

Adaptive Multi-Agent Orchestration with Human-in-the-Loop for Automated Code Generation

Shivam Parsad¹, Shalini Pandey², Raj kumar Mishra³, Mayank Patel⁴, Divya Pachauri⁵

^{1,2,3,4}Student, Department of Computer Science and Engineering
(Artificial Intelligence and Machine Learning)

⁵Assistant Professor, Department of Computer Science and Engineering
(Artificial Intelligence and Machine Learning)

¹shivamparsad1001@gmail.com ²pshalinip2004@gmail.com

³mishra.raj.edu@gmail.com ⁴ms8190561@gmail.com

⁵divya@nitra.ac.in

Abstract—Automated code-generation projects have been revolutionized by rapidly developing LLMs, which can take natural language instructions and convert them to executable program code. However, an agent’s limited reasoning ability, failure to account for edge cases, and inability to systematically correct errors leads to unreliable output generated by single-agent systems. New research is investigating the multi-agent architectures that allow multiple specialized agents to collaborate and complete complex tasks (planning, coding, and verifying) more effectively. However, many working methods still perform with fixed sequential pipelines, limiting the amount of adaptive decision-making when there are problems/errors. OrchestratorFlow is a multi-agent automated code generation framework that implements an adaptive orchestration method to coordinate the work of its specialized agents (Planning, Design, Coding, Reviewer, Tester). The Orchestrator dynamically re-routes tasks to the appropriate agents based on feedback mechanisms that structure the successes and failures of what the agents have accomplished in their collaborative efforts. This allows the agents to collaboratively refine the generated code iteratively through adaptive orchestration of the work effort. OrchestratorFlow can also be used to incorporate human in the loop participation, allowing users to intervene when ambiguities arise in the planning or design stages. Results from experiments show significant improvements in the reliability of generated code along with fewer runtime errors as compared to that of the historical single-agent approach or the current linear multi-agent approach.

Index Terms—Large Language Models, Code Generation, Multi-Agent Systems, Adaptive Orchestration, Human-in-the-Loop, Software Automation.

I Introduction

Recent progress in Large Language Models (LLMs) has significantly changed the way software development processes are automated. Modern language models generate code from natural language, assist in debugging, and automate repetitive programming tasks. AI-powered tools such as ChatGPT, Claude, and Gemini have demonstrated impressive capabilities; however, they share a critical structural constraint: they are single-model systems [1]. Every task - whether it requires security expertise, database design, or frontend logic - is handled by the same generalist model, with no specialist agents, no role delegation, and no coordinated multi-agent pipeline.

Traditional single-agent LLM systems rely on a single model to interpret instructions and produce code directly. While this works for simple tasks, it fails when problems require multiple reasoning steps or careful validation. Generated code may contain logical mistakes, incomplete implementations, or unhandled edge cases. Without structured verification mechanisms, these errors propagate through the development process.

To address these limitations, researchers have explored multi-agent architectures where different AI agents collaborate to perform specialized roles. MetaGPT [4] and ChatDev [5] demonstrated that role-based agent pipelines produce better software than single models. However, these systems introduced a new problem: all agents are always active, wasting tokens and propagating hallucinations across the pipeline [1], [3]. He et al. [2] empirically confirmed this cost. The integration of LLMs with domain-specific multi-agent pipelines has also been explored in manufacturing contexts [9] and recommendation systems [10], confirming that specialized agent coordination provides meaningful advantages across diverse domains.

I-A Research Background

Guo et al. (2024) [1]: Present the first unified survey of LLM-based Multi-Agent systems, analyzing over 50 studies. Six open challenges are identified - cascading hallucination, scalability, and orchestration complexity - all motivating OrchestratorFlow. Critically, all reviewed systems use always-active agent pools, with no demand-driven activation proposed.

He, Treude & Lo (2025) [2]: Systematically review 71 primary studies. Their empirical ChatDev case study - Snake (success in 2 attempts) vs. Tetris (success only on attempt 10) - directly demonstrates the cost of always-active fixed pipelines. Dynamic agent self-reorganization is proposed as a top priority.

