From Automation to Autonomy: A Survey of Agentic Workflows in CI/CD Orchestration

Ali Amar, Ibrahim Qaiser, Ayesha Kanwal*
School of Electrical Engineering and Computer Science, National University of Sciences and Technology, Islamabad, Pakistan
***Correspondence**: ayesha.kanwal@seecs.edu.pk

Citation | Amar. A, Qaiser. I, Kanwal. A, "From Automation to Autonomy: A Survey of Agentic Workflows in CI/CD Orchestration", IJIST, Special Issue pp 476-492, May 2026
Received| March 25, 2026 **Revised|** May 04, 2026 **Accepted|** May 10, 2026 **Published|** May 14, 2026.

Continuous Integration and Continuous Deployment (CI/CD) pipelines are foundational to modern software delivery, yet remain reliant on rigid, pre-defined scripts that lack the flexibility to handle unforeseen anomalies. While prior surveys have examined Large Language Model (LLM) agents for general software engineering tasks, none have focused specifically on CI/CD orchestration and the transition from automation to autonomy. Through a structured review of 72 papers published between 2023 and 2025 and sourced from IEEE Xplore, ACM Digital Library, and arXiv, this survey addresses that gap. The 72 studies are distributed across three primary application domains: autonomous code generation and repair (28 papers), intelligent verification and environment setup (23 papers), and incident management with root cause analysis (21 papers). We propose the PARA (Perception, Action, Reasoning, Reflection) framework as an operational lens for analyzing agentic CI/CD systems. Comparative analysis of five representative systems yields concrete performance figures: SWE-agent resolves 12.5% of issues against a 3.8% scripted baseline; the DEI multi-agent committee reaches 34.3% versus 27.3% for single-agent baselines; CXXCrafter achieves 71.2% success on C/C++ builds compared with 45% for general-purpose agents; MACOG reaches 74.02% on Terraform synthesis, dropping to 61.45% when its Security Prover is ablated; and Flow reports a 67% reduction in Mean Time to Resolution for incident triage. Reported task success across the 72 studies ranges from 15% to 75% as a function of task complexity, and self-correction loops add a further 4–5% per iteration at exponential token cost. Challenges spanning economic viability, security risks, and reliability concerns are systematically analyzed. We conclude that the shift from scripted automation to autonomous agents represents a significant evolution in DevOps practices toward intent-driven orchestration, and outline future directions, including Knowledge Graph-augmented LLMs and standardized Agent-Tool Protocols.

Keywords: CI/CD; Agentic AI; Large Language Models; Multi-Agent Systems; DevOps

Introduction:

Software operations have advanced through successive layers of abstraction. Early practice was manual and difficult to scale; scripting languages later allowed task replication but retained brittle dependencies. The rise of Continuous Integration/Continuous Deployment (CI/CD) pipelines marked the maturation of "DevOps 1.0," in which automation became standardized yet remained strictly imperative [1]. A pipeline could build and deploy code, but a transient network error would halt it because the orchestrator could not judge whether a retry was safe. "DevOps 2.0" subsequently introduced declarative infrastructure through technologies such as Kubernetes and Terraform, where engineers specified the desired state rather than the steps to reach it [2]. Reconciliation loops improved resilience, but the logic within these orchestrators was still hard-coded by human developers, so they could self-heal known issues, such as a crashed pod, while failing on novel faults, such as logical deadlocks in database migrations or cascading service dependencies.

The current transition, referred to as "DevOps 3.0" or "Software Engineering 3.0" [3], integrates autonomous agents powered by Large Language Models (LLMs), using them as core reasoning engines. Unlike rule-based orchestrators, such agents can interpret natural-language intent, decompose complex objectives into executable plans, and dynamically invoke tools to achieve those plans [4]. AI-augmented DevOps frameworks embed LLMs and machine learning directly into the delivery lifecycle to support autonomous and traceable decision-making [5], effectively decoupling *intent* from *implementation*. An engineer operating in this paradigm may state a goal such as "deploy this feature to staging and ensure it handles a 20% traffic spike," and the agent derives the necessary steps: provisioning resources, configuring load balancers, monitoring latency, and rolling back if performance degrades.

Several operational bottlenecks now shift from human cognition to computational reasoning, especially for routine failure triage and environment setup. Agentic workflows make "self-healing" systems possible, systems that analyze root causes and patch code rather than just restarting services [6]. They support "autonomous fuzzing" that evolves test drivers based on compiler feedback [7]. They also make "intelligent incident management" practical, with agents acting as first responders who triage logs and suggest fixes before human intervention is required.

However, this autonomy brings serious risks. The probabilistic nature of LLMs introduces non-determinism into a domain that values predictability. Hallucinations, such as an agent generating a destructive command, create real security threats [8]. The economic cost of processing millions of tokens for routine log analysis also calls into question the return on investment of these systems [9].

We address four research questions: **RQ1:** What architectural patterns characterize agentic CI/CD systems? **RQ2:** What application domains have demonstrated value? **RQ3:** What challenges limit production deployment? **RQ4:** What future directions are most promising?

Research Objectives:

The four research questions above translate into four corresponding, measurable survey objectives. **O1** (RQ1): to derive an operational taxonomy of agent capabilities in CI/CD settings and to map it explicitly to pre-existing cognitive-agent models, producing a framework usable as an analytical lens across heterogeneous systems. **O2** (RQ2): to identify the application domains in which agentic CI/CD systems have demonstrated measurable value and to quantify their reported performance ranges per domain. **O3** (RQ3): to systematically characterize the economic, security, reliability, and organizational barriers that currently constrain production deployment, drawing on the reported evidence from the 72 surveyed studies. **O4** (RQ4): to consolidate the research directions that are most likely to close the

identified gaps, with explicit recommendations for benchmarking, standardization, and human–agent oversight.

Novelty and Contributions:

Prior surveys of LLM-based software engineering agents organize the literature either by developer roles [1] or by lifecycle phase [4], and general AIOps surveys [2] treat CI/CD as one of many downstream concerns. None of these organizations captures the specific operational demands of CI/CD orchestration, which combines deterministic deployment guarantees, environment reproducibility, and continuous observability. The present work contributes four elements that are absent from prior surveys. First, it introduces the PARA (Perception, Action, Reasoning, Reflection) framework as an operational taxonomy for agent capabilities in CI/CD contexts, explicitly contrasted against role-based, lifecycle-based, and communication-based [10] taxonomies, and anchored in the BDI and OODA cognitive-agent models (Section 4). Second, it focuses the analysis on the automation-to-autonomy transition specific to CI/CD, rather than general software engineering [11]. Third, it provides a comparative synthesis of five representative systems (SWE-agent, CSR-Bench, CXXCrafter, MACOG, and Flow) with explicit empirical performance figures and baselines (Table 3). Fourth, it systematically characterizes the economic, security, reliability, and organizational challenges that separate current prototypes from production-grade deployment, and identifies Knowledge Graph-augmented LLMs, Neuro-Symbolic hybrids, and standardized Agent-Tool Protocols as the directions most likely to close the identified gaps.

Literature Review and Related Work:

Several surveys examine LLM agents in software engineering contexts. Table 1 positions this survey against prior work.

