WHITEPAPER

Infrastructure as Code

How teams can leverage modern engineering practices to deploy immutable infrastructure on Azure

INTRODUCTION

Infrastructure has traditionally run on-premise using fragile bare metal servers, with heroics performed by seasoned ops engineers through endless patching and unique configuration. Few things slow down software delivery more than infrastructure needs. In the modern world of software development, technology advances have produced:

- Virtualized infrastructure with VMs, containers, or even “serverless”
- Cloud computing for scalable, reliable infrastructure
- Managed infrastructure services like IaaS and PaaS

Teams can take advantage of these capabilities by adopting infrastructure as code (IaC). By leveraging IaC principals, infrastructure is handled with a defined software development lifecycle (SDLC) where changes are made safely and easily.

Liatrio developed and implemented a robust SDLC process to define, test, and deploy infrastructure. This pattern allows teams to confidently deploy their infrastructure on the Azure cloud platform. This whitepaper goes through what teams need for creating IaC:

- Adopting IaC principals and SDLC workflow
- Utilizing industry standard Terraform as an IaC orchestrator tool
- Managing infrastructure deployments with ADO Pipelines
- Leveraging Microsoft Azure ecosystem with ADO as the centralized SDLC tool suite and Azure Cloud for infrastructure deployments

MODERN APPROACH TO MANAGING INFRASTRUCTURE

We treat our infrastructure as a product, just like our applications. Modern practices to managing infrastructure should follow the same pattern as application development, including:

- Everything as code
- All changes are code reviewed
- Code-level and functional testing
- Fast feedback loops
- Automated delivery and controls with pipelines

Infrastructure as Code

Infrastructure as Code is a paradigm for managing infrastructure resources' life cycle in the same way that the application code life cycle is managed. This involves defining all resources within the environment in a declarative language, like Terraform, and using a version control system, like Git, and managing changes to the infrastructure through SDLC processes. Creation, deletion and modification of every resource in an environment is then represented as a change to the code within the version control system. This is similar to the process of managing the software delivery life cycle of a software application.

IaC offers the ability to test, track and automate the life cycle of cloud resources in a consistent and repeatable way. The ability to test, validate and automate changes to the infrastructure leads to a reduction in unexpected failures within the provisioning and change management of your infrastructure. Because changes to the infrastructure live in source control, it allows the opportunity to use tools like pull requests and approvals to make changes to the code, and therefore the infrastructure. Using strong engineering practices and automation, changes to the infrastructure become traceable back to a specific change in the code.

Immutable Infrastructure

No changes or configurations should happen outside of the source code and SDLC. Immutability should be taken seriously and no changes should be made outside of the initial desired state of delivered infrastructure. All changes should be made via code so the infrastructure's state being tracked remains accurate to avoid configuration drift.

Terraform for IaC

Terraform is our preferred tool to orchestrate our infrastructure using code. Terraform allows infrastructure to be expressed as code and allows us to track and maintain the state changes to infrastructure so that changes can be made only as needed and configuration drift is minimized or eliminated.

Centralized SDLC

Using the Azure DevOps (ADO) platform, we are able to manage and orchestrate the entire lifecycle of infrastructure. ADO's tool suite enables a complete CI/CD pipeline to deploy infrastructure and a single location to track work, code, and deployments.

SOFTWARE ENGINEERING PRACTICES FOR IAC

Certain prerequisites and process changes are required in order to enable a successful IaC workflow. Below we describe several technical processes to assist with this, however it is worth noting that changes with respect to working norms and team interactions are equally important as they create the social contracts needed to maintain IaC over time.

Managing IaC with Git

ADO Repos provides a fully featured Git-based implementation for version control of code. All IaC code should live under version control, allowing the code to be shared, reviewed and tracked.

Utilizing Git enables peer review and approval processes for change management of infrastructure through branching and merging. Developers have the ability to checkout the code locally, create a branch and make changes to the code. When changes are ready to be introduced into the environment, a pull request is opened for peer review and approval. After approval is received by the required parties involved, the change can then be validated and introduced into the deployment process. This workflow is very common in software delivery.

