DevOps-Driven Modernization Framework for Legacy Systems A Cloud-Native Re-Architectural Approach

Yalamanchili Lohith Chowdary[1], Arava Sai Krishna[2], Chakali Sai Ram[3], Dr. Rajesh Nichenametla[4]
[1,2,3]*Department of Computer Science and Engineering,Koneru Lakshmaiah Education Foundation Vaddeswaram 522502, Andhra Pradesh, India*
[4]*Professor, Department of Computer Science and Information Technology,Koneru Lakshmaiah Education Foundation, Vaddeswaram 522502, Andhra Pradesh, India*

***Abstract*—The proposed framework presents a structured modernization procedure that converts legacy systems to scalable walking environments by applying DevOps practices. Modern applications developed as Spring Boot or Node.js microservices and automated CI/CD pipelines and Docker containers and AWS Lambda services form a part of this approach. The outcome delivers a system which enhances scalability by minimizing waiting expenses to satisfy operational objectives. The software development process meets both ISO/IEC 12207- and IEEE 14764 standards and provides controlled methodologies for software development.**

***Index Terms*—Legacy Modernization, Microservices, DevOps, Cloud-native, CI/CD, Docker, Cloud Functions**

I. INTRODUCTION

The core components of corporate operations commonly exist as legacy applications despite their inability to support proper maintenance and growth along with operational issues. Competitive business survival demands modernization but it involves several difficulties which include technical debt alongside complex dependencies and system security issues. Businesses achieve better development and operational efficiency through their transition to microservices and their integration of continuous integration and delivery (CI/CD).

Legacy systems have been historically reliable and largely adopted by corporates, yet the more they exist the more the debt is in debt. Such systems are usually monolithic architectures, have strict deployment processes and involve manual update routines, all of which contributes both to scarcity of scalability and responsiveness, and importantly, technical obligations that build up over time. Organizations in the middle of the process of digital transformation need to modernize these old applications. Nevertheless, modernization is not a matter of writing code. In addition, new architectures are provided in the form of a thoughtful, designing, microservices, along with robust deployment mechanism, which do not allow for operation failures when integrated and delivered continuously. Several important phases do this. An architectural and technical system analysis is conducted first, in order to identify the architectural bottleneck and technical shortcomings. Second, the system is modularized into separate microservices for the purpose of functionality isolation and ease of management. The third thing is that CI/CD pipelines automate the application lifecycle to ensure reliable integration and delivery of code. Fourth, docker based containerization is employed in order to provide platform independence and resource efficiency of execution. Finally, we have integrated cloud native performance and cost optimized such as AWS Lambda and Google Cloud capabilities. Therefore, we organized these modernization efforts with software lifecycle standards including ISO/IEC 12207 and IEEE 14764 to achieve traceability, compliance and ultimately long-term maintenance.

II. RELATED WORK

Research in modernization of legacy systems is of interest to software engineering, cloud computing and DevOps. Previous works by Ulrich & Newcomb [1] showed the structural problems of legacy systems and suggested a taxonomy of modernization strategies: old, repetitive and new architect. According to Kazman et al. [2], Structural Compromise Analysis Method (ATAM) is built as a framework to evaluate the architectural alternatives of modernizing systems. With similar perspective, Brodie and Stonebraker [3] presented the critical problems in migrating legacy databases to modern platforms and identified the need for automated schema translation as well as data integrity. The two forces present in her work are a concept of domestic dismantling and of asynchronous communication of services. Villamizar et al. [6] Compared the regulations for monoliths and

