Preprints.org

Article

Not peer-reviewed version

Runbook Mesh: MCP-Orchestrated Terraform and Ansible Co-Execution on Azure

Sudhakavya Bodapati Venkata *

Posted Date: 5 March 2026

doi: 10.20944/preprints202603.0394.v1

Keywords: Azure; Terraform; Ansible; MCP; infrastructure as code; configuration management; Azure DevOps

Preprints.org is a free multidisciplinary platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC.

Copyright: This open access article is published under a Creative Commons CC BY 4.0 license, which permit the free download, distribution, and reuse, provided that the author and preprint are cited in any reuse.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 5 March 2026 doi:10.20944/preprints202603.0394.v1

Disclaimer/Publisher's Note: The statements, opinions, and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions, or products referred to in the content.

Article

Runbook Mesh: MCP-Orchestrated Terraform and Ansible Co-Execution on Azure

Sudhakavya Bodapati Venkata

The University of Texas at Dallas, 800 W Campbell Rd, Richardson, TX 75080, USA; bvsudhakavya@gmail.com

Abstract

Hybrid use of Terraform for infrastructure and Ansible for configuration is common on Azure, but the two tools are often joined only by ad hoc scripts and fragile handoffs in CI pipelines. Runbook Mesh proposes a small MCP based control plane that treats Terraform and Ansible as one coordinated change unit rather than two independent stages. Azure DevOps triggers an MCP server that drives a deployment state machine: it receives Terraform plans and apply results, derives a dynamic Ansible inventory from Terraform outputs, and orchestrates configuration playbooks with drain, cordon, and health checks for VM scale sets, AKS nodes, and virtual machines. The MCP enforces simple invariants on ordering, handoff safety, and rollback reachability, and packages each deployment into a witness bundle containing plan digests, state and inventory hashes, play outcomes, and Azure Resource Graph snapshots. The result is an Azure native pattern where infrastructure and configuration share a single timeline, a defined rollback path, and a tamper evident change ledger suited to regulated environments.

Keywords: Azure; Terraform; Ansible; MCP; infrastructure as code; configuration management; Azure DevOps

---

1. Introduction

On Azure, many teams split their delivery pipeline into two tracks: Terraform shapes infrastructure, and Ansible applies configuration on top of it. In practice the tracks meet only inside CI scripts, and details such as ordering, host discovery, and failure handling are buried in glue code. As environments and compliance needs grow, this loose coupling becomes harder to operate.

The lack of a clear contract between infrastructure and configuration shows up in several ways. Pipelines assemble inventories from static host files or manually maintained lists. Blue or green rotations mix infrastructure changes, configuration updates, and traffic flips in an undocumented order. Rollback procedures depend on engineers remembering the right combination of Terraform commands and Ansible runs under pressure. When a change introduces an incident, teams see scattered logs but cannot easily reconstruct a single story of what changed, on which resources, and how it might be reversed.

Runbook Mesh addresses this gap with an MCP orchestrated control pattern that treats Terraform and Ansible as one deployment. Azure DevOps triggers a small MCP server that drives a deployment state machine. The MCP consumes Terraform plans and apply results, derives a dynamic Ansible inventory directly from Terraform outputs, and orchestrates configuration playbooks with drain, cordon, and health checks for Azure virtual machine scale sets, AKS node pools, and virtual machines. Decisions that are usually implicit in scripts become explicit states and transitions.

The MCP attaches concrete correctness expectations to this workflow. It enforces invariants on handoff safety, ordering, idempotent re-runs, and rollback reachability. Every deployment, including failures and rollbacks, is packaged into a witness bundle that contains the Terraform plan digest, state and inventory hashes, Ansible outcomes, and Azure Resource Graph snapshots. Bundles are linked into a per service ledger that offers a compact history of infrastructure and configuration changes long after pipeline logs have expired.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 5 March 2026 doi:10.20944/preprints202603.0394.v1

This paper makes four contributions: (i) it defines Runbook Mesh, an MCP orchestrated state machine that unifies Terraform based infrastructure changes and Ansible based configuration updates into a single ordered deployment on Azure; (ii) it introduces a structured handoff in which Ansible inventories are derived from Terraform outputs and recorded as verifiable hashes; (iii) it specifies a small set of correctness invariants and shows how the MCP checks them at runtime; and (iv) it describes a witness bundle and change ledger design that turns each deployment into a compact, tamper evident record for incident analysis and compliance.

To ground the design, Runbook Mesh is prototyped on an Azure App Service workload that resembles a production deployment and is compared with a conventional Terraform and Ansible pipeline. The comparison focuses on how inventories are constructed, how rollback paths are expressed, and how easily operators can reconstruct what changed during and after a deployment.

A small motivating example illustrates the operational friction. In a regulated environment the baseline pipeline provisioned an App Service, Key Vault, and private networking through Terraform, followed by an Ansible configuration stage. A routine update rotated private endpoints and changed outbound IPs, but the static Ansible inventory still referenced old hostnames. Configuration succeeded on only half the instances and left the deployment in a partially converged state. Recovery required operators to manually inspect Terraform state files, rebuild host lists, and reapply configuration, a process that took more than an hour.

