Journal Publication of International Research for Engineering and Management (JOIREM)
Volume: 04 Issue: 11 | Nov-2026
ISSN (O) 3107-6696

DevOps Automation Platform for Deployment, Monitoring and Self-Healing of CloudNative Web Applications

Pratik Hanvate[1], Sumit Hate[2], Harsh Khadatkar[3], Rohit Butle[4], Roshan Nimje [5] Mohini Nikam[6]
12,34,5.6Students at Department of Computer Engineering, Bapurao Deshmukh College of Engineering, Sevagram, Wardha, Maharashtra, India, and carried out this work as their final-year B.Tech mega project under the guidance of **Prof. F. N. Mawale** *(Project Guide) and* **Dr. S. W. Mohod** *(Head of Department).*

----------***----------

Abstract - Modern software teams must build, test, deploy and monitor applications reliably while reducing repetitive manual work. Manual or partially automated deployment workflows lead to inconsistent environments, slower releases, delayed detection of failures and recovery that depends on human intervention. Existing research and practice address continuous integration and delivery, containerization, orchestration, infrastructure automation and monitoring largely as separate topics; however, the value of each depends on how well they are integrated into one verifiable workflow. This paper presents a comprehensive literature review and the architectural proposition for a DevOps Automation Platform for Deployment, Monitoring and Self-Healing of Cloud-Native Web Applications. The proposed workflow connects Git and GitHub for version control, GitHub Actions for automated validation, build and gated deployment, Docker for versioned container packaging, Terraform-based Infrastructure as Code and Amazon Web Services (AWS) for repeatable cloud provisioning, and Kubernetes for orchestration. Health endpoints and liveness and readiness probes provide monitoring, and a bounded self-healing mechanism restarts or replaces unhealthy workloads and escalates to manual review or rollback when its retry limit is exhausted. The paper details the six-layer architecture, the 8-stage automated delivery workflow, the bounded recovery algorithm, security considerations and six evaluation metrics: pipeline success rate, build duration, deployment duration, mean recovery time, recovery success rate and manual intervention count. At the 50% completion stage, the scope, architecture, module design and validation plan are defined, while repository setup, the CI/CD workflow, Docker packaging and monitoring configuration are in progress. No measured performance results are claimed; they will be reported only after end-to-end pipeline runs and controlled failure tests are executed and recorded. Future work focuses on completing integration, controlled failure injection and evidence-based evaluation. INDEX TERMS DevOps, CI/CD, GitHub Actions, Docker, Kubernetes, AWS, Terraform, Infrastructure as Code, monitoring, self-healing, cloud-native applications, literature review.

INDEX TERMS- DevOps, CI/CD, GitHub Actions, Docker, Kubernetes, AWS, Terraform, Infrastructure as Code, monitoring, self-healing, cloud-native applications, literature review.

I. INTRODUCTION

Releasing software quickly and reliably is now a core requirement for any web service. Cloud computing [1] supplies on-demand infrastructure, but it does not by itself make delivery reliable. In a traditional workflow a developer builds an application locally, copies files to a server, configures the environment by hand and deploys it with little automated validation. This approach is slow, hard to reproduce and prone to differences between development and production environments. DevOps addresses these problems by combining development and operations practices to improve collaboration, delivery speed, repeatability and service reliability [2]-[4].

Conventional deployment practice shows four practical limitations. First, manual or partially automated steps introduce configuration errors and make releases inconsistent. Second, tools such as version control, build servers, containers and cloud consoles are often used in isolation, so a change is not traced automatically from commit to running service. Third, failures are noticed late because health visibility depends on people watching dashboards. Fourth, recovery from a failed workload usually needs manual intervention, which increases downtime. Continuous integration and delivery (CI/CD) automate the build and release path [5], microservice and cloud-native architectures make frequent deployment practical [6], containers give a portable artifact [7], and orchestrators restart failed workloads [8], [9]; yet the literature rarely shows these elements working together as one evidence-backed workflow.

To address this gap, this paper proposes a DevOps Automation Platform for cloud-native web applications. A code change in a GitHub repository triggers a GitHub Actions workflow that validates the change, builds a versioned Docker image and deploys it to an AWS-based Kubernetes environment whose infrastructure is described with Terraform. Health endpoints and Kubernetes probes monitor the deployed workload, and a bounded self-healing mechanism restarts or replaces an unhealthy workload and records the event. Crucially, recovery is limited by a retry threshold and escalates to manual review, so automation stays controlled and observable.