Table 1. Comparison with Related Surveys

Survey	Year	Focus	Taxonomy	CI/CD	Papers	Key Limitation
[11]	2023	LLMs for SE	None	Minimal	395	Pre-agentic era
[1]	2024	Multi-Agent SE	Role-based	Limited	106	No operational taxonomy
[4]	2024	SE Agents	Lifecycle	Partial	85	No CI/CD focus
[10]	2024	Multi-Agent	Communication	None	100+	General-purpose scope
[2]	2025	AIOps+LLMs	Task-based	Moderate	150+	Broad IT operations
This Survey	2026	Agentic CI/CD	PARA	Full	72	—

[11] offer a broad mapping of LLMs across SE tasks but predate the agentic shift; their survey includes no framework for analyzing agent autonomy. [4] introduce a lifecycle-based categorization yet treat CI/CD as a side concern, with no analysis of pipeline-specific challenges such as tokenomics or agent security. [1][10] study multi-agent collaboration patterns but organize them by roles and communication protocols rather than operational capabilities, leaving open the question of how agents interact with infrastructure tooling. [2] come closest in scope, surveying AIOps with LLMs, but cover IT operations broadly without examining the particular requirements of CI/CD orchestration, such as deterministic deployment guarantees and environment reproducibility. Our survey fills these gaps with its CI/CD-specific focus, the PARA operational framework that maps directly to pipeline tooling, and a structured analysis of production deployment challenges across economic, security, and organizational dimensions.

Gap Analysis of Recent Agentic CI/CD Literature:

Publications from 2025 have rapidly expanded the agentic-SE literature, but leave three gaps specific to CI/CD orchestration unaddressed. First, recent general-purpose agent surveys [6][8][12] concentrate on code generation benchmarks and agent evaluation

methodology without examining pipeline-level concerns such as build reproducibility, incident triage, or infrastructure-as-code synthesis. Second, recent system contributions target individual pipeline stages in isolation: CSR-Bench [7] evaluates research-repository deployment, CXXCrafter [13] addresses C/C++ build automation, KVFlow [14] optimizes prefix caching for multi-agent workflows, and MACOG [15] targets Terraform synthesis, but no 2025 work unifies these stages under a shared capability lens. Third, while [2] survey AIOps in the LLM era, their scope is IT operations at large, and CI/CD-specific deterministic-guarantee requirements, token-cost viability, and agent security are treated only in passing. The present survey is positioned to fill these three gaps by providing a CI/CD-focused capability taxonomy (Section 4), a cross-stage performance synthesis (Section 5), and a production-readiness assessment (Section 5.5) grounded in the most recent 2023–2025 evidence.

Survey Scope and Approach:

This survey examines LLM-based agentic systems for CI/CD orchestration from 2023–2025, focusing on systems that demonstrate autonomous decision-making rather than simple automation. We searched IEEE Xplore, ACM Digital Library, and arXiv using terms: ("LLM agent" OR "autonomous agent" OR "multi-agent") AND ("CI/CD" OR "DevOps" OR "deployment" OR "AIOps"). We supplemented automated search with snowballing from key papers and consulted industrial blogs from major DevOps platforms.

The search process proceeded in two phases. The automated database search returned 214 candidate papers after removing duplicates. Forward and backward snowballing from the ten most highly cited papers in the initial set contributed an additional 38 candidates, yielding 252 papers for screening. All three authors independently screened titles and abstracts against the inclusion criteria described below, with disagreements resolved through discussion. This screening reduced the pool to 103 full-text candidates, of which 72 met all inclusion criteria after full-text review. The full identification, screening, eligibility, and inclusion pipeline is summarized in Figure 1.

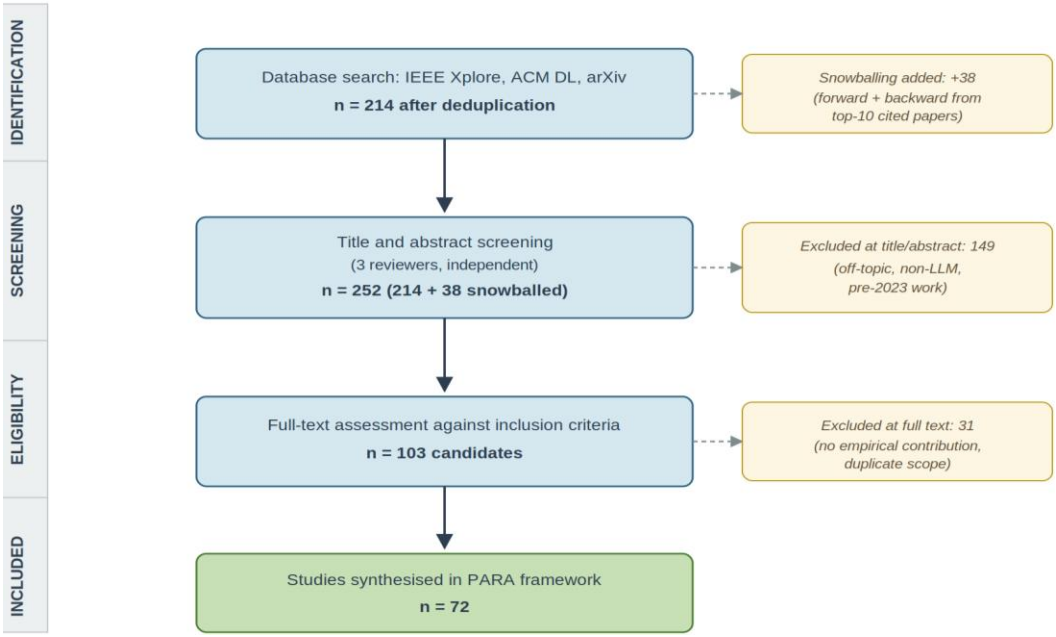

Figure 1. Paper selection pipeline. Identification from IEEE Xplore, ACM Digital Library, and arXiv yielded 214 deduplicated candidates; snowballing added 38, giving 252 records for title and abstract screening. Full-text assessment against inclusion criteria reduced the pool to 103 candidates, of which 72 were included in the final synthesis.

Selection Criteria: We included papers that (1) incorporate LLMs as reasoning engines (not just code completion), (2) target CI/CD/operations domains, and (3) provide empirical

evaluation or architectural contributions. We excluded traditional ML-DevOps papers without LLM agents, pure theoretical frameworks, and pre-2023 work. This yielded 72 papers across three primary domains: code generation/repair (28 papers), verification/testing (23 papers), and incident management (21 papers). Domain assignment was based on each paper's primary contribution: papers whose main evaluation targeted bug fixing, patch generation, or code synthesis were assigned to code generation/repair; those focused on test creation, fuzzing, environment setup, or build automation to verification/testing; and those addressing log analysis, root cause analysis, incident triage, or infrastructure management to incident management. Papers spanning multiple domains were assigned based on the evaluation setting that received the most substantial treatment.

We note two scope limitations. First, industrial systems with proprietary architectures (e.g., internal CI/CD agents at large technology companies) are underrepresented because they lack peer-reviewed publications; we partially compensate through industrial blog references where available. Second, the rapid pace of preprint publication means that some arXiv papers included here have not yet undergone formal peer review, though we prioritized those with empirical evaluations to mitigate this concern. A further methodological caveat is that inter-rater agreement during screening was resolved through discussion rather than a pre-registered Cohen's kappa or similar quantitative metric, in line with the qualitative scoping aims of the survey; a fully pre-registered screening protocol would be appropriate for a systematic review with inferential claims, but was not pursued here because the goal of the present work is structured synthesis rather than meta-analytic aggregation.

