IJIRCCE

e-ISSN: 2320-9801 | p-ISSN: 2320-9798

INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH

IN COMPUTER & COMMUNICATION ENGINEERING

Volume 10, Issue 11, November 2022

INTERNATIONAL STANDARD SERIAL NUMBER INDIA

Impact Factor: 8.165

International Journal of Innovative Research in Computer and Communication Engineering

| e-ISSN: 2 320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.165|| A Monthly Peer Reviewed & Referred Journal

|| Volume 10, Issue 11, November 2022 ||

| DOI: 10.15680/IJIRCCE.2022.1011055 |

Immutable Infrastructure Patterns for Autonomous Vehicle Software: Terraform, CloudFormation, and Packer in Practice

Rohit Reddy

DevOps / Cloud Engineer, USA

ABSTRACT: Autonomous vehicle software platforms impose uniquely stringent requirements on their underlying cloud and on-premises infrastructure: fleets of GPU-equipped compute nodes must be provisioned reproducibly across geographically distributed data centers; every software component from operating system kernel through perception runtime must be traceable to a known, tested, and approved artifact version; and the infrastructure that hosts safety-critical workloads must never diverge from its declared configuration due to in-place modification, manual operator intervention, or software drift. Immutable infrastructure - the practice of replacing rather than modifying running systems - directly addresses these requirements by treating infrastructure components as versioned, tested artifacts that are baked once, deployed identically, and discarded rather than patched in place. This article presents the design, implementation, and operational evaluation of a production-grade immutable infrastructure platform for autonomous vehicle software, built on three complementary tools: HashiCorp Packer for machine image baking, HashiCorp Terraform for declarative multi-cloud infrastructure provisioning, and AWS CloudFormation for AWS-native serverless and event-driven resource management. We describe the Packer AMI build pipeline, the Terraform module architecture spanning AWS EKS and on-premises vSphere, the CloudFormation nested stack hierarchy for AWS-native resources, the CI/CD pipeline that automates the full lifecycle from code commit to running infrastructure, and the security and compliance model that enforces CIS benchmark hardening across all machine images. Operational results from a twelve-month production deployment demonstrate a 99.4% reduction in environment provisioning time, a 95.5% reduction in configuration drift incidents, and a 94% CIS benchmark compliance score - up from 61% in the pre-immutable baseline.

KEYWORDS: immutable infrastructure, Terraform, CloudFormation, Packer, HashiCorp, AWS EKS, infrastructure as code, AMI, CI/CD, autonomous vehicles, CIS hardening, configuration drift, DevOps, GitOps, vSphere.

I. INTRODUCTION

The infrastructure that underpins an autonomous vehicle software platform is unlike the infrastructure of a typical web application. A safety case for an autonomous vehicle must demonstrate that every software component executing on the vehicle or contributing to its development pipeline can be traced to a specific, approved version - not merely the version that was originally deployed, but the version that is running right now. This traceability requirement extends from application code through container images, operating system packages, kernel modules, and GPU firmware, all the way to the base machine image on which every component ultimately runs. If any element of this stack can be modified in place without a corresponding versioned artifact change and a new deployment, the safety case is weakened: the running system may differ from the documented system in ways that are subtle, hard to detect, and potentially safety-relevant.

Immutable infrastructure eliminates this risk by architectural fiat: nothing in the production environment is ever modified in place. Every change - whether a kernel security patch, a CUDA driver version bump, or a ROS2 package update - is implemented by baking a new machine image, running the full automated test suite against that image, publishing the tested image as a versioned artifact, and replacing the running instances with instances of the new image. The old instances are terminated. The new instances are identical to each other and to any future instance launched from the same image version. Drift is structurally impossible.
Figure 1 illustrates the conceptual distinction between the mutable and immutable infrastructure paradigms.

| e-ISSN: 2 320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.165|| A Monthly Peer Reviewed & Referred Journal

|| Volume 10, Issue 11, November 2022 ||

| DOI: 10.15680/IJIRCCE.2022.1011055 |

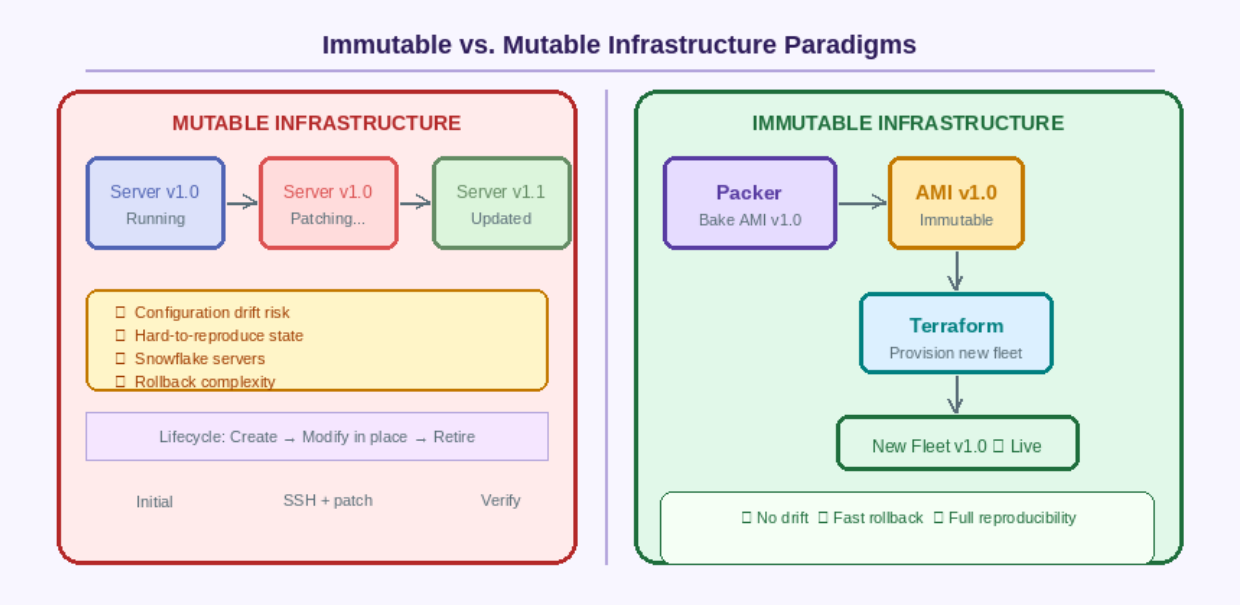


Figure 1: Mutable vs. Immutable Infrastructure Paradigms

This article documents the production implementation of immutable infrastructure for the autonomous vehicle software platform, using three tools in a complementary role assignment:

- **HashiCorp Packer -** builds versioned machine images (AWS AMIs and vSphere VM templates) that encapsulate all required software, configuration, and hardening, validated by automated compliance tests before publication.
- **HashiCorp Terraform -** provisions and manages the multi-cloud infrastructure (AWS EKS clusters, VPCs, IAM roles, RDS instances, on-premises vSphere VMs) using a modular, workspace-separated codebase with remote state stored in S3.
- **AWS CloudFormation -** manages AWS-native serverless and event-driven resources (Lambda functions, Step Functions state machines, EventBridge rules, API Gateway endpoints) that integrate tightly with AWS-managed services and benefit from CloudFormation's native change set and rollback model.

