

DevOps AI: Intelligent Multi-Agent Platform

¹ Asst. Prof. Bhaghyashri B. Jadhav, ² Mr. Vansh Babashaheb Kirtishahi, ³ Ms. Gauri Shrikant Kulkarni,

⁴ Mr. Vignesh Deepak Patil, ⁵ Mr. Shreyash Ankush Rakhunde

Dr. D. Y. Patil Pratishthan's College of Engineering, Salokhenagar, Kolhapur

Department of Computer Science and Engineering (Data Science)

Corresponding Author: Asst. Prof. Bhaghyashri B. Jadhav

E-mail: bhagyadyp27@gmail.com

Abstract

Modern DevOps pipelines rely on a chain of specialised tools for version control, containerisation, deployment, monitoring, documentation and database administration. Each tool automates its own task, yet engineers still move between interfaces, remember tool-specific commands and stitch the workflow together by hand, which raises the risk of configuration mistakes and slows troubleshooting. This paper presents the design of a multi-agent DevOps platform in which a locally hosted large language model acts as a central orchestrator. A user submits a request in plain language through a web dashboard; the orchestrator interprets the intent, selects one of five specialised agents (Deployment, Docker, Monitoring, Documentation or Database), invokes the relevant tool through a restricted execution layer, and returns the outcome with an explanation. The platform is built on Ollama, LangGraph, Node.js, React, Docker, MongoDB, Redis, Prometheus and Grafana. Running the model locally keeps project data and execution context inside the organisation's own infrastructure. Operations that can change or destroy state, such as production deployment, rollback, container removal or database deletion, pass through a risk-evaluation step and require explicit human approval, and every action is written to an audit log. We report the Phase-I outcomes, namely the architecture, agent responsibilities, risk policy and workflow, and define an evaluation protocol covering task accuracy, deployment success, execution time, monitoring detection, documentation quality, recovery rate, resource use and human intervention. Quantitative results will follow the prototype evaluation. The work offers a reusable, extensible and locally deployable blueprint for conversational DevOps automation.

Keywords: AI agents, DevOps, large language models, multi-agent systems.

I. INTRODUCTION

A. Background

DevOps grew out of a simple observation: software reaches users faster and more reliably when the people who write it and the people who run it share tools, responsibilities and feedback. Over the past decade the idea has matured into a broad family of practices that includes continuous integration and delivery, configuration management, containerisation, centralised logging, monitoring, infrastructure as code and database change management [1]. Each practice is supported by dedicated tooling, and the combined effect is a software delivery process that is faster and more repeatable than the manual release procedures it replaced. Adoption has not been effortless, however. Systematic reviews have examined both the benefits of DevOps and the obstacles organisations meet while putting it into practice [2], [3].

One obstacle is the sheer number of moving parts. Even a modest web application deployed to the cloud depends on a chain of configured services, from source repositories and build servers to runtime environments [4]. A developer working on such a project may switch several times in one afternoon between a Git hosting service, a CI/CD dashboard, the Docker command line, a cloud console, a database client and a metrics panel. Every one of these tools has its own vocabulary, credentials, configuration syntax and failure modes. The automation each tool provides is valuable, but the coordination between tools is still done by people, and that is where time is lost and mistakes creep in. A mistyped environment variable, a container started with the wrong image tag or a deployment pushed to the wrong environment rarely comes from a lack of skill. It comes from the cognitive load of juggling many interfaces at once.

The problem is sharper for small teams and student projects, which seldom have a dedicated operations engineer. In such settings the same few people write the code, build the images, deploy the service, watch the dashboards and write the documentation. Reducing the overhead of these routine tasks would let them spend more time on the software itself. At the same time, the people in these settings are often the ones with the least experience of the toolchain, so an interface that explains what it is doing and asks before doing anything risky is more useful to them than one that simply hides complexity.

B. LLM-Based Agents as a Coordination Layer

Large language models (LLMs) offer a new way to attack this coordination problem. Because they can turn a loosely worded request into a structured plan, they can sit between the user and the toolchain. The idea of an agent, a software entity that perceives its environment, makes decisions and acts to meet goals, has a long history in computer science [17]. What is new is that LLMs can now serve as the reasoning core of such agents. Prompting strategies that interleave reasoning with actions allow a model to plan, call an external tool and revise its plan from the observation that comes back [18], and related

work shows that language models can learn to decide when and how to call external tools through programming interfaces [19]. These capabilities make an LLM a plausible DevOps coordinator, because DevOps work is essentially a sequence of tool calls in which the next step depends on the result of the previous one.

The same capability creates a hazard. When a model reads text from logs, repository files or database records, an attacker can plant instructions in that text and steer the model into actions the user never intended. This class of problem, known as indirect prompt injection, has been demonstrated against real applications that integrate LLMs with external data [20]. A platform that lets a model touch deployment pipelines and databases must therefore be designed on the assumption that model output and model input are both untrusted. This assumption shapes the architecture presented in this paper.