Proposed Taxonomy: The PARA Framework:

Agency in SE 3.0 rests on four core capabilities, which we call the PARA framework (Figure 2). This framework addresses **RQ1** by providing an operational lens for understanding agent architectures in CI/CD contexts.

Existing agent taxonomies focus on roles (developer, tester) or lifecycle phases (design, implementation), but CI/CD orchestration demands an understanding of how agents interact with pipeline infrastructure. The PARA framework meets this need by breaking agent capabilities into four concrete dimensions that map directly to DevOps tooling and observability.

Table 2. Formal mapping of classical agent models onto the PARA capabilities and their concrete CI/CD manifestations.

PARA Capability	BDI Component [16]	OODA Phase	CI/CD Manifestation
Perception	Beliefs (state)	Observe	Ingest logs, metrics (Prometheus), AST, PR/ticket text; ACIs [17] summarize noisy terminal output.
Reasoning	Desires (goals)	Orient + Decide	Decompose the deployment goal into provisioning, configuration, and validation (CoT [18], RGD).
Action	Intentions (plans)	Act	Invoke kubectl, git, and terraform through CLIs/APIs/SDKs [19]; sandboxed execution.
Reflection	(not modeled)	(feedback only)	Dry-run (terraform plan), self-critique, Actor–Critic revision [18]; first-class in PARA.

PARA draws on and adapts well-known agent theories for the CI/CD setting. The BDI (Belief-Desire-Intention) model [16] supplies the cognitive foundation: an agent's Beliefs about system state correspond to PARA's Perception, its Desires (deployment goals) inform Reasoning, and its Intentions (committed plans) drive Action. The OODA (Observe-Orient-Decide-Act) loop from decision theory follows a parallel mapping, where Observe aligns with

Perception, Orient and Decide with Reasoning, and Act with Action. PARA goes beyond both models by treating Reflection as a first-class capability, addressing the self-correction requirement that is vital in CI/CD but absent from classical agent theories.

Formally, the PARA loop for one CI/CD decision cycle can be written as s_{t+1} = Refl(Act(Plan(Perc(s_t), g))), where Perc maps the raw environment state s_t into a structured observation, Plan is the Reasoning policy that produces a candidate plan for goal g, Act executes the plan through the tool interface, and Refl critiques the execution trace and either accepts the new state or re-invokes Plan with revised beliefs. Classical BDI and OODA supply Perc, Plan, and Act but leave Refl implicit; PARA makes it explicit, which is essential in CI/CD because the cost of a single unreflected Act on production infrastructure can be catastrophic.

PARA Capabilities:

Perception: The ability to ingest and process multimodal data: unstructured logs, structured metrics from Prometheus, code through ASTs, and natural-language tickets and PRs [20]. In CI/CD, agents must reconcile signals from heterogeneous sources, such as correlating a Prometheus alert with a recent Git commit and an open Jira ticket describing the same symptom. The quality of downstream reasoning depends directly on the fidelity of this ingestion, which is why Agent-Computer Interfaces (ACIs) that present structured, summarized views of the environment have proved particularly important.

Action (Tool Use): The execution of commands through defined interfaces such as APIs, CLIs, or SDKs. This is the "hands" of the agent, enabling commands such as *kubectl get pods*, *git commit*, or *terraform apply*. Executable code actions produce better agent performance than natural language planning alone. The Action layer is also where safety constraints must be enforced: every tool invocation is a potential side-effect on production infrastructure, making permission boundaries and sandboxing (Section 5.5.2) inseparable from Action.

Reasoning (Planning): The capacity to decompose a high-level goal into logical steps through Chain-of-Thought prompting, with multi-LLM approaches such as RGD improving debugging through iterative reasoning cycles. In CI/CD orchestration, Reasoning must handle both planning (breaking a deployment goal into provisioning, configuration, and validation steps) and backward diagnosis (tracing a failing test to its root cause through the dependency graph). Systems that attempt end-to-end execution without intermediate planning consistently underperform those that produce explicit step-by-step plans, as demonstrated by the CSR-Bench results in Section 5.3.

Reflection (Self-Correction): The ability to critique outputs and correct errors. Agents run "mental checks" or dry runs (e.g., *terraform plan*) to verify correctness before applying changes [21]. A traditional CI pipeline treats a failed build as a terminal state; a reflective agent treats it as feedback, re-examining its own patch and generating a revised attempt. This loop enables the 4–5% per-iteration accuracy improvements reported in LLM-ARC, at an exponential token cost discussed in Section 5.5.1.

To illustrate how the four capabilities interact in practice, consider an incident response scenario. A Prometheus alert fires, indicating elevated 5xx error rates on a production microservice (Perception). The agent queries recent deployment history and correlates the spike with a commit merged 20 minutes earlier, then formulates a hypothesis that the new database migration introduced a schema mismatch (Reasoning). It executes a targeted rollback of the offending commit and triggers a canary redeployment (Action). Finally, it monitors the error rate for five minutes post-rollback and, finding it returned to baseline, confirms the fix and updates the incident ticket with a root cause summary (Reflection). Each PARA capability feeds into the next, forming the iterative loop that underpins autonomous orchestration.

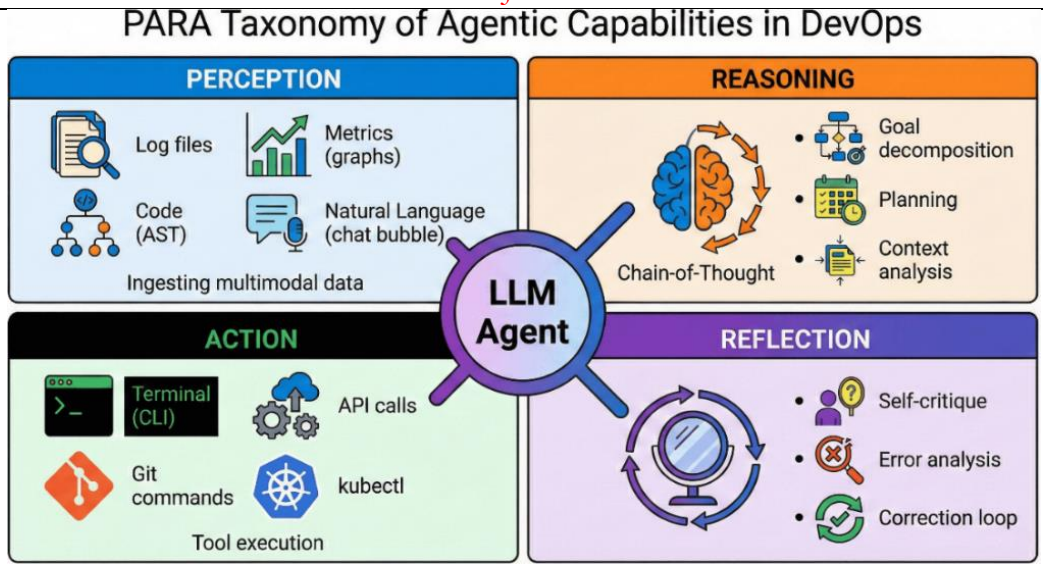

Figure 2. The PARA Framework: Agent capabilities in CI/CD orchestration (Perception, Action, Reasoning, Reflection) with the LLM Agent coordinating all four. The four quadrants surround a central LLM Agent that orchestrates capability invocation. Perception (upper left) ingests multimodal CI/CD inputs (log files, Prometheus metrics, code ASTs, natural-language tickets, and PRs). Reasoning (upper right) consumes the structured observation and performs Chain-of-Thought goal decomposition and planning. Action (lower left) executes the plan through sandboxed tool interfaces (terminal/CLI, Git, kubectl, REST APIs; Section 5.5.2). Reflection (lower right) closes the loop via self-critique and error analysis. Arrows flow clockwise Perception → Reasoning → Action → Reflection, and the Reflection→Reasoning back-edge realizes the self-correction loop formalized in the preceding paragraph.