Tran et al. (2025) [3]: Introduce a five-dimensional collaboration framework covering Actors, Type, Structure, Strategy, and Coordination with mathematical formalization. Dynamic coordination (DAG-based runtime agent activation) is most relevant to OrchestratorFlow.

Hong et al. (2023) [4]: MetaGPT encodes software team workflows as Standardized Operating Procedures into role-specific LLM prompts. Establishes that structured role-based collaboration substantially improves code quality; however, static role activation incurs unnecessary cost.

Qian et al. (2023) [5]: ChatDev implements a simulated software company with LLM-powered agents. While demonstrating feasibility of multi-agent development, the always-on pipeline and absence of adaptive routing limit efficiency.

Zhao et al. (2026) [9]: Introduced an LLM-driven multi-agent manufacturing framework designed for intelligent shopfloor environments, consisting of five specialized modules. The system demonstrates how LLM reasoning enables dynamic, adaptive decision-making without pre-training, paralleling OrchestratorFlow’s orchestration philosophy.

Tong et al. (2026) [10]: Introduce SAGE, a five-agent recommendation system with asynchronous coordination, sequential activation, and Monte Carlo robustness validation ($N = 1000$, $R > 0.85$), showing that specialized multi-agent architectures consistently outperform monolithic approaches.

Pan et al. (2026) [11]: Proposed a modular multi-agent architecture that enables natural language communication with graph-based digital twins. The grounded architecture achieves 100% correctness in city-scale deployments, illustrating that decomposing complex queries across specialized agents reduces hallucinations.

II Methodology

II-A System Architecture

OrchestratorFlow receives a natural language programming task and processes it through specialized agents coordinated by an orchestration layer. Unlike traditional code generation systems, it distributes the problem-solving process, with the orchestrator acting as the central controller managing agent execution and information flow.

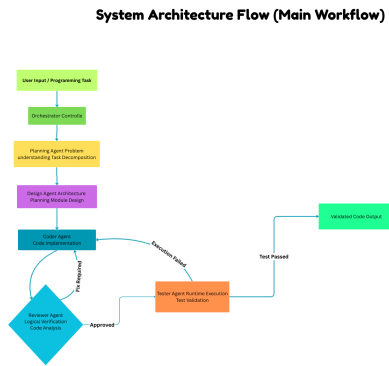


Fig. 1. OrchestratorFlow Agent Pipeline - System Architecture Flow (Main Workflow)

II-B Agent Design

The OrchestratorFlow system employs five specialized agents, each responsible for a specific stage of the development pipeline.

Planning Agent: Analyzes the user’s programming request and generates a high-level plan. Responsibilities include problem decomposition, algorithm identification, and step-by-step solution planning.

Design Agent: Transforms the plan into a structured software architecture. Defines key components such as functions, modules, classes, and data structures.

Coder Agent: Generates executable source code based on plan and design specifications. Also invoked during iterative correction cycles.

Reviewer Agent: Evaluates generated code for logical correctness, completeness, and design consistency. Produces: `review_status = {fix, pass}` with a `feedback_message`.

Tester Agent: Executes generated code in a controlled environment and verifies program behavior. Produces: `test_status = {fail, pass}`. On failure, routes back to Coder.

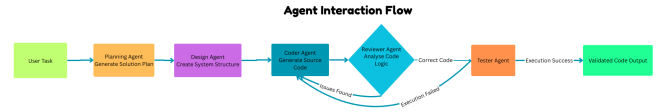


Fig. 2. Agent Interaction Flow - Inter-Agent Communication and Feedback Loops

II-C Orchestrator Control Mechanism

The orchestrator manages the execution sequence of agents and maintains global system state:

```

State = { user_input, plan, design,
          code, review_status, test_status }
  
```

Each agent reads from and updates this shared state. Based on updated values, the orchestrator determines which agent executes next, enabling flexible coordination while avoiding rigid sequential pipelines.

