LOOM: A Team-of-Agents Architecture for High-Performance Graph Analytics Workflows with Arkouda and Arachne

Asha Saxena
Department of Data Science
New Jersey Institute of Technology
Newark, NJ, USA
as4689@njit.edu

David A. Bader
Department of Data Science
New Jersey Institute of Technology
Newark, NJ, USA
bader@njit.edu

***Abstract*—Teams of large language model (LLM) agents now plan, code, and verify their own work, yet nearly all agentic frameworks assume a dataset fits in the memory of one machine, excluding the billion-edge graphs that motivate high-performance computing and where a single poor analytical step can waste hours of supercomputer time. We present LOOM, a team-of-agents architecture for large-scale graph analytics on Arkouda and its graph extension Arachne. LOOM organizes role-specialized agents around a shared, typed artifact, and contributes five mechanisms that distinguish it from generic agentic pipelines: a Graph Workflow Intermediate Representation (GW-IR) that is statically checked so ill-typed workflows are rejected before any cluster cycles are spent; telemetry-grounded reflection that feeds per-locale load imbalance, communication volume, and time to solution from the Chapel-based server back to the agents; invariant-based verification against graph-theoretic identities that hold without ground truth; a Security and Access Control Agent that enforces resource quotas and audits data access before any cluster cycles are spent; and a Learning/Adaptation Agent that replaces the hand-crafted cost model with a data-driven predictor trained on accumulated provenance telemetry. LOOM is implemented as an open-source framework that runs end to end, available at https://github.com/Bader-Research/LOOM.**

***Index Terms*—multi-agent systems, large language models, large-scale graph analytics, high-performance computing, Arkouda, Arachne, workflow orchestration, security, adaptive cost models**

I. Introduction

Autonomous agents built on large language models (LLMs) now plan multi-step tasks, call external tools, write and debug code, and critique their own outputs [1]–[3]. Multi-agent variants assign specialized roles to several cooperating models and have produced strong results on software engineering and data-science benchmarks [4]–[8]. These systems share a quiet but consequential assumption: the data they reason over fits in the memory of one machine, and the tools they call are ordinary single-node libraries. That assumption fails precisely where high-performance computing is indispensable. Connectome reconstructions such as the H01 dataset already exceed one hundred million edges [9], cybersecurity telemetry and web-scale social graphs reach billions of edges, and exploratory analysis at this scale demands distributed-memory parallelism rather than a laptop.

A second problem compounds the first. When an agent operates a supercomputer, mistakes are expensive. A hallucinated pipeline that symmetrizes a graph twice, a community-detection variant chosen without regard to the graph's degree distribution, or a kernel launched with a degree of parallelism that starves half the locales can each cost hours of wallclock time and substantial energy before any error is noticed. Single-agent loops that reflect only on textual outputs [3] have no principled way to catch these failures, because the signal that matters lives in the runtime: load imbalance across compute nodes, communication volume, and memory high-water marks.

We present LOOM, a team-of-agents architecture that couples role-specialized LLM agents with Arkouda [10], an open-source framework that exposes a Python interface similar to NumPy and pandas over distributed arrays, and its graph analytics extension Arachne [11]. Arkouda is built on the Chapel parallel programming language [12] and runs a back-end server across the locales of an HPC cluster. Arachne adds scalable graph kernels including BFS, connected components, triangle counting, k-truss, community detection [13], subgraph isomorphism [9], [14], and property-graph queries [15]. LOOM is publicly available at https://github.com/Bader-Research/LOOM.

LOOM departs from prior agentic frameworks in five ways:

1) **A typed, verifiable workflow plan** via a GW-IR (Section IV).
2) **Telemetry-grounded performance reflection** (Section V).
3) **Invariant-based verification without ground truth** (Section VI).
4) **A Security and Access Control Agent** that enforces resource quotas and data-access policies before any cluster cycles are spent (Section XI).
5) **A Learning/Adaptation Agent** that replaces the hand-crafted cost model with a data-driven predictor (Section XII).