Cognitive Architectures:

The dominant architectural pattern for DevOps agents is ReAct (Reasoning + Acting) [22]. In a ReAct loop, the agent generates a "Thought," performs an "Action," and waits for an "Observation" (the action's output) before generating the next Thought. This allows dynamic adjustment: when an agent tries to access a file that does not exist, the Observation ("File not found") triggers adaptive reasoning rather than a static failure.

For memory and context management, agents rely on two main mechanisms. Retrieval-Augmented Generation (RAG) lets agents pull in relevant documentation, historical incident reports, or code snippets from vector databases to inform current reasoning [23]. Alongside RAG, Knowledge Graphs (KGs) provide structured "world models" for handling relationships. Causal Knowledge Graphs help agents trace dependencies not explicitly stated in logs, for example, recognizing that Service A depends on Service B based on network topology definitions.

Results and Discussion:

Architecture Patterns in Agentic Systems:

Early coding assistants such as GitHub Copilot operated as single-agent systems, but complex orchestration tasks increasingly call for **Multi-Agent Systems (MAS)**. MAS offers two key advantages (Figure 3). First, *specialization* lets agents take on distinct roles (Developer, Tester, Ops Manager), allowing smaller, fine-tuned models for specific tasks [24]. Second, *collaboration* pools collective intelligence; the DEI framework showed that agent "committees" achieved a 34.3% resolve rate compared with 27.3% for single agents [25].

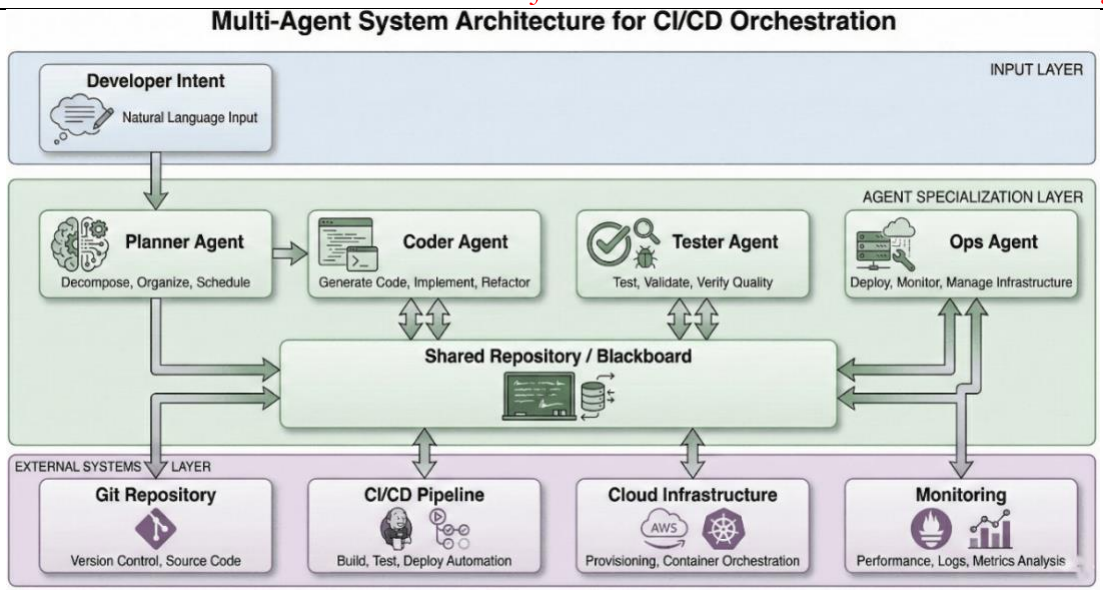

Figure 3. Multi-Agent System Architecture: Specialized agents coordinate via a blackboard pattern, interfacing with Git, CI/CD pipelines, and cloud infrastructure.

Agents coordinate through two main patterns: the Blackboard Pattern, where agents read from and write to a shared persistent state [26], and Message Passing, where agents exchange structured messages.

Domain 1: Autonomous Code Generation and Repair:

Agents such as SWE-agent and OpenDevin act as autonomous developers, connecting to the shell, navigating the repository, and working to solve issues end-to-end. A key innovation here is the Agent-Computer Interface (ACI**)**. LLMs are not built for raw terminal output, which tends to be verbose and noisy. ACIs present simplified, structured views of the file system and linter outputs, cutting token usage and helping the model focus on the signals that matter. Context engineering for multi-agent systems combines intent translation, semantic retrieval, and specialized sub-agents to improve code generation accuracy [27]. CodeAgent showed clear gains by pairing tool use with agent systems for real-world repository-level coding challenges.

The most valuable capability in this domain is the Self-Correction Loop. When an agent submits a patch and the CI build fails, it reads the error logs. Rather than stopping, it enters a "Repair" loop: analyzing the error, modifying the code, and resubmitting. Work on LLM-ARC (Actor-Critic based self-correction) shows that over multiple iterations, accuracy in fixing code compilation issues can improve by 4–5%. The agent in effect "learns" from the compiler, treating the error message as a grounding signal to refine its hypothesis. Self-collaboration approaches let a single LLM role-play as multiple experts (analyst, coder, tester), producing improved code through internal dialogue without requiring multiple model deployments [28].

Across these systems, two design patterns recur. The first is the Agent-Computer Interface (ACI), introduced with SWE-agent. Systems that expose raw shell output to the LLM tend to do worse on task success than those that provide structured file navigation, linting, and build output summaries, so ACIs are now a common scaffolding choice. The second is multi-agent role specialization, which appears either as explicit agent committees (DEI, MetaGPT, CodePori) or as a single model role-playing multiple expert personas (self-collaboration, self-organized agents). Both forms produce gains of roughly 15–25% over single-agent pipelines on repository-level benchmarks. Context engineering is often the limiting factor: if intent translation, semantic retrieval, and sub-agent delegation are not carefully designed, the extra agents add coordination overhead without improving accuracy.

Domain 2: Intelligent Verification and Fuzzing:

Verification remains a major bottleneck in CI/CD. Before agents can generate or test code, they must reliably set up execution environments. [29] showed that even leading agents struggle with dependency resolution and environment configuration, with success rates falling sharply when projects involve non-standard build systems or implicit OS-level dependencies. For test generation, the "Assured" principle filters tests that (1) compile, (2) pass, and (3) increase coverage, countering hallucination where tests reference non-existent functions. This filtering does impose a trade-off: in practice, 40–60% of agent-generated tests are discarded during the assurance stage, raising token expenditure per useful test produced.