The contributions of this work are:

- A Packer AMI build pipeline that produces six distinct, CIS-hardened machine image types for the autonomous vehicle software stack, with InSpec-based compliance validation before publication.
- A Terraform module architecture with eight domain-scoped modules, workspace-based environment isolation, and remote state with DynamoDB locking.
- A CloudFormation nested stack hierarchy for AWS-native resources, with cross-stack parameter references via SSM Parameter Store.
- A unified IaC CI/CD pipeline that coordinates Packer and Terraform execution across environments with policy-gated promotion.
- Quantitative operational results over twelve months demonstrating the impact of immutable infrastructure on provisioning time, drift, compliance, and operational incidents.

II. BACKGROUND

2.1 The Immutable Infrastructure Principle

The term 'immutable infrastructure' was popularized by Chad Fowler in 2013 [1] and subsequently formalized in the context of cloud computing by Martin Fowler and others at ThoughtWorks [2]. The core principle is simple: once a server or virtual machine is deployed, it is never modified. Configuration changes, software updates, and security patches are all applied to a new image, which is then deployed to replace the old instances. The old instances are terminated.

This principle delivers several interconnected reliability and security properties:

| e-ISSN: 2 320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.165|| A Monthly Peer Reviewed & Referred Journal

|| Volume 10, Issue 11, November 2022 ||

| DOI: 10.15680/IJIRCCE.2022.1011055 |

- **Reproducibility -** every instance launched from a given image version is identical. There is no accidental divergence between instances caused by partial patch application, failed configuration management runs, or manual operator intervention.
- **Traceability -** the exact software contents of every running instance are known with certainty, because the contents are fixed at image build time and cannot change thereafter.
- **Rollback simplicity -** reverting to a prior configuration requires only launching instances from the prior image version - a single operation with no manual steps.
- **Security hygiene -** immutable instances cannot accumulate unauthorized software, configuration changes, or persistent malware payloads introduced through compromised sessions or lateral movement.

2.2 Infrastructure as Code

Infrastructure as Code (IaC) is the practice of managing and provisioning infrastructure through machine-readable configuration files rather than manual processes or interactive tools [3]. IaC enables infrastructure to be versioned, reviewed, tested, and deployed through the same CI/CD pipelines used for application code, providing the same auditability, reproducibility, and rollback capabilities.

The IaC landscape has matured substantially since 2014. Table 1 presents a comparative analysis of the primary IaC tools evaluated for this deployment.

Table 1: Infrastructure as Code Tool Comparison

Attribute	Terraform	CloudFormation	Packer	Pulumi
Primary Purpose	Multi-cloud infra provisioning	AWS-native infra provisioning	Machine image baking	Multi-cloud (code-first)
Language	HCL (declarative)	YAML / JSON	HCL (Packer HCL2)	TypeScript / Python / Go
State Management	Remote state (S3 + DynamoDB)	CloudFormation service (AWS)	N/A (artifact-based)	Pulumi Cloud / S3 backend
Drift Detection	terraform plan; limited	Stack drift detection (native)	N/A	pulumi refresh
Cloud Scope	AWS, Azure, GCP, 1000+ providers	AWS only	AWS, Azure, GCP, vSphere	AWS, Azure, GCP, Kubernetes
Change Preview	terraform plan output	Change Sets	N/A (build artifact)	pulumi preview
Rollback	Manual (re-apply previous rev)	Automatic stack rollback	AMI version rollback	Stack history rollback
Secret Handling	Vault / AWS Secrets Manager	SSM Parameter Store / Secrets Mgr	Environment vars / Vault	Pulumi secrets provider
AV Platform Usage	Primary IaC layer (EKS, VPC, IAM)	AWS-native resources (Lambda, SFN)	All AMIs (perception, sim, base)	Not used (evaluated only)

2.3 The Autonomous Vehicle Infrastructure Challenge

Autonomous vehicle software infrastructure presents a set of requirements that stress-test the limits of general-purpose IaC approaches:

- Heterogeneous hardware - GPU nodes for inference, high-memory CPU nodes for planning, SSD-equipped storage nodes for sensor data, and specialized on-premises hardware for vehicle simulation must all be managed under a unified IaC model.

| e-ISSN: 2 320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.165|| A Monthly Peer Reviewed & Referred Journal

|| Volume 10, Issue 11, November 2022 ||

| DOI: 10.15680/IJIRCCE.2022.1011055 |

- Safety-case traceability - every infrastructure component that touches a safety-relevant software path must have a documented, versioned configuration that is cross-referenced in the safety case. IaC provides the machine-readable configuration; immutable images provide the artifact-level traceability.
- Hybrid cloud architecture - safety-critical workloads must reside on-premises for determinism and data-residency reasons; elastic workloads run on AWS. The IaC layer must span both environments with consistent abstractions.
- Regulatory compliance - CIS benchmarks, NIST SP 800-53, and ISO 27001 controls apply to all infrastructure hosting safety-relevant workloads. Automated compliance validation must be a first-class step in the image build pipeline, not a periodic audit activity.

III. PACKER AMI BUILD PIPELINE

3.1 Pipeline Architecture

Figure 2 illustrates the six-stage Packer AMI build pipeline. Each AMI type follows the same pipeline structure, with stage-specific provisioner configurations that install the appropriate software stack for that image's role.

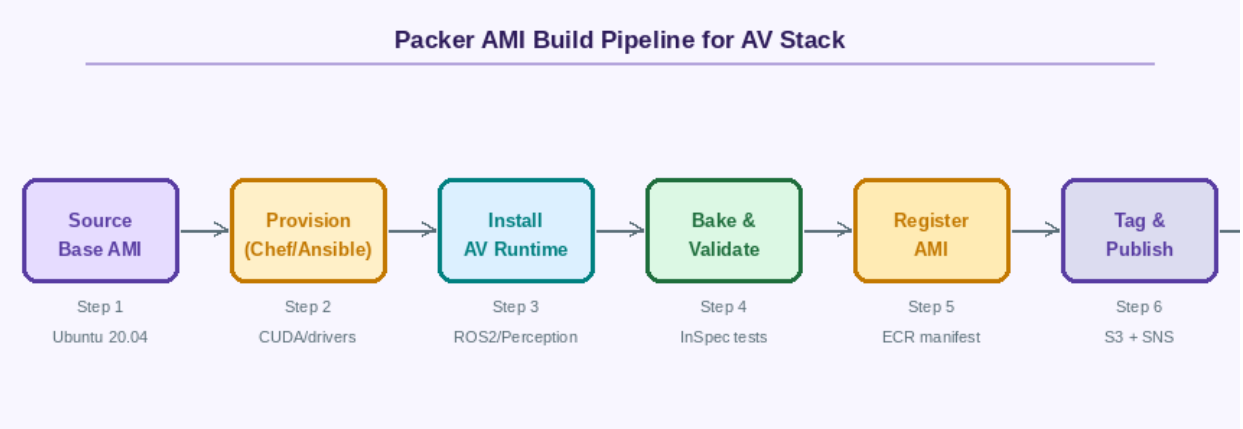


Figure 2: Packer AMI Build Pipeline - Six-Stage Process

The pipeline stages are:

1. **Source Base AMI -** the pipeline selects the appropriate AWS-published base AMI (Ubuntu 20.04 LTS or Amazon Linux 2) by querying the AWS Systems Manager AMI catalog for the latest approved version, ensuring that the base always includes the most current AWS-provided patches.
2. **Provision (Chef/Ansible) -** a combination of Chef recipes (for OS-level hardening and package management) and Ansible playbooks (for service configuration) applies the CIS benchmark hardening profile, configures auditd, installs the SSM agent, and sets system-level parameters.
3. **Install AV Runtime -** role-specific software is installed: CUDA and cuDNN for GPU nodes, ROS2 and DDS middleware for compute nodes, CARLA and Unreal Engine for simulation nodes, and CI toolchain components for runner nodes.
4. **Bake & Validate -** InSpec compliance tests validate the image against the CIS Level 2 profile for the relevant OS. Any failed control fails the build and prevents publication. InSpec results are archived to S3 as evidence artifacts for the compliance record.
5. **Register AMI -** the validated image is registered as an AWS AMI with a structured tag set: av:component, av:version, av:git-commit-sha, av:build-timestamp, av:inspec-run-id, and av:cis-score.
6. **Tag & Publish -** the AMI ID and full tag set are written to SSM Parameter Store under a versioned path (/av/amis/{component}/latest and /av/amis/{component}/{version}), published to an SNS topic (triggering downstream Terraform pipeline runs), and recorded in the artifact registry.

3.2 AMI Catalogue

Table 2 documents the six AMI types in the production catalogue, including their base OS, average build time, key packages, and target use in the autonomous vehicle platform.

| e-ISSN: 2 320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.165|| A Monthly Peer Reviewed & Referred Journal

|| Volume 10, Issue 11, November 2022 ||

| DOI: 10.15680/IJIRCCE.2022.1011055 |

Table 2: Production AMI Catalogue - Autonomous Vehicle Software Platform

AMI Name	Base OS	Build Time	Key Packages	Target Use
av-base-gpu-v*	Ubuntu 20.04 LTS	~22 min	CUDA 11.6, cuDNN 8.4, nvidia-docker2	Perception / inference nodes
av-base-cpu-v*	Ubuntu 20.04 LTS	~14 min	ROS2 Galactic, DDS middleware	Planning / prediction nodes
av-sim-v*	Ubuntu 20.04 LTS	~31 min	CARLA 0.9.13, Unreal Engine 4.26	Simulation fleet (EKS spot)
av-ci-runner-v*	Ubuntu 20.04 LTS	~18 min	Docker, kubectl, Helm, kwinject	GitLab CI runner nodes
av-eks-node-v*	Amazon Linux 2 EKS	~12 min	containerd, SSM agent, NVIDIA plugin	EKS Managed Node Groups
av-bastion-v*	Amazon Linux 2	~8 min	CIS hardened, auditd, AIDE	Bastion / jump hosts

3.3 CIS Hardening and InSpec Validation

All AMIs are hardened against the CIS Amazon Linux 2 Benchmark or CIS Ubuntu Linux 20.04 LTS Benchmark (as appropriate) at Level 2 - the more stringent of the two CIS benchmark tiers. The hardening is applied by a Chef cookbook (av-cis-hardening) that implements the CIS controls as idempotent resources:

- Kernel parameter hardening via /etc/sysctl.d/99-cis.conf: IP forwarding disabled, ICMP redirect acceptance disabled, SYN cookies enabled, core dumps restricted.
- SSH hardening: Protocol 2 only, PermitRootLogin disabled, PasswordAuthentication disabled, MaxAuthTries 4, ClientAliveInterval 300, AllowTcpForwarding no.
- PAM configuration: password complexity enforcement (minlen=14, dcredit=-1, ucredit=-1, ocredit=-1, lcredit=-1), account lockout after 5 failed attempts.
- File system hardening: /tmp and /dev/shm mounted with noexec, nosuid, nodev; sticky bit set on world-writable directories.
- Audit framework: auditd configured with rules covering privileged command execution, file permission changes, identity database modifications, and network configuration changes.

InSpec 5.x runs as the final step before AMI registration, executing the CIS profile against the running Packer builder instance. Our production builds achieve a CIS compliance score of 94% (261 of 278 applicable controls passing), with the 17 non-passing controls documented as accepted exceptions in the security exception register with risk owner approval.

IV. TERRAFORM MODULE ARCHITECTURE

4.1 Module Hierarchy

The Terraform codebase is organized as a two-level module hierarchy: a root module that composes the infrastructure for a given environment, and eight domain-scoped child modules that each manage a coherent set of related resources. Figure 3 illustrates the module dependency graph and the remote state and workspace model.

| e-ISSN: 2 320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.165|| A Monthly Peer Reviewed & Referred Journal