Contributions. We (i) present LOOM's role-specialized architecture; (ii) define the GW-IR and static validation; (iii) introduce telemetry-grounded reflection; (iv) describe

invariant-based verification; (v) introduce a Security and Access Control Agent for policy governance; (vi) introduce a Learning/Adaptation Agent for data-driven kernel selection; and (vii) release an open-source implementation. LOOM builds on our prior study [16] finding that LLMs excel at composing established kernels but rarely invent new ones.

II. Background and Related Work

A. Arkouda and Arachne

Arkouda answers a practical need in interactive data science: the ability to manipulate terabyte-scale arrays from a familiar Python session without writing parallel code [10]. A data scientist issues NumPy-like operations from a Jupyter notebook; the Arkouda client serializes these into requests to a Chapel server executing in parallel across cluster locales and returning compact results or symbolic handles. Arachne extends this model to graphs [11], maintaining distributed graph representations and exposing kernels for traversal, connectivity, dense-subgraph and motif discovery, and community detection—several of which scale to graphs with billions of edges and have set performance records on connectome-scale property graphs [9], [13], [15]. Arachne is open source.[1]

B. LLM Agents and Tool Use

The ReAct pattern interleaves reasoning traces with tool calls [1]; Toolformer shows that models can learn when to invoke tools [2]; Reflexion adds verbal self-critique across attempts [3]. Multi-agent frameworks such as AutoGen [4], MetaGPT [5], and CAMEL [6] coordinate several role-playing agents, and Data Interpreter automates end-to-end notebook workflows [7]. AgentBench surveys the resulting capabilities and limitations [8]. What these systems lack, for our setting, is any notion of distributed execution cost, any typed contract between the agent's plan and a parallel runtime, and any feedback channel richer than text. LOOM supplies all three.

C. Multi-Agent Coordination

LOOM's coordination substrate revisits two classical ideas. The blackboard architecture lets independent knowledge sources contribute opportunistically to a shared solution structure under the direction of a control component [17]; we adopt the GW-IR as that shared structure. The contract-net protocol assigns subtasks to the most suitable specialist through announcement and bidding [18]; the orchestrator uses an analogous mechanism to route GW-IR subgoals. Recasting these protocols around a typed graph-analytics IR, with an HPC runtime in the loop, is what makes the combination new.

III. The LOOM Architecture

LOOM separates *what* to compute from *how* to compute it at scale, and assigns each concern to a specialist. A user states an analytical goal in natural language. The team collaboratively constructs, validates, executes, and refines a GW-IR that realizes the goal, with all coordination flowing through a shared blackboard. Table I summarizes all eight agents, including the two new ones introduced in this paper.

[1]https://github.com/Bears-R-Us/arkouda-njit

```
load(src="edges.pq")        : EdgeSet
build_graph(.)              : DiGraph
symmetrize(.)               : Graph[sym]
largest_cc(.)               : Graph[sym, 1cc]
community(., variant=?)     : Partition
per_block_triangles(.,.)    : ScalarField[block]
report(.)                   : Summary
```

Fig. 1. A GW-IR fragment for "find communities in the network and rank them by internal triangle density." Bracketed tags are refinement attributes. The variant for `community` is left open for the Algorithm Strategist to fill using the cost model.

A. Coordination on the Blackboard

All agents share one blackboard that holds the current GW-IR, the most recent telemetry, and a growing provenance record. The orchestrator acts as the control component in the blackboard sense [17]: it inspects the partially constructed GW-IR, identifies the next open subgoal, and announces it. Specialists "bid" by proposing GW-IR fragments annotated with confidence and estimated cost, and the orchestrator commits the strongest proposal [18]. Because the shared artifact is typed rather than textual, conflicting proposals are reconciled by type checking and cost comparison rather than by another round of free-form debate. The control loop is given in Algorithm 1.

IV. The Graph Workflow Intermediate Representation

The GW-IR is the heart of LOOM. It is a directed acyclic graph whose nodes are typed Arachne operations and whose edges carry typed graph artifacts. The type system is deliberately small but expressive enough to catch the errors that matter in graph analytics. Core artifact types include `Graph`, `DiGraph`, `PropertyGraph`, `VertexSet`, `EdgeSet`, `Partition` (a labeling of vertices), and `ScalarField` (a per-vertex or per-edge value). Each type carries refinement attributes, for example whether a `Graph` is symmetric, weighted, or known to be a single connected component.