II-D Adaptive Routing Strategy

A key innovation of OrchestratorFlow is its adaptive routing mechanism, which dynamically determines the next agent based on evaluation results. Traditional pipelines follow a fixed order (Planner → Coder → Reviewer → End) with no adaptation when errors occur.

II-E Human-in-the-Loop Integration

To improve decision reliability, OrchestratorFlow employs a Human-in-the-Loop (HITL) mechanism that is triggered when task ambiguity is detected by the planning agent, when design

TABLE I
ADAPTIVE ROUTING LOGIC - ORCHESTRATORFLOW DECISION TABLE

Triggering Condition	Current Agent	Next Agent	Action Taken
Reviewer detects logical issue (<code>review_status = fix</code>)	Reviewer	Coder Agent	Re-generate code with reviewer feedback
Reviewer approves code (<code>review_status = pass</code>)	Reviewer	Tester Agent	Proceed to runtime execution tests
Tester detects runtime failure (<code>test_status = fail</code>)	Tester	Coder Agent	Re-generate code with test failure report
Tester passes all checks (<code>test_status = pass</code>)	Tester	End Workflow	Return verified code as final output
Planning ambiguity detected	Planning Agent	Human-in-the-Loop	Request user clarification on task intent
Design conflict detected	Design Agent	Human-in-the-Loop	Request user guidance on architecture decision

constraints are inconsistent with system limitations, or when additional user requirements are introduced. This reflects the principle observed in SAGE [10], where human expert review is reserved for high-complexity cases.

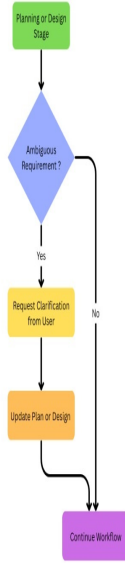


Fig. 3. Human-in-the-Loop Interaction Flow - Ambiguity Detection and User Clarification

II-F Extensibility

The modular design supports future expansion through additional specialized agents (e.g., Security Analyst, Optimization Agent), dynamic reconfiguration of routing logic, and integration with different LLM backends. This aligns with [9],

where LLM-based agents were deployed without retraining across diverse shopfloor configurations.

II-G Algorithm

The formal algorithm for the adaptive multi-agent code generation process is presented in Algorithm 1.

Algorithm 1 OrchestratorFlow Multi-Agent Generation Algorithm

Require: `user_input` - natural language programming task

Ensure: Verified executable code

```

1: plan ← PLANNINGAGENT(user_input)
2: design ← DESIGNAGENT(plan)
3: code ← CODERAGENT(design)
4: while true do
5:   review ← REVIEWERAGENT(code)
6:   if review.status = "fix" then
7:     code ← CODERAGENT(design, review.feedback)
8:   continue
9:   end if
10:  test ← TESTERAGENT(code)
11:  if test.status = "fail" then
12:    code ← CODERAGENT(design, test.feedback)
13:  continue
14:  end if
15: break
16: end while
17: return code // verified output

```

III Results

III-A Experimental Setup

The evaluation was conducted using programming tasks derived from common algorithmic problems. Three configurations were compared: (1) **Single-Agent System** - a single LLM generates code directly without verification; (2) **Linear Multi-Agent Pipeline** - a fixed pipeline without adaptive routing;

(3) **OrchestratorFlow** - the proposed system with five agents and adaptive orchestration.

III-B Evaluation Metrics

Four metrics were defined: Code Correctness, Runtime Error Rate, Iteration Count, and Task Success Rate.

III-C Performance Comparison

Table II summarizes quantitative performance across all three configurations.

TABLE II
PERFORMANCE COMPARISON BETWEEN CODE GENERATION SYSTEMS

System	Success Rate	Error Rate	Avg. Iter.
Single-Agent	52%	34%	1
Linear Multi-Agent	71%	21%	2
OrchestratorFlow	88%	9%	3

The adaptive multi-agent architecture significantly improves task success rates, achieving 88% success versus 52% for single-agent and 71% for linear pipeline systems.