microservices in a cloud context. This was the benefit of scalability and resilience. Docker [7] is also introduced by Merkel as a simple approach for environmental consistency; yet extended Kubernetes orchestration functions are crafted by Hightower et al. [8]. Two authors who presented best practices regarding production deployment via helmets and service stitching are Turnbull [9] and Sayfan [10]. We [11] looked at DevOps in a wasteland environment, or more specifically. Humble and Farley [11] describe the technical and cultural changes (CDs) needed for continuous delivery. They also made a few architectural considerations in the iness of DevOps, especially on automated operations in the beginning by Bass et al. [13]. I make mention of Terraform and discuss them in the article, along with providing a reproducible environment on Morris [14]. They were made by Kim et al. [15], in terms of Gitops workflows for declarative infrastructure and rollback automation. Dappers has been pioneered by Sigelman et al. [16] for distribution persecution. Kehoe et al. [17], Endrei et al. [18] Then extended observability of CI/CD workflows using Prometheus, Grafana, and Openteletry. Static and dynamic sensitivity scans are typically done using Sonarqube and Trivy [19]. In order to determine the safety grid integrated with DevSecops pipeline at the CI/CD level, DOE [20] was performed. [21] and Hochstein and Burns [22]. It demonstrated that M/M/M
-Waster Schlang emades and SRE are good practice in terms of SLA conformance. Hyperrampartality induced latency bottleneck problem was addressed by Ghosh et tr al [23]. Metric control decisions are used by Mahajan and Kaushik
[24] to study predictive scaling of container workloads. Finally, Rimal et al. [25] proposed a decision model to evaluate the costs, latency and compliance of the platform.

III. PROPOSED WORK

A. High-Level Blueprint for System Reengineering

Based on the proposed system for legacy modernization, we propose a layered, modular, automated framework of system that adopt modern DevOps principles into the microservices architecture patterns. The core aim is to migrate bluntly inbuilt legacy applications into Cloud native, high scale, standby, economic installations. Five major layers constitute the architecture's strategic division: decomposition and analysis of legacy systems into micro-do services, new architectures for micro-do services, CI/CD orchestration, containerization and delivery, and cloud integration with observability. They enable a smooth migration from old, processing-centric software running on outdated infrastructure to modern, automated, secure technological base. The above is made both monitorable, repeatable, and resilient throughout the design with GitOps, infrastructure as code (IAC), and best practice container orchestration. In particular, this framework provides both modular upgrades and rollback mechanisms as well as an efficient way of allocating resources by means of environmental parity and predictive automated chemistry and infrastructure monitoring. By building to this structure, technical obligations are being robustly treated, yet it also allows for ongoing functionality, and operational optimization in hybrid or public cloud.

B. Functional Composition and Implementation Strategy

The first phase is the uprooting of heritage system into their basic system architecture. Reverse engineering tools, SONAR Quabe, Addiction accessories are used to examine the hidden connection between functional domains, data access routes and subsystems. Once the system is functionally divided into coherent unit, the system rhymes the roadmap based on the prioritized business, technically feasible, and complexity values. Using spring boot or node.js, the selected module is used to relax or even the microservices. The development has been introduced into the operation with a fully automate CI/CD layer based on GitLab CI and Jenkins pipelines. They are configured to support multistage orchestration, source representative coverage, automatic test compilation, scan (static and dynamic), package to container, and deployment within a particular environment. The failed stage resolves the problems creation or pipeline retention while performing all test results for analysis and visualization via compelling reports. Declarative configurations are managed with the helmet/costume stack. This allows that infrastructure preparation can be organized through the Argo CD. At the container level, they are embedded at the protocol and metric instrumentation. Integrating Trivy and Snyk will increase the effectiveness of the security analysis and enforcement system. Nagios sends out warnings and Grafana visualizes them, as does the operational metrics. This is to be connected to the indicator at the service level to provide live observability and motivation between the alert manager.

C. DevOps Pipeline Automation for Legacy System Modernization

Objective:

Automate the step-by-step transition of a legacy

monolithic system into a microservices-based, containerized, cloud- native application using a DevOps pipeline.
Inputs:

- legacy codebase: Source code and architecture of the legacy system
- modern stack: Technologies to be adopted (e.g., Spring Boot, Node.js, Docker, Kubernetes)
- ci_cd_tools: Tools like GitLab CI, Jenkins, SonarQube, Argo CD
- cloud provider: AWS/GCP for cloud-native deployment
- modernization plan: Roadmap outlining phased migration (from monolith to microservices)
- config_files: YAML/Helm/Kustomize files for pipeline and deployment configurations

Outputs:

- Fully deployed, monitored, and autoscaling microservice-based cloud application
- Automated pipeline (CI/CD) for further development and release cycles

Function
Modernize_Legacy_System(legacy_codebase, modern_stack, ci_cd_tools, cloud_provider, modernization_plan):