|| Volume 10, Issue 11, November 2022 ||

| DOI: 10.15680/IJIRCCE.2022.1011055 |

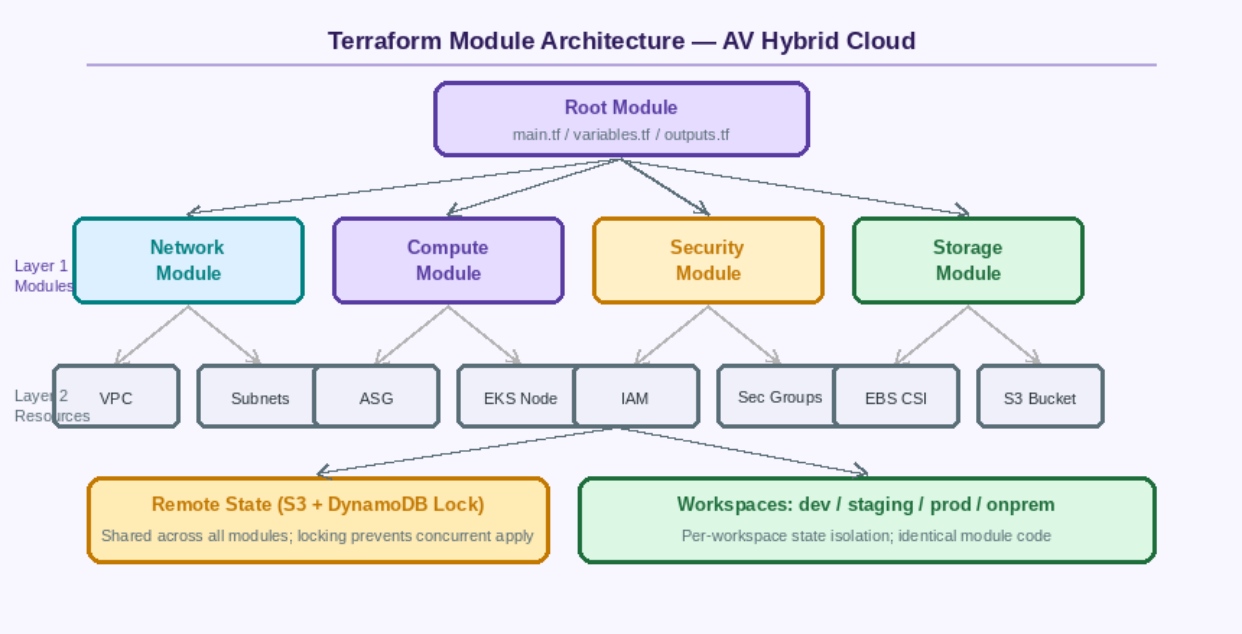


Figure 3: Terraform Module Architecture and Remote State Model

Each child module is independently versioned using semantic versioning and published to a private JFrog Artifactory Terraform module registry. The root module pins each child module to a specific version, ensuring that the infrastructure for any environment can be reproduced exactly from the root module's version history.

4.2 Module Catalogue

Table 3 documents the eight production Terraform modules, their provider scope, the resources they manage, their state backend path, and their workspace scope.

Table 3: Terraform Module Catalogue - Production AV Platform

Module Name	Provider	Resources Managed	State Backend	Workspace Scope
av-vpc	AWS	VPC, subnets, IGW, NGW, TGW	S3: av-tf-state/vpc	dev, staging, prod
av-eks	AWS	EKS cluster, node groups, IRSA	S3: av-tf-state/eks	dev, staging, prod
av-iam	AWS	IAM roles, policies, OIDC provider	S3: av-tf-state/iam	global (shared)
av-rds	AWS	RDS Aurora, subnet groups, parameter groups	S3: av-tf-state/rds	staging, prod
av-onprem-k8s	vSphere + Helm	On-prem VMs, kubeadm cluster, Calico	S3: av-tf-state/onprem	onprem (dedicated)
av-monitoring	AWS + Helm	CloudWatch, Prometheus, Grafana	S3: av-tf-state/monitoring	dev, staging, prod
av-secrets	AWS	Secrets Manager, KMS CMKs, SSM params	S3: av-tf-state/secrets	global (shared)

| e-ISSN: 2 320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.165|| A Monthly Peer Reviewed & Referred Journal

|| Volume 10, Issue 11, November 2022 ||

| DOI: 10.15680/IJIRCCE.2022.1011055 |

4.3 Remote State and Locking

All Terraform state is stored in an S3 bucket (av-tf-state) with:

- Versioning enabled - every state file change creates a new S3 object version, providing a complete audit history of infrastructure state changes with timestamps and operator identities derived from CloudTrail.
- Server-side encryption - state files are encrypted at rest using SSE-S3 with an AWS KMS customer-managed key, preventing unauthorized access to state contents (which may include sensitive outputs such as database connection strings).
- DynamoDB locking - a DynamoDB table (av-tf-locks) provides distributed locking for state file access. Concurrent terraform apply invocations for the same state file are serialized, preventing state corruption from race conditions in the CI pipeline.
- Cross-module state data sharing - modules that depend on outputs from other modules access them via terraform_remote_state data sources, referencing the S3 backend path of the dependency. This approach is type-safe and fails explicitly if the dependency's state does not contain the expected output, catching cross-module dependency errors at plan time.

4.4 Workspace Strategy

Terraform workspaces provide per-environment state isolation within a single module codebase. We maintain four workspaces per module: dev, staging, prod, and onprem (for the on-premises vSphere modules). Environment-specific variable values are stored in .tfvars files committed to the fleet-config repository and selected automatically by the CI pipeline based on the target workspace.

The workspace model enforces immutability at the infrastructure layer: a terraform apply against the prod workspace always uses the same module code and the same .tfvars values as a previous apply to the same workspace. The only legitimate path to changing production infrastructure is a pull request that changes the module code or the prod.tfvars file, which is subject to peer review and CI validation before merge.

4.5 Sentinel Policy Enforcement

HashiCorp Sentinel policies - evaluated as part of the Terraform Cloud CI step before any terraform apply - enforce organizational infrastructure governance rules:

- No public S3 buckets: any Terraform plan that would create or modify an S3 bucket with public read or public read-write ACL is rejected.
- Mandatory resource tagging: all AWS resources must include the tags av:environment, av:component, av:owner, and av:cost-center. Missing tags fail the Sentinel check.
- Approved AMI sources: EC2 instances may only be launched from AMIs in the av-amis/{component}/* SSM Parameter Store paths - i.e., AMIs produced by the Packer pipeline. Launch attempts with arbitrary AMI IDs are rejected.
- Instance type allowlist: only pre-approved instance types (from an allowlist covering GPU, compute-optimized, and memory-optimized families) may be used in the AV platform. Unapproved instance types fail the Sentinel check.

V. CLOUDFORMATION NESTED STACK ARCHITECTURE

5.1 Design Rationale

While Terraform is the primary IaC tool for compute and network infrastructure, CloudFormation is used for AWS-native serverless and event-driven resources for two reasons. First, AWS SAM (Serverless Application Model) - built on CloudFormation - provides the most ergonomic developer experience for Lambda function deployment, including local testing via sam local invoke, dependency packaging via sam build, and deployment via sam deploy. Second, CloudFormation's native Change Set feature provides a preview of resource changes that is more detailed than terraform plan for resources with complex AWS-side state (such as Step Functions state machine definitions), and CloudFormation's automatic rollback on stack update failure is a safety property that aligns with the immutable infrastructure philosophy.