C. Related Work

Recent studies have begun to explore LLM-driven automation across the DevOps landscape, and it is useful to group them by the activity they address.

Infrastructure management. Yang et al. examined how LLM-based agents manage cloud infrastructure through four interfaces: software development kits, command-line tools, infrastructure-as-code platforms and web portals [5]. Their work frames the question of which interface an agent should use, and it shows that cloud operations are a realistic target for agentic automation. A related line of work treats the infrastructure code itself as the artefact the model produces, with generative AI acting as a copilot that writes and revises infrastructure-as-code across the DevSecOps lifecycle [12].

ChatOps. ChatOps brings operational commands into a conversational interface. One recent study uses an LLM-powered ChatOps approach to automate and monitor the deployment pipeline of a microservice application [6], and another proposes a low-code approach that composes ChatOps capabilities for microservice systems with the help of large language models [7]. Both demonstrate that natural-language access to pipelines is practical, but both are concerned mainly with microservice deployment and monitoring rather than with the wider set of tasks that surround it.

Multi-agent software engineering. Other researchers assign different lifecycle tasks to different agents. One framework builds on Semantic Kernel to coordinate agents across autonomous development lifecycles [8], while an agentic architecture for mobile DevOps spans activities from build prediction to incident resolution [9]. These works show that specialisation and coordination can be combined, although they are tied to specific frameworks or application domains.

Security of infrastructure code. A cluster of studies applies LLMs to security. An agentic workflow detects and remediates vulnerabilities in infrastructure-as-code [10], and a separate approach targets the detection of security smells in infrastructure-as-code configurations [11]. These studies are valuable in themselves, and they also remind us that automated changes to infrastructure carry security consequences.

Risks of LLM-generated and automated artefacts. A systematic review of the security of code generated by LLMs documents the need to assess such code before trusting it [13]. Work on GitHub Actions discusses how workflow misconfiguration creates security problems in CI/CD systems [14], and a systematic review of cybersecurity in DevOps environments surveys the wider landscape of threats [15]. Security has also been studied specifically for the monitoring phase of the DevOps cycle, where a hybrid intrusion-detection system combines ensemble learning with continuous monitoring [16].

D. Research Gap and Motivation

Taken together, the existing work establishes three points. LLM agents can operate infrastructure; conversational interfaces can drive pipelines; and LLM-generated actions must be treated with caution. What the literature offers less of is a unified design in which several specialised agents share context and cooperate across the everyday operations of a project, namely deploying, containerising, monitoring, documenting and inspecting data, under a single orchestrator and a single safety policy. Most of the studies above concentrate on one activity, whether that is pipeline generation, infrastructure provisioning, vulnerability remediation or lifecycle coordination.

Two further concerns motivate this work. The first is where the model runs. Sending deployment logs, configuration files and database details to an external AI service is not acceptable in every organisation, which makes a locally hosted model attractive even though it may be less capable than a large hosted one. The second is trust. A model that can execute commands can also execute the wrong command, so any practical design must limit what the model is allowed to do, record what it did and keep a person in control of consequential actions.

E. Objectives

The study pursues the following objectives:

- 1) To design a modular architecture in which an LLM-based orchestrator routes natural-language requests to specialised DevOps agents.
- 2) To define a controlled tool-execution layer that separates the model's reasoning from the execution of commands.
- 3) To specify a risk-classification and human-approval policy for operations that can change or destroy state.
- 4) To keep the AI component local, so that project data and execution context remain within the organisation's infrastructure.
- 5) To define a reproducible evaluation protocol with test cases and metrics for the prototype.

F. Contributions

This paper makes the following contributions:

- A layered architecture in which a central orchestrator, powered by a locally hosted LLM, routes natural-language requests to five specialised agents: Deployment, Docker, Monitoring, Documentation and Database.
- A controlled tool-execution layer that separates the model's reasoning from the actual execution of commands, using allow-listed, schema-validated operations.
- A three-tier risk policy with mandatory human approval for high-impact operations, together with an audit trail of every request, decision and result.
- An evaluation protocol with defined test cases and metrics, so that the prototype can be assessed reproducibly.

The study is guided by one research question: can an LLM-driven multi-agent architecture simplify and automate heterogeneous DevOps operations while preserving execution reliability, observability and human control? The remainder of the paper is organised as follows. Section II describes the methodology step by step, Section III reports the Phase-I results and the evaluation plan, Section IV discusses their implications and limitations, and Section V concludes with future directions.

II. METHODOLOGY

The work followed ten sequential steps, from identifying the problem to defining how the finished system will be tested. Each step feeds the next: the activities found in Subsection II-A determine the requirements of Subsection II-B, the requirements shape the architecture of Subsection II-C, and so on until the evaluation design of Subsection II-J closes the loop by describing how the result will be judged against the original requirements. Throughout, the description is of a design that is being implemented; statements about behaviour are statements of intent, to be verified in the evaluation phase.