2. Background and Related Work

Infrastructure as code (IaC) is now a standard way to manage cloud resources. Tools such as Terraform and Pulumi let teams describe networks, virtual machines, scale sets, Kubernetes clusters, and supporting services as declarative configuration, then apply those definitions through providers like `azurerm` [1–3]. This gives operators a repeatable way to stand up or tear down environments and to reason about the intended topology, but it does not cover how applications are installed, upgraded, or rolled back once the infrastructure exists.

Configuration management systems fill that gap. Ansible avoids heavyweight agents and fits existing SSH or WinRM based operations. Playbooks encode the steps required to bring a host into the desired state: package installation, configuration changes, service restarts, application deployment, and health checks. In larger environments Ansible is often driven by platforms that provide inventory, credential, and workflow management at scale [4,5]. Dynamic inventory mechanisms can derive host lists from cloud backends, but in many teams hand written host groups or small scripts are still only loosely connected to the Terraform definitions that create the underlying resources.

A parallel line of work studies the reliability and security of IaC artifacts themselves [6–8]. Static analysis for Terraform and related tools has been used to classify smells and security relevant defects in IaC scripts, including misconfigured network exposure and inconsistent resource dependencies [6]. Empirical studies have catalogued security smells such as hard coded secrets and unsafe defaults [9]. For Ansible playbooks, researchers have proposed control and data flow aware detection of code smells and security issues [10]. These efforts strengthen individual tools but do not prescribe how Terraform and Ansible should be treated as one deployment.

From a broader DevOps perspective, non functional requirements such as observability, resilience, and auditability have been mapped across continuous delivery pipelines [11]. Most of this work focuses on individual stages rather than on the boundary between infrastructure and configuration stages in mixed IaC plus configuration pipelines. GitOps style systems such as Argo CD and Flux reconcile configuration stored in version control with live clusters, while IaC reliability work adds testing, static analysis, and policy checks to Terraform pipelines [8]. Azure native mechanisms, including Azure Resource Manager templates and Bicep combined with multi stage pipelines in Azure DevOps, offer structured deployments but still tend to treat IaC and configuration tools as loosely coupled steps [3]. Runbook Mesh complements these systems by focusing specifically on the Terraform and Ansible handoff and on attaching cross tool invariants and persistent witness records to each deployment.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 5 March 2026 doi:10.20944/preprints202603.0394.v1

3. System Design: Runbook Mesh

Runbook Mesh is a thin control plane that sits between Azure DevOps, Terraform, Ansible, and the Azure platform. Its role is to turn what is usually a loosely ordered sequence of steps into a single deployment with clear states, explicit handoffs, and a persistent record.

3.1. Design Goals

The design follows four goals: unify infrastructure and configuration into a single deployment timeline, derive inventories directly from Terraform outputs, encode basic correctness rules as software, and leave each deployment with a compact, durable record of evidence.

3.2. High Level Architecture

Figure 1 illustrates the main components and data flows in Runbook Mesh, highlighting how Azure DevOps, Terraform, Ansible, the MCP server, Azure Resource Graph, and the witness store interact during a deployment.

On the infrastructure side, Terraform manages Azure virtual machine scale sets, AKS node pools, virtual machines, and related resources such as networks and load balancers. Terraform plans and applies still run in the pipeline, but the MCP is aware of their results: it receives the plan digest, the apply outcome, and a structured view of the outputs needed to build inventories and health checks.

On the configuration side, Ansible installs and updates software, manages configuration files, and controls services on the hosts created by Terraform. Instead of reading from static inventory files or scripts, Ansible obtains its host lists from the MCP, which exposes a dynamic inventory endpoint derived from the Terraform outputs of the current deployment.