Deploying research code is especially challenging because of incomplete documentation and implicit dependencies. CSR-Bench evaluates agents on research repositories and finds that multi-step reasoning is essential; agents that break build tasks into discrete subtasks (dependency identification, configuration generation, iterative error resolution) achieve notably higher success than those attempting end-to-end builds. For C/C++ building, CXXCrafter shows that domain-specific agents with tailored tool interfaces outperform general-purpose agents (71.2% vs. 45% success rate), suggesting that the cost of developing specialized tool wrappers is well justified by the accuracy gains. AutoML-Agent [30] applies these patterns to ML pipeline automation, where environment reproducibility is equally important.

Two patterns stand out across these verification systems. First, build-environment reproducibility is where most of the difficulty actually lives. [29] report that dependency-resolution failures cause most failed research-repository deployments, and CSR-Bench reaches the same conclusion through its multi-step decomposition results. Second, the gap between language-specific and general-purpose agents is not closing as models scale up. CXXCrafter's 26-point lead over general-purpose baselines and the specialized architecture of AutoML-Agent both suggest that, for verification work, a well-designed tool interface and good domain priors matter more than simply using a larger base model. For teams building verification agents, this means the investment should go into tool wrappers that encode build-system idioms, not into chasing bigger models.

Domain 3: Incident Management and Root Cause Analysis:

The "Operations" phase is where agents serve as Site Reliability Engineers (SREs), looking after the production environment. Log analysis is the classic "needle in a haystack" problem: LLMs have limited context windows, yet production logs can span gigabytes. A single Kubernetes cluster can produce over 10 GB of logs daily, far exceeding what current models can process at once. Recent surveys on AIOps in the LLM era identify multi-stage pipelines as the prevailing pattern: (1) preprocessing to filter key logs and prune token overflow, (2) retrieval-augmented generation (RAG) to incorporate historical incident data, and (3) LLM-based reasoning to synthesize findings into actionable reports. The evidence suggests that architectural support, preprocessing combined with RAG, matters more than raw model capability alone; models with RAG pipelines consistently outperform larger models without retrieval support on incident triage benchmarks.

Incident response often requires following Troubleshooting Guides (TSGs), which are typically written as unstructured wiki documents that tend to become outdated or ambiguous. [31] introduced Flow, a framework for modularized agentic workflow automation that parses unstructured procedures into structured workflows represented as Directed Acyclic Graphs (DAGs). Because the workflow structure is explicit, Flow can spot independent branches and run them in parallel, combining the reliability of structured processes with the flexibility of LLM-based interpretation and cutting Mean Time to Resolution (MTTR) substantially. This structured approach also strengthens auditability, since each resolution step is traceable

through the DAG, a property that is increasingly valued in regulated industries where incident remediation must meet compliance requirements.

In the infrastructure domain, MACOG (Multi-Agent Code-Orchestrated Generation) targets the generation of Terraform infrastructure. An LLM might produce Terraform code that is syntactically valid but insecure (e.g., creating a public S3 bucket) or wasteful (e.g., provisioning oversized instances). MACOG tackles this with a multi-agent pipeline: a Planner agent proposes an architecture, a Coder agent generates the HCL, a Security Prover validates against Open Policy Agent (OPA) policies with feedback to the Coder on violations, and a Sandbox runs Terraform plan to catch logical errors before deployment. An ablation study showed that removing the Security Prover dropped the success rate from 74.02% to 61.45%, confirming that policy-based guardrails are essential. Related multi-agent patterns appear in adjacent infrastructure domains [32][33].

The main design choice in this domain is how much structure to impose on agent reasoning. Flow sits at one end: it parses unstructured troubleshooting guides into Directed Acyclic Graphs before execution, which gives up some flexibility in interpretation but gains auditability and parallel execution. MACOG takes a similar approach with its Planner, Coder, Prover, and Sandbox stages, each with a clearly defined output. Pure ReAct-based incident agents sit at the other end; they are faster to prototype but do not easily meet the compliance-logging requirements (SOC 2, ISO 27001) of regulated deployments, as discussed in Section 5.5.4. Hybrid patterns that combine formal-language scaffolding with LLM flexibility [34] look like a reasonable compromise, but empirical evidence from production incident streams outside industry-published case studies on smart cities and autonomous networks is still thin.

Table 3. Performance Synthesis of Key Agentic Systems

System	Domain	Architecture	Performance	Baseline
SWE-agent [3]	Issue Resolution	ReAct + ACI	12.5%	3.8%
DEI [25]	Code Repair	Multi-Agent	34.3%	27.3%
CXXCrafter [13]	C/C++ Building	Single + Tools	71.2%	45%
CSR-Bench [7]	Research Deploy	Benchmark	38.5%	N/A
MACOG [15]	IaC (Terraform)	Blackboard MAS	74.0%	54%
Flow [31]	Incident Mgmt	DAG-Guided	67% MTTR↓	Manual

Key Performance Insights: (1) Multi-agent systems show 15–25% improvement over single agents for complex tasks (DEI vs. single agent). (2) Domain-specific tool integration (ACI, Security Prover) provides larger gains than pure model scaling. (3) Current systems achieve 40–75% success on well-defined tasks, suggesting production readiness requires human-in-the-loop hybrid approaches. (4) Self-correction loops (Reflection) improve accuracy by 4–5% per iteration, but with exponential token costs.

Critical Challenges:

Economic Viability: Tokenomics:

Token consumption remains a primary economic barrier to adoption. An agentic loop that retries a task ten times while carrying the full conversation history consumes token volumes that are orders of magnitude higher than a single-shot request, so principled memory management is essential for long-running agent sessions.

The TALE (Token-Aware Layered Execution) framework proposes strategies such as "Event-based Pruning," cutting token usage by roughly 67% through intelligent history summarization that keeps only successful reasoning paths while discarding dead-end branches. KVFlow similarly introduces efficient prefix caching for speeding up LLM-based multi-agent workflows, targeting the computational overhead of context management in complex agentic systems.

Security: The Lethal Trifecta:

Allowing an agent to execute code introduces what we call "The Lethal Trifecta": Filesystem Access, Privilege Escalation, and Network Access [35]. Each vector carries distinct risks: filesystem access opens the door to credential theft, network access permits data exfiltration, and privilege escalation allows container escape.

Agents are vulnerable to indirect prompt injection, where a malicious PR title could theoretically manipulate an agent into executing harmful commands. Guardrails such as Llama Firewall act as proxies, scanning user inputs for adversarial patterns (e.g., "Ignore previous instructions") and scanning agent outputs for dangerous code patterns (e.g., hardcoded credentials or dangerous syscalls) before execution.

Incidents have been documented in which agents generated destructive shell commands (e.g., *rm -rf **) in response to ambiguous user instructions such as "clean up." Strict sandboxing is therefore mandatory: agents should execute in ephemeral Docker containers without root privileges and with read-only volume mounts. Chaos-engineering techniques have been proposed to systematically probe and improve the robustness of LLM-based multi-agent systems [35].

Reliability and Determinism:

DevOps practice demands deterministic behavior, but LLMs are probabilistic: an agent may repair a defect on one invocation and fail on the identical input on the next [36]. Trustworthy human–agent collaboration in this setting requires direct treatment of these reliability concerns, and recent work on multi-agent decision-making identifies inter-agent coordination and communication as the principal open challenges [37].