A. Problem Identification

We began by listing the routine activities that force developers and operators to work across several tools: source-code management, CI/CD configuration, application deployment, Docker image creation, container management, system monitoring, log analysis, documentation, database operations and error diagnosis. The list was drawn from the DevOps literature discussed in Section I and from the day-to-day workflow of a typical web-application project. These ten activities define the functional scope of the platform.

The activities were then grouped by the kind of knowledge and the kind of tool each one needs. Deployment and CI/CD configuration depend on repository structure and hosting services; image and container work depends on the Docker engine; monitoring and log analysis depend on metric and log sources; documentation depends on a view of the whole project; and database operations depend on the data store and its access rules. This grouping produced the five agent domains used in the rest of the design. Table I shows how the activities map to agents.

TABLE I
MAPPING OF DEVOPS ACTIVITIES TO AGENTS

DevOps activity	Responsible agent
Source-code management, CI/CD configuration, application deployment	Deployment Agent
Docker image creation, container management	Docker Agent
System monitoring, log analysis, error diagnosis	Monitoring Agent
Documentation generation	Documentation Agent
Database operations	Database Agent

Error diagnosis is shared in practice, because a failing deployment may require evidence from logs, containers and the database. The orchestrator handles such cross-cutting requests by decomposing them into sub-tasks for several agents (Subsection II-D).

B. Requirement Analysis

Requirements were separated into functional and non-functional groups and written as testable statements, so that each one can later be checked against the prototype. Functionally, the platform must accept natural-language requests, identify the task, select an agent, execute authorised commands, manage containers, support deployment workflows, collect monitoring data, analyse logs, generate documentation, perform controlled database operations, show execution status and keep a history of past runs.

Non-functionally, it must be modular, so that one agent can be replaced without affecting the others; scalable, so that additional projects and agents can be supported; secure, so that the model cannot reach beyond its allowed operations; auditable, so that every action can be traced; simple to operate; capable of running its AI component locally; extensible, so that new agents can be added; and subject to human approval for sensitive actions. Of these, security, auditability and local execution received the most weight in later design decisions, because they address the trust and data-control concerns raised in Section I-D.

C. Layered Architecture and Technology Stack

The architecture has four layers, shown in Fig. 1. The user-interface layer is a React.js dashboard for project management, task entry, execution status, logs, monitoring views, generated documents and recommendations. The backend layer uses Node.js, Express.js, REST APIs and Socket.IO to handle authentication, project data, agent requests and real-time status updates. The AI orchestration layer holds the language model and the agent workflow and is described in Subsection II-D. The tool-execution layer wraps the external systems the agents may touch. Keeping these layers separate means that the interface, the model and the tools can each be replaced without rewriting the others.

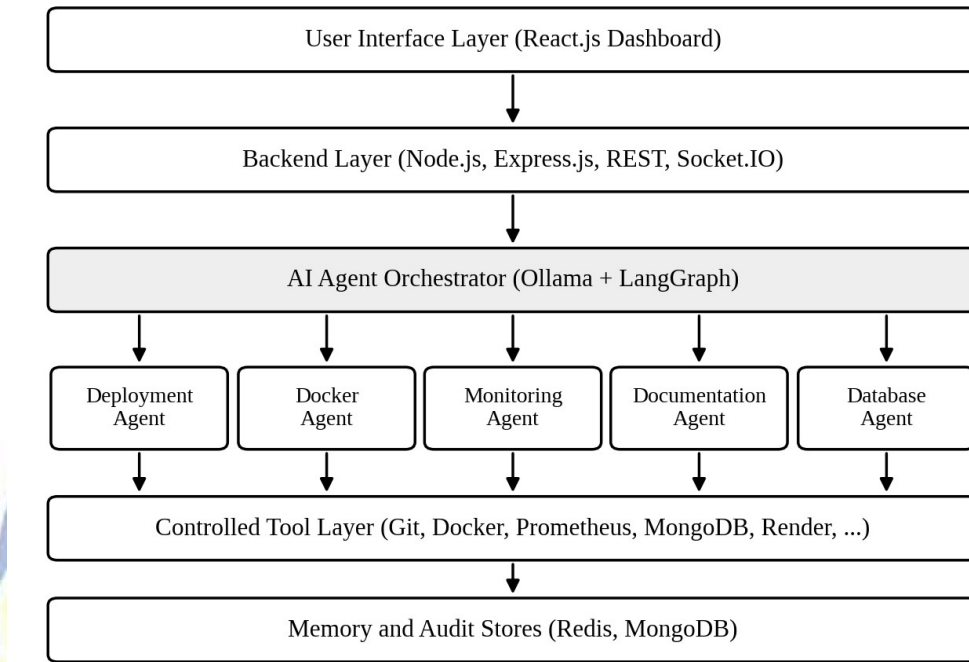


Fig. 1. Layered architecture of the proposed platform.

Table II lists the technologies chosen for each layer and the reason for each choice. Ollama was selected to serve open-weight models on local hardware, which supports the requirement for local AI execution. LangGraph was selected because it represents an agent workflow as a graph of states and transitions, which makes control flow explicit and inspectable. Node.js and React.js were chosen for a uniform JavaScript codebase across backend and interface; Docker, Prometheus and Grafana are widely used tools for containerisation and monitoring, which makes the results easier to reproduce elsewhere.

