiv:2411.04468.
9. Cemri, M., et al. (2025). Why Do Multi-Agent LLM Systems Fail? NeurIPS 2025. arXiv:2503.13657.
10. Kim, Y., et al. (2025). Towards a Science of Scaling Agent Systems. arXiv:2512.08296.
11. Ehtesham, A., et al. (2025). A Survey of Agent Interoperability Protocols: MCP, ACP, A2A and ANP. arXiv:2505.02279.
12. Li, G., Hammoud, H. A. A. K., Itani, H., Khizbullin, D., & Ghanem, B. (2023). CAMEL: Communicative Agents for “Mind” Exploration of Large Language Model Society. NeurIPS 2023. arXiv:2303.17760.
13. Qian, C., et al. (2023). ChatDev: Communicative Agents for Software Development. ACL 2024. arXiv:2307.07924.
14. Yang, J., et al. (2024). SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. NeurIPS 2024. arXiv:2405.15793.
15. Qian, C., et al. (2024). Scaling Large-Language-Model-based Multi-Agent Collaboration. ICLR 2025. arXiv:2406.07155.

Appendix A. Prototype Source Code

Listing 1: TCA-DARE Python Prototype (tca_dare_runner.py)

```
import time
import json
import urllib.request
import tiktoken

# Local Ollama Configuration
OLLAMA_URL = "http://localhost:11434/api/generate"
OLLAMA_MODEL = "llama3"

tokenizer = tiktoken.get_encoding("cl100k_base")


class TCADARERouter:
    """Layer 1: Task Complexity Evaluator & Router."""

    def __init__(self, tau_1: float = 30.0, tau_2: float = 70.0):
```

```
        self.tau_1 = tau_1
        self.tau_2 = tau_2
        self.weights = [0.25, 0.25, 0.25, 0.25]

    def extract_features(self, query: str) -> dict:
        tokens = tokenizer.encode(query)
        f_len = min(len(tokens) / 100.0, 1.0) * 100.0

        academic_keywords = [
            "gpt", "llama", "arxiv", "agent", "metagpt",
            "react", "benchmark", "dataset", "transformer"
        ]
        f_ent = min(sum(1 for kw in academic_keywords if kw in query.lower()) * 20.0, 100.0)

        hop_indicators = [
            "compare", "difference", "between", "relate",
            "why", "how do", "across"
        ]
        f_hop = 100.0 if any(ind in query.lower() for ind in hop_indicators) else 10.0

        syn_indicators = [
            "synthesize", "survey", "review", "comprehensive",
            "all papers", "summarize"
        ]
        f_syn = 100.0 if any(ind in query.lower() for ind in syn_indicators) else 10.0

        return {"f_len": f_len, "f_ent": f_ent, "f_hop": f_hop, "f_syn": f_syn}

    def compute_complexity(self, features: dict) -> float:
        return sum(self.weights[i] * list(features.values())[i] for i in range(4))

    def route_query(self, query: str) -> tuple:
        features = self.extract_features(query)
        c_score = self.compute_complexity(features)
        if c_score < self.tau_1:
            return "Route_C1_Single_Agent", c_score
        elif self.tau_1 <= c_score < self.tau_2:
            return "Route_C2_Dual_Pipeline", c_score
        else:
            return "Route_C3_Full_Orchestration", c_score


class OllamaAgentExecutionEngine:
    """Layer 2 & 4: Specialist Agent Execution & Telemetry Capture."""

    @staticmethod
    def call_agent(agent_role: str, query: str, context: str = "") -> tuple:
        t_start = time.time()
        prompt_text = (
            f"You are agent role: {agent_role}.\n"
            f"Query: {query}\n"
            f"Context: {context}\n"
            f"Provide a concise academic response."
        )
        prompt_tokens = len(tokenizer.encode(prompt_text))
```

```
        payload = {"model": OLLAMA_MODEL, "prompt": prompt_text, "stream": False}
        req = urllib.request.Request(
            OLLAMA_URL,
            data=json.dumps(payload).encode("utf-8"),
            headers={"Content-Type": "application/json"}
        )
        # No fallback: if the local model cannot be reached, stop with an
        # error so that no telemetry is recorded without a real model call.
        try:
            with urllib.request.urlopen(req, timeout=600) as response:
                result = json.loads(response.read().decode("utf-8"))
        except Exception as exc:
            raise RuntimeError(
                f"Ollama call failed for {agent_role} at {OLLAMA_URL}: {exc}"
            ) from exc
        if "response" not in result:
            raise RuntimeError(f"Unexpected Ollama response for {agent_role}: {result}")
        response_text = result["response"]

        completion_tokens = len(tokenizer.encode(response_text))
        total_tokens = prompt_tokens + completion_tokens
        elapsed_latency = time.time() - t_start

        telemetry = {
            "tokens": total_tokens,
            "cost": 0.0,  # local inference: no API cost is incurred or estimated
            "latency": round(elapsed_latency, 2),
            "calls": 1
        }
        return response_text, telemetry


def run_pipeline_experiment(query: str) -> dict:
    """End-to-end TCA-DARE execution pipeline."""
    router = TCADARERouter()
    route, score = router.route_query(query)
    total_tokens, total_cost, total_latency, total_calls = 0, 0.0, 0.0, 0

    if route == "Route_C1_Single_Agent":
        _, tel = OllamaAgentExecutionEngine.call_agent("A_ext", query)
        total_tokens += tel["tokens"]
        total_cost += tel["cost"]
        total_latency += tel["latency"]
        total_calls += tel["calls"]
    elif route == "Route_C2_Dual_Pipeline":
        docs, t1 = OllamaAgentExecutionEngine.call_agent("A_src", query)
        draft, t2 = OllamaAgentExecutionEngine.call_agent("A_syn", query, context=docs)
        _, t3 = OllamaAgentExecutionEngine.call_agent("A_chk", draft, context=docs)
        for t in [t1, t2, t3]:
            total_tokens += t["tokens"]
            total_cost += t["cost"]
            total_latency += t["latency"]
            total_calls += t["calls"]
    else:
```

```
        agents = ["A_src", "A_ext", "A_aud", "A_syn", "A_chk"]
        context_data = ""
        for agent in agents:
            res, tel = OllamaAgentExecutionEngine.call_agent(agent, query, context=context_data)
            context_data += " " + res
            total_tokens += tel["tokens"]
            total_cost += tel["cost"]
            total_latency += tel["latency"]
            total_calls += tel["calls"]

    return {
        "Route": route,
        "Complexity_Score": round(score, 2),
        "Total_Tokens": total_tokens,
        "Total_Cost": total_cost,
        "Total_Latency": round(total_latency, 2),
        "Total_Calls": total_calls
    }


if __name__ == "__main__":
    sample_queries = [
        "What is the parameter size of Llama 3?",
        "Compare the reasoning mechanisms between ReAct and Reflexion across agent benchmarks.",
        "Synthesize a comprehensive literature review across all papers on multi-agent LLM orchestration
protocols including MCP, ACP, A2A, and ANP."
    ]
    for q in sample_queries:
        print(run_pipeline_experiment(q))
```

Supplementary Files

This is a list of supplementary files associated with this preprint. Click to download.

- tcadarerunner.py