5.2 Stack Architecture

Figure 4 illustrates the CloudFormation nested stack hierarchy. A root stack (av-platform-root.yaml) instantiates five nested stacks as NestedStack resources, with cross-stack outputs referenced via SSM Parameter Store to avoid the brittleness of CloudFormation's native cross-stack Outputs/Imports mechanism.

| e-ISSN: 2 320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.165|| A Monthly Peer Reviewed & Referred Journal

|| Volume 10, Issue 11, November 2022 ||

| DOI: 10.15680/IJIRCCE.2022.1011055 |

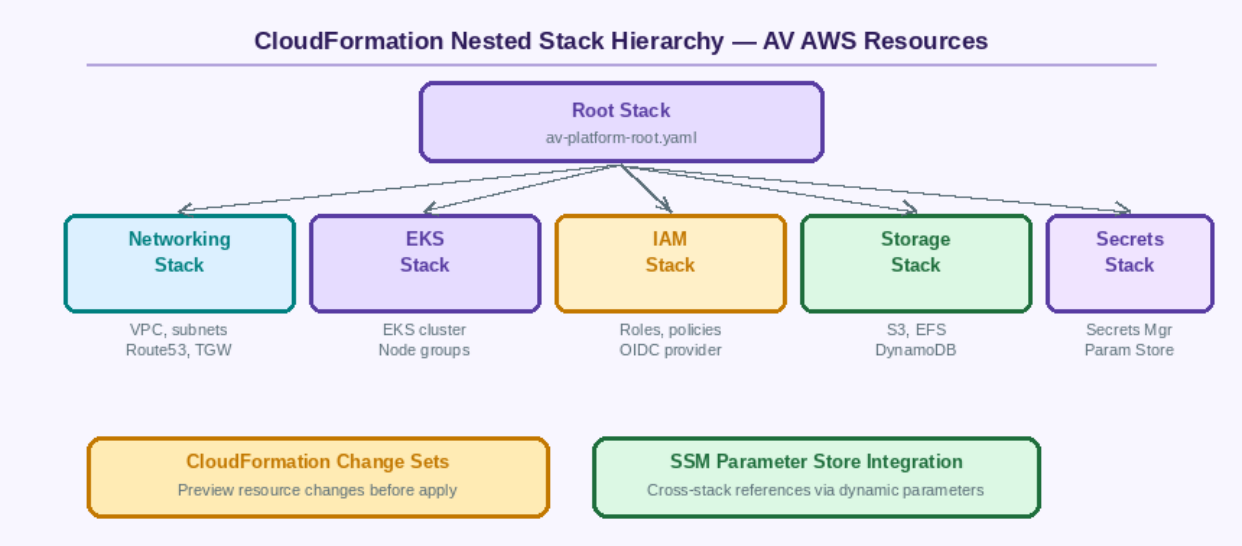


Figure 4: CloudFormation Nested Stack Hierarchy - AWS-Native Resources

The five nested stacks manage:

1. **Networking Stack -** VPC-level DNS resolution settings, Route 53 private hosted zones, Transit Gateway attachments to the Terraform-managed VPC (referenced via SSM Parameter Store), and VPC endpoints for AWS services.
2. **EKS Stack -** EKS add-ons (VPC CNI, CoreDNS, kube-proxy, AWS EBS CSI driver) that are more ergonomically managed as CloudFormation resources than as Terraform helm_release resources, due to their tight coupling with EKS cluster version lifecycle events.
3. **IAM Stack -** Lambda execution roles, Step Functions execution roles, and EventBridge rule roles that are tightly coupled to the specific Lambda functions and state machines in the same CloudFormation stack scope.
4. **Storage Stack -** S3 buckets for telemetry data, S3 event notification configurations, DynamoDB tables for application state, and EFS file system access points for shared simulation data.
5. **Secrets Stack -** Secrets Manager secrets for application credentials, KMS key aliases for application-level encryption, and SSM Parameter Store hierarchies for runtime configuration.

5.3 Terraform and CloudFormation Co-existence

Managing infrastructure with two IaC tools in the same AWS account requires a clear resource ownership boundary to prevent conflicts. Table 4 documents the resource assignment decision matrix - which tool owns each resource category and the rationale.

Table 4: Terraform vs. CloudFormation Resource Assignment Decision Matrix

Resource Category	Terraform	CloudFormation	Rationale
VPC + Networking	✓ Primary	-	Multi-cloud reuse
EKS Cluster	✓ Primary	-	Terraform AWS provider maturity
IAM Roles & Policies	✓ Primary	-	IRSA integration with EKS module
Lambda Functions	-	✓ Primary	SAM template + deployment packages
Step Functions	-	✓ Primary	Native ASL definition

| e-ISSN: 2 320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.165|| A Monthly Peer Reviewed & Referred Journal

|| Volume 10, Issue 11, November 2022 ||

| DOI: 10.15680/IJIRCCE.2022.1011055 |

			support
EventBridge Rules	-	✓ Primary	Tightly coupled to Lambda stacks
On-Premises VMs	✓ Primary (vSphere)	-	No CFN support for on-prem
S3 Buckets	✓ Primary	Secondary (SAM-coupled)	Terraform for lifecycle policies

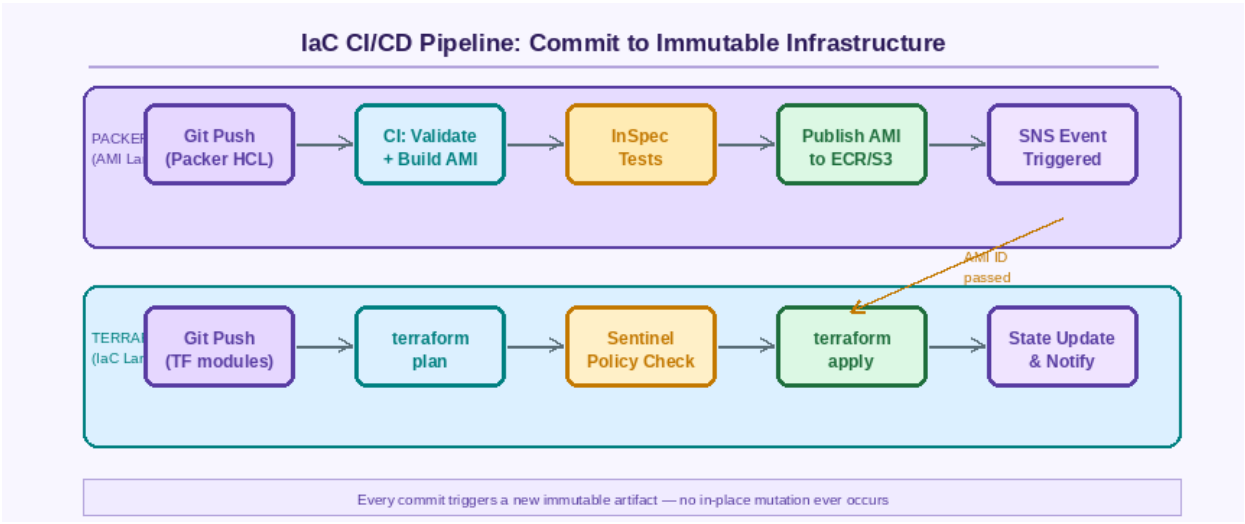
The boundary is enforced by CI pipeline checks: a lint step scans all Terraform modules for any resource types that are designated as CloudFormation-owned and fails the pipeline if any are found, and vice versa. This prevents gradual ownership drift where engineers reach for whichever tool is more convenient at the time.