The primary contributions of this work are:
1) An 8-stage automated delivery workflow that unifies commit, validation, build, scanning, deployment, monitoring, recovery and evaluation.
2) A six-layer architecture that connects GitHub, GitHub Actions, Docker, Terraform, AWS and Kubernetes into one traceable pipeline.
3) A bounded self-healing algorithm that limits automated retries and escalates to manual review or rollback.
4) A security and reliability design based on least privilege, protected secrets, reviewed infrastructure plans and pinned dependencies.
5) Six measurable evaluation metrics and seven validation tests (T1-T7) defined before implementation is complete.
6) A critical literature review that identifies the integration and evidence gap addressed by the platform.

The remainder of this paper is organized as follows: Section II reviews related literature. Section III details the 8-stage workflow, system architecture, security considerations and evaluation metrics. Section IV reports the current development status. Section V discusses implications and technical limitations. Finally, Section VI outlines future work.

II. LITERATURE REVIEW

This section reviews the academic and practitioner literature that underpins the conceptual and technical structuring of the proposed platform. The landscape of DevOps, containerization, infrastructure automation and reliability engineering is vast, spanning publications across IEEE, ACM, Elsevier and practitioner sources. The discussion is organized thematically to contextualize why the technological and architectural choices were made.

II-A. CI/CD AND DEVOPS PRACTICE

Continuous integration and delivery automate the integration, validation and release of application changes. Humble and Farley [2] describe the deployment pipeline as an automated path from commit to release, in which every change passes through build, test and deployment stages. Shahin et al. [5] provide a systematic review of CI, delivery and deployment approaches, tools, challenges and practices, concluding that automation brings earlier feedback and more frequent releases but that pipelines depend on meaningful tests, reliable infrastructure and careful management of deployment gates.

The wider DevOps literature links these practices to organizational performance. Leite et al. [4] survey DevOps concepts and challenges, and Forsgren et al. [3] show statistically that delivery speed and stability improve together when teams automate and measure their pipelines. Balalaie et al. [6] report how a cloud-native, microservice-oriented architecture enables DevOps adoption, since small independently deployable units are easier to build, test and release automatically. These studies motivate the choice of GitHub Actions as the workflow engine, triggered by push and pull-request events, and the decision to define explicit metrics instead of assuming that automation improves delivery.

II-B. CONTAINERIZATION AND ORCHESTRATION

Containers package an application with its runtime dependencies into a portable image. Merkel [7] explains how Docker reduces differences between environments by letting one artifact move unchanged from build to deployment. Image size, base-image trust and vulnerability scanning remain concerns, which is why the proposed pipeline includes an optional scan stage before an image is published.

Kubernetes grew out of Google's experience with Borg and Omega [8], [9]. It provides scheduling, service discovery, rolling updates and restart behaviour for containerized workloads. Liveness probes decide whether a container should be restarted, and readiness probes decide whether it can receive traffic. Verma et al. [9] note that automatic restart and rescheduling of failed tasks is central to running services reliably at scale. At the same time, incorrect probes or resource settings can themselves cause instability, and a systematic mapping of microservice research [15] stresses that servicelevel health checking is an architectural concern, not an afterthought.

II-C. INFRASTRUCTURE AS CODE

Infrastructure as Code (IaC) defines cloud resources declaratively so that provisioning is repeatable, reviewable and version-controlled [10]. Terraform produces an execution plan that can be inspected before changes are applied. Rahman et al. [11] map IaC research and highlight recurring quality and security problems such as hard-coded secrets, over-broad permissions and unreviewed changes. These findings directly shape the platform's security design: plans are reviewed before application, credentials come from protected secret stores or role-based identities, and permissions follow the principle of least privilege.

II-D. MONITORING, SELF-HEALING AND RELIABILITY

The Site Reliability Engineering literature [12] stresses that monitoring is useful only when signals are meaningful and alerts are actionable; collecting telemetry without an incident response plan does not improve reliability. Self-healing has its roots in autonomic computing, where Kephart and Chess [13] describe a closed loop of monitoring, analysing, planning and executing corrective actions. In practice a self-healing system restarts a process or replaces an unhealthy workload in response to a known failure condition. Automated recovery must be bounded, observable and tested, otherwise repeated restarts may hide

Journal Publication of International Research for Engineering and Management (JOIREM)
Volume: 04 Issue: 11 | Nov-2026
ISSN (O) 3107-6696

a deeper defect. Basiri et al. [14] describe chaos engineering, the practice of injecting controlled failures to verify that a system tolerates them. The proposed platform adopts the same idea in a modest form: safe, controlled failures are injected and the detection time, recovery action and time to healthy state are recorded.