TABLE II
TECHNOLOGY STACK

Layer	Technology	Role
Interface	React.js	Dashboard, task entry, status and log display
Backend	Node.js, Express.js, Socket.IO	REST API, authentication, real-time updates
AI	Ollama, LangGraph	Local model serving; agent workflow as a state graph
Tools	Git/GitHub, Docker, Render, Linux CLI, Postman	Source control, containers, deployment, system operations, API testing
Monitoring	Prometheus, Grafana	Metric collection and visualisation
Data	MongoDB, Redis	Durable records; short-lived context and cache

D. Orchestrator Design

The orchestrator is the core of the platform. It accesses a local model through Ollama, and its workflow is expressed as a LangGraph state graph in which each node is a processing stage and each edge is a permitted transition. A shared state object travels through the graph (Table III), and each node reads from it and writes its result back. For every request the orchestrator carries out the following stages:

- 1) Interpret the request and extract the user's intent and any parameters (for example, the target environment or service

name).

- 2) Decompose the request into sub-tasks when it involves more than one operation.
- 3) Select the responsible agent and the tools it will need.
- 4) Retrieve context, such as project settings and earlier results, from the memory layer.
- 5) Generate a structured action proposal, not a free-form shell command.
- 6) Pass the proposal through the risk evaluator and, when required, wait for human approval.
- 7) Execute the approved action through the tool layer and collect its output.
- 8) Interpret the result, detect errors, verify the outcome and prepare a response for the dashboard.

TABLE III
ORCHESTRATOR STATE FIELDS

Field	Content
Request	Original text, user identity, project and timestamp
Intent	Task type and extracted parameters
Plan	Ordered list of sub-tasks and the agent assigned to each
Context	Project settings and earlier results retrieved from memory
Proposal	Structured action awaiting validation and risk evaluation
Risk tier	Low, medium or high, with the reason for the classification
Approval	Pending, granted or denied, with approver and time
Result	Tool output, verification outcome and any error
Retries	Number of attempts made for the current sub-task

Two design choices deserve explanation. First, the model is asked to produce structured output (an intent record or an action proposal) rather than prose or shell text. A structured record can be validated mechanically, which is much safer than parsing a command out of free text. Second, the graph is not a straight line. If a stage fails, for example because a proposal does not match its schema, the graph can route back to an earlier node and ask the model to try again, up to a configurable maximum number of retries. After that limit the request is escalated to the user with an explanation of what went wrong, instead of looping indefinitely. When a request spans several agents, the orchestrator runs the sub-tasks in the planned order and passes the output of one to the next through the shared state.

E. Specialised Agent Design

Each agent owns one domain and exposes a limited set of operations. Table IV summarises their responsibilities and the tools they use. The subsections that follow describe how each agent works.

TABLE IV
AGENT RESPONSIBILITIES AND ASSOCIATED TOOLS

Agent	Main responsibilities	Tools used
Deployment	Repository analysis, application detection, deployment configuration, build, deployment, health check, rollback support	Git/GitHub, Render, Linux CLI
Docker	Dockerfile generation and review, image build, container create/start/stop/restart, Docker Compose, health check	Docker, Linux CLI
Monitoring	CPU, memory, container status, service availability, application errors, log anomalies	Prometheus, Grafana, logs
Documentation	README, API, deployment and configuration documents, architecture descriptions, technical reports	Git/GitHub, Postman, MongoDB metadata
Database	Query execution, inspection, schema analysis, backup, restore, performance analysis, documentation	MongoDB

Deployment Agent

The Deployment Agent follows a fixed sequence: analyse the repository, detect the application type, prepare the deployment configuration, build, deploy, run a health check and return the result. Repository analysis looks at manifest files and directory structure to decide, for instance, whether the project is a Node.js service or a static site. The agent then prepares the matching build and start commands and environment settings. After deployment it calls the service's health endpoint and reports success or the failure evidence. Because production deployment and rollback change a live system, both are placed in the high-risk tier (Subsection II-H).

Docker Agent

The Docker Agent follows a parallel sequence from Dockerfile to image, container and health check. It can generate a

Dockerfile for a detected application, or review an existing one and suggest changes such as a smaller base image or the removal of unnecessary build tools. It builds and tags images, creates containers and starts, stops and restarts them. For multi-container projects it manages Docker Compose definitions. Removing a container is treated as a high-risk operation, whereas building an image or starting a container is medium risk because both can be reversed.

Monitoring Agent

The Monitoring Agent does not only fetch numbers; it combines metrics and logs so that it can answer diagnostic questions such as why CPU usage is high. It queries Prometheus for CPU, memory, container status and service availability, reads recent application logs, and passes both to the model for interpretation. The model's answer is expected to cite the evidence it used, for example the time window and the metric, so that the user can check the explanation against the Grafana dashboard. All monitoring operations are read-only and fall in the low-risk tier.