III-D Code Reliability - Success Rate Comparison

Fig. 4 illustrates the task success rate comparison. The three systems are colored distinctly for clarity.

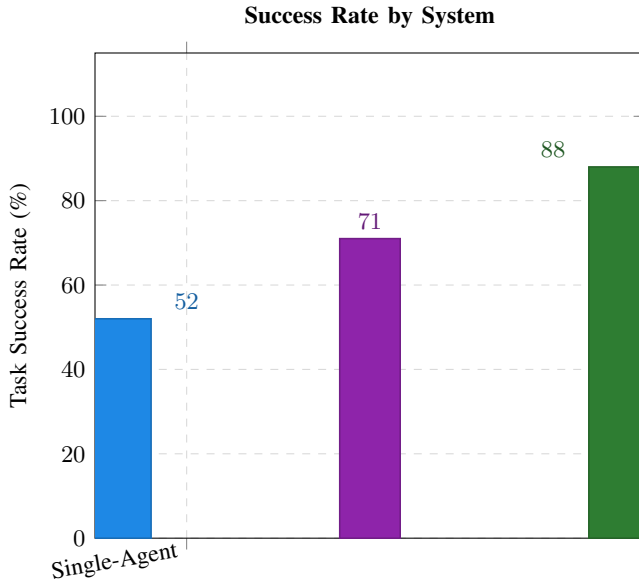


Fig. 4. Task success rate. OrchestratorFlow achieves 88% vs. 52% (single-agent) and 71% (linear pipeline).

III-E Error Reduction - Runtime Error Rate

Fig. 5 presents the runtime error rate comparison (lower is better).

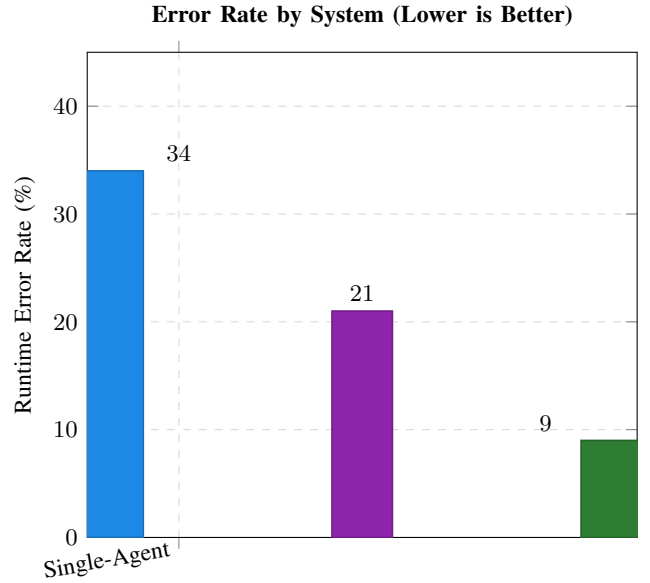


Fig. 5. Runtime error rate. OrchestratorFlow achieves 9% - a 73.5% reduction versus single-agent systems.

The inclusion of both Reviewer and Tester agents significantly reduces runtime errors. OrchestratorFlow reduces errors from 34% (single-agent) to just 9% - a 73.5% reduction.

III-F Iterative Refinement Analysis

Fig. 6 shows average iterations per task for each configuration.