Mitigations include the "Reviewer" pattern, where a second, separate agent critiques the first, creating a check-and-balance system. Grounding is also important: StepFly's Query Preparation Plugins ensure the agent cannot hallucinate metric names; it must pick from a valid list. LogSage's use of RAG ensures the agent's reasoning draws on real historical data, not just training weights. Formal-LLM integrates formal language specifications with natural language for controllable LLM-based agents.

Unified modeling frameworks help by providing structured approaches to integrating active and passive core agents [38]. Evaluation-driven development approaches propose process models and reference architectures that treat evaluation as a continuous function rather than a final checkpoint [39]. Balancing autonomy with alignment remains an open problem, motivating multi-dimensional taxonomies for autonomous LLM-powered multi-agent architectures.

Production Deployment Gaps:

Production deployment faces organizational barriers on top of the technical ones. DevOps demands predictable behavior for compliance audits, yet LLM agents produce probabilistic outputs, calling for hybrid approaches where agents handle exploratory tasks while deterministic systems run the critical paths. In regulated industries such as finance and healthcare, every automated action must leave an auditable trace; current agent frameworks seldom generate the structured logs needed to satisfy SOC 2 or ISO 27001 requirements. When agents fail, traditional debugging falls short because reasoning chains may span hundreds of opaque LLM calls, requiring specialized observability tools that current platforms do not yet offer. Additionally, the infrastructure overhead (sandboxing, guardrails, context caching) may outweigh automation benefits for narrow use cases. Economic viability depends on targeting high-value, high-frequency tasks such as flaky test triage rather than full pipeline autonomy.

Future Directions:

Agentic Knowledge Graphs: Current RAG systems retrieve text chunks without grasping relationships. Future systems will use Agentic KGs where agents both read from and write to

the graph, updating knowledge bases with incident findings. Concretely, an Agentic KG for CI/CD would encode service dependencies, historical failure modes, and resolution strategies as a structured graph. When an incident occurs, the agent would traverse the graph to identify causal chains. For example, it could recognize that Service A's latency spike is likely caused by Service B's recent schema migration, because a dependency edge links the two, and a prior incident node records the same pattern. After resolution, the agent writes the new finding back into the graph, building an organizational memory that outlasts individual team members. The primary research challenge is maintaining graph consistency as multiple agents write concurrently and as the underlying infrastructure evolves.

Neuro-Symbolic Hybrids: Pure LLMs struggle with strict logic (e.g., dependency resolution). We anticipate Neuro-Symbolic agents combining LLM creativity with SMT solver rigor for configuration management requiring provable correctness. Not all CI/CD constraints are equally suited to symbolic treatment. Version compatibility checks and resource quota satisfaction (e.g., ensuring a Terraform plan does not exceed account-level CPU limits) are naturally expressed as constraint satisfaction problems amenable to SMT solvers. In contrast, tasks that require contextual judgment, such as deciding whether a flaky test failure warrants a retry or signals a genuine regression, remain better handled by LLM-based reasoning. We expect the most effective architectures to partition the problem: the LLM generates candidate plans in natural language, a symbolic verifier checks them against formal specifications, and the LLM revises any plan that fails verification, creating a generate-verify-revise loop that draws on the strengths of each approach.

Standardization: The industry needs a standardized Agent-Tool Protocol (ATP), similar to LSP, enabling any agent to connect to any tool. MCP-Zero demonstrates active tool discovery [40], Meta-Agent-Workflow streamlines tool usage [41], and Agent Net proposes decentralized coordination [42]. Without such a protocol, every agent-tool integration requires bespoke wrappers, creating a combinatorial explosion as both the number of agents and the number of DevOps tools grow. An ATP would define a common schema for tool capability advertisement, invocation, and result reporting, allowing a newly developed agent to discover and use existing pipeline tools without custom adapters. The emergence of the Model Context Protocol (MCP) in 2024–2025 represents an early step in this direction, though its current scope is broader than CI/CD and lacks the domain-specific semantics (e.g., deployment rollback guarantees, idempotency declarations) that pipeline orchestration demands.

Open Problems: Four cross-cutting challenges remain. **Benchmark standardization** for agentic CI/CD is lacking, since no widely adopted harness captures end-to-end build-test-deploy-monitor orchestration, and the community needs **open datasets** of anonymized failure traces, build logs, and incident reports for reproducible evaluation. Formal **cost–quality tradeoff** models would let practitioners predict whether agentic approaches are economically justified for a given pipeline complexity. Calibrated **human–agent oversight** between full human-in-the-loop approval and fully autonomous execution remains underexplored. Finally, **privacy-preserving cross-organization learning** via federated learning and differential privacy could unlock proprietary incident data, though applying these techniques to sequential tool-using reasoning traces is technically open. Concrete recommendations for addressing these problems are given in Section 7.2.

Implications, Recommendations, and Conclusion:

Answering the Research Questions:

Table 4 consolidates the survey findings against the four research questions introduced in Section 1, linking each question to its corresponding objective, the paper section in which it is addressed, and the quantitative evidence that supports the answer.

Table 4. Mapping of research questions to objectives, paper sections, and supporting evidence.

RQ	Obj.	Section(s)	Key Finding and Supporting Evidence
RQ1	O1	4, 5.1	Agentic CI/CD systems are characterized by the PARA capabilities (Perception, Action, Reasoning, Reflection), mapped onto BDI and OODA in Table 2 and realized over ReAct loops (Section 4.2). Multi-agent designs outperform single-agent baselines for complex tasks: DEI committees reach 34.3% resolution vs. 27.3% for single agents.
RQ2	O2	5.2–5.4	Three domains have demonstrated value across the 72 surveyed studies: code generation/repair (28 papers), verification/environment setup (23), and incident management (21). Reported task-success rates span 15–75% depending on complexity, with domain-specific examples SWE-agent 12.5% (vs. 3.8% scripted), CXXCrafter 71.2% (vs. 45%), MACOG 74.0%, and Flow 67% MTTR reduction Table 3.
RQ3	O3	5.5	Production deployment is constrained on four axes: tokenomics (event-based pruning cuts cost ~67% [9]); security (the Lethal Trifecta of filesystem, network, and privilege access [35]) reliability (LLM stochasticity clashing with deterministic pipeline requirements [36][37]); and organizational readiness (observability and oversight gaps, 5.5.4).
RQ4	O4	6	The most promising directions are Agentic Knowledge Graphs, Neuro-Symbolic hybrids with SMT verification, a standardized Agent-Tool Protocol (ATP) extending MCP [40][41][42], and open end-to-end CI/CD benchmarks with shared failure-trace datasets.

