International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V6I1P116
Eureka Vision Publication | Volume 6, Issue 1, 144-147, 2025

Original Article

Transforming DevOps via Automated Environment Provisioning

Priyadarshini Jayakumar[1], Vipin Jose[2]
[1,2] Independent Researcher, USA.

Received On: 20/02/2025 ***Revised On: 08/03/2025*** ***Accepted On: 21/03/2025*** ***Published On: 24/03/2025***

Abstract *- This white paper examines how automated environment provisioning can enhance DevOps in large-scale organizations. By replacing manual processes with Infrastructure as Code (IaC) and Continuous Integration and Continuous Deployment - CI/CD pipelines, teams can rapidly and reliably create secure, scalable environments. Highlighting a US based Telecom giant's use of composable infrastructure, the paper demonstrates benefits such as sub-two-hour environment provisioning, high availability, and improved customer experiences. Automated provisioning emerges as a key enabler of speed, consistency, and innovation in modern DevOps practices.*

Keywords *- DevOps, Automated Environment Provisioning, Infrastructure as Code (IaC), Continuous Integration (CI), Continuous Deployment (CD), CI/CD Pipelines.*

1. Introduction

DevOps has revolutionized the way we approach software development and IT operations, emphasizing collaboration, continuous integration, and delivery. A critical aspect of this transformation is the ability to scale or descale environments on a need basis, which significantly enhances the efficiency and reliability of the software and its usage during peak traffic such as a holiday sales event. This white paper explores the impact of automated environment provisioning as a DevOps practice, illustrating how a completely automated, pipeline-based environment build can transform traditional IT models.

2. The Traditional Model

Traditionally, environment provisioning is often a separate manual, time-consuming process involving setting up of hardware, installing operating systems, configuring software, and ensuring that all dependencies are installed and configured. Development is often built and tested against a pre-built environment, and this leads to different behavior across the various environments, such as development, testing, staging, and production. This isolated approach is prone to errors, inconsistencies, and delays, leading to unstable environments for the code features and prolonged development cycles. Scaling of the environment is not often a development requirement since the functionality and environment configurations are supported by different teams and designed in isolation.

3. Automated Environment Provisioning Model

Features in development are built along with environment configurations and designed to work best not just in isolation but also with scaling, redundancy and failover as part of the requirement. Automated environment provisioning utilizes infrastructure as code (IaC) principles and continuous deployment pipelines (like Git) to create, configure, and manage environments programmatically. The entire environment, including its configuration, hereafter referred as *stack*, is defined in code and deployed using a pipeline. Each stage in the environment like security, networking, monitoring are handled as environment specific configurations applied to each feature revision. After completion of a SDLC cycle, this environment can then be automatically torn down releasing the infrastructure resources. This approach ensures that environments are consistent, reproducible, and scalable.

4. Setting up Automated Environment Provisioning

Transitioning from a traditional development model to an automated infrastructure and code build model requires continuous integration and delivery (CI/CD) where the software being built must meet core architectural guidelines such as deployability, modifiability, and testability and can be reliably integrated, tested and released through a pipeline to an environment automatically. It aims at building, testing, and releasing software at greater speed and frequency. Setting up the Environment might seem as an additional overhead during build; especially during development and unit testing and they can be developed as discreet components, where each of the sections are built separately during development but compiled and deployed together in higher environments.

5.1. High-Level Components of Automated Environment Provisioning Model

Automated environment provisioning relies on a set of integrated components that together enable consistent, reproducible, and scalable deployments. These components ensure infrastructure, code, and configurations are version-controlled, automated, and adaptable to ever changing business needs.

5.1.1. Repository

- Stores version-controlled artifacts needed to build the application.
- Separates feature-based environment configuration and baselined environment configurations for the various parts of the Software Development LifeCycle (SDLC)

5.1.2. Release Branch

- Manages different versions of files through branch labeling.
- Ensure that committed changes beyond a point-in-time label remain isolated.
- Enables simultaneous use of artifacts across multiple release-specific stacks.