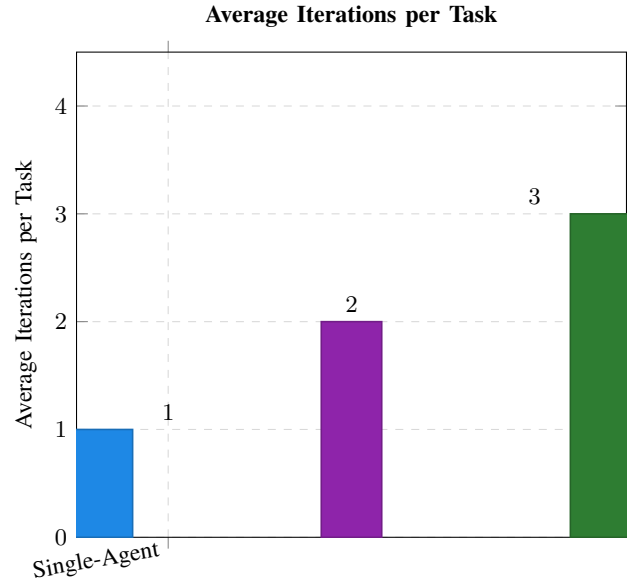


Fig. 6. Average iterations per task. OrchestratorFlow requires more iterations, each enabling refinement that produces higher-quality code.

While OrchestratorFlow requires more iterations, the iterative correction process contributes to higher reliability, justified by the substantial improvements in Figs. 4 and 5.

IV Discussion

IV-A Impact of Adaptive Multi-Agent Collaboration

Multi-agent collaboration significantly enhances problem-solving capabilities. Unlike single-agent models, OrchestratorFlow distributes responsibilities among specialized agents, mimicking real-world software development workflows. This mirrors the LLM-based manufacturing scheduler of [9], where specialized module separation (TM and DM) substantially improved decision stability and adaptability.

IV-B Comparison with Existing Systems

Table III provides a feature comparison between OrchestratorFlow and representative existing systems.

TABLE III
ARCHITECTURAL FEATURE COMPARISON

Feature	ChatDev	MetaGPT	AutoGen	DyLAN	Ours
Agent Activation	Always-On	Always-On	Manual	Perf.-Based	Task-Need
Dormant Agents	×	×	×	Partial	✓
Dynamic Routing	×	×	×	×	✓
HITL Support	×	×	Partial	×	✓
Iterative Correction	Partial	Partial	Partial	Partial	✓
Cost Efficiency	×	×	Partial	Partial	✓
Hallucination Mitigation	Partial	Partial	Partial	Partial	✓

Table III reveals a consistent pattern: agent activation is static and always-on, workflows are fixed at design time, and task-semantic routing does not exist. OrchestratorFlow addresses these gaps through adaptive routing, iterative correction, and human-in-the-loop integration.

IV-C Benefits of Adaptive Orchestration

The adaptive routing mechanism offers three primary advantages. First, *iterative error correction* ensures that when the reviewer detects logical issues or the tester identifies runtime errors, the orchestrator automatically routes the workflow back to the coder agent. Second, *improved code reliability* is achieved by combining logical verification and runtime testing stages. Third, *flexible workflow management* via shared system state enables dynamic decision-making during task execution. This parallels Pan et al. [11], whose modular decision-agent pipeline decomposes complex queries into verifiable sub-steps.

IV-D Human-in-the-Loop Collaboration

Certain programming problems require contextual knowledge or clarification beyond agent capability. The HITL mechanism reduces the risk of incorrect solutions, allows human expertise to guide the AI system, and improves solution quality. SAGE [10] similarly employs human expert review for high-complexity scenarios, demonstrating that hybrid human-AI oversight is a generalizable principle for responsible multi-agent deployment.

IV-E Limitations

Despite its advantages, the system has limitations. First, the overall effectiveness of the system is constrained by the capability of the foundational language models used for agent reasoning. Second, the evaluation focuses on relatively small programming tasks; more complex problems may require additional coordination mechanisms. Third, the iterative correction mechanism may increase computational cost. Addressing these limitations remains an important direction for future research.

V Conclusion

This work introduces OrchestratorFlow, a dynamic multi-agent architecture designed to automate software code generation. The proposed system introduces an orchestration layer that dynamically coordinates interactions between specialized agents responsible for planning, design, implementation, verification, and testing. Unlike traditional single-agent systems or fixed multi-agent pipelines, OrchestratorFlow uses adaptive routing to determine the next step based on verification feedback.