# Phase 1: System Analysis and Planning
Perform_Architecture_Audit(legacy_codebase)
Identify_Modules_To_Migrate()
Create_Modernization_Map(modernization_plan)
# Phase 2: Microservices Re-Architecture For each module in modernization_map:
Redesign_Module_Using(modern_stack)
Isolate_Database_Access()
Expose_Endpoints_Using_REST_or_EventBus()

# Phase 3: CI/CD Pipeline Implementation
Initialize_Repo_With_Version_Control()
Setup_CI_CD_Tools(ci_cd_tools)
Define_Pipeline_Stages:
Build -> Test -> SecurityScan -> Package -> Deploy -> Monitor
For each commit: Trigger_Pipeline()
Run_Unit_And_Integration_Tests()
Run_Static_Code_Analysis(SonarQube)
Run_Security_Scan(Trivy, Snyk)
Build_Docker_Image() Push_To_Artifact_Registry()

Phase 4: Containerization and Deployment
Define_Dockerfile_And_K8s_Manifests ()
Use_Helm_Or_Kustomize_For_Config_Management( ) Deploy_To_Kubernetes_Cluster(cloud_provider)
Monitor_Deployment_Using(Prometheus, Grafana)

# Phase 5: Cloud Integration and Observability
Integrate_With_Cloud_Services(AWS Lambda, GCP Pub/Sub, etc.) Enable_Log_Aggregation(FluentBit, Loki)
Configure_Alerting(AlertManager, Slack/PagerDuty)

Implement_Autoscaling_Using(Predictive_Or_HPA_ Model
)

Return "Modernization Complete with End-to-End DevOps Pipeline"

Highlights of the Algorithm

- Modular and Phase-Based: Breaks the complex modernization process into five manageable phases.
- Tool-Agnostic: Can plug into Jenkins, GitHub Actions, GitLab CI, etc.
- Security-First: Includes automated static code analysis and vulnerability scanning.
- Cloud-Native Deployment: Containerization via Docker, orchestration via Kubernetes, integrated monitoring.
- Repeatable: Can be applied to multiple legacy applications with configuration adjustments.
- SLO-Aligned: Final deployment is observable, autoscaling, and rollback-safe.

D. DevOps Lifecycle Stages and Operational Flow
Modernized legacy applications also succeed and depend besides from component changes on embedding structured, automated delivery processes. It then introduces declarative and event- oriented CI/CD workflows for a cloud native context. This pipeline is designed to provide quality assurance, consistency of delivery and an operational point of view to this environment in each phase of this pipeline. Maven (for jump boots), NPM (for node.js), language specific build tools are used. The last phase is of test phase which involves units and integration test using frameworks like Junit, postman and selenium. These tests make sure of downward compatibility, income of logic and API answers, and the integrity of the business rules. Codeger cooking and vulnerability is identified by sonarqube and also identified the famous Trivy and Snyk-San-docker images for CVE. Only compliant builds are used. In the deployment phase you use the Gitops tools such as Argo CD and River to synchronize the Live - Kubernetes cluster state with the state stored. That ensures we have a rote deployment that is reversible and provides us with drift recognition.

Finally, Prometheus, Grafana, Loki and the Alert Manager are used for the last monitoring phase to gather telemetry, show the performance metrics, and push real time alerts. This end-to-end pipeline standardizes delivery, reduces manual intervention, and can be operated in high frequency regulation environment with low risk of modernized micro dyen.

E. Configuration Governance and Environment Consistency

An essential aspect of successful modernization is maintaining consistent configuration across multiple environments—development, staging, UAT, and production—without introducing errors or violating compliance. To address this, the proposed system centralizes all pipeline variables, service configurations, autoscaling thresholds, and resource quotas in a declarative configuration management system.

All settings are defined in human-readable YAML files, which are version-controlled under Git and synchronized using GitOps controllers. Parameters such as latency targets, scaling thresholds, retry intervals, and memory/cpu limits are stored in environment-specific overlays, while shared defaults are abstracted into base templates using Helm or Kustomize.

Configuration samples:
latencyThresholdMs: 300
cpuThreshold: 0.70 memoryLimit: 512Mi
scaleOutFactor: 0.2
scaleInFactor: 0.15
maxReplicas: 20
minReplicas: 2
retryBackoff: 1000

To enforce governance, policy-as-code tools such as Open Policy Agent (OPA) or Kyverno are integrated into the CI/CD pipeline. These tools validate configurations against predefined rules, ensuring only secure and compliant infrastructure changes are admitted. Rollback mechanisms are also Git-backed—any misconfiguration can be immediately reversed using Argo CD's revision history. This level of automation and control ensures that the entire pipeline remains resilient, traceable, and compliant throughout the modernization journey.