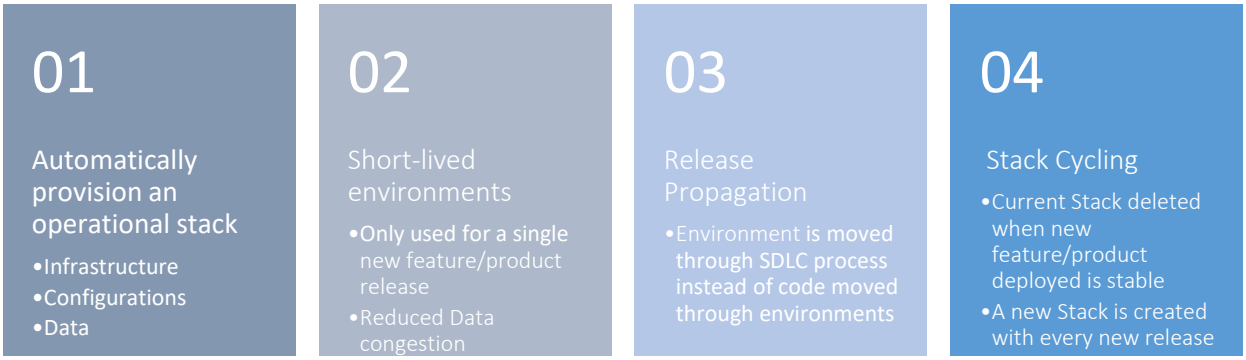


Figure 1. What is Automated Environment Provisioning

5.1.3. Infrastructure-as-Code (IaC)

- Defines deterministic provisioning of infrastructure (e.g., VMs, containers, databases).
- Tightly couples infrastructure definition with application requirements.
- Ensures all infrastructure is accessible and modifiable as part of a release branch.

5.1.4. Application Code

- Contains business logic, compiled into executable binaries.
- Runs on provisioned infrastructure created by Infrastructure as Code (IaC).
- Accessible and modifiable per release branch, allowing versioned deployment.

5.1.5. Configuration (Config)

- Stores persisted-runtime-settings affecting application behavior (e.g., feature toggles).
- Supports external configuration management for secrets and dynamic variables.
- Version-controlled alongside application and infra-artifacts.
- Updated endpoints and credential pointers for downstream connections (e.g.: - environments used by downstream applications)

5.1.6. Pipelines

- Static Code Analysis for feature code standards and Infrastructure as Code (IaC) standards
- Automates complex orchestration (infrastructure provisioning, builds, testing, deployment).
- Chains modular steps for CI/CD processes.
- Fully integrated with release branches for controlled automation.

5.1.7. Config Templates

- Allows scaling and reconfiguration of environments without redeploying infra.
- Provides reusable, preset templates to maintain consistency across stacks.

5.1.8. Testing Components

- Test Definitions: Provide deterministic pass/fail criteria for functional and performance validation (Scaling, Failover, etc.)
- Testing Framework: Executes structured tests (unit, mock, end-to-end, and sanity tests).

5.1.9. Orchestrator

- Automates release-specific lifecycle management:
  - ✓ Creates and deletes stacks.
  - ✓ Provisions infrastructure and deploys application code.
  - ✓ Updates config dynamically and applies templates.
  - ✓ Handle rollback gracefully, when required.

5.10. Stack & Stack Manager

- Stack: Runnable collection of provisioned infrastructure containing code and config.
- Stack Manager: Automates stack progression through release stages (development → canary → production and rollback).
- Monitors telemetry, testing outcomes, and live traffic percentage allocations.

5.11. Application Monitoring & Health Check

- Continuously validates live stack performance.
- Runs automated sanity checks and compares telemetry against baselines.
- Automatically routes or disables traffic to anomalous stacks.

5.12. Application Gateway

- Serves as a load balancer/proxy for routing traffic across stacks.
- Enables segment-based traffic control (blue/green, canary, full rollout).