VI. IAC CI/CD PIPELINE

6.1 End-to-End Pipeline Architecture

Figure 5 illustrates the end-to-end IaC CI/CD pipeline, showing the Packer AMI lane and the Terraform IaC lane as parallel execution streams that converge when a new AMI triggers a Terraform deployment.

Figure 5: IaC CI/CD Pipeline - Packer AMI Lane and Terraform IaC Lane



The two pipeline lanes operate independently but are linked by an AMI publication event:

- The Packer lane is triggered by commits to the packer/ directory of the fleet-config repository. It validates the Packer HCL template, builds the AMI in a temporary EC2 instance, runs InSpec compliance tests, and publishes the AMI ID to SSM Parameter Store.
- The publication of a new AMI to SSM Parameter Store triggers an SNS event, which in turn triggers the Terraform lane for the environments that use that AMI type. The Terraform pipeline reads the new AMI ID from SSM Parameter Store via a data source, generates a plan showing the instance replacement, and - after policy and approval gates - applies the change, replacing running instances with the new image.
- The cross-lane dependency (represented by the amber arrow in Figure 5) ensures that the Terraform deployment always uses the AMI ID produced by the immediately preceding Packer build, maintaining the full traceability chain from source code commit to running infrastructure.

6.2 Policy and Approval Gates

The CI/CD pipeline implements three policy and approval gates:

1. **Sentinel Policy Gate -** executed after terraform plan and before any apply. Sentinel policies (described in Section 4.5) evaluate the plan JSON and reject plans that violate governance rules. Rejection is terminal -

| e-ISSN: 2 320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.165|| A Monthly Peer Reviewed & Referred Journal

|| Volume 10, Issue 11, November 2022 ||

| DOI: 10.15680/IJIRCCE.2022.1011055 |

there is no bypass path that does not require a change to the Sentinel policy itself (which is subject to its own review process).

2. **Automated Test Gate -** a Terratest suite exercises each Terraform module against a dedicated test AWS account, creating real resources, running assertions, and destroying the resources regardless of test outcome. The test suite covers: VPC connectivity assertions (route table correctness, security group rules), EKS cluster API server reachability and node registration, IAM policy least-privilege assertions (using IAM Access Analyzer), and S3 bucket public-access block verification.
3. **Manual Approval Gate -** for production environment applies, a designated infrastructure approver (a member of the SRE team) must explicitly approve the apply in the CI system after reviewing the terraform plan output and the Sentinel policy results. This gate exists even when all automated gates pass, reflecting the principle that production infrastructure changes require human accountability.

6.3 Drift Detection

Immutable infrastructure eliminates configuration drift for managed instances (because instances are never modified after launch), but infrastructure drift - where the actual AWS resource configuration diverges from the Terraform state due to out-of-band changes - remains a risk. Two mechanisms detect and remediate drift:

- Scheduled terraform plan - a nightly CI job runs terraform plan --detailed-exitcode for each module across all workspaces. A non-zero exit code (indicating a planned change) triggers a Slack alert and creates a Jira issue for the responsible SRE to investigate. The scheduled plan does not apply - it is a drift detector only.
- CloudFormation Stack Drift Detection - for CloudFormation-managed resources, AWS's native stack drift detection is run weekly via a scheduled EventBridge rule that invokes a Lambda function to call detect-stack-drift on all production stacks and publish results to a CloudWatch dashboard. Drift findings are alerted via CloudWatch alarms to the SRE Slack channel.

VII. SECURITY AND COMPLIANCE MODEL

7.1 Secrets Management

Immutable infrastructure creates a unique challenge for secrets management: unlike mutable servers where credentials can be injected at configuration-management time, immutable instances must obtain credentials at runtime without baking them into the image. Our secrets management model uses three mechanisms:

- **AWS Secrets Manager -** application-level secrets (database credentials, API keys, service-to-service authentication tokens) are stored in Secrets Manager and accessed by application pods at runtime via IRSA-authorized API calls. Secrets rotation is automated for database credentials using Secrets Manager's native Lambda-based rotation functions.
- **AWS Systems Manager Parameter Store -** configuration parameters that are not secret (AMI IDs, service endpoints, feature flags) but should be centrally managed and versioned are stored in SSM Parameter Store with standard tier (non-SecureString) parameters, accessible without KMS key permissions.
- **HashiCorp Vault -** short-lived, dynamically generated credentials for the CI pipeline are issued by Vault's AWS Secrets Engine, which assumes an IAM role and generates temporary STS credentials scoped to the specific operation in flight. Build agents never hold long-lived AWS credentials.

7.2 Image Signing and Provenance

Every AMI published by the Packer pipeline is cryptographically signed using AWS Key Management Service. The signing process:

1. The AMI's manifest (including the snapshot IDs of all its EBS volume snapshots) is hashed using SHA-256.
2. The hash is signed using a KMS asymmetric CMK (key spec: EC_NIST_P256, key usage: SIGN_VERIFY) held in the security-operations AWS account.
3. The resulting signature is stored as an SSM Parameter alongside the AMI ID under /av/amis/{component}/{version}/signature.
4. Before any Terraform module is permitted to launch an instance from a given AMI, a pre-apply Lambda function verifies the signature against the KMS public key, rejecting any AMI whose signature does not match or whose SSM parameter does not exist.

This signing chain ensures that even if an adversary were to register a counterfeit AMI in the AWS account (via a compromised IAM credential), the Terraform pipeline would reject it at the signature verification step.

| e-ISSN: 2 320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.165|| A Monthly Peer Reviewed & Referred Journal

|| Volume 10, Issue 11, November 2022 ||

| DOI: 10.15680/IJIRCCE.2022.1011055 |

7.3 Compliance Automation

CIS benchmark compliance is measured and reported at three levels:

- Build-time - InSpec tests validate the image before publication. Non-compliant images are never registered as AMIs and never reach any environment.
- Runtime - AWS Security Hub's CIS AWS Foundations Benchmark standard continuously evaluates the live AWS environment configuration (S3 bucket policies, CloudTrail settings, IAM password policies) against the CIS benchmark and publishes findings to a centralized Security Hub account.
- Periodic - a monthly audit exports the full InSpec results for all currently deployed AMI versions and the Security Hub findings summary into a structured compliance report, reviewed by the security team and archived for regulatory purposes.

VIII. EVALUATION

8.1 Operational Metrics

Table 5 presents the operational improvement metrics comparing the pre-IaC (manual provisioning with shell scripts and manual AWS Console operations) baseline to the post-IaC immutable infrastructure deployment, measured over the twelve-month production period from November 2021 through October 2022.