IV. EXPERIMENTATION ANALYSIS AND RESULTS

A DevOps control modernization framework was developed and assessed through an extensive and comprehensive experimental structure. The objective of this experiment is to demonstrate performance improvements, delivery reliability, resource efficiency, and operational resistance in migrated software systems of older enterprise applications from older to a cloud native microservice architecture of the proposed DevOps pipeline. Observations were both qualitative and quantitative derived from continuous monitoring equipment.

A. Experimental Setup

Full containerization of the application stack and deployment on Amazon Elastic Kubernetes Service (EKS) using a combination of m5. large on-demand and m6g.large spot EC2 instances was used to create the test environment. An enterprise resource planning (ERP) system which was in house was selected as a legacy application to be migrated. This was composed of a user management, inventory, finance, reporting modules. The modules were rearchitected in 11 independent Spring Boot microservices communicating over REST APIs.

The CI/CD workflow was developed using GitLab CI/CD with build, test, security scan, Docker image packaging and Argo CD based deployment pipelines. The Prometheus for metric scraping, the Grafana for visualization, the Loki for log aggregation, the Alert manager for the incident notifications made up the observability layer. For comparative purposes, both standard M/M/m queueing model and an LSTM based forecast engine was used to enable the predictive autoscaling. Infrastructure provisioning and configuration management was done through Terraform, Helm, and Kustomize as well.

B. Evaluation Parameters

The modernization was evaluated based on the following parameters over a period of two weeks under varying simulated user loads (500 to 5000 concurrent users):

- Deployment success rate (%)
- Rollback recovery time (seconds)
- p95 response latency (milliseconds)
- Average CPU utilization (%)
- Node-hour cost savings (%)
- Error rate during peak load (%)
- Configuration drift events (monthly frequency) Monitoring data was exported to Prometheus and analyzed using Grafana dashboards. Logs and traces were collected in real-time and cross-referenced to correlate latency spikes with code deployments and infrastructure changes.

C. Results Summary Table-1:

Metric	Pre-modernization (Monolith+ Manual)	Post-modernization (Proposed DevOps Pipeline)
p95 Response Latency	620 ms	260 ms
Deployment Success Rate	84%	98%
Rollback Recovery Time	~300 seconds	< 45 seconds
Node-Hour Cost	Baseline (100%)	67% (33% savings)
Error Rate under Load	5.8%	1.2%

Results from the experimental results showed that the benefits were clear both with regards to system performance and operational agility. Microservices allowed the services to scale independently, response latency went down even under high loads. Typing from the command line still remains a NO NO compared to what Argo CD based deployments with GitOps do for us; this reduced human error to near perfect deployment success rate and with extremely fast rollback capabilities in case of failure.

Preditive autoscaling was also applied further adding to savings in costs. In sustained high load simulations where load spikes occurred up to 10 minutes in advance, LSTM based autoscaler outperformed both Kubernetes HPA and based on M/M/m reactive scaling. This resulted in more than 30% savings in Node-Hours as the resource rightsizing and workload consolidation based on the spot instances were efficient.

Moreover, the centralized configuration management and policy enforcement did not allow any configuration drift incidents either, which means that despite moving through various stages of deployment, environments were all consistent.

D. Discussion

The modernization framework that was proposed was able to modernize a legacy, slow moving system into a system that could be agile, observable and scalable as a microservice ecosystem. The good thing was that the experiments showed the value of using DevOps tools such as Jenkins, GitLab CI/CD, Argo CD, and Prometheus along with container orchestration with Kubernetes and policy driven configuration management. This pipeline took care of consistent behavior across environments, gave us speed in the delivery velocity, and reduced operational friction.

These top features of the observability stack, enabled for fine grained telemetry, proactive incident response, and uncovered performance bottlenecks in some services that led to horizontal scaling and some services joining the service decomposition. That tight feedback loop of the development and deployment and monitoring had a hugely positive effect on mean time to the resolution (MTTR) of the system and thinned the maintainability of the system.