II-E. SYNTHESIS AND RESEARCH GAP IDENTIFICATION

Synthesizing the literature across these domains reveals a highly specific, actionable research void: the intersection of end-to-end workflow integration, bounded self-healing, and reproducible evidence. Existing studies and tools are either strong on one stage of the lifecycle (pipelines, containers, orchestration, IaC or monitoring) or descriptive rather than evidence-based. Docker does not manage recovery, a pipeline does not observe a running service, and monitoring does not repair it. The proposed platform targets this exact niche: it demonstrates the delivery lifecycle as one connected workflow and evaluates it through reproducible test cases, defined metrics and recorded evidence. Table I summarizes the reviewed literature and its relevance.

TABLE I. Summary of Reviewed Literature and Relevance to the Proposed Platform

Reference	Year	Methodology / Technology	Application	Key Contribution	Relevance to Proposed Platform
Humble and Farley [2]	2010	Deployment pipeline; automated build, test, release	Release engineering	Defines commit-to-release automation	Basis for the proposed CI/CD workflow.
Forsgren et al. [3]	2018	Survey-based statistical study	DevOps performance	Links automation and fast feedback to delivery performance	Motivates measured evaluation metrics.
Leite et al. [4]	2019	Literature survey	DevOps concepts	Maps DevOps practices and open challenges	Frames collaboration and automation scope.
Shahin et al. [5]	2017	Systematic review	CI/CD	Reviews approaches, tools, challenges and practices	Pipeline reliability concerns addressed by gated deployment.
Balalaie et al. [6]	2016	Experience report; microservice migration	Cloud-native migration	Shows cloud-native architecture enabling DevOps	Supports the cloud-native deployment target.
Merkel [7]	2014	Docker containers	Application packaging	Portable, consistent runtime artifact	Basis of the container build stage.
Burns et al. [8]	2016	Borg, Omega and Kubernetes lessons	Container orchestration	Design lessons for container management	Basis for probes, replicas and restarts.
Verma et al. [9]	2015	Borg cluster manager	Large-scale cluster management	Automatic restart and rescheduling of failed tasks	Supports workload self-recovery design.
Morris [10]	2016	IaC practice	Infrastructure provisioning	Declarative, repeatable infrastructure	Basis for Terraform provisioning.
Rahman et al. [11]	2019	Systematic mapping	IaC research	Maps IaC themes, quality and security concerns	Motivates reviewed plans and least privilege.
Beyer et al. [12]	2016	SRE practice	Reliability and monitoring	Actionable alerts and reliability targets	Guides health checks and alert thresholds.
Kephart and Chess [13]	2003	Autonomic computing	Self-managing systems	Monitor, analyse, plan, execute loop	Conceptual basis of the recovery loop.
Basiri et al. [14]	2016	Chaos engineering	Resilience testing	Controlled failure experiments	Basis for controlled failure injection tests.
Pahl and Jamshidi [15]	2016	Systematic mapping	Microservices	Characterizes microservice architecture research	Context for service-level health checks.
Proposed platform	2026	GitHub Actions, Docker, Terraform, AWS, Kubernetes	Cloud-native web application delivery	Integrated build, deploy, monitor and bounded recover workflow	Evidence-based end-to-end validation.

Activate Win
Go to Settings to

III. MATERIALS AND METHODS

management Infrastructure provisioning IaC research Reliability and monitoring Self-managing systems Microservices Cloud-native web application delivery This section describes the core methodology, extensively breaking down the 8-stage automated delivery workflow, and outlines the system architecture, security considerations and evaluation metrics required to execute the platform. The project follows an iterative approach in which each component is configured and checked individually before being integrated into the complete workflow.

III-A. THE 8-STAGE AUTOMATED DELIVERY WORKFLOW

To minimize manual effort and release errors, the platform does not rely on a developer performing deployment steps by hand. Instead, it uses a structured, sequential 8-stage workflow that is triggered by a source-code change, validates it, and delivers it to the target environment with traceability at each stage.