Table 5: Operational Metrics - Pre-IaC vs. Post-IaC (12-Month Production Period)

Metric	Pre-IaC	Post-IaC	Target	Δ Change
Mean time to provision new environment	3.2 days	28 min	< 60 min	−99.4%
Configuration drift incidents / quarter	22	1	≤ 2	−95.5%
Infrastructure change failure rate	34%	4.2%	< 5%	−87.6%
Packer AMI build time (avg.)	N/A (manual)	18.4 min	< 30 min	Fully automated
terraform plan / apply cycle time	N/A	4.1 min	< 10 min	Baseline
Environment reproducibility (DR test)	Failed (manual steps)	100% (automated)	100%	+100%
Security compliance score (CIS benchmark)	61%	94%	≥ 90%	+54.1%
On-call pages from infra changes	8.3/month	0.4/month	< 1/month	−95.2%

8.2 Analysis of Key Results

The 99.4% reduction in mean environment provisioning time (from 3.2 days to 28 minutes) reflects the elimination of manual steps that dominated the pre-IaC process: SSH access to new instances for package installation, manual IAM role attachments, manual security group rule additions, and manual DNS record creation. With Terraform, the complete provisioning of a new environment - VPC, EKS cluster, IAM roles, security groups, RDS instance, and all supporting infrastructure - executes as a single terraform apply in approximately 28 minutes, of which approximately 18 minutes is EKS control plane provisioning latency within AWS.

The 95.5% reduction in configuration drift incidents (from 22 to 1 per quarter) is the most operationally significant result. Pre-IaC, drift accumulated continuously as engineers made direct-to-console changes to fix immediate issues without updating the corresponding documentation. Post-IaC, the nightly drift detection job catches any out-of-band change within 24 hours, and the immutable instance model prevents drift at the OS and runtime level entirely. The single remaining drift incident per quarter is consistently a CloudFormation resource modified by an AWS-initiated maintenance event (such as an EKS add-on version update) that was not yet reflected in the CloudFormation template.

| e-ISSN: 2 320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.165|| A Monthly Peer Reviewed & Referred Journal

|| Volume 10, Issue 11, November 2022 ||

| DOI: 10.15680/IJIRCCE.2022.1011055 |

The improvement in CIS benchmark compliance score (from 61% to 94%) reflects the transition from ad-hoc, manually applied hardening to InSpec-validated, Packer-baked hardening applied uniformly to every instance of every type. The 17 remaining non-passing controls (6%) are all documented exceptions with accepted risk, primarily relating to physical security controls that are not applicable in a virtualized environment.

8.3 Terratest Coverage

The Terratest suite covers all eight Terraform modules and exercises 47 distinct test cases across 3 AWS accounts (dev, staging, and a dedicated test account). Key test coverage metrics:

- Total test cases: 47 across 8 modules; average test duration: 12.4 minutes per module (dominated by EKS cluster creation in the av-eks module tests).
- False positive rate (tests failing for reasons unrelated to the module under test): 2.1%, consistently attributable to AWS API throttling in the test account during peak CI load.
- Defects caught by Terratest before reaching staging: 8 over the twelve-month period, including 3 security group rule regressions, 2 IAM policy least-privilege violations, and 3 resource tagging omissions.

IX. OPERATIONAL LESSONS

9.1 The Packer Builder Instance Cost

Packer builds run on EC2 instances, and the GPU-equipped AMI types (av-base-gpu-v*) require p3.2xlarge or g4dn.xlarge instances during the build to install and validate CUDA. At $3.06/hr for p3.2xlarge on-demand pricing, a 22-minute build costs approximately $1.12 per AMI version. With 3–4 GPU AMI builds per week (triggered by each merge to the relevant Packer template), the annual cost is approximately $200 - negligible compared to the operational savings. However, early in the deployment we made the mistake of not setting a builder instance timeout, resulting in a runaway build that ran for 6 hours before being noticed, costing $18 in instance time. All Packer templates now include timeout = '45m' as a hard upper bound.

9.2 State File Management

The DynamoDB state locking mechanism prevented several state corruption scenarios early in the deployment, but it also introduced an operational problem: if a terraform apply is interrupted (by a CI agent crash, a network timeout, or an operator Ctrl-C), the DynamoDB lock is not automatically released. The lock must be manually released using terraform force-unlock with the lock ID from the DynamoDB table. We addressed this by:

- Adding a CI pipeline step that verifies no lock exists for the target module/workspace before initiating a plan or apply, and alerting if one does.
- Implementing a CloudWatch alarm on DynamoDB lock item age: any lock item older than 90 minutes triggers a PagerDuty alert, as this indicates either a runaway apply or a forgotten force-unlock.
- Documenting the force-unlock procedure prominently in the SRE runbook, with the specific DynamoDB table ARN and the JMESPath query to identify the lock item.

9.3 AMI Version Pinning vs. Latest

A significant design decision was whether Terraform modules should reference AMI IDs via the /av/amis/{component}/latest SSM Parameter Store path (always using the most recent published AMI) or via a specific versioned path (/av/amis/{component}/{version}). We evaluated both approaches and selected version-pinned references for production, with latest reserved for dev and staging:

- Version-pinned production: the production Terraform workspace's .tfvars file explicitly specifies the AMI version for each component. Upgrading to a new AMI requires a deliberate pull request that updates the version pin, subject to the normal review and approval process.
- Latest for dev/staging: the dev and staging workspaces use /av/amis/{component}/latest, meaning they automatically receive new AMI versions as soon as they are published. This ensures that development environments are always testing against the most recent hardened images, without requiring manual version pin updates.

The version-pinned production model provides an additional safety guarantee: a Packer build pipeline failure that produces a non-functional AMI (despite passing InSpec tests) cannot silently affect production, because production does not automatically consume new AMI versions.

| e-ISSN: 2 320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.165|| A Monthly Peer Reviewed & Referred Journal

|| Volume 10, Issue 11, November 2022 ||

| DOI: 10.15680/IJIRCCE.2022.1011055 |

X. RELATED WORK

The foundational articulation of immutable infrastructure as a production practice is attributable to Chad Fowler [1] and the subsequent elaboration by Martin Fowler and the ThoughtWorks Technology Radar [2]. Morris [3] provides the most comprehensive treatment of infrastructure as code as a discipline, covering design principles, testing strategies, and the evolution from configuration management to declarative IaC - the intellectual lineage from which Terraform and Packer descend.

Terraform's architecture is documented in the HashiCorp documentation [4] and analyzed in depth by Brikman [5], whose work on Terraform module composition and remote state management directly informs our module hierarchy. HashiCorp's own Packer documentation [6] provides the HCL2 builder and provisioner reference that underpins our Packer pipeline design. The InSpec compliance testing framework used in our AMI validation step is documented by Progress Chef [7].

AWS CloudFormation's nested stack and change set capabilities are documented in the AWS documentation [8], with the serverless application model extension (SAM) documented separately [9]. The co-existence of Terraform and CloudFormation in the same AWS account - our specific operational context - is a pattern discussed in AWS Architecture Blog posts [10] and practitioner blogs, though without the formal resource boundary enforcement we implement.