Every operation node declares preconditions and postconditions over these types. A plan that pipes a `DiGraph` into a kernel that assumes symmetry is rejected by static checking; a plan that feeds an edge-valued `ScalarField` where a vertex-valued one is required never reaches the cluster. Figure 1 shows a small GW-IR.

Static validation traverses the DAG in topological order, propagating types and refinements and checking each node's preconditions against the inferred types of its inputs (Algorithm 2). The check is inexpensive, runs entirely on the client, and yields a precise, machine-readable diagnostic that the orchestrator routes back to the agent that introduced the offending node. The GW-IR doubles as a provenance and reproducibility record.

V. Telemetry-Grounded Performance Reflection

Generic agentic reflection critiques text [3]. In distributed graph analytics, the decisive feedback is numeric and comes

TABLE I
ROLES IN THE LOOM AGENT TEAM. EACH AGENT READS AND WRITES THE SHARED GW-IR ON THE BLACKBOARD. NEW AGENTS ARE SHADED.

Agent	Responsibility	Primary inputs → outputs
Orchestrator	Decomposes the user goal into GW-IR subgoals, routes them to specialists via a contract-net announcement, and decides when the workflow is complete.	user goal, blackboard state → subgoal assignments, termination decision
Data Engineer	Resolves data sources, infers schema, builds the distributed graph (symmetrization, deduplication, largest connected component), and issues server requests.	raw sources, GW-IR ingest nodes → graph handles, schema, server calls
Algorithm Strategist	Selects kernels and variants for each analytical step using a cost model over graph statistics.	GW-IR analysis nodes, graph statistics → kernel and variant choices
Performance Engineer	Reads runtime telemetry and proposes layout, parallelism, and aggregation changes; re-plans hot nodes.	telemetry, cost model → tuned execution parameters
Verifier	Statically validates the GW-IR and checks results against graph-theoretic invariants and spot-check subgraphs.	GW-IR, results → pass/fail with diagnostics
Interpreter	Translates verified results into domain language, figures, and a provenance summary for the user.	verified results, provenance → natural-language report
Security & Access Control	Enforces resource quotas and data ACLs; appends a tamper-evident provenance entry for every committed operation before any cluster cycles are spent.	GW-IR fragment, ACL, quota config → clearance token or structured rejection
Learning/Adaptation	Trains a gradient-boosted cost model on accumulated provenance telemetry; replaces the hand-crafted model incrementally and falls back to it when data are insufficient.	provenance telemetry → trained predictor, updated kernel recommendations

Algorithm 1 LOOM control loop (SACA addition in **bold**)

$G \leftarrow$ empty GW-IR; $prov \leftarrow \emptyset$
seed G from the user goal (orchestrator)
while G has an open subgoal s and budget remains **do**
  announce s; collect typed fragment bids from specialists
  commit best fragment f to G by type and cost
  if Verifier rejects G statically **then**
    return diagnostics to responsible agent; **continue**
  end if
  if SACA rejects f (quota / ACL violation) then
    return policy diagnostic; continue
  end if
  if f is executable and its inputs are ready **then**
    execute on Arkouda/Arachne; record results, telemetry
    $prov \leftarrow prov \cup \{f, params, telemetry, \textbf{SACA-token}\}$
    if telemetry indicates an inefficiency **then**
      Performance Engineer proposes revised f; rerun
    end if
    if Verifier's invariants fail on results **then**
      route to Algorithm Strategist for an alternative
    end if
  end if
end while
Interpreter renders verified results and $prov$ for user

Algorithm 2 Static GW-IR validation

Require: GW-IR G with operation library L
for node n in topological order of G **do**
  $\tau_{\text{in}} \leftarrow$ inferred types/refinements of n's inputs
  if τ_{in} violates $pre(n)$ in L **then**
    return REJECT(n, expected $pre(n)$, got τ_{in})
  end if
  annotate n's outputs with $post(n)$
end for
return ACCEPT

from the runtime. After each kernel, the Arkouda server returns a structured telemetry record placed on the blackboard. The record includes time to solution, peak memory per locale, the coefficient of variation of work across locales as a load-imbalance measure, the volume of inter-locale communication, and, where available, aggregation buffer statistics. The performance-engineer agent reads this record and proposes concrete, typed changes to the GW-IR rather than prose.