The architecture also incorporates human-in-the-loop interaction, allowing the system to request clarification from users when planning or design ambiguities arise and a modular design supporting future expansion. Experimental evaluation demonstrates 88% task success rate (vs. 52% single-agent), and only 9% runtime error rate (vs. 34% single-agent), confirming the potential of multi-agent collaboration and adaptive orchestration. These findings are consistent with recent research on multi-agent architectures built on large language models.

Future research will focus on expanding the evaluation to larger programming datasets, integrating additional agent roles (optimization, security analysis), and exploring learning-based orchestration strategies.

References

- [1] T. Guo, X. Chen, Y. Wang, R. Chang et al., “Large Language Model based Multi-Agents: A Survey of Progress and Challenges,” *Proc. 33rd Int. Joint Conf. Artif. Intell.*, 2024.
- [2] J. He, C. Treude, and D. Lo, “LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision, and the Road Ahead,” *ACM Trans. Softw. Eng. Methodol.*, 2024.
- [3] K.-T. Tran et al., “Multi-Agent Collaboration Mechanisms: A Survey of LLMs,” arXiv:2501.06322, 2025.
- [4] S. Hong et al., “MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework,” *ICLR 2024*, arXiv:2308.00352, 2023.
- [5] C. Qian et al., “Communicative Agents for Software Development (ChatDev),” arXiv:2307.07924, 2023.
- [6] Q. Wu et al., “AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation,” arXiv:2308.08155, 2023.
- [7] Z. Liu et al., “Dynamic LLM-Agent Network (DyLAN),” arXiv:2310.02170, 2023.
- [8] G. Li et al., “CAMEL: Communicative Agents for Mind Exploration of Large LLM Society,” *NeurIPS 2023*, arXiv:2303.17760, 2023.
- [9] Z. Zhao et al., “A Large Language Model-Based Multi-Agent Manufacturing System for Intelligent Shopfloors,” *Adv. Eng. Inform.*, vol. 69, p. 103888, 2026.
- [10] J. Tong et al., “SAGE: A Theory-Informed LLM-Based Multi-Agent Recommendation System,” *Proc. AAAI 2026*, 2026.
- [11] Y. Pan et al., “LLM-Enabled Multi-Agent Framework for Natural Language Interaction with Graph-Based Digital Twins,” *Autom. Constr.*, vol. 183, p. 106791, 2026.