III-A1. Stage 1: Source Commit and Trigger

A developer commits code or opens a pull request in the GitHub repository. The event triggers the CI/CD workflow for that source revision R. Version control provides change history, review and a traceable identifier that is later reused as the image tag.

III-A2. Stage 2: Validation

The workflow checks out revision R, installs dependencies and runs the configured linting, unit tests and validation checks. If any check fails, the pipeline stops before deployment, the failure log is recorded and the developer notified. This validation gate prevents a faulty revision from reaching the runtime.

III-A3. Stage 3: Image Build and Tagging

After successful validation, the application and its dependencies are built into a Docker image using the repository Dockerfile. A traceable version tag derived from the source revision is applied so that every running container can be linked back to a commit and a pipeline run.

III-A4. Stage 4: Scan and Publish

The image may optionally be scanned for known vulnerabilities and is then published to the selected container registry, or passed directly to the deployment stage. Pinned or controlled action versions and trusted base images reduce supply-chain risk.

III-A5. Stage 5: Deployment

The deployment stage updates the target AWS and Kubernetes environment using the deployment manifests. Infrastructure is provisioned with Terraform, whose plan is reviewed before it is applied. Deployment proceeds only after all required validation steps have succeeded.

III-A6. Stage 6: Health Monitoring

Once deployed, the application is observed through its health endpoint, Kubernetes liveness and readiness probes, container status, logs and resource metrics. A deployment is considered successful only after the first healthy response is recorded.

III-A7. Stage 7: Bounded Self-Healing

If a probe fails, the failure event is recorded and an alert is raised. While the attempt count is within the retry limit N, a

Journal Publication of International Research for Engineering and Management (JOIREM)
Volume: 04 Issue: 11 | Nov-2026
ISSN (O) 3107-6696

restart or workload replacement is triggered and health is rechecked. When the limit is exhausted, the deployment is marked unhealthy and escalated for manual review or rollback. This bounded loop prevents endless restarts from masking a deeper defect.

III-A8. Stage 8: Logging, Alerting and Evaluation

Timestamps of failure detection and recovery are written to an audit log. Pipeline run history, build and deployment durations and recovery timestamps are collected so that the metrics in Section III-D can be computed from actual run data rather than assumed.

III-B. SYSTEM ARCHITECTURE AND TECHNOLOGIES

The architecture of the proposed platform relies on automation-first, cloud-native technologies, ensuring a repeatable and traceable path from commit to running service. As the platform represents a complex amalgamation of modern tooling, the distinct roles of GitHub, GitHub Actions, Docker, Terraform, AWS and Kubernetes are explicitly defined [16]-[20]. Figure 1 logically maps the six layers of the application.

III-B1. Source Control Layer (Git and GitHub)

Git and GitHub provide change history, collaboration, pull request review and workflow triggering. Application code, Dockerfiles, Kubernetes manifests, Terraform files and workflow definitions are all version-controlled, so that configuration is documented and reproducible. Secrets are never committed; they are stored in the platform's protected secret store.

III-B2. CI/CD Automation Layer (GitHub Actions)

GitHub Actions [16] runs workflow jobs on push and pullrequest events. A job sequence typically comprises checkout, dependency installation, linting or testing, image build, optional image scan and gated deployment. Job permissions are restricted to the minimum needed, and third-party actions are pinned to controlled versions.

III-B3. Containerization Layer (Docker)

Docker [17] builds a container image from the application and its dependencies. The image is tagged with a version derived from the source revision and stored in a registry before deployment. The same image runs in every environment, which removes the environment-specific differences that cause failures in manual deployment.

III-B4. Infrastructure and Orchestration Layer (Terraform, AWS and Kubernetes)

Terraform [19] describes the required AWS [20] resources declaratively and produces a plan for review before changes are applied, supporting repeatable provisioning. Kubernetes [18] manifests define workload replicas, ports, resource requests and health probes, and handle rolling updates and restarts. The exact deployment mechanism depends on the selected target environment.

III-B5. Monitoring and Self-Healing Layer

Monitoring uses application health endpoints, probes, container status, logs and resource metrics. Recovery actions are linked to clearly defined conditions and produce evidence in logs or events. Algorithm 1 demonstrates the conceptual workflow of the bounded recovery cycle, which intercepts repeated failures and escalates them instead of retrying indefinitely.