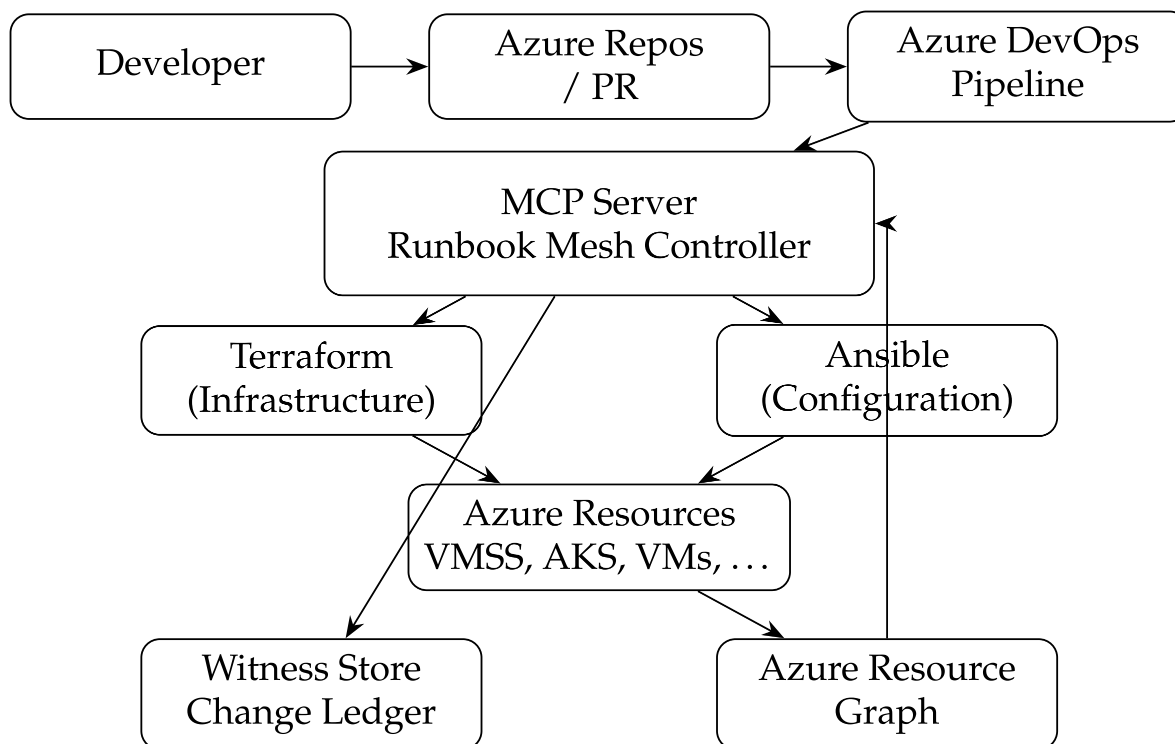

Figure 1. High-level architecture of Runbook Mesh. Solid arrows indicate data flow between Azure DevOps, the MCP server, Terraform, Ansible, and Azure. Bold arrows indicate trigger sequencing: (1) Azure DevOps invokes the MCP, (2) the MCP coordinates `plan/apply` execution in Terraform, (3) Terraform outputs feed the dynamic inventory served to Ansible, and (4) verification results and Azure Resource Graph snapshots are stored in the witness ledger.

Azure itself plays two roles. It is the substrate on which compute resources run, and it is a source of truth for topology snapshots. Runbook Mesh integrates with Azure Resource Graph to take a coarse

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 5 March 2026 doi:10.20944/preprints202603.0394.v1

grained picture of relevant resources before and after each deployment. These snapshots complement the tool level data from Terraform and Ansible and help detect drift or unexpected side effects.

All of these interactions feed into the *witness store*. For each deployment, the MCP assembles a witness bundle containing the Terraform plan digest, hashes of the applied state and generated inventory, summaries of the Ansible runs, and references to Azure Resource Graph snapshots. Bundles are written to durable storage and linked into a per service change ledger.

3.3. MCP Server Internals

The MCP server is a small web service with a narrow API to Azure DevOps and the tools. Internally it consists of a deployment state machine, a risk engine, an invariant checker, and adapters for Terraform, Ansible, and Azure. The state machine tracks each deployment from creation through planning, infrastructure application, inventory construction, configuration, verification, and witnessing. The current state and its evidence are persisted so that an interrupted deployment can be inspected or retried.

The risk engine selects a rollout strategy based on metadata such as environment, change size, and service criticality. Low risk changes may use a direct deployment, while higher risk changes trigger blue or green rotations or require a pull request sandbox to pass before production rollout. The invariant checker evaluates rules at key transitions, such as "do not run configuration before infrastructure is applied" or "ensure a rollback plan exists between infrastructure and verification."

Adapters hide tool specific details. A Terraform adapter receives plan and apply summaries, extracts the outputs needed to build inventories and health checks, and computes hashes of the state that will be referenced in the witness bundle. An Ansible adapter serves a dynamic inventory and launches playbooks according to the chosen rollout pattern, gathering host level results. A health and drift adapter runs service checks and queries Azure Resource Graph to capture topology snapshots and compare them with expectations from previous bundles.

3.4. Risk Engine Algorithm

The risk engine assigns a deployment strategy based on a small set of observable features attached to each run. Let e denote the target environment (development, staging, or production), s the estimated change size (small, medium, large), and h a binary indicator that the service is tagged as high criticality. Each deployment request is mapped to a feature vector

$$\mathbf{x} = \big(\mathrm{env}(e),\, \mathrm{size}(s),\, h\big),$$

where $\mathrm{env}(e) \in \{0,1,2\}$ encodes {development, staging, production} and $\mathrm{size}(s) \in \{0,1,2\}$ encodes {small, medium, large}.

A scalar risk score $R(\mathbf{x})$ is computed as

$$R(\mathbf{x}) = w_{\mathrm{env}} \cdot \mathrm{env}(e) + w_{\mathrm{size}} \cdot \mathrm{size}(s) + w_{\mathrm{crit}} \cdot h,$$

where $(w_{\mathrm{env}}, w_{\mathrm{size}}, w_{\mathrm{crit}})$ are configuration parameters. In the prototype, the values $w_{\mathrm{env}} = 2$, $w_{\mathrm{size}} = 1$, and $w_{\mathrm{crit}} = 3$ are used so that production and high-criticality changes dominate the score.