A. A Cost Model for Kernel Selection

Before committing a kernel, the Algorithm Strategist consults a lightweight analytic cost model parameterized by graph statistics that Arachne reports cheaply: the number of vertices n, the number of edges m, the maximum and average degree, and an estimate of the diameter. For a traversal step the model contrasts the top-down and bottom-up directions of BFS, preferring bottom-up steps when the frontier is large relative to the remaining unvisited set. For community detection the model weighs label propagation, which is cheap and communication-light but lower quality, against the higher-quality but heavier variants studied in our Arachne work [13], choosing based on m, the target quality, and the user's stated time budget. The point is not that the cost model is exact; it is that algorithm selection is grounded in graph structure and runtime cost rather than in an LLM's generic prior, which our prior study shows can be unreliable for performance-critical choices [16].

B. Closing the Loop

When telemetry reveals an inefficiency, the performance engineer acts. High load imbalance suggests a different vertex-to-locale mapping or a degree-aware partition. Heavy communication relative to computation suggests batching through aggregation or switching to a kernel variant with a lower communication term. A memory high-water mark near capacity suggests streaming a subgraph or coarsening before the expensive step. Each remedy is expressed as an edit to the GW-IR, re-validated statically, and re-executed, and the before-and-after telemetry is retained in provenance.

VI. Correctness via Invariant-Based Verification

At billion-edge scale there is rarely a ground-truth answer to compare against, so the verifier checks identities that must hold for any correct result. Examples that LOOM uses include: the sum of vertex degrees equals twice the edge count; a `Partition` returned by community detection or connected components covers the vertex set exactly once, with no vertex unassigned and none assigned twice; the number of connected components does not increase when edges are added; triangle counts produced by two independent estimators agree within tolerance; and a k-truss is a subgraph of the input with every edge in at least $k-2$ triangles. Beyond global identities, the verifier draws small induced subgraphs, recomputes a quantity locally with an independent method, and compares—a property-based spot check that is cheap relative to the full computation. When an invariant fails, the verifier reports which one and on which artifact, and the orchestrator routes the failure to the Algorithm Strategist for an alternative kernel or to the Data Engineer for a data-quality fix.

VII. Open-Source Implementation

LOOM is implemented as an open-source Python framework and is publicly available. The GW-IR, its type lattice of graph-property artifacts, and the static validator of Algorithm 2 form the core. The eight agents and the contract-net control loop of Algorithm 1 are separate modules that communicate only through the typed blackboard. Importantly, the cost model, telemetry-grounded reflection, invariant checks, SACA policy engine, and LAA model training are ordinary deterministic code rather than model prompts, so the correctness and performance machinery is reproducible and unit-tested; the language model is used only for natural-language goal decomposition and for rendering the final report.

Two execution backends sit behind a single interface. A simulation backend builds a graph and computes real results—including connected components, community labels, and triangle counts—while synthesizing per-locale telemetry as a function of the chosen kernel and data layout, so that the full pipeline including the SACA checks and the LAA fall-back can be exercised without a cluster. A second backend adapts the production Arkouda and Arachne stack, issuing requests to a running Chapel server. The framework ships with a test suite that exercises the static validator, the invariant checks, the SACA policy engine, and the full pipeline including the reflection and adaptation loops.

VIII. A Worked Example

Consider a neuroscientist who asks, in plain language, "In the H01 cortical connectome, find the densely interconnected neuron communities and tell me which ones are richest in recurring three-neuron motifs." The H01 dataset has on the order of one hundred million edges and does not fit comfortably on a workstation [9].