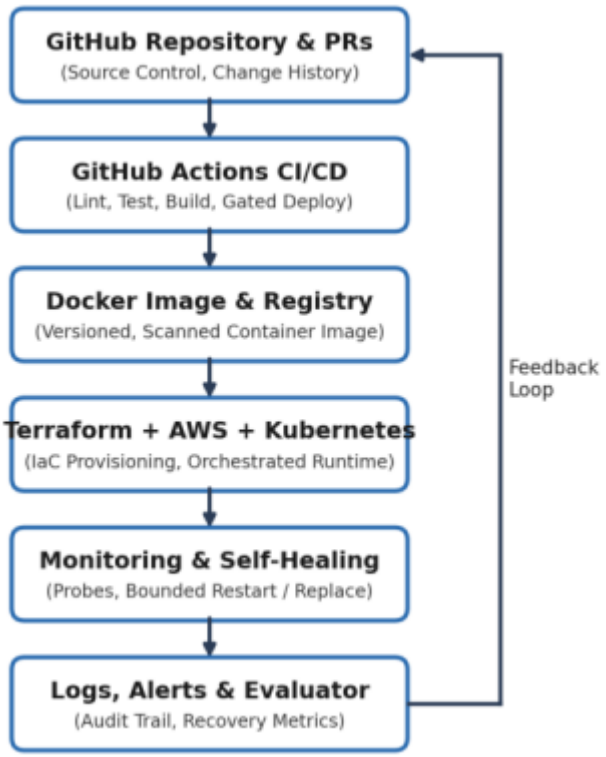


Fig. 1. System Architecture mapping the flow from the GitHub repository through the CI/CD pipeline, the Docker image and registry, the Terraform/AWS/Kubernetes runtime and the monitoring and self-healing layer, down to logs, alerts and evaluation, with a feedback loop to the source repository.

Algorithm 1 Bounded Build, Deploy, Monitor and Recover Cycle

Require: Source revision R, configuration C, health policy H, retry limit N
Ensure: Deployed version with recorded health status and logs
1: attempts-0
2: Checks - RunValidation(R) {lint, unit tests}
3: if Checks failed then

4: Log failure; return ABORT {stop before deploy}
5: end if
6: Image - Build(R); Tag(Image, sha(R))
7: Publish(Image) {registry or deploy stage}
8: Deploy(Image, C)
9: while service running do
10: status- Probe(H.endpoint)
11: if status = healthy then continue
12: else
13: Log failure with timestamp; attempts - attempts +1
14: if attempts ≤N then
15: Restart or replace workload; log result
16: Recheck health; record recovery time
17: else
18: Mark deployment UNHEALTHY; raise alert
19: return ESCALATED {manual review / rollback}
20: end if
21: end if
22: end while
23: return Deployed

III-C. SECURITY AND RELIABILITY CONSIDERATIONS

Because the platform connects a code repository to cloud resources, security is treated as part of the design rather than as an add-on. The following measures are planned. Credentials are stored in protected secrets or role-based identity mechanisms and are never committed to the repository. CI/CD workflows and cloud resources follow the principle of least privilege, with proper IAM permissions and controlled network configuration on AWS. Terraform plans are reviewed before changes are applied to prevent unintended modification. Action versions are pinned, trusted base images are used where practical and images may be scanned for known vulnerabilities. Automated recovery loops are limited and every action is logged. Finally, no availability or performance improvement is claimed without repeatable measurements.

III-D. PERFORMANCE EVALUATION METRICS

The platform will be evaluated with the following metrics, computed from actual run data once end-to-end validation is complete. They are defined now so that they can be applied consistently.

1) Pipeline Success Rate measures the proportion of successful workflow runs:

$S = (R/R) \times 100\%$ (1)

where R, is the number of successful workflow runs and R is the total number of runs in the GitHub Actions run history.

2) Build and Deployment Duration measure the time needed to deliver a version:

$D = \text{thealthy start}$ (2)

where tstart is the start of the deployment stage and thealthy is the time of the first successful health check. Build duration is measured similarly from workflow start and completion timestamps.

3) Mean Recovery Time measures how quickly service is restored after an injected failure:

$MTTR = (1/n)\ \Sigma\ (trec.i- tfail,i)$ (3)

where trailu is the injection time of the i-th controlled failure, trec,i is the time at which the service is healthy again, and n is the number of failure tests.