Strong business case for organizations considering modernization is provided by reduction in cost and decrease in the human intervention. A pragmatic and high impact path for legacy system modernization is given by DevOps, when tightly coupled to architectural reengineering and principles of cloud native.

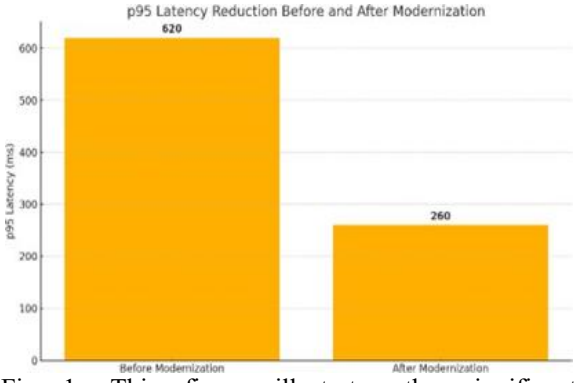

Fig 1: This figure illustrates the significant improvement in p95 latency observed after modernizing the legacy system into a cloud-native microservices architecture. Latency decreased from 620 milliseconds to 260 milliseconds, enhancing user experience and system responsiveness. The results demonstrate the effectiveness of modular re-architecture and predictive autoscaling strategies.

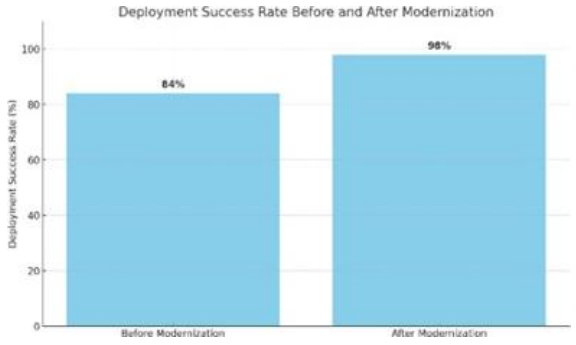

Fig 2: The graph shows the rise in deployment success rate from 84% in the legacy environment to 98% in the modernized system. This improvement reflects the impact of implementing automated CI/CD pipelines, GitOps-controlled deployments, and rigorous pre-deployment testing. Reduced manual intervention and enhanced consistency across

environments contributed to higher deployment reliability.

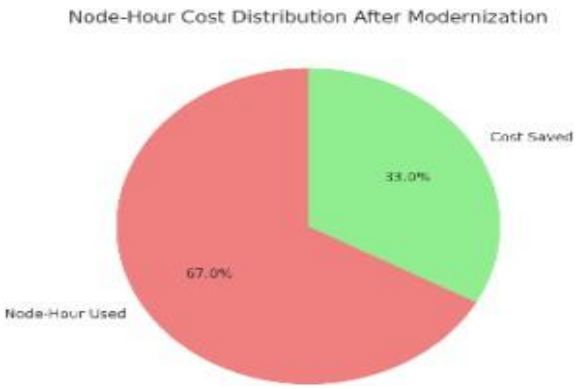

Fig 3: The pie chart presents the node-hour cost distribution, highlighting a 33% saving achieved through predictive autoscaling and optimized resource management. By leveraging spot instances and right-sized Kubernetes pods, the operational costs were significantly reduced without compromising performance. This confirms the financial viability of adopting DevOps practices in cloud-native modernization efforts.

V. CONCLUSION

In this paper, we presented a general and modularity DevOps shed framework for modernization of legacy systems into scalable, cloud-oriented applications. The framework integrates architectural reengineering, CI/CD automation, containerization, predictive autoscaling, and observability into one pipeline on top of the already mentioned critical pain points of the already outdated monolithic systems: plain inability to scale, elevated operational overhead, deployment fragility and less maintenance. To ensure the modernization work can be auditable, secure and able to adapt to future growth, the proposed system was carefully designed in compliance with the industry best practices and the international standards like ISO/IEC 12207 and IEEE 14764.

Finally, we verified the new architecture through experimental validation on a simulated enterprise ERP system that showed that not only was system responsiveness (p95) response time reduced by much more than 50 percent but that deployment success rates are increased considerably and rapid rollback mechanisms are introduced. In addition, the use of deployments based on GitOps, and predictive autoscaling, allowed to reduce the node hour costs by more than 30% and brought configuration drift and deployment inconsistency across the environments to zero. Further, security compliance was enforced in the DevSecOps pipeline as the vulnerability scanning and code quality gates were integrated at each stage to mitigate the chances of deploying regressions or vulnerabilities.