The orchestrator decomposes the goal into ingest, community detection, motif density, and reporting subgoals and seeds the GW-IR. **SACA verifies the user holds a valid data-access token for the H01 source and has sufficient node-hour quota before any data are loaded.** The data engineer resolves the connectome source, builds the graph as `Graph[sym, 1cc]`, and the verifier confirms the degree-sum identity. The Algorithm Strategist consults the cost model (LAA-trained after prior runs, or hand-crafted on first deployment) and selects a community-detection variant [13] balancing quality and the interactive time budget. Community detection runs; telemetry shows acceptable balance. For per-community motif counts, the first execution reports high inter-locale communication; the performance engineer restricts motif search to within-community edges and batches candidate validation, re-validates the GW-IR edit, and re-runs. The verifier confirms that the community labeling partitions the vertex set and that motif counts on a sampled community match an independent recomputation. The interpreter returns a ranked list of communities with their motif densities, a natural-language explanation, and the GW-IR plus telemetry as a reproducible record.

Running the reference implementation on a planted-community graph of 1,500 vertices: the strategist selected the Louvain variant; the performance engineer switched to the within-community batched strategy, lowering inter-locale communication and load imbalance; and the verifier confirmed all three invariants—the degree sum equaled twice the edge count, the labeling covered every vertex, and the internal triangle total (13,225) did not exceed an independently recomputed global count (13,270).

IX. Evaluation Methodology

The reference implementation provides the substrate for empirical study. The design targets six questions: does the GW-IR reduce wasted cluster time; does telemetry-grounded reflection improve performance; does invariant checking catch real errors; is the end-to-end system usable by non-experts; does SACA correctly enforce policy without false positives; and does the LAA improve kernel-selection accuracy over the hand-crafted baseline.

Datasets. We use public graphs spanning regimes (Table II): synthetic Kronecker graphs from the Graph500 generator for controlled scaling [19], the GAP Benchmark Suite graphs for traversal and centrality workloads [20], web and social graphs that reach billions of edges, and the H01 connectome as a property-graph case study [9].

TABLE II
REPRESENTATIVE PUBLIC DATASETS FOR EVALUATION.

Dataset	Scale (edges)	Role
Graph500 Kronecker	2^{26} to 2^{36}	controlled scaling [19]
GAP suite	10^8 to 10^9	traversal, centrality [20]
Web/social graphs	up to 10^9+	community detection
H01 connectome	$\sim 1.5 \times 10^8$	property-graph motifs [9]

Baselines. We compare LOOM against (i) a single tool-using agent over the same Arachne kernels, in the spirit of ReAct [1]; (ii) a generic multi-agent framework without the GW-IR, telemetry loop, or verifier [4]; and (iii) an expert human using Arachne directly, as a quality and time reference.

Metrics. We measure the static plan-rejection rate and the cluster time it saves; end-to-end time to solution and total cost; the invariant-failure detection rate on a suite of deliberately seeded errors; the load-imbalance and communication improvements attributable to the performance engineer; task success rate and time for non-expert participants; SACA policy-rejection accuracy on a suite of synthetic over-quota and unauthorized-access requests; and LAA kernel-selection accuracy versus the hand-crafted baseline on the GAP suite, with mean absolute percentage error of time-to-solution and load-imbalance predictions as secondary metrics.

X. DISCUSSION, LIMITATIONS, AND FUTURE WORK

LOOM inherits the failure modes of its underlying models. Agents can still propose poor plans; the contribution is to make those plans cheap to reject through static typing and cheap to correct through telemetry and invariants, rather than to eliminate error. The cost model is approximate and must be calibrated per system—addressed by the LAA, which improves selection as telemetry accumulates. The invariant suite covers many but not all kernels, and constructing strong invariants for every analytic is ongoing work. SACA's effectiveness depends on the quality of the administrator-supplied ACL and quota configuration; an incorrectly configured policy may be too permissive or too restrictive. The LAA requires a meaningful provenance dataset before outperforming the hand-crafted baseline, and may generalize poorly across hardware tiers without retraining.

Promising directions include extending the GW-IR to streaming and dynamic graphs, supporting human-in-the-loop checkpoints where a scientist approves expensive steps, and generalizing the approach beyond Arachne to other distributed analytics backends that expose typed kernels.