4) Recovery Success Rate measures the proportion of failures recovered automatically:

$Q = (Nrec/Ninj) \times 100\%$ (4)

where Nrec is the number of successful automatic recoveries and Ninj is the number of controlled failures injected. In addition, the **manual intervention count** records the number of steps that require a person during a normal release, using a documented release checklist.

IV. RESULTS AND CURRENT DEVELOPMENT STATUS

As a mega-project currently in its active development lifecycle, this section outlines the achieved architectural milestones, development status and the planned validation of the platform. Quantitative results from complete pipeline execution are slated for the next phase.

IV-A. ARCHITECTURAL DESIGN AND CONFIGURATION STATUS

The 50% milestone represents completed project planning, requirement analysis, system design and architecture definition, together with progress toward implementing and integrating the major components. The project topic, problem statement and objectives are defined, and the six-layer architecture and module breakdown are designed. The GitHub repository and source-control workflow, the CI/CD workflow definition and Docker packaging are in progress. Cloud deployment configuration and Terraform infrastructure configuration are planned or in progress, and monitoring and health checks have a defined design with implementation in progress. Table II summarizes the status of every work item.

TABLE II. Progress Status of Work Items at the 50% Milestone

Work Item	Status
Topic, problem statement, objectives	Defined
Architecture and module breakdown	Designed
GitHub repository and source control	In progress
CI/CD workflow definition	In progress
Docker packaging	In progress
Cloud deployment configuration	Planned / in progress
Terraform configuration	Planned / in progress
Monitoring and health checks	Design defined; in progress
Self-healing failure scenarios	Design defined; testing pending
End-to-end validation and metrics	Pending

IV-B. PLANNED VALIDATION TESTS

Seven validation tests have been defined, each with an expected behaviour and the evidence to be captured. **T1** (valid code change) expects the CI workflow to complete all required checks. **T2** (invalid code or failing test) expects the pipeline to stop before deployment. **T3** (container build) expects an image with a traceable tag. **T4** (deployment health) expects the application to answer the configured health check. **T5** (application becomes unhealthy) expects monitoring to detect the failure. **T6** (controlled workload failure) expects the supported restart or recovery action to restore service, with before and after timestamps. **T7** (credential and configuration check) expects that secrets are not exposed in logs or in the repository.

IV-C. PENDING INTEGRATION

While the individual components are being configured and checked in isolation, the seamless, uninterrupted flow from commit to verified deployment is still being completed. Specifically, the successful CI/CD workflow run on the group's own repository, the image build and run consistency check, the deployment configuration for the selected AWS or Kubernetes environment, and the monitoring thresholds and notifications remain to be finalized.

No measured results from actual pipeline execution, deployment, monitoring and self-healing tests are available yet, and no numerical performance improvement is claimed. Screenshots and logs from real executions will be captured, and the metrics in Section III-D will be computed from the recorded run data, before any comparison with manual deployment is made.

V. DISCUSSION

V-A. PLATFORM ADVANTAGES AND ACADEMIC POSITIONING

The proposed platform addresses several critical gaps identified in contemporary DevOps practice. By connecting source control,validation, packaging, deployment, monitoring and recovery, it makes every running version traceable to a commit, a pipeline run and an image tag. Containerization removes environment differences between build and deployment, and Infrastructure as Code makes the target environment reproducible and reviewable.

Furthermore, placing a validation gate before deployment and a retry limit on recovery keeps automation controlled instead of unrestricted. Compared with manual deployment, the workflow reduces repetitive steps; compared with using individual tools in isolation, it defines in advance how the connection will be evidenced through test cases and metrics. The platform builds on widely used, documented technologies [16]-[20] rather than proposing new ones, so its value lies in integration and verification.

V-B. LIMITATIONS

Despite the robust architectural proposition, the platform's current design and developmental state exhibit several acknowledged technical limitations:

1) Work in Progress: The complete CI/CD pipeline and its components may not yet be fully operational in the target environment, and some proposed functions require further implementation, integration and testing.

2) Tool and Environment Dependency: The system depends on the availability and correct configuration of GitHub Actions, Docker, Kubernetes, AWS and Terraform. Cloud resource limits, configuration errors, network issues and software compatibility may affect deployment and testing.

3) Limited Self-Healing Scope: Monitoring and selfhealing are limited to the checks and recovery actions that are configured and verified. Advanced features such as automated rollback, intelligent failure prediction and comprehensive security monitoring require further development.