Implications and Recommendations:

Theoretically, the PARA framework shows that classical BDI and OODA models require Reflection to be elevated to a first-class capability in CI/CD, because the cost of an unreflected Action on production infrastructure is unbounded. Practically, the 15–75% success ranges across the 72 studies indicate that agentic approaches are production-viable only for well-scoped, high-frequency tasks (incident triage, flaky-test classification, dependency resolution); full pipeline autonomy is premature, and hybrid architectures (agents propose, deterministic systems execute) should be the default deployment pattern. For industry, the DevOps 3.0 transition is an organizational shift more than a tooling upgrade, and requires new competencies in prompt engineering, agent-trace observability, and calibrated human–agent oversight. **For researchers,** we recommend (i) Knowledge Graph–LLM integration for grounding, (ii) end-to-end CI/CD benchmarks and open failure-trace datasets spanning build–test–deploy–monitor, (iii) formal cost–quality tradeoff models, and (iv) calibrated-autonomy oversight between full human-in-the-loop and fully autonomous execution. **For practitioners,** we recommend (i) ephemeral non-root Docker sandboxes with read-only mounts and guardrail proxies before any production tool access, (ii) staged rollout starting with log summarization and flaky-test triage, (iii) agent-trace instrumentation through distributed-tracing platforms, and (iv) deterministic-fallback architectures on safety-critical paths to preserve SOC 2 and ISO 27001 auditability.

Limitations:

Three limitations should be noted. First, the 72-paper scope excludes proprietary industrial agent deployments, which are not publicly documented; the conclusions, therefore, apply primarily to the open research literature. Second, the performance figures in Table 3 are drawn from heterogeneous benchmarks (SWE-bench, CSR-Bench, terraform suites, incident-

triage datasets) that differ in task definition, baseline, and metric, so we report qualitative patterns and the authors' own numbers rather than attempting a meta-analytic aggregation, weighted confidence intervals, or significance tests across incomparable settings. Third, screening disagreements were resolved through discussion rather than a pre-registered Cohen's kappa, consistent with the qualitative scoping orientation of the survey.

Conclusion:

The transition from scripted automation to LLM-driven autonomy in CI/CD orchestration is bounded by economic, security, reliability, and organizational constraints that model scaling alone will not dissolve. Success will depend on the right scaffolding: Memory (RAG and Knowledge Graphs), Perception (Agent-Computer Interfaces), Action (sandboxed tool interfaces), and Reflection (guardrails and dry-run verification). It will also depend on beginning with well-scoped, high-frequency tasks rather than full pipeline autonomy. The next wave of progress will come from Agentic Knowledge Graphs, Neuro-Symbolic hybrids, and a standardized Agent-Tool Protocol.

Acknowledgements:

The authors would like to thank the reviewers for their constructive feedback, which helped improve the quality of this paper. The authors declare no conflict of interest.

References:

[1] Junda He, Christoph Treude, David Lo, "LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision and the Road Ahead," *arXiv:2404.04834*, 2024, [Online]. Available: https://arxiv.org/abs/2404.04834

[2] Lingzhe Zhang, Tong Jia, Mengxi Jia, Yifan Wu, Aiwei Liu, Yong Yang, Zhonghai Wu, Xuming Hu, Philip S. Yu, Ying Li, "A Survey of AIOps in the Era of Large Language Models," *arXiv:2507.12472*, 2025, [Online]. Available: https://arxiv.org/abs/2507.12472

[3] Chunqiu Steven Xia, Yinlin Deng, "Demystifying LLM-Based Software Engineering Agents," *Proc. ACM Softw. Eng.*, vol. 2, 2025, [Online]. Available: https://dl.acm.org/doi/10.1145/3715754

[4] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, Huaming Chen, "From LLMs to LLM-based Agents for Software Engineering: A Survey of Current, Challenges and Future," *arXiv:2408.02479*, 2025, [Online]. Available: https://arxiv.org/abs/2408.02479

[5] M. S. C. Mr. Balajee Asish Brahmandam, Mr. Vishal Narender Punjabi, "AI-Augmented DevOps: Autonomous Software Delivery with Large Language Models," *IJIRMP*, vol. 13, no. 3, 2025, [Online]. Available: https://www.ijirmps.org/papers/2025/3/232448.pdf

[6] Yihong Dong, Xue Jiang, "A Survey on Code Generation with LLM-based Agents," *arXiv:2508.00083v1*, 2025, [Online]. Available: https://arxiv.org/html/2508.00083v1

[7] Yijia Xiao, Runhui Wang, Luyang Kong, Davor Golac, Wei Wang, "CSR-Bench: Benchmarking LLM Agents in Deployment of Computer Science Research Repositories," *arXiv:2502.06111*, 2025, [Online]. Available: https://arxiv.org/abs/2502.06111

[8] Asaf Yehudai, Lilach Eden, Alan Li, Guy Uziel, Yilun Zhao, Roy Bar-Haim, Arman Cohan, Michal Shmueli-Scheuer, "Survey on Evaluation of LLM-based Agents," *arXiv:2503.16416*, 2025, [Online]. Available: https://arxiv.org/abs/2503.16416

[9] Hatalis, K., Christou, D., Myers, J., Jones, S., Lambert, K., Amos-Binks, A., Dannenhauer, Z., & Dannenhauer, D, "Memory Matters: The Need to Improve Long-Term Memory in LLM-Agents," *Proc. AAAI Symp. Ser.*, vol. 2, no. 1, pp. 277–280, 2024, doi: https://doi.org/10.1609/aaaiss.v2i1.27688.

[10] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V.

Chawla, Olaf Wiest, Xiangliang Zhang, "Large Language Model based Multi-Agents: A Survey of Progress and Challenges," *arXiv:2402.01680*, 2024, [Online]. Available: https://arxiv.org/abs/2402.01680

[11] Xinyi Hou, Yanjie Zhao, "Large Language Models for Software Engineering: A Systematic Literature Review," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 8, pp. 1–79, 2024, [Online]. Available: https://dl.acm.org/doi/10.1145/3695988

[12] Yixin Liu, Guibin Zhang, Kun Wang, Shiyuan Li, Shirui Pan, "Graph-Augmented Large Language Model Agents: Current Progress and Future Prospects," *arXiv:2507.21407*, 2025, [Online]. Available: https://arxiv.org/abs/2507.21407

[13] Zhengmin Yu, Yuan Zhang, Ming Wen, Yinan Nie, Wenhui Zhang, Min Yang, "CXXCrafter: An LLM-Based Agent for Automated C/C++ Open Source Software Building," *arXiv:2505.21069*, 2025, [Online]. Available: https://arxiv.org/abs/2505.21069

[14] Zaifeng Pan, Ajjkumar Patel, Zhengding Hu, Yipeng Shen, Yue Guan, Wan-Lu Li, Lianhui Qin, Yida Wang, Yufei Ding, "KVFlow: Efficient Prefix Caching for Accelerating LLM-Based Multi-Agent Workflows," *arXiv:2507.07400*, 2025, [Online]. Available: https://arxiv.org/abs/2507.07400

[15] Dheer Toprani, Vijay K. Madisetti, "LLM Agentic Workflow for Automated Vulnerability Detection and Remediation in Infrastructure-as-Code," *IEEE Access*, vol. 13, 2025, [Online]. Available: https://ieeexplore.ieee.org/document/10965635

[16] Thorsten Händler, "Balancing Autonomy and Alignment: A Multi-Dimensional Taxonomy for Autonomous LLM-powered Multi-Agent Architectures," *arXiv:2310.03659*, 2023, [Online]. Available: https://arxiv.org/abs/2310.03659

[17] Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, Zhi Jin, "CodeAgent: Enhancing Code Generation with Tool-Integrated Agent Systems for Real-World Repo-level Coding Challenges," *arXiv:2401.07339*, 2024, [Online]. Available: https://arxiv.org/abs/2401.07339

[18] Haolin Jin, Zechao Sun, Huaming Chen, "RGD: Multi-LLM Based Agent Debugger via Refinement and Generation Guidance," *arXiv:2410.01242*, 2024, [Online]. Available: https://arxiv.org/abs/2410.01242

[19] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, Heng Ji, "Executable Code Actions Elicit Better LLM Agents," *arXiv:2402.01030*, 2024, [Online]. Available: https://arxiv.org/abs/2402.01030

[20] V. Gogineni, "LLM-Powered Multi-Agent Systems: A Technical Framework for Collaborative Intelligence Through Optimized Knowledge Retrieval and Communication," *2025 6th Int. Conf. Artif. Intell. Robot. Control. AIRC 2025*, pp. 452–456, 2025, doi: 10.1109/AIRC64931.2025.11077480.