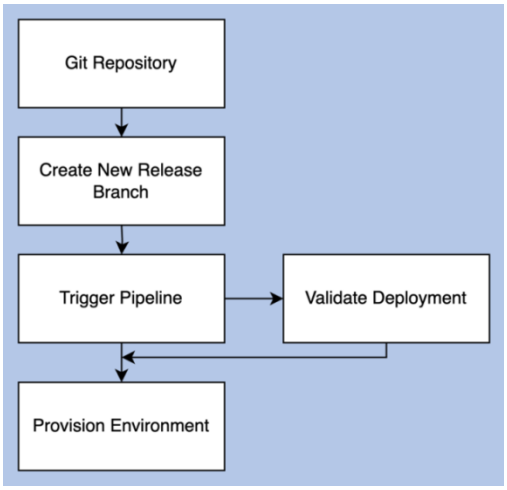


Figure 2. Automated Environment Provisioning - Pipeline Overview

5.2. Example set-up of a Pipeline

A pipeline can be created in GitLab https://gitlab.com/ and 2 modes of operation can be defined:

- CREATE_STACK - For creating a new environment.
- DESTROY_STACK - To destroy an existing environment.

And following variables should also be defined:

- MODE - The mode of operation - CREATE_STACK or DESTROY_STACK
- ENV - The type of environment i.e., non-production vs Production
- NAME - The name of the environment.
- The pipeline implementation properties can be defined in a yaml file, */gitlab-ci.yml* which defines the stages of the pipeline.
- The common activities to be done in a pipeline job can be described under /gitlab-ci/.common.gitlab-ci.yml
- Each environment specific jobs can be defined separately by creating a file /gitlab-ci/ .env.{env-type}.rules.gitlab-ci.yml where env type can be configured or defined.
- A new docker/cloud image can be created for this pipeline, to exclusively install any required software libraries.

5.3. Recommended Environment Progression

Below Diagram describes the ideal progression of a stack through stages from development to full production for maximum output efficiency.

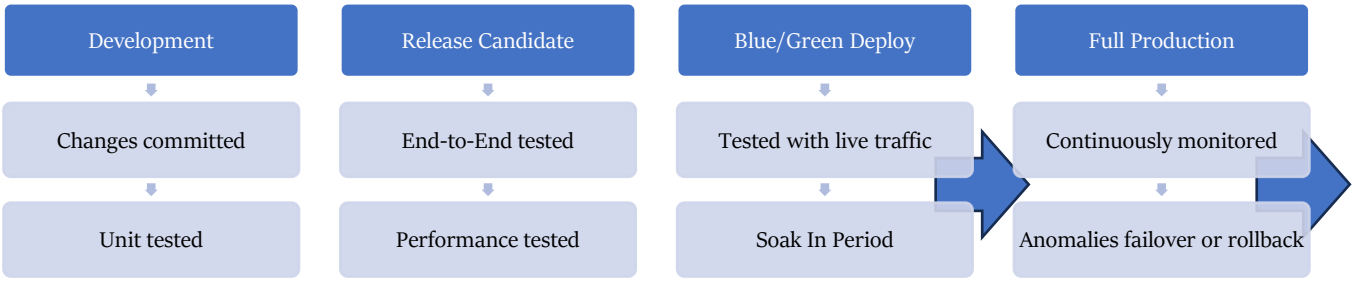


Figure 3. Example of a Stack Progression

5.4. Implementing Monitoring and Logging

Automated provisioning should be accompanied by robust monitoring and logging to track the health and performance of environments. Environment Metrics help identify and resolve issues quickly. Tools like Prometheus, Grafana, and ELK stack are examples of tools that can be integrated into the application. A US based telecom giant uses blue-green pipeline monitoring reports for tracking the successful stages in the pipeline for any given environment with the respective environment placeholder.

5.5. Ensuring Security and Compliance

Security practices should be integrated into the provisioning process, including the use of secure configurations, automated vulnerability scanning, and compliance checks. Provision to auto-rotate secrets per stack allows for the keys to be active only during the availability of the stack. This ensures that environments are secure and compliant with industry standards and regulations.

6. Benefits of Automated Environment Provisioning

6.1. Key Benefits include

- Consistency and Reproducibility: Automated provisioning ensures that stacks are built from the same configuration scripts, eliminating discrepancies between various stages of the SDLC.
- Speed and Efficiency: Features are tested in relation to a specific environment configuration version which reduces the risk of environment related defects and speeds up rollout to production.
- Scalability: Features are built to be scaled up or down based on demand, optimizing resource usage and cost.
- Security: Automated provisioning integrates security practices into the pipeline and thereby the stacks, ensuring that environments are compliant with security standards and policies. Also, secrets such as keys are created and destroyed per stack provisioned, providing automatic secret key rotations.