Documentation Agent

The Documentation Agent builds documents from actual project state, which helps keep them current. Its outputs include README files, API documentation, deployment guides, architecture descriptions, configuration documents and technical reports. It reads the repository, the stored project metadata and, where available, API collections, and drafts text for the user to review. Generated documents are saved as drafts and are not committed to the repository without confirmation.

Database Agent

The Database Agent treats every statement as potentially state-changing until it has been classified. It supports inspection, schema analysis, query execution, backup, restore, performance analysis and documentation of the data model. Read-only queries run in the low-risk tier. Operations that write, restore or delete data are high-risk and need explicit approval. Query results returned to the model are limited in size and are treated as data, not as instructions.

F. Controlled Tool Execution

Agents never run arbitrary commands. They request named operations from the tool layer, and each operation has a declared schema, an allow-list of permitted arguments and a fixed execution wrapper. Fig. 2 shows an example of the structured proposal that the model must produce. The tool layer checks the proposal against the schema for the named operation. A proposal that names an unknown operation, includes an argument outside the allow-list or carries a value of the wrong type is rejected before it reaches any tool, and the rejection is fed back to the orchestrator as an error.

```
{
  "agent": "docker",
  "operation": "build_image",
  "args": {
    "context": "./app",
    "tag": "app:1.0"
  },
  "target_env": "staging"
}
```

Fig. 2. Example of a structured action proposal.

Several further safeguards apply. Credentials are held by the tool layer, not placed in prompts, so the model never sees secrets. Each operation runs through a fixed wrapper that builds the actual command from validated fields; the model never supplies a command string. Where possible, commands run with the least privilege needed, and file-system access is limited to the project workspace. Text that enters the model from logs, repository files or database content is treated as data, never as instructions, which reduces the exposure to indirect prompt injection [20]. This separation of reasoning from execution is the main safeguard against the well-documented risk that LLM-generated code and commands can be insecure [13] and against the workflow misconfiguration and credential-exposure problems reported for CI/CD systems such as GitHub Actions [14], [15].

G. Context and Memory

Two stores give the orchestrator continuity. Redis holds short-lived material: session context, intermediate results and cached responses. Its fast access suits data that is needed during a single multi-step operation and can be discarded afterwards. MongoDB holds durable records: users, projects, agent configurations, execution history, logs, task records and documentation metadata. Together they let a multi-step operation, such as build, then deploy, then verify, carry its context from one stage to the next.

When a new request arrives, the context-retrieval stage loads only what is relevant: the settings of the current project, the outcome of the last few related tasks and any open approval requests. Limiting the retrieved context serves two purposes. It keeps the prompt within the context window of a locally hosted model, which is usually smaller than that of a large hosted one, and it reduces the amount of project information exposed to the model on each call.

H. Risk Classification and Human Approval

Not every action deserves the same level of caution. The risk evaluator places each proposed action into one of three tiers (Table V). Read-only actions run automatically, reversible changes run after a lightweight confirmation, and high-impact

actions cannot run without explicit approval from an authorised user. The classification does not depend on the model's own opinion of how risky an action is. It uses fixed rules over the operation name, the target environment and the type of resource affected, so that the same operation always receives the same tier. An operation aimed at a production environment is promoted by one tier, which is why a deployment to staging is medium risk and a deployment to production is high risk.

TABLE V
RISK TIERS AND APPROVAL POLICY

Tier	Typical operations	Policy
Low	Status queries, log and metric reads, documentation generation, read-only database queries	Execute automatically; log the action
Medium	Image build, container start/restart, staging deployment, Dockerfile changes	Execute after user confirmation; log the action
High	Production deployment, rollback, container removal, database write/restore/delete, infrastructure modification	Block until explicit approval; verify result; log the action

For high-risk actions the workflow is: the model proposes the action, the risk evaluator classifies it, the dashboard shows the exact operation, target and expected effect to an authorised user, the user approves or denies, and only then does the tool layer execute. After execution the orchestrator verifies the result, for instance by running a health check after a deployment or confirming that a container is no longer present after removal. A denied request is recorded and returned to the user with the reason, and the model is not allowed to resubmit the same action unchanged.

I. Audit Logging

Every request produces a record that captures who asked, what was asked, which agent was chosen, what action was proposed, how it was classified, whether approval was given, which tool ran, what it returned and how the outcome was verified. Table VI lists the fields. These records are stored in MongoDB and surfaced in the dashboard, which makes each decision traceable after the fact. They also serve a second purpose in the evaluation: the same records provide the raw data for computing task accuracy, execution time and the number of human interventions.

TABLE VI
AUDIT RECORD FIELDS

Field	Description
Requester	User identity and project
Request text	The natural-language request as submitted
Agent and operation	Selected agent and the named operation
Proposal	Structured action, including arguments
Risk tier	Assigned tier and the rule that produced it
Approval	Decision, approver and timestamps
Execution	Tool invoked, start and end time, exit status
Verification	Result of the post-execution check

J. Evaluation Design

To judge the platform fairly, we defined the evaluation before the experiments, so that the measurements cannot be shaped by the results.