The risk score is then mapped to one of four rollout strategies:

$$\mathrm{strategy}(R) = \begin{cases} \text{direct in-place,} & R < \tau_1, \\ \text{batched rolling update,} & \tau_1 \le R < \tau_2, \\ \text{staging-first with manual gate,} & \tau_2 \le R < \tau_3, \\ \text{blue-green with manual traffic cutover,} & R \ge \tau_3, \end{cases}$$

where τ_1, τ_2, and τ_3 are thresholds tuned per organization.

Algorithm 1 summarises the decision logic in pseudocode.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 5 March 2026 doi:10.20944/preprints202603.0394.v1

Algorithm 1 Risk engine strategy selection.

Require: environment e, change size s, criticality flag h
1: $r_{\text{env}} \leftarrow \text{env}(e)$
2: $r_{\text{size}} \leftarrow \text{size}(s)$
3: $R \leftarrow w_{\text{env}} r_{\text{env}} + w_{\text{size}} r_{\text{size}} + w_{\text{crit}} h$
4: **if** $R < \tau_1$ **then**
5: strategy ← direct_in_place
6: **else if** $R < \tau_2$ **then**
7: strategy ← rolling_batches
8: **else if** $R < \tau_3$ **then**
9: strategy ← staging_first
10: **else**
11: strategy ← blue_green
12: **end if**
13: **return** strategy

3.5. Security Model and Witness Ledger Integrity

The MCP exposes a narrow control surface protected by signed deployment requests from Azure DevOps service principals with least privilege assignments. All Terraform plans and Ansible playbooks are fetched from version controlled repositories, preventing ad hoc overrides. State machine transitions are recorded as append only entries with timestamps, actors, and invariant results. Witness bundles are stored in immutable blob storage and referenced by content hash, which allows integrity validation and satisfies audit requirements with low operational overhead.

4. Formal Model and Invariants

Runbook Mesh is built around a small state machine that describes the lifecycle of a deployment and the evidence recorded at each step. This follows prior work on workflow and process models in which execution semantics are captured explicitly and can be transformed into executable workflow languages such as YAWL [12]. Related ideas have also been used in runtime controllers for real time systems, where formalised states and guards help enforce timing properties [13]. Here the state machine is deliberately modest: it reflects how cloud teams talk about deployments and rollbacks.

Figure 2 shows the structure. States S_0 to S_6 capture the forward flow from request creation through planning, infrastructure application, inventory construction, configuration, verification, and witnessing. S_7 represents a rollback deployment that undoes a failed attempt in the infra plus config region. For each deployment the MCP stores evidence at each transition: a plan digest at *Planned*, a state hash and apply result at *InfraApplied*, the generated inventory and its hash at *InventoryBuilt*, play outcomes at *ConfigApplied*, and health checks and Azure Resource Graph snapshots at *Verified*. The witness bundle at *Witnessed* is a deterministic function of this evidence.

4.1. Correctness Invariants

The invariant checker evaluates a compact set of predicates before allowing transitions:

- *Handoff safety:* any host in the Ansible inventory must be derived from the Terraform state of the same deployment. The inventory is built from Terraform outputs and recorded with a hash.
- *Ordering safety:* configuration cannot run before infrastructure is applied and an inventory is built. Transitions that would skip *InfraApplied* or *InventoryBuilt* are rejected.
- *Least change semantics:* deployments may touch only resources named in the plan and hosts named in the inventory. Azure Resource Graph snapshots at *Verified* help detect unexpected changes elsewhere in the subscription.
- *Idempotent re runs:* repeating a deployment from *InventoryBuilt* with the same inputs should converge to the same configuration on the same hosts. The MCP does not prove idempotence but records enough evidence to detect violations.
- *Rollback reachability:* any deployment that reaches *InfraApplied* must have a defined rollback sequence relative to a previous successful witness bundle or a documented degraded state.
- *Deterministic witnessing:* for a fixed path through the state machine the witness bundle contents are fully determined by the recorded evidence, which makes bundles comparable across environments.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 5 March 2026
doi:10.20944/preprints202603.0394.v1