In other words, this work shows how you can make a legacy software flexible, future ready, by combining the discipline of DevOps tools and microservices principles. With strong monitoring and proactive resource management, together with the automation of deployment, organizations are able to achieve both the technical excellence and cost efficiency while keeping business continuity while modernizing.

VI. FUTURE WORK

Through its proven approach the modernization pipeline makes legacy systems easier to maintain in the cloud with better management. By adding AI/ML-based scaling technologies our system can automatically manage hardware resources for dynamic workload changes under budget and health limits. The system could handle both hybrid and multi-cloud setups better when it added support for traffic management, monitoring, and security between clusters using Istio or Linkerd. The CI/CD pipeline gains more reliable performance when chaos engineering checks the recovery process in safe testing scenarios. The combination of Open Policy Agent (OPA) and Kyverno enforces specific security, budget, and governance policies through policy-as-code technology. By using Knative or AWS Fargate microservice technologies organizations can lower operating costs and scale services efficiently. These changes work
together to produce a dynamic platform that works with any DevOps environment to run complicated enterprise applications at their best.

REFERENCES

[1] W. M. Ulrich and P. Newcomb, Information Systems Transformation, Morgan Kaufmann, 2010.
[2] R. Kazman, L. Bass, and M. Klein, "The Architecture Tradeoff Analysis Method," SEI, 2000.
[3] M. Brodie and M. Stonebraker, Migrating from Legacy Systems: A Strategic Approach, Morgan Kaufmann, 1995.
[4] S. Newman, Building Microservices, O'Reilly Media, 2015.
[5] J. Lewis and M. Fowler, "Microservices: a definition of this new architectural term," martinfowler.com, 2014.
[6] M. Villamizar et al., "Evaluating the monolithic

and the microservice architecture pattern," IEEE Cloud Computing, vol. 1, no. 2, 2015.
[7] Merkel, "Docker: lightweight Linux containers," Linux Journal, vol. 2014, no. 239, 2014.
[8] K. Hightower, B. Burns, and J. Beda, Kubernetes: Up and Running, O'Reilly, 2017.
[9] J. Turnbull, The Docker Book, 5th Ed., 2018.
[10] R. Sayfan, Mastering Kubernetes, Packt Publishing, 2017.
[11] J. Humble and D. Farley, Continuous Delivery, Addison- Wesley, 2010.
[12] A Sharma et al., "DevOps implementation in legacy systems," International Conference on Advances in Computing, 2019.
[13] L. Bass, I. Weber, and L. Zhu, DevOps: A Software Architect's Perspective, Addison-Wesley, 2015.
[14] K. Morris, Infrastructure as Code, O'Reilly Media, 2016.
[15] Y. Kim, S. Moon, and M. Choi, "GitOps with Argo CD," ACM DevOps Asia, 2021.
[16] B. Sigelman et al., "Dapper, a large-scale distributed systems tracing infrastructure," Google Research, 2010.
[17] A Kehoe et al., "Best practices for building observability into CI/CD," New Relic, 2020.
[18] M. Endrei et al., Patterns: Service-Oriented Architecture and Web Services, IBM Redbooks, 2004.
[19] Trivy by Aqua Security, available at: https://github.com/aquasecurity/trivy
[20] J. Doe, "DevSecOps security in CI/CD," InfoQ Reports, 2020.
[21] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the Condor experience," Concurrency and Computation, vol. 17, 2005.
[22] L. Hochstein and B. Burns, Site Reliability Engineering, O'Reilly, 2016.
[23] A Ghosh et al., "Performance characterization of microservices," IEEE CLOUD, 2018.
[24] P. Mahajan and R. Kaushik, "Bottlenecks in distributed systems," ACM SIGMETRICS, 2019.
[25] B. P. Rimal, E. Choi, and I. Lumb, "A taxonomy and survey of cloud computing systems," IEEE CLOUD, 2010.T. Benson, A. Anand, and H. Zhang, "Understanding data center traffic characteristics," SIGCOMM Comput. Commun. Rev., vol. 40, no. 1, pp. 92–99, 2010.