Test environment

A test environment containing a sample web application, a MongoDB instance, a Docker host and a Prometheus–Grafana stack will be prepared. The sample application will be small but realistic, with a front end, an API and a database, so that deployment, container, monitoring and database tasks all have something meaningful to act on. The language model will run on the same class of hardware that a small team could afford, and the hardware specification will be reported so that latency figures can be interpreted.

Test suite and procedure

A suite of natural-language requests will be run against the environment, beginning with the five cases listed in Table VIII and extended with variations in wording, so that the evaluation measures understanding of intent rather than memorisation of a fixed phrase. Each request will be repeated several times because LLM output varies between runs. For each run we will record whether the correct agent and operation were selected, whether the task finished, how long it took, how many times a person had to intervene, and the CPU and memory consumed by the model and the platform.

Scoring and analysis

A task counts as correct only if the right agent and operation were chosen and the final state of the environment matches the expected state, which is checked by a script, not by reading the model's own report. Documentation quality will be rated by several human reviewers using a fixed rubric covering accuracy, completeness and clarity, and agreement between reviewers will be reported. Results will be summarised as means with standard deviations or confidence intervals across the repeated runs.

Safety tests

In addition to ordinary tasks, the suite will include cases built to stress the safety design: ambiguous requests, requests for destructive operations that must trigger approval, malformed model output that must be rejected by schema validation, and logs or documents containing injected instructions that must not be followed.

The metrics and the test cases are given in Section III. Fig. 3 summarises the end-to-end workflow that results from these steps.

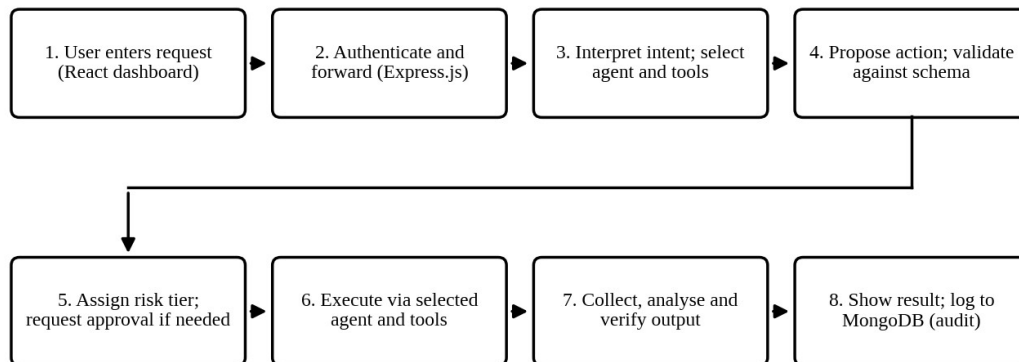


Fig. 3. End-to-end execution workflow.

III. RESULTS

At the time of writing the work is at the prototype stage. We therefore report what Phase-I has produced and the protocol by which the system will be measured. We do not report accuracy, latency or success-rate figures, because these have not yet been measured and any such numbers would be unsupported.

A. Phase-I Outcomes

Phase-I produced five concrete outcomes. First, a modular architecture with a central orchestrator and five specialised agents (Subsections II-C to II-E). Second, a common workflow that takes a natural-language request through interpretation, agent selection, tool execution, result analysis and dashboard reporting (Fig. 3). Third, a controlled tool-execution design that keeps reasoning separate from execution (Subsection II-F). Fourth, a three-tier risk and approval policy with audit logging (Subsections II-H to II-I). Fifth, a defined technology stack: Ollama for local model serving, LangGraph for agent orchestration, Node.js and Express.js for backend services, React.js for the interface, and Docker, MongoDB, Redis, Prometheus and Grafana for supporting infrastructure.

B. Evaluation Metrics

Table VII lists the metrics that will be collected in the experimental phase. The “Measured value” column is intentionally empty and will be completed with data from the test runs.

TABLE VII
EVALUATION METRICS

Metric	Definition	Measured value
Task execution accuracy	Correct tasks ÷ total tasks	[to be measured]
Agent-selection accuracy	Correctly routed requests ÷ total requests	[to be measured]
Deployment success rate	Successful deployments ÷ total deployments	[to be measured]
Docker task success	Successful container operations ÷ total	[to be measured]
Monitoring detection rate	Correctly identified events ÷ injected events	[to be measured]
Database operation success	Successful operations ÷ total	[to be measured]

Metric	Definition	Measured value
Documentation quality	Mean reviewer score on fixed rubric	[to be measured]
Average execution time	Mean seconds per task	[to be measured]
LLM response latency	Mean model response time	[to be measured]
Human interventions	Manual interventions per task	[to be measured]
Error recovery rate	Failures recovered ÷ total failures	[to be measured]
Resource consumption	CPU and RAM of model and platform	[to be measured]

C. Planned Test Cases

Table VIII lists the initial test suite.