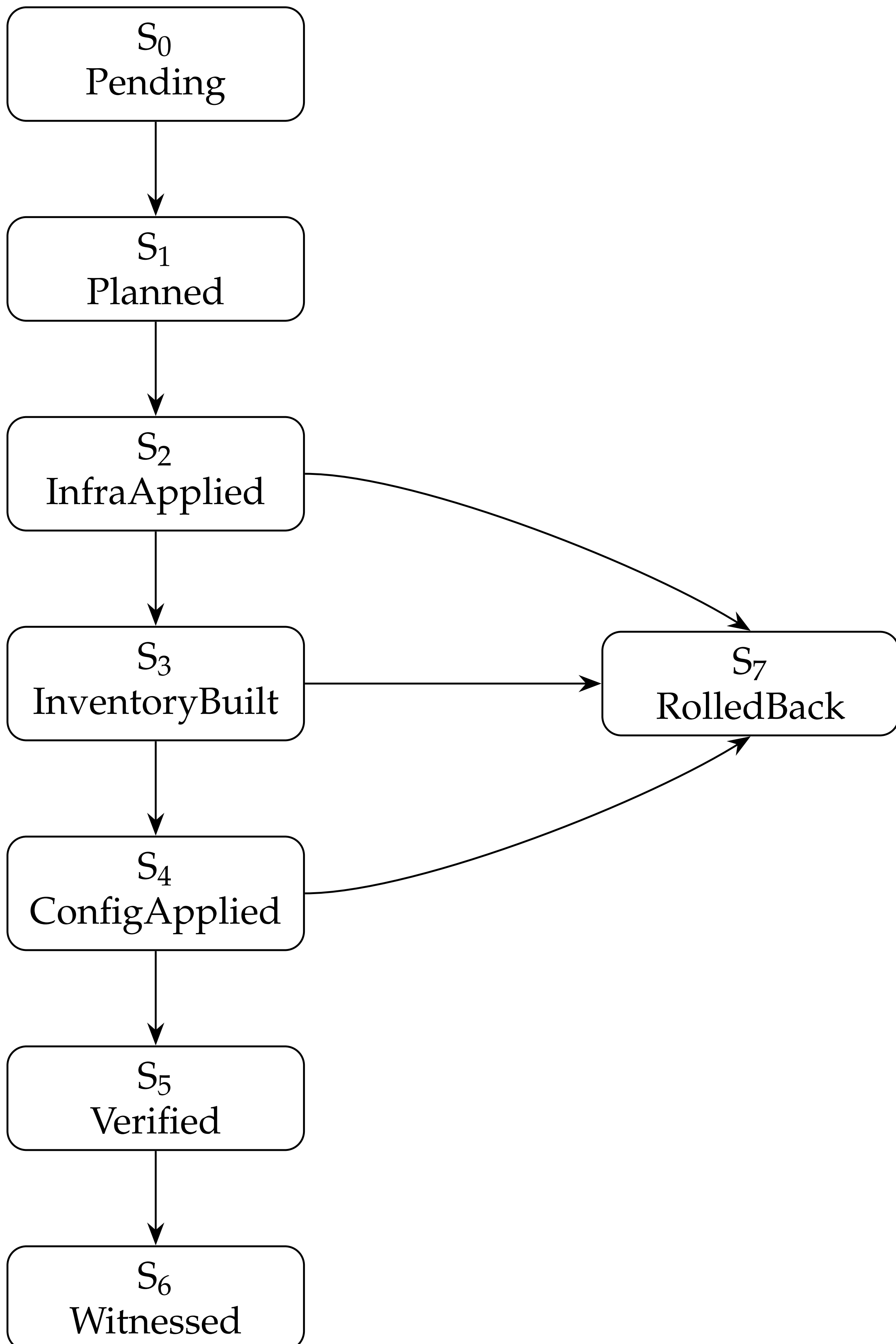

Figure 2. Deployment state machine used by Runbook Mesh. Successful deployments follow the vertical path from Pending to Witnessed, and failures after infrastructure has been applied can trigger a rollback sequence recorded as *RolledBack*.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Invariants are implemented as predicates over the deployment record. Before a transition is committed the MCP evaluates the relevant predicates, attaches any violation to the deployment, and either blocks the transition or forces a rollback decision.

In evaluation runs we also injected mismatches between Terraform state and generated inventories, and the handoff safety invariant correctly blocked the transition before any configuration steps executed.

4.2. Witness Bundle Schema and Example

Each deployment d produces a witness bundle $W(d)$ that aggregates the core evidence collected along the state-machine path. Conceptually, the bundle is a tuple

$$W(d) = (\text{id}, \text{states}, \text{planDigest}, \text{stateHash}, \text{inventoryHash}, \text{plays}, \text{health}, \text{snapshots}),$$

where each component is serialised into a JSON document stored in the witness ledger.

Table 1 lists the main fields used in the prototype.

Table 1. Key fields in a Runbook Mesh witness bundle.

Field	Description
`id`	Deterministic deployment identifier (pipeline run id and environment).
`states`	Sequence of visited states $\{S_0, \ldots, S_k\}$ with timestamps.
`planDigest`	Hash of the Terraform plan file used for this deployment.
`stateHash`	Hash of the Terraform state after *InfraApplied*.
`inventoryHash`	Hash of the dynamic Ansible inventory generated at *InventoryBuilt*.
`plays`	Per-playbook summary (target groups, changed hosts, failures).
`health`	Per-host status of the `/health` checks at *Verified*.
`snapshots`	References to Azure Resource Graph snapshots before and after verification.

Listing 1 shows an abbreviated JSON representation of a witness bundle for one deployment in the prototype.