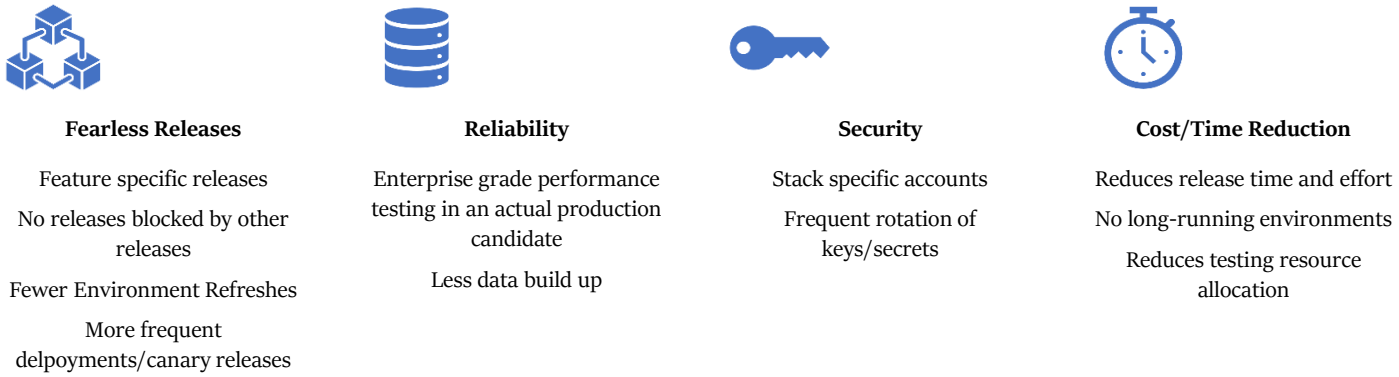


Figure 4. Key Components of Cloud Computing Infrastructure

7. Quantifiable Outcomes

By adopting automated environment provisioning powered by composable infrastructure, large enterprises can achieve remarkable outcomes. A US based telecom giant achieved.

- Rapid Environment Deployments: Fully functional, production-grade environments spun up in under two hours instead of days.
- 100% Availability: Ensured uninterrupted service delivery, even under high traffic conditions.
- Enhanced Customer Engagement: Improved experience and increased engagement driven by precise, targeted A/B testing.
- Environment Compliance: Keeping up with package updates and security patches are integrated into the SDLC and part of the feature rollout.

8. Conclusion

Automated environment provisioning is a game-changer for DevOps, enabling teams to build, manage, and scale environments efficiently and reliably. By leveraging pipeline-based environment builds, organizations can ensure that environments remain stable and secure throughout the SDLC process. This approach not only accelerates development cycles but also enhances the quality and security of the software being developed. The ability to tear-down environments after a specific SDLC cycle paves way to low-cost infrastructure set-up and maintenance. As organizations continue to adopt DevOps practices, automated environment provisioning will become an essential component of their strategy, driving innovation and excellence in software development and operations.

References

[1] HashiCorp, "*Terraform: Infrastructure as Code*. HashiCorp, 2021". [Online]. Available: https://www.terraform.io

[2] Amazon Web Services, "*AWS CloudFormation*". Amazon, 2021. [Online]. Available: https://aws.amazon.com/cloudformation

[3] Red Hat, "*Ansible: Provision Infrastructure*". Red Hat, 2021. [Online]. Available: https://www.ansible.com

[4] Jenkins Project, "*Jenkins Pipeline*". Jenkins, 2021. [Online]. Available: https://www.jenkins.io/doc/book/pipeline

[5] Prometheus Authors, "*Prometheus Monitoring*". Prometheus, 2021. [Online]. Available: https://prometheus.io

[6] T-Mobile, "*Autopilot" (Patent Pending)*. T-Mobile US, Inc., 2025.