TABLE VIII
PLANNED TEST CASES AND EXPECTED FLOW

ID	Request	Expected flow
T1	"Deploy the latest version of the application."	Repository → build → deployment → health check → result
T2	"Build the Docker image and start the container."	Dockerfile → build → image → container → health check
T3	"Why is the application showing high CPU usage?"	Metrics → Monitoring Agent → analysis → explanation
T4	"Generate deployment documentation."	Project information → Documentation Agent → document
T5	"Check the database status and show recent errors."	Database Agent → query → analysis → result

The suite will later be extended with negative cases: ambiguous requests, requests for destructive operations (which must trigger approval), malformed model output, and logs that contain injected instructions.

IV. DISCUSSION

The Phase-I results show that the main architectural problem, the fragmentation of DevOps work across many tools, can be addressed by a thin conversational layer that does not replace those tools. The platform's value is not the language model on its own. It comes from combining the model's interpretation of intent with specialised agents, controlled tool execution, shared context, monitoring, human approval and result analysis. Keeping each responsibility in its own agent also means that deployment logic, container handling, monitoring and documentation can change independently.

The design is consistent with directions in recent work. Yang et al. show that LLM agents can operate infrastructure through several interfaces [5], ChatOps studies show that natural-language access to pipelines and monitoring is practical [6], [7], and multi-agent studies show that specialised agents can be coordinated across lifecycle activities [8], [9]. The present design differs in bringing deployment, containers, monitoring, documentation and database work under one orchestrator with one risk policy, and in using a locally hosted model. The comparison is architectural only; without measurements we cannot claim that the platform performs better than any of these systems.

A. Security Implications

Security shaped the design more than any other concern. A command written by a language model cannot be assumed safe. Studies of LLM-generated code report insecure output [13], and work on GitHub Actions describes how workflow misconfiguration can expose credentials or allow command injection [14]. The design answers these risks in layers: structured proposals instead of raw commands, allow-listed operations, credentials withheld from the model, untrusted text treated as data, risk tiers with human approval and a complete audit trail. These measures reduce, but do not remove, the risk. Their effectiveness has to be tested with the negative cases listed in Section III-C.

B. Limitations

- **No experimental data yet.** The claims in this paper concern design; reliability, latency and accuracy remain to be measured.
- **Model capability.** Locally hosted models are usually smaller than hosted commercial models, so intent recognition and multi-step planning may be weaker, and response time depends on the available hardware.
- **Approval fatigue.** If too many actions require confirmation, users may approve without reading. The tier boundaries will need tuning against real usage.
- **Scope of tools.** The prototype targets a specific tool set (GitHub, Docker, MongoDB, Prometheus, Grafana, Render). Generalisation to other stacks is not yet demonstrated.
- **Run-to-run variation.** LLM output is non-deterministic, so the evaluation must use repeated runs rather than single observations.

C. Expected Benefits

If the evaluation confirms the design, the platform should offer a single point of interaction for common operations, less manual coordination between tools, a shared view of execution and monitoring data, straightforward addition of new agents, local control of project data and a traceable record of every action.

V. CONCLUSION

This paper presented the design of an LLM-driven multi-agent platform for DevOps operations. A central orchestrator, backed by a locally hosted model, interprets natural-language requests and routes them to Deployment, Docker, Monitoring, Documentation and Database agents. These agents act only through a controlled tool layer, while a three-tier risk policy, human approval for high-impact actions and a full audit log keep the automation accountable.

Phase-I established the architecture, agent responsibilities, workflow, risk policy, technology stack and evaluation protocol. It did not yet produce performance data, and the paper makes no claims beyond that. The next phase will implement the orchestrator and the five agents, integrate the tools, and run the test suite to measure task accuracy, deployment reliability, monitoring detection, documentation quality, latency, resource use and the amount of human intervention needed.

Future Scope

- A Security/DevSecOps Agent for dependency and container scanning, secret detection, infrastructure-as-code analysis, compliance checks and CI/CD security review, building on recent LLM-based work on infrastructure-code security [10]–[12].
- Anomaly-aware monitoring that combines metric analysis with intrusion- and anomaly-detection techniques studied for the continuous-monitoring phase of DevOps [16].
- Comparison of several local models, and of local versus hosted models, in terms of accuracy, latency and cost.
- Support for additional platforms such as Kubernetes and major cloud providers, and a study with practising engineers to measure usability and trust.