- [12] A. Vaswani et al., "Attention Is All You Need," *Adv. Neural Inf. Process. Syst.*, vol. 30, 2017.
- [13] T. Brown et al., "Language Models are Few-Shot Learners," *Adv. Neural Inf. Process. Syst.*, vol. 33, pp. 1877–1901, 2020.
- [14] Z. Qin, D. Johnson, and Y. Lu, "Dynamic Production Scheduling towards Self-Organizing Mass Personalization: A Multi-Agent Dueling Deep Reinforcement Learning Approach," *J. Manuf. Syst.*, vol. 68, pp. 242–257, 2023, 10.1016/j.jmsy.2023.03.003.
- [15] J. Xie, X. Li, L. Gao, and L. Gui, "A Hybrid Genetic Tabu Search Algorithm for Distributed Flexible Job Shop Scheduling Problems," *J. Manuf. Syst.*, vol. 71, pp. 82–94, 2023, 10.1016/j.jmsy.2023.09.002.
- [16] L. Huang, D. Tang, Z. Zhang, H. Zhu, Q. Cai, and S. Zhao, "An Iterated Greedy Algorithm Integrating Job Insertion Strategy for Distributed Job Shop Scheduling Problems," *J. Manuf. Syst.*, vol. 77, pp. 746–763, 2024, 10.1016/j.jmsy.2024.10.014.
- [17] Y. Gui, Z. Zhang, D. Tang, H. Zhu, and Y. Zhang, "Collaborative Dynamic Scheduling in a Self-Organizing Manufacturing System Using Multi-Agent Reinforcement Learning," *Adv. Eng. Inf.*, vol. 62, p. 102646, 2024, 10.1016/j.aei.2024.102646.
- [18] Y. Gui, D. Tang, H. Zhu, Y. Zhang, and Z. Zhang, "Dynamic Scheduling for Flexible Job Shop Using a Deep Reinforcement Learning Approach," *Comput. Ind. Eng.*, vol. 180, p. 109255, 2023, 10.1016/j.cie.2023.109255.
- [19] Y. G. Kim, S. Lee, J. Son, H. Bae, and B. D. Chung, "Multi-Agent System and Reinforcement Learning Approach for Distributed Intelligence in a Flexible Smart Manufacturing System," *J. Manuf. Syst.*, vol. 57, pp. 440–450, 2020, 10.1016/j.jmsy.2020.11.004.
- [20] T. Wu, S. He, J. Liu, S. Sun, K. Liu, Q.-L. Han et al., "A Brief Overview of ChatGPT: The History, Status Quo and Potential Future Development," *IEEE/CAA J. Autom. Sin.*, vol. 10, pp. 1122–1136, 2023, 10.1109/JAS.2023.123618.
- [21] M. R. Garey, D. S. Johnson, and R. Sethi, "The Complexity of Flowshop and Jobshop Scheduling," *Math. Oper. Res.*, vol. 1, no. 2, pp. 117–129, 1976, 10.1287/moor.1.2.117.
- [22] C. Jian, J. Ping, and M. Zhang, "A Cloud Edge-Based Two-Level Hybrid Scheduling Learning Model in Cloud Manufacturing," *Int. J. Prod. Res.*, vol. 59, pp. 4836–4850, 2021, 10.1080/00207543.2020.1779371.
- [23] Q. Liu, Z. Gao, J. Li, S. Li, and L. Zhu, "Research on Optimization of Dual-Resource Batch Scheduling in Flexible Job Shop," *Comput. Mater. Contin.*, vol. 76, pp. 2503–2530, 2023, 10.32604/cmc.2023.040505.
- [24] S. Agashe, Y. Fan, and X. E. Wang, "LLMCoordination: Evaluating and Analyzing Multi-Agent Coordination Abilities in Large Language Models," arXiv:2310.03903, 2023.
- [25] M. Andriushchenko, F. Croce, N. Flammarion et al., "AgentHarm: A Benchmark for Measuring Harmfulness of LLM Agents," arXiv:2410.09024, 2024.
- [26] X. Bo, Z. Zhang, Q. Dai, X. Feng, L. Wang, R. Li, X. Chen, and J.-R. Wen, "Reflective Multi-Agent Collaboration based on Large Language Models," in *Adv. Neural Inf. Process. Syst.*, vol. 37, 2024.
- [27] W. Chen, Y. Su, J. Zuo, C. Yang et al., "AgentVerse: Facilitating Multi-Agent Collaboration and Exploring Emergent Behaviors in Agents," arXiv:2308.10848, 2023.
- [28] J. Dagdelen, A. Dunn, S. Lee, N. Walker et al., "Structured Information Extraction from Scientific Text with Large Language Models," *Nat. Commun.*, vol. 15, no. 1, p. 1418, 2024.