XI. SECURITY AND ACCESS CONTROL AGENT

A. Motivation

Shared HPC clusters operate under institutional allocation policies: each project is granted a budget of node-hours, a memory ceiling per locale, and access to a defined set of datasets. Section X acknowledges that "resource quotas and policy enforcement remain necessary" but provides no enforcement mechanism. The Security and Access Control Agent (SACA) fills this gap as the seventh member of the LOOM team.

B. Agent Design

SACA operates in a blocking role: no GW-IR fragment may be committed to the execution queue without a SACA clearance token. Its responsibilities span three phases:

- **Pre-execution policy check:** enforces per-user and per-project node-hour and memory quotas by inspecting cost estimates attached to each GW-IR fragment before execution.
- **Data access auditing:** audits each GW-IR node against a dataset access-control list (ACL), rejecting operations that reference data sources outside the requesting user's authorization scope.
- **Provenance logging:** appends a signed, timestamped entry to a tamper-evident provenance ledger for every committed operation, recording the proposing agent, the clearance decision, and the runtime outcome.

Because the GW-IR makes every action explicit and typed before execution, SACA can perform all three checks statically—before any cluster cycles are spent—preserving LOOM's core principle that expensive failures are caught early. Algorithm 1 reflects the updated control loop: the SACA clearance check (lines 9–11) is inserted between static GW-IR validation and execution.

C. Threat Model and Scope

SACA addresses the *insider-automation* threat model: a legitimate user whose agent team, through hallucination, runaway loops, or misconfigured goals, issues operations that exceed authorized scope. It does not address external adversaries or malicious model weights; those require infrastructure-level controls outside LOOM's scope. SACA's ACL and quota configuration are provided by the cluster administrator and loaded at LOOM startup, keeping its policy external to and independent of the LLM agents it governs. This design ensures that the security boundary is not subject to prompt injection or agent miscommunication.

XII. LEARNING/ADAPTATION AGENT

A. Motivation

The Algorithm Strategist's hand-crafted cost model "is not exact and must be calibrated per system" (Section V), and Section X explicitly identifies learning the cost model from accumulated telemetry as a promising future direction. A fixed model cannot capture hardware-specific performance characteristics across diverse cluster configurations, nor improve from experience. The Learning/Adaptation Agent (LAA) addresses this directly as the eighth LOOM agent.

B. Data Collection and Model Architecture

Every LOOM run retains its GW-IR and per-kernel telemetry in provenance. The LAA treats this provenance as a growing training dataset. For each executed kernel it extracts a feature vector: graph statistics (n, m, max and average degree, diameter

estimate), kernel variant selected, locale count, and hardware tier. The prediction targets are the three telemetry quantities that the performance engineer uses for reflection: time to solution, per-locale load imbalance (coefficient of variation of work), and inter-locale communication volume.

The LAA trains a gradient-boosted regression model (one per prediction target) on accumulated provenance records. Gradient boosting is chosen for three reasons aligned with LOOM's design philosophy: it is sample-efficient and performs well with tens to hundreds of training examples, which is realistic for early deployment; its predictions are explainable via feature importances, allowing the Algorithm Strategist to communicate why a kernel was selected; and it is fully deterministic, keeping the cost model reproducible and unit-testable—consistent with LOOM's principle that correctness and performance machinery should not depend on stochastic model outputs. The LAA exposes the same interface as the original hand-crafted cost model; the Algorithm Strategist requires no modification.

C. Training Protocol and Evaluation

On first deployment, the LAA falls back to the original hand-crafted cost model, ensuring full functionality without prior runs. After each run, the LAA triggers an incremental model update if the provenance store has grown by more than a configurable threshold (default: ten new records). A cross-validation gate rejects updates whose leave-one-out error on held-out provenance records exceeds that of the current model, preventing degradation from outlier runs caused by unusual cluster load conditions.

The LAA is evaluated against the hand-crafted baseline using kernel-selection accuracy on the GAP suite [20] as the primary metric, with mean absolute percentage error of time-to-solution and load-imbalance predictions on held-out provenance records as secondary metrics. End-to-end time-to-solution improvement attributable to LAA-guided selection is measured as a tertiary metric.