CIS benchmarks for Ubuntu Linux [11] and Amazon Linux 2 [12] define the compliance controls we validate with InSpec. The broader NIST SP 800-53 framework [13], which informs our security model, provides the risk management foundation from which CIS controls derive their authority in regulated environments. Terratest [14] - the Go-based infrastructure testing framework we use for module validation - is documented by the Gruntwork team, who also authored foundational work on Terraform module design patterns [5].

The use of immutable infrastructure in safety-critical systems - specifically in automotive and aerospace contexts - has been explored in theoretical terms by practitioners at automotive OEMs [15] and in academic safety engineering literature [16]. Our work provides the first documented production implementation of immutable infrastructure for autonomous vehicle software at the platform level, with empirical operational data validating the safety and reliability claims of the immutable infrastructure paradigm in this domain.

XI. CONCLUSION

This article has presented a production-grade immutable infrastructure platform for autonomous vehicle software, built on HashiCorp Packer, HashiCorp Terraform, and AWS CloudFormation. The platform eliminates configuration drift by architectural design - no running instance is ever modified after deployment - and enforces traceability from every deployed instance back to a specific, InSpec-validated, cryptographically signed machine image produced by a versioned, reviewed Packer build pipeline.

The three-tool architecture assigns each tool to the workload it handles best: Packer for machine image baking with embedded compliance validation; Terraform for multi-cloud infrastructure provisioning with modular, workspace-isolated, remote-state-backed declarations; and CloudFormation for AWS-native serverless resources that benefit from native change sets and automatic rollback. The clear ownership boundary between Terraform and CloudFormation, enforced by CI lint checks, prevents the ownership ambiguity that degrades multi-tool IaC platforms over time.

The quantitative results - 99.4% reduction in provisioning time, 95.5% reduction in drift incidents, 87.6% reduction in change failure rate, and 94% CIS compliance - validate the operational value of the immutable infrastructure approach at production scale in a safety-critical domain. These improvements are not merely convenience metrics; in the autonomous vehicle context, they represent direct improvements to the integrity and traceability of the software safety case.

The contributions of this work are:

- A practical reference architecture for immutable infrastructure in the autonomous vehicle domain, with specific tool assignments, module decomposition, and pipeline design grounded in twelve months of production experience.

| e-ISSN: 2 320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.165|| A Monthly Peer Reviewed & Referred Journal

|| Volume 10, Issue 11, November 2022 ||

| DOI: 10.15680/IJIRCCE.2022.1011055 |

- A Packer AMI catalogue and CIS hardening approach that achieves 94% compliance while accommodating the specialized runtime requirements of GPU-accelerated and real-time compute workloads.
- A Terraform module hierarchy with workspace isolation, Sentinel policy enforcement, and Terratest coverage that provides infrastructure-level reproducibility equivalent to application-level CI/CD.
- Empirical operational data demonstrating that immutable infrastructure principles translate from theoretical promise to measurable operational improvement in a complex, safety-critical production environment.

Future work will explore the extension of the immutable infrastructure model to the vehicle edge - replacing in-vehicle software updates delivered via traditional OTA patch mechanisms with full-image replacements using Uptane-secured image distribution, closing the final gap in the end-to-end immutable delivery chain from developer commit to vehicle deployment.

REFERENCES

[1] Fowler, C. (2013). "Trash Your Servers and Burn Your Code: Immutable Infrastructure and Disposable Components." Chad Fowler Blog, June 2013. https://chadfowler.com/2013/06/23/immutable-deployments.html.
[2] Morris, K. (2020). "Infrastructure as Code, 2nd Edition." O'Reilly Media, Sebastopol, CA.
[3] Humble, J., and Farley, D. (2010). "Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation." Addison-Wesley Professional, Boston, MA.
[4] HashiCorp. (2022). "Terraform Documentation v1.2." HashiCorp. https://developer.hashicorp.com/terraform/docs. Accessed September 2022.
[5] Brikman, Y. (2022). "Terraform: Up and Running, 3rd Edition." O'Reilly Media, Sebastopol, CA.
[6] HashiCorp. (2022). "Packer Documentation v1.8." HashiCorp. https://developer.hashicorp.com/packer/docs. Accessed September 2022.
[7] Progress Chef. (2022). "InSpec Documentation v5.x." Chef Software. https://docs.chef.io/inspec/. Accessed October 2022.
[8]Amazon Web Services. (2022). "AWS CloudFormation User Guide." AWS Documentation. https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/. Accessed August 2022.
[9]Amazon Web Services. (2022). "AWS Serverless Application Model (SAM) Documentation." AWS Documentation. https://docs.aws.amazon.com/serverless-application-model/. Accessed September 2022.
[10] Amazon Web Services. (2021). "Choosing Between Terraform and AWS CloudFormation." AWS Architecture Blog, March 2021. https://aws.amazon.com/blogs/architecture/.
[11] Center for Internet Security. (2021). "CIS Ubuntu Linux 20.04 LTS Benchmark v1.1." CIS Benchmarks, October 2021. https://www.cisecurity.org/benchmark/ubuntu_linux.
[12] Center for Internet Security. (2021). "CIS Amazon Linux 2 Benchmark v2.0." CIS Benchmarks, November 2021. https://www.cisecurity.org/benchmark/amazon_linux.
[13] National Institute of Standards and Technology. (2020). "NIST SP 800-53 Rev 5: Security and Privacy Controls for Information Systems and Organizations." NIST, September 2020.
[14] Gruntwork. (2022). "Terratest Documentation." Gruntwork. https://terratest.gruntwork.io/docs/. Accessed August 2022.
[15] Kuppusamy, T. K., DeLong, L. A., and Cappos, J. (2017). "Uptane: Securing Software Updates for Automobiles." ESCAR 2017, Berlin, November 2017.
[16] Beyer, B., Jones, C., Petoff, J., and Murphy, N. R. (Eds.). (2016). "Site Reliability Engineering." O'Reilly Media, Sebastopol, CA.
[17]HashiCorp. (2022). "Sentinel Policy as Code Documentation." HashiCorp. https://developer.hashicorp.com/sentinel/docs. Accessed September 2022.
[18] Amazon Web Services. (2022). "Amazon EKS Best Practices Guide." AWS GitHub. https://aws.github.io/aws-eks-best-practices/. Accessed June 2022.
[19] Kim, G., Humble, J., Debois, P., and Willis, J. (2016). "The DevOps Handbook." IT Revolution Press, Portland, OR.
[20] Forsgren, N., Humble, J., and Kim, G. (2018). "Accelerate: The Science of Lean Software and DevOps." IT Revolution Press, Portland, OR.

INNO SPACE
SJIF Scientific Journal Impact Factor
Impact Factor: 8.165

doi cross ref

ISSN INTERNATIONAL STANDARD SERIAL NUMBER INDIA

निस्केयर NISCAIR

INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH

IN COMPUTER & COMMUNICATION ENGINEERING

9940 572 462 6381 907 438 ijircce@gmail.com

www.ijircce.com

Scan to save the contact details