Automated Delivery with Pipelines

The full benefits of IaC are realized when the process of validating, testing and deploying infrastructure is codified and automated via pipelines. Azure DevOps provides a mechanism for defining tasks that should be performed on a code repository and the frequency at which these tasks should be performed through Azure Pipelines. Pipelines provide fast feedback to developers when the changes that are committed to Git do not work as expected or do not pass validation tests.

By creating Pipelines as Code in Azure DevOps, we are able to store all steps in code so we know when the process of building or testing our infrastructure or application changes. Using Azure DevOps Pipelines and a good branching model, we are able to automate these simple verification tasks so that errors can be caught early in the process and before the change is introduced into the main code base.

Static Code Analysis/Linting

Linting is a common example of a very inexpensive check, which can be very expensive if not caught early in the process. A linter is a tool that checks to ensure that code is structured properly but it does not check the functionality of the code and can typically be run with a single command in a few seconds. If a linting problem is introduced into the codebase it may cause problems at deployment time, at which time the approval and change process must be started again from the beginning to fix the linting problem. To ensure that linting errors are not introduced into the codebase, a pipeline should be used to run the linter against every change that is made to the code. If the linter fails, the pipeline should fail and prevent the code from moving forward in the approval process.

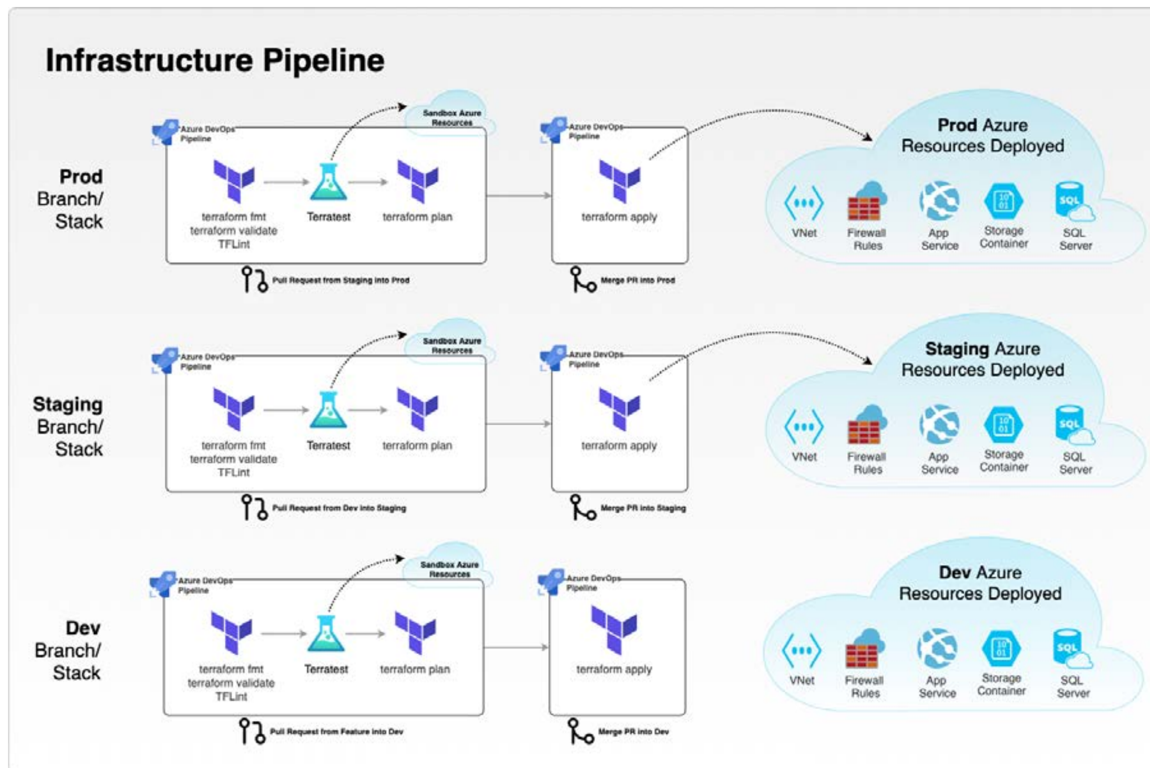
Testing & Validation

In software development, code requires validation and testing to ensure that the software will run properly and as expected. The same is true with IaC. Some validation tasks are more expensive than others because they may take more time or use more resources. Pipelines can be used to define that these more expensive tests are run less frequently, but always before code is introduced into the codebase and prepared to be deployed.

Environment Management with GitOps