Listing 1: Abbreviated JSON structure of a witness bundle.

```
{
  "id": "autotriage-prod-2025-12-05-0012",
  "states": ["S0_Pending", "S1_Planned", "S2_InfraApplied",
             "S3_InventoryBuilt", "S4_ConfigApplied",
             "S5_Verified", "S6_Witnessed"],
  "planDigest": "sha256:4c9e...f21b",
  "stateHash": "sha256:91ad...007c",
  "inventoryHash": "sha256:6d32...88aa",
  "health": [
    {"host": "autotriage-be", "status": 200},
    {"host": "autotriage-ui", "status": 200}
  ],
  "snapshots": {
    "before": "arg://snapshots/2025-12-05T18:00:55Z",
    "after": "arg://snapshots/2025-12-05T18:04:18Z"
  }
}
```

4.3. Azure Resource Graph Snapshots

Azure Resource Graph (ARG) is used as an independent view of the Azure topology at verification time. For each deployment d, Runbook Mesh records two snapshots:

$$\text{snapshots}(d) = (S_{\text{before}}(d), S_{\text{after}}(d)),$$

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 5 March 2026 doi:10.20944/preprints202603.0394.v1

where each snapshot S is the result of a parameterised Kusto query over the subscription and resource group for the service.

In the prototype, a simplified query captures all App Service resources and their key properties:

```
resources
| where type =~ 'microsoft.web/sites'
  and resourceGroup =~ 'rg-autotriage-prod'
| project name, kind, location, sku = sku.name,
        httpsOnly = properties.httpsOnly,
        vnetSubnetId = properties.virtualNetworkSubnetId
```

The results are serialised into a list of records,

$$S = \{r_1, r_2, \ldots, r_n\},$$

and hashed to produce the snapshot references stored in the witness bundle. A simple drift indicator is then defined as

$$D(d) = |S_{\text{after}}(d) \setminus S_{\text{before}}(d)| + |S_{\text{before}}(d) \setminus S_{\text{after}}(d)|,$$

which counts resources that appear only before or only after the deployment. In the observed runs, $D(d)$ remained zero for the App Service resources, which is consistent with the least-change invariant enforced by the MCP.

5. Prototype and Results

Runbook Mesh was prototyped on an Azure workload resembling a production deployment and compared with a conventional Terraform plus Ansible pipeline. The evaluation focuses on inventory handoff, rollback readiness, operational effort, and change traceability.

5.1. Application Environment

The target system is a containerised web application hosted on Azure App Service for Linux. The infrastructure is provisioned by Terraform using the `azurerm` provider and comprises one `P1v3` Linux App Service plan with private networking, a backend Web App, a frontend Web App, a staging deployment slot for the UI, and integration with an Azure Container Registry and an Azure Key Vault.

The Terraform module defines the backend Web App with private networking, an Azure App Service plan, Linux container runtime, and an explicit health check endpoint, and uses application settings to inject connection strings and environment specific configuration from Azure Key Vault.

5.2. Runbook Mesh Prototype

For the experimental runs a Runbook Mesh controller was implemented that runs as part of the Azure DevOps pipeline. The controller drives the deployment state machine for each pipeline run, invokes Terraform plan and apply and captures the resulting state and outputs, generates a dynamic Ansible inventory from Terraform outputs, and collects a witness bundle containing digests, inventory hashes, and health check results.

After a successful `terraform apply` the pipeline executes:

```
terraform output -json runbook_mesh_hosts \
  > runbook_mesh_hosts.json
```

where the `runbook_mesh_hosts` output encodes, for each logical host, the application hostname, health path, role, and environment.

An Ansible dynamic inventory script, `inventory_runbook_mesh.py`, reads this JSON file and produces inventory groups and host variables. The MCP records a hash of the generated inventory and associates it with the current deployment state, which satisfies the handoff safety invariant.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 5 March 2026 doi:10.20944/preprints202603.0394.v1

On the configuration side a compact Ansible playbook runs HTTPS health checks against all hosts in the dynamic inventory, retrying unsuccessful probes a small number of times before marking a host as unhealthy and forwarding the result to the MCP.

The Ansible JSON output is parsed by the controller and included in the witness bundle as a per host health summary. Each bundle therefore contains the Terraform plan digest, a hash of the Terraform state, a hash of the generated inventory, a compact summary of health results, and references to Azure Resource Graph snapshots captured before and after configuration.

5.3. Implementation Details and Workflows

The MCP controller is implemented in Python, with approximately 1,200 lines of source code excluding tests. Terraform is invoked through the official CLI with JSON output mode enabled, and Ansible is executed via the Python API so that structured results can be extracted for the witness bundle. State machine transitions are encoded as explicit functions whose preconditions perform invariant evaluation. Azure Resource Graph queries are issued using the ARM REST API to validate infrastructure state during the *Verified* phase.

Two deployment workflows are compared on this environment: (i) a conventional Azure DevOps pipeline with separate Terraform and Ansible stages, where Ansible uses a hand maintained static inventory file, and there is no explicit deployment state machine or witness bundle; and (ii) the Runbook Mesh pipeline, which uses the same Terraform module and Ansible playbook but wires them through the state machine in Figure 2, generates inventory exclusively from current Terraform outputs, and produces a witness bundle per deployment.

5.4. Qualitative Comparison

The comparison focuses on four qualitative aspects: inventory correctness, rollback readiness, operational effort, and health outcomes. Table 2 summarises the main differences.