REFERENCES

- [1] Authors as listed in publisher record, “Mapping DevOps Capabilities to the Software Life Cycle: A Systematic Literature Review”, Information and Software Technology, 2025. [Open \(title search link\)](#) [\[verify authors/DOI\]](#)
- [2] Authors as listed in publisher record, “DevOps Adoption in Software Development Organizations: A Systematic Literature Review”, Proc. 4th Int. Conf. Advanced Research in Computing (ICARC), 2024. [Open \(title search link\)](#) [\[verify authors/DOI\]](#)
- [3] Authors as listed in publisher record, “Challenges Affecting the Successful Adoption of DevOps Practices: A Systematic Literature Review”, Proc. 4th Int. Conf. Advanced Research in Computing (ICARC), 2024. [Open \(title search link\)](#) [\[verify authors/DOI\]](#)
- [4] Authors as listed in publisher record, “Setting Up a CI/CD Pipeline in the Cloud for a Web Application”, Int. Conf. Expert Clouds and Applications (ICOECA), 2024. [Open \(title search link\)](#) [\[verify authors/DOI\]](#)
- [5] Z. Yang, A. Bhatnagar, Y. Qiu, T. Miao, P. T. J. Kon, Y. Xiao, Y. Huang, M. Casado, and A. Chen, “Cloud Infrastructure Management in the Age of AI Agents”, ACM SIGOPS Operating Systems Review, vol. 59, no. 2, pp. 1–8, 2025. <https://doi.org/10.1145/3759441.3759443>
- [6] Authors as listed in publisher record, “LLM-Powered ChatOps Approach to Microservice Deployment Pipeline Automation and Monitoring”, IEEE Int. Conf. E-Business Engineering (ICEBE), 2025. [Open \(title search link\)](#) [\[verify authors/DOI\]](#)
- [7] S.-K. Wang, S.-P. Ma, G.-H. Lai, and C.-H. Chao, “ChatOps for Microservice Systems: A Low-Code Approach Using Service Composition and Large Language Models”, Future Generation Computer Systems, vol. 161, pp. 518–530, 2024. <https://doi.org/10.1016/j.future.2024.07.029>
- [8] Authors as listed in publisher record, “Semantic Kernel for Software Engineering: A Multi-Agent Framework for Autonomous Development Lifecycles”, Int. Conf. Electronics, Communication and Aerospace Technology (ICECA), 2025. [Open \(title search link\)](#) [\[verify authors/DOI\]](#)
- [9] Authors as listed in publisher record, “ARIA: An Agentic Release Intelligence Architecture for Autonomous End-to-End Mobile DevOps—From Build Prediction to Incident Resolution”, Int. Conf. Artificial Intelligence, Systems, and Emerging Technologies, 2026. [Open \(title search link\)](#) [\[verify authors/DOI\]](#)
- [10] D. Toprani and V. K. Madisetti, “LLM Agentic Workflow for Automated Vulnerability Detection and Remediation in Infrastructure-as-Code”, IEEE Access, vol. 13, pp. 69175–69181, 2025. <https://doi.org/10.1109/ACCESS.2025.3560911>
- [11] Authors as listed in publisher record, “SecLLM: Enhancing Security Smell Detection in IaC With Large Language Models”, IEEE Access, 2025. [Open \(title search link\)](#) [\[verify authors/DOI\]](#)
- [12] Authors as listed in publisher record, “Generative AI as an Infrastructure Copilot: Automating Infrastructure-as-Code Across the DevSecOps Lifecycle”, Automated Software Engineering, 2026. [Open \(title search link\)](#) [\[verify authors/DOI\]](#)
- [13] Authors as listed in publisher record, “State of the Art of the Security of Code Generated by LLMs: A Systematic Literature Review”, Int. Conf. Software Engineering Research and Innovation (CONISOFT), 2024. [Open \(title search link\)](#) [\[verify authors/DOI\]](#)
- [14] Authors as listed in publisher record, “Mitigating Security Issues in GitHub Actions”, IEEE/ACM Int. Workshop on Engineering and Cybersecurity of Critical Systems and Software Vulnerability, 2024. [Open \(title search link\)](#) [\[verify authors/DOI\]](#)
- [15] Authors as listed in publisher record, “Cybersecurity in DevOps Environments: A Systematic Literature Review”, IEEE Access, 2025. [Open \(title search link\)](#) [\[verify authors/DOI\]](#)
- [16] Authors as listed in publisher record, “Enhancing DevOps Continuous Monitoring Phase: Hybrid Intrusion Detection and Ensemble Learning System (HIDEELS)”, IEEE Access, 2026. [Open \(title search link\)](#) [\[verify authors/DOI\]](#)

- [17] M. Wooldridge and N. R. Jennings, “**Intelligent Agents: Theory and Practice**”, The Knowledge Engineering Review, vol. 10, no. 2, pp. 115–152, 1995. <https://doi.org/10.1017/S0269888900008122>
- [18] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafraan, K. Narasimhan, and Y. Cao, “**ReAct: Synergizing Reasoning and Acting in Language Models**”, Proc. Int. Conf. Learning Representations (ICLR), 2023. <https://arxiv.org/abs/2210.03629>
- [19] T. Schick, J. Dwivedi-Yu, R. Dessi, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom, “**Toolformer: Language Models Can Teach Themselves to Use Tools**”, Advances in Neural Information Processing Systems (NeurIPS), 2023. <https://arxiv.org/abs/2302.04761>
- [20] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz, “**Not What You’ve Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection**”, Proc. ACM Workshop on Artificial Intelligence and Security (AISec), 2023. <https://arxiv.org/abs/2302.12173>