4) Evaluation Pending: Production-level scalability, high availability and recovery performance have not been established. The evaluation is limited to the test cases and environment available for the project.

VI. CONCLUSION AND FUTURE WORK

This paper has detailed the architecture, literature review and conceptual framework of a DevOps Automation Platform for Deployment, Monitoring and Self-Healing of CloudNative Web Applications. Motivated by the risks of manual deployment and the isolation of individual DevOps tools, the platform connects Git and GitHub, GitHub Actions, Docker, Terraform, AWS and Kubernetes into one traceable workflow with health monitoring and bounded self-healing.

By executing a carefully designed 8-stage workflow, the platform aims to replace error-prone manual steps with validated, repeatable automation, while a retry limit and escalation path keep recovery controlled and observable. The security design and the six evaluation metrics ensure that results can be checked rather than assumed.

Moving forward, the work will be centered on completing end-to-end integration, finalizing deployment and monitoring configuration, and implementing and demonstrating at least one controlled self-healing scenario. The metrics will then be computed from actual run data. Longer-term extensions include security scanning and software supply-chain checks, progressive delivery such as canary or blue-green releases, centralized logs and dashboards, automated rollback with alerting, infrastructure

drift checks and policy validation, and evaluation of deployment and recovery performance under repeatable test conditions.

ACKNOWLEDGMENT

The authors express sincere gratitude to Prof. F. N. Mawale (Project Guide) and Dr. S. W. Mohod (Head of Department), Department of Computer Engineering, Bapurao Deshmukh College of Engineering, Sevagram, Wardha, Maharashtra, India, and to Prof. M. N. Thakare (Principal) for their continuous academic guidance, technical feedback and the facilities provided for the conceptualization and development of this final-year B.Tech mega project.

REFERENCES

[1] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," National Institute of Standards and Technology, Gaithersburg, MD, USA, Spec. Publ. 800-145, 2011.

[2] J. Humble and D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Boston, MA, USA: Addison-Wesley, 2010.

[3] N. Forsgren, J. Humble, and G. Kim, Accelerate: The Science of Lean Software and DevOps. Portland, OR, USA: IT Revolution Press, 2018.

[4] L. Leite, C. Rocha, F. Kon, D. Milojicic, and P. Meirelles, "A Survey of DevOps Concepts and Challenges," ACM Comput. Surv., vol. 52, no. 6, Art. no. 127, 2019.

[5] M. Shahin, M. A. Babar, and L. Zhu, "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices," IEEE Access, vol. 5, pp. 3909-3943, 2017.

[6] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture," IEEE Softw., vol. 33, no. 3, pp. 42-52, May/Jun. 2016.

[7] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," Linux J., vol. 2014, no. 239, Art. no. 2, 2014.

[8] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," Commun. ACM, vol. 59, no. 5, pp. 50-57, 2016.

[9] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in Proc. 10th Eur. Conf. Comput. Syst. (EuroSys), Bordeaux, France, 2015, Art. no. 18.

[10] K. Morris, Infrastructure as Code: Managing Servers in the Cloud. Sebastopol, CA, USA: O'Reilly Media, 2016.

[11] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams, "A systematic mapping study of infrastructure as code research," Inf. Softw. Technol., vol. 108, pp. 65-77, 2019.

[12] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, Eds., Site Reliability Engineering: How Google Runs Production Systems. Sebastopol, CA, USA: O'Reilly Media, 2016.

[13] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," Computer, vol. 36, no. 1, pp. 41-50, Jan. 2003.

[14] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos Engineering," IEEE Softw., vol. 33, no. 3, pp. 35-41, May/Jun. 2016.

[15] C. Pahl and P. Jamshidi, "Microservices: A Systematic Mapping Study," in Proc. 6th Int. Conf. Cloud Comput. Serv. Sci. (CLOSER), Rome, Italy, 2016, pp. 137-146.

[16] GitHub, "GitHub Actions Documentation." [Online]. Available: https://docs.github.com/actions

[17] Docker Inc., "Docker Documentation." https://docs.docker.com/ [Online]. Available:

[18] The Kubernetes Authors, "Kubernetes Documentation." [Online]. Available: https://kubernetes.io/docs/

[19] HashiCorp, "Terraform Documentation." https://developer.hashicorp.com/terraform/docs [Online]. Available:

[20] Amazon Web Services, "AWS Documentation." [Online]. Available: https://docs.aws.amazon.com/