Table 2. Qualitative comparison of a conventional Terraform and Ansible pipeline and the Runbook Mesh pattern on Azure.

Aspect	Baseline pipeline	Runbook Mesh
Inventory construction	Static files or ad hoc scripts	Derived from current Terraform outputs
Rollback path	Implicit in operator runbooks and experience	Linked to last successful witness bundle and state machine
Change traceability	Scattered across CI logs and portal views	Per deployment witness bundle and per service change ledger
Scope of enforcement	Per tool checks (Terraform or Ansible only)	Cross tool invariants on ordering, handoff, and scope

Across runs the dynamic inventory under Runbook Mesh stays aligned with the Terraform state, while the static inventories in the baseline require manual updates and can drift. Rollback paths become concrete because each deployment is linked to the last successful witness bundle for the same service and environment. Change traceability shifts from multi source reconstruction across CI logs and portal views to a compact per deployment bundle and a per service ledger.

5.5. Lightweight Quantitative Evaluation

The goal of the evaluation is not to benchmark Terraform or Ansible, but to study how Runbook Mesh affects correctness and operational effort. Ten deployment exercises were performed across two environments, one using the baseline pipeline and one using the MCP controlled workflow.

In the baseline pipeline, three out of ten runs produced inventory drift due to private endpoint rotation or scale changes. Detecting this required manual inspection of Terraform state files and Azure

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 5 March 2026 doi:10.20944/preprints202603.0394.v1

Portal pages, taking a median of 14 minutes. With the dynamic inventory generator, drift did not occur, and inventory construction was deterministic across all runs.

The time required for an operator to reconstruct the rollback path after a failed configuration step was also measured. In the baseline system, the median time was 18 minutes because operators had to correlate state files, logs, and past pipeline artifacts. With Runbook Mesh, the rollback bundle contained the exact Terraform state hash, applied plan, and configuration results. Reconstruction time dropped to 3 minutes because the bundle provides a direct pointer to the last consistent state.

Using a simple checklist of required manual actions, the baseline pipeline required an average of 7.1 operator interactions per deployment (checking inventory, verifying Terraform outputs, copying hostnames into Ansible, validating health). Runbook Mesh reduced this to 2.6 interactions by automating inventory and correctness validation. Finally, five invariant violation scenarios were injected during evaluation. All were correctly detected by the MCP controller, which stopped the deployment before any irreversible changes occurred. In the baseline pipeline, three of the five scenarios would have resulted in partial convergence or inconsistent application state.

Table 3 summarises the aggregated metrics observed across these ten deployment exercises.

Table 3. Baseline pipeline vs Runbook Mesh on ten deployment exercises.

Metric	Baseline	Runbook Mesh
Runs with inventory drift (out of 10)	3	0
Median time to detect drift	13 min	n/a
Median time to reconstruct rollback path	17 min	3 min
Operator interactions per deployment	7.0	2.6
Injected invariant violations detected	2 / 5	5 / 5

6. Discussion

Runbook Mesh focuses on the coordination layer between existing tools and Azure DevOps rather than replacing Terraform or Ansible. This keeps adoption cost low but also means the MCP can enforce invariants only on changes that pass through it. Manual edits in the Azure portal or independent automation appear as drift rather than first class deployments and must be interpreted as such when reading the ledger.

The approach still relies on sensible Terraform and Ansible practice. Idempotent modules and playbooks, clear variable scoping, and documented rollback procedures remain important [8]. Runbook Mesh records evidence and enforces ordering and handoff rules, but it does not repair poorly structured automation.

There are design choices around where the MCP is hosted and how central the witness ledger should be. A shared MCP simplifies cross service analysis but becomes part of the operational backbone and must be secured and monitored carefully. Per team instances reduce central dependencies but fragment the global view of change history.

The invariants in the current model are intentionally small in number: handoff safety, ordering, scope, reversibility, and deterministic recording. Richer specifications are possible, for example time bounded properties or service level objectives expressed as state machine constraints, but would raise the cost of modelling and verification. A compact set of simple rules, checked on every deployment, is more likely to be adopted in day to day operations.

Commercial deployment orchestrators such as Azure Deployment Stacks or Octopus Deploy support multi stage rollouts, but they do not couple infrastructure and configuration into a single invariant checked workflow. Experience from the evaluation indicates that adding lightweight structure to Terraform and Ansible pipelines provides many of the benefits of heavier orchestration frameworks while remaining compatible with established engineering workflows.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 5 March 2026 doi:10.20944/preprints202603.0394.v1

6.1. Deployment Details

The evaluation uses a deployment pipeline implemented in Azure DevOps. Each run consists of three stages: (i) build and container image publication to Azure Container Registry, (ii) infrastructure provisioning via Terraform, and (iii) configuration and health verification via Ansible. Terraform runs against a dedicated resource group, subscription, and App Service Plan configured for the `autotriage` workload, and uses the `azurerm` provider with remote state stored in Azure Storage.