XIII. Conclusion

We presented LOOM, a team-of-agents architecture that brings supercomputer-scale graph analytics within reach of natural-language interaction on Arkouda and Arachne. The original three mechanisms—typed GW-IR with static checking, telemetry-grounded reflection, and invariant-based verification—are extended with a Security and Access Control Agent that enforces resource quotas and data-access policies statically before any cluster cycles are spent, and a Learning/Adaptation Agent that replaces the hand-crafted cost model with a gradient-boosted predictor trained on accumulated provenance telemetry. Together, the eight-agent LOOM architecture makes large-scale graph analysis accessible, correct, cost-aware, policy-governed, and adaptive. We see LOOM as a step toward agentic systems that respect the realities of high-performance computing rather than assuming them away.

Acknowledgment

This research was funded in part by NSF grant numbers CCF-2109988, OAC-2402560, and CCF-2453324.

References

[1] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing reasoning and acting in language models," in *ICLR*, 2023.

[2] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom, "Toolformer: Language models can teach themselves to use tools," in *NeurIPS*, 2023.

[3] N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning," in *NeurIPS*, 2023.

[4] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. H. Awadallah, R. W. White, D. Burger, and C. Wang, "AutoGen: Enabling next-gen LLM applications via multi-agent conversation," *arXiv preprint arXiv:2308.08155*, 2023.

[5] S. Hong *et al.*, "MetaGPT: Meta programming for a multi-agent collaborative framework," in *ICLR*, 2024.

[6] G. Li, H. A. A. K. Hammoud, H. Itani, D. Khizbullin, and B. Ghanem, "CAMEL: Communicative agents for 'mind' exploration of large language model society," in *NeurIPS*, 2023.

[7] S. Hong *et al.*, "Data interpreter: An LLM agent for data science," *arXiv preprint arXiv:2402.18679*, 2024.

[8] X. Liu *et al.*, "AgentBench: Evaluating LLMs as agents," *arXiv preprint arXiv:2308.03688*, 2023.

[9] M. Dindoost, O. A. Rodriguez, B. Bryg, I. Koutis, and D. A. Bader, "HiPerMotif: Novel parallel subgraph isomorphism in large-scale property graphs," in *IEEE HPEC*, 2025, pp. 1–8.

[10] M. Merrill, W. Reus, and T. Neumann, "Arkouda: Interactive data exploration backed by Chapel," in *CHIUW*, 2019, p. 28.

[11] O. A. Rodriguez, Z. Du, J. Patchett, F. Li, and D. A. Bader, "Arachne: An Arkouda package for large-scale graph analytics," in *IEEE HPEC*, 2022, pp. 1–7.

[12] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the Chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, 2007.

[13] F. Li, Z. Du, and D. A. Bader, "Designing parallel algorithms for community detection using Arachne," in *IEEE HPEC*, 2025, pp. 1–7.

[14] M. Dindoost, O. A. Rodriguez, S. Bagchi, P. Pauliuchenka, Z. Du, and D. A. Bader, "VF2-PS: Parallel and scalable subgraph monomorphism in Arachne," in *IEEE HPEC*, 2024, pp. 1–7.

[15] O. A. Rodriguez, F. V. Buschmann, Z. Du, and D. A. Bader, "Property graphs in Arachne," in *IEEE HPEC*, 2023, pp. 1–7.

[16] A. B. Nia, M. Dindoost, and D. A. Bader, "Evaluating efficiency and novelty of LLM-generated code for graph analysis," in *IEEE HPEC*, 2025, pp. 1–8.

[17] B. Hayes-Roth, "A blackboard architecture for control," *Artificial Intelligence*, vol. 26, no. 3, pp. 251–321, 1985.

[18] R. G. Smith, "The contract net protocol: High-level communication and control in a distributed problem solver," *IEEE Trans. Comput.*, vol. C-29, no. 12, pp. 1104–1113, 1980.

[19] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the Graph 500," in *CUG*, 2010.

[20] S. Beamer, K. Asanović, and D. Patterson, "The GAP benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.