[21] Yoichi Ishibashi, Yoshimasa Nishimura, "Self-Organized Agents: A LLM Multi-Agent Framework toward Ultra Large-Scale Code Generation and Optimization," *arXiv:2404.02183*, 2024, [Online]. Available: https://arxiv.org/abs/2404.02183

[22] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, Lingming Zhang, "Agentless: Demystifying LLM-based Software Engineering Agents," *arXiv:2407.01489*, 2024, [Online]. Available: https://arxiv.org/abs/2407.01489

[23] Siru Liu, Allison B. McCoy, "Improving large language model applications in biomedicine with retrieval-augmented generation: a systematic review, meta-analysis, and clinical development guidelines," *J. Am. Med. Inform. Assoc.*, vol. 32, no. 4, 2025, [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/39812777/

[24] Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, "MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework," *arXiv:2308.00352*, 2023, [Online]. Available:

https://arxiv.org/abs/2308.00352

[25] Zhuoyun Du, Chen Qian, Wei Liu, Zihao Xie, YiFei Wang, Rennai Qiu, Yufan Dang, Weize Chen, Cheng Yang, Ye Tian, Xuantang Xiong, Lei Han, "Multi-Agent Collaboration via Cross-Team Orchestration," *arXiv:2406.08979*, 2024, [Online]. Available: https://arxiv.org/abs/2406.08979

[26] Zeeshan Rasheed, Malik Abdul Sami, Kai-Kristian Kemell, Muhammad Waseem, Mika Saari, Kari Systä, Pekka Abrahamsson, "CodePori: Large-Scale System for Autonomous Software Development Using Multi-Agent Technology," *arXiv:2402.01411*, 2024, [Online]. Available: https://arxiv.org/abs/2402.01411

[27] Muhammad Haseeb, "Context Engineering for Multi-Agent LLM Code Assistants Using Elicit, NotebookLM, ChatGPT, and Claude Code," *arXiv:2508.08322*, 2025, [Online]. Available: https://arxiv.org/abs/2508.08322

[28] Yihong Dong, Xue Jiang, Zhi Jin, Ge Li, "Self-collaboration Code Generation via ChatGPT," *arXiv:2304.07590*, 2023, [Online]. Available: https://arxiv.org/abs/2304.07590

[29] Louis Milliken, Sungmin Kang, Shin Yoo, "Beyond pip install: Evaluating LLM Agents for the Automated Installation of Python Projects," *arXiv:2412.06294*, 2024, [Online]. Available: https://arxiv.org/abs/2412.06294

[30] Patara Trirat, Wonyong Jeong, Sung Ju Hwang, "AutoML-Agent: A Multi-Agent LLM Framework for Full-Pipeline AutoML," *arXiv:2410.02958*, 2024, [Online]. Available: https://arxiv.org/abs/2410.02958

[31] Boye Niu, Yiliao Song, Kai Lian, Yifan Shen, Yu Yao, Kun Zhang, Tongliang Liu, "Flow: Modularized Agentic Workflow Automation," *arXiv:2501.07834*, 2025, [Online]. Available: https://arxiv.org/abs/2501.07834

[32] Anna Kalyuzhnaya, Sergey Mityagin, "LLM Agents for Smart City Management: Enhancing Decision Support Through Multi-Agent AI Systems," *Smart Cities*, vol. 8, no. 1, p. 19, 2025, doi: https://doi.org/10.3390/smartcities8010019.

[33] Masoud Shokrnezhad, Tarik Taleb, "An Autonomous Network Orchestration Framework Integrating Large Language Models with Continual Reinforcement Learning," *arXiv:2502.16198*, 2025, [Online]. Available: https://arxiv.org/abs/2502.16198

[34] Zelong Li, Wenyue Hua, Hao Wang, He Zhu, Yongfeng Zhang, "Formal-LLM: Integrating Formal Language and Natural Language for Controllable LLM-based Agents," *arXiv:2402.00798*, 2024, [Online]. Available: https://arxiv.org/abs/2402.00798

[35] Joshua Owotogbe, "Assessing and Enhancing the Robustness of LLM-based Multi-Agent Systems Through Chaos Engineering," *arXiv:2505.03096*, 2025, [Online]. Available: https://arxiv.org/abs/2505.03096

[36] Krishna Ronanki, "Facilitating Trustworthy Human-Agent Collaboration in LLM-based Multi-Agent System oriented Software Engineering," *arXiv:2505.04251*, 2025, [Online]. Available: https://arxiv.org/abs/2505.04251

[37] C. Sun, S. Huang, and D. Pompili, "LLM-Based Multi-Agent Decision-Making: Challenges and Future Directions," *IEEE Robot. Autom. Lett.*, vol. 10, no. 6, pp. 5681–5688, 2025, doi: 10.1109/LRA.2025.3562371.

[38] Amine Ben Hassouna, Hana Chaari, Ines Belhaj, "LLM-Agent-UMF: LLM-based Agent Unified Modeling Framework for Seamless Design of Multi Active/Passive Core-Agent Architectures," *arXiv:2409.11393*, 2024, [Online]. Available: https://arxiv.org/abs/2409.11393

[39] Boming Xia, Qinghua Lu, Liming Zhu, Zhenchang Xing, Dehai Zhao, Hao Zhang, "Evaluation-Driven Development and Operations of LLM Agents: A Process Model

and Reference Architecture," *arXiv:2411.13768*, 2024, [Online]. Available: https://arxiv.org/abs/2411.13768

[40] Xiang Fei, Xiawu Zheng, Hao Feng, "MCP-Zero: Active Tool Discovery for Autonomous LLM Agents," *arXiv:2506.01056*, 2025, [Online]. Available: https://arxiv.org/abs/2506.01056

[41] Xiaoyu Tan, Bin Li, "Meta-Agent-Workflow: Streamlining Tool Usage in LLMs through Workflow Construction, Retrieval, and Refinement," *WWW Companion 2025 - Companion Proc. ACM Web Conf. 2025*, 2025, [Online]. Available: https://dl.acm.org/doi/10.1145/3701716.3715247

[42] Yingxuan Yang, Huacan Chai, Shuai Shao, Yuanyi Song, Siyuan Qi, Renting Rui, Weinan Zhang, "AgentNet: Decentralized Evolutionary Coordination for LLM-based Multi-Agent Systems," *arXiv:2504.00587*, 2025, [Online]. Available: https://arxiv.org/abs/2504.00587