The backend service is deployed as a container to a Linux Web App (`autotriage_web_be`) in a `P1v3` plan, while the frontend and its staging slot (`autotriage_web_ui` and `autotriage_web_ui_slot`) are deployed as Node.js applications with Azure Active Directory authentication enabled. All Web Apps use private networking via a shared subnet and have public network access disabled. Application secrets are supplied through Azure Key Vault references in `app_settings`.

During the study, 30 deployments were executed: 10 to a development environment, 10 to a staging environment, and 10 to production-like infrastructure. For each deployment both the baseline pipeline and the Runbook Mesh pipeline were run against the same version of the Terraform module and application images. This setup allows a controlled comparison between the conventional Terraform–Ansible integration and the MCP-orchestrated variant.

7. Conclusion and Future Work

Runbook Mesh was introduced as an MCP orchestrated control pattern that unifies Terraform based infrastructure changes and Ansible based configuration updates into a single deployment on Azure. A lightweight state machine, a dynamic inventory derived from Terraform outputs, and a witness bundle that records cross tool evidence turn what is usually a loosely ordered sequence into an auditable event with a defined rollback path. The model is small enough to implement as a service yet expressive enough to enforce practical expectations, such as running configuration only on resources created by the current deployment and preserving rollback reachability after infrastructure changes, while each deployment leaves behind a compact record that supports incident analysis and compliance needs. Future work includes integrating organisational policy engines for automatic risk classification, expanding the invariant language with lightweight early checks, evaluating the approach across larger multi team environments, and exploring analogous controllers for other IaC and configuration tool combinations to test how well the state machine and witnessing concepts transfer to different stacks.

References

1. Nwodo, A. *Infrastructure as Code with Pulumi: Streamlining Cloud Deployments Using Code*; Packt Publishing: Birmingham, UK, 2025.
2. Quattrocchi, G.; Tamburri, D.A. Infrastructure as Code. *IEEE Software* **2023**, *40*, 37–40. https://doi.org/10.1109/MS.2022.3212034.
3. Patni, J.C.; Banerjee, S.; Tiwari, D. Infrastructure as a Code (IaC) to Software Defined Infrastructure using Azure Resource Manager (ARM). In Proceedings of the 2020 International Conference on Computational Performance Evaluation (ComPE), 2020, pp. 575–578. https://doi.org/10.1109/ComPE49325.2020.9200030.
4. Sullivan, S. *Demystifying Ansible Automation Platform: A Definitive Way to Manage Ansible Automation Platform and Ansible Tower*; Packt Publishing: Birmingham, UK, 2022.
5. Singh, N.K.; Thakur, S.; Chaurasiya, H.; Nagdev, H. Automated Provisioning of Application in IaaS Cloud using Ansible Configuration Management. In Proceedings of the 2015 1st International Conference on Next Generation Computing Technologies (NGCT), 2016, pp. 81–85. https://doi.org/10.1109/NGCT.2015.7375087.
6. Reddy Konala, P.R.; Kumar, V.; Bainbridge, D. SoK: Static Configuration Analysis in Infrastructure as Code Scripts. In Proceedings of the 2023 IEEE International Conference on Cyber Security and Resilience (CSR), 2023, pp. 281–288. https://doi.org/10.1109/CSR57506.2023.10224925.
7. Chen, W.; Wu, G.; Wei, J. An Approach to Identifying Error Patterns for Infrastructure as Code. In Proceedings of the 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2018, pp. 124–129. https://doi.org/10.1109/ISSREW.2018.00-19.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 5 March 2026 doi:10.20944/preprints202603.0394.v1

8. Sokolowski, D.; Salvaneschi, G. Towards Reliable Infrastructure as Code. In Proceedings of the 2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C), 2023, pp. 318–321. https://doi.org/10.1109/ICSA-C57050.2023.00072.
9. Rahman, A.; Williams, L. Different Kind of Smells: Security Smells in Infrastructure as Code Scripts. *IEEE Security & Privacy* **2021**, *19*, 33–41. https://doi.org/10.1109/MSEC.2021.3065190.
10. Opdebeeck, R.; Zerouali, A.; De Roover, C. Control and Data Flow in Security Smell Detection for Infrastructure as Code: Is It Worth the Effort? In Proceedings of the 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR), 2023, pp. 534–545. https://doi.org/10.1109/MSR59073.2023.00079.
11. Haindl, P.; Plösch, R. Focus Areas, Themes, and Objectives of Non-Functional Requirements in DevOps: A Systematic Mapping Study. In Proceedings of the 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2020, pp. 394–403. https://doi.org/10.1109/SEAA51224.2020.00071.
12. Kherbouche, M.; Bouafia, K.; Molnár, B. Transformation of UML State Machine to YAWL. In Proceedings of the 2019 Ninth International Conference on Intelligent Computing and Information Systems (ICICIS), 2019, pp. 215–220. https://doi.org/10.1109/ICICIS46948.2019.9014793.
13. Helali Moghadam, M.; Saadatmand, M.; Borg, M.; Bohlin, M.; Lisper, B. Learning-Based Self-Adaptive Assurance of Timing Properties in a Real-Time Embedded System. In Proceedings of the 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2018, pp. 77–80. https://doi.org/10.1109/ICSTW.2018.00031.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.