- [29] Z. Ding, Z. Liu, Z. Fang, K. Su, L. Zhu, and Z. Lu, "Multi-Agent Coordination via Multi-Level Communication," in *Adv. Neural Inf. Process. Syst.*, vol. 37, 2024.
- [30] Y. Du, S. Li, A. Torralba, J. B. Tenenbaum, and I. Mordatch, "Improving Factuality and Reasoning in Language Models through Multiagent Debate," arXiv:2305.14325, 2023.
- [31] A. Fournay, G. Bansal, H. Mozannar et al., "Magentic-One: A Generalist Multi-Agent System for Solving Complex Tasks," arXiv:2411.04468, 2024.
- [32] M. H. Garcia, C. Couturier, D. M. Diaz, A. Mallick, A. Kyrillidis, R. Sim, V. Ruhle, and S. Rajmohan, "Exploring How LLMs Capture and Represent Domain-Specific Knowledge," arXiv:2504.16871, 2025.
- [33] F. Harer, H.-G. Fill, I. Vasic et al., "Specification and Evaluation of Multi-Agent LLM Systems - Prototype and Cybersecurity Applications," arXiv:2506.10467, 2025.
- [34] Q. Liu, N. Wang, J. Li, T. Ma, F. Li, and Z. Gao, "Research on Flexible Job Shop Scheduling Optimization Based on Segmented AGV," 2023.
- [35] R. Li, W. Gong, L. Wang, C. Lu, and X. Zhuang, "Surprisingly Popular-Based Adaptive Memetic Algorithm for Energy-Efficient Distributed Flexible Job Shop Scheduling," *IEEE Trans. Cybern.*, pp. 1–11, 2023, 10.1109/TCYB.2023.3280175.
- [36] C. Liu, H. Zhu, D. Tang, Q. Nie, T. Zhou, L. Wang et al., "Probing an Intelligent Predictive Maintenance Approach with Deep Learning and Augmented Reality for Machine Tools in IoT-Enabled Manufacturing," *Rob. Comput. Integr. Manuf.*, vol. 77, p. 102357, 2022, 10.1016/j.rcim.2022.102357.
- [37] T. Zhou, D. Tang, H. Zhu, and L. Wang, "Reinforcement Learning with Composite Rewards for Production Scheduling in a Smart Factory," *IEEE Access*, vol. 9, pp. 752–766, 2020.
- [38] Y. Du, J. Li, C. Li, and P. Duan, "A Reinforcement Learning Approach for Flexible Job Shop Scheduling Problem with Crane Transportation and Setup Times," *IEEE Trans. Neural Netw. Learn. Syst.*, pp. 1–15, 2022, 10.1109/TNNLS.2022.3208942.
- [39] R. Liu, R. Piplani, and C. Toro, "Deep Reinforcement Learning for Dynamic Scheduling of a Flexible Job Shop," *Int. J. Prod. Res.*, pp. 1–21, 2022, 10.1080/00207543.2022.2058432.
- [40] S. Luo, "Dynamic Scheduling for Flexible Job Shop with New Job Insertions by Deep Reinforcement Learning," *Appl. Soft Comput.*, vol. 91, p. 106208, 2020, 10.1016/j.asoc.2020.106208.
- [41] X. Wang, L. Zhang, T. Lin, C. Zhao, K. Wang, and Z. Chen, "Solving Job Scheduling Problems in a Resource Preemption Environment with Multi-Agent Reinforcement Learning," *Rob. Comput. Integr. Manuf.*, vol. 77, p. 102324, 2022, 10.1016/j.rcim.2022.102324.
- [42] R. Chen, B. Yang, S. Li, and S. Wang, "A Self-Learning Genetic Algorithm Based on Reinforcement Learning for Flexible Job-Shop Scheduling Problem," *Comput. Ind. Eng.*, vol. 149, p. 106778, 2020, 10.1016/j.cie.2020.106778.
- [43] Z. Qin, "Self-Organizing Manufacturing Network: A Paradigm towards Smart Manufacturing in Mass Personalization," *J. Manuf. Syst.*, vol. 13, 2021.
- [44] K. Alexopoulos, P. Mavrothalassitis, E. Bakopoulos, N. Nikolakis, and D. Mourtzis, "Deep Reinforcement Learning for Selection of Dispatch Rules for Scheduling of Production Systems," *Appl. Sci.*, vol. 15, 2025, 10.3390/app15010232.
- [45] S. Wang, J. Wan, D. Zhang, D. Li, and C. Zhang, "Towards Smart Factory for Industry 4.0: A Self-Organized Multi-Agent System with Big Data Based Feedback and Coordination," *Comput. Netw.*, vol. 101, pp. 158–168, 2016, 10.1016/j.comnet.2015.12.017.