This effort seeks to apply the GitOps approach to replace legacy deployment tools with manifest files in a Git repository and deployment pipeline in Azure DevOps. This approach uses Git to maintain the definition of the desired state of each environment, the path to request and approve changes, as well as a log of changes in the commit history.

GitOps uses Git commits as the mechanism for defining an environment change as well as triggering of the change. In this model, developers trigger changes to environments by merging pull requests into the IaC deploy repository. Changes from a developer's branch should only be merged into dev. To promote changes to higher environments PRs should be created from dev to staging and then from staging to prod. Developers should never commit directly to an environment branch, nor should they create a PR from their own branch to a branch other than dev.



IAC DEPLOY REPOSITORY

The infrastructure code that defines each environment lives in a single Git repository. In this repo, the Terraform code that describes the infrastructure is contained inside a **terraform** directory with environment specific variables stored within separate *.tfvars* files which influence environment configurations. . A deeper dive into recommendations and set up for Terraform in a deploy repository and pipeline will be explored in a separate Liatrio publication.

Contents & Structure

- The pipeline code for deploying IaC (ex: azure-pipelines.yml)
- The Terraform that describes the environment (ex: terraform/main.tf)
- Environment-specific values to be passed to Terraform (ex: dev.tfvars, staging.tfvars, prod.tfvars)
- Branches that represent each environment deployment (ex: dev, staging, prod branch)

```
terraform
├─ main.tf          primary terraform code
├─ backend.tf
├─ outputs.tf
├─ variables.tf
├─ versions.tf
│
├─ dev.tfvars       environment specific configuration
├─ staging.tfvars
└─ prod.tfvars
```

Environment-Specific Values

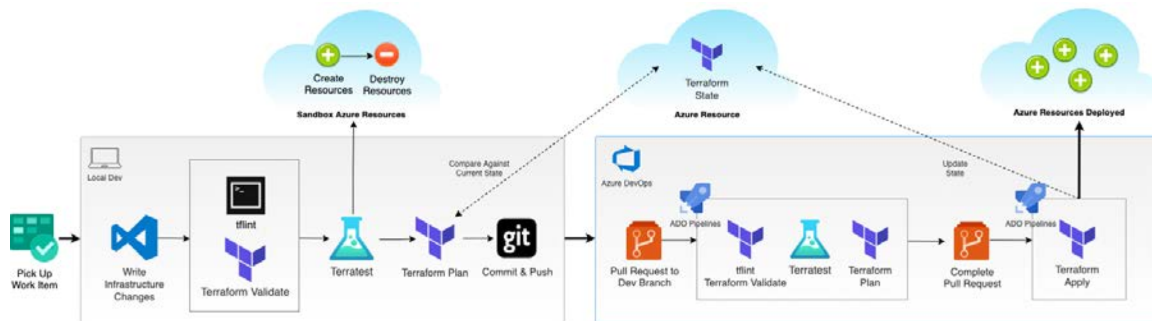
The Terraform configuration that describes each environment is not only represented by the Terraform code in the **terraform** directory, but also by the specific environment variable files e.g. dev.tfvars, staging.tfvars, prod.tfvars. These files contain specific configuration for the inputs to the Terraform for their respective environment. Any similar Terraform configuration between environments can be stored in a file like **shared-config.auto.tfvars** that lives in the same folder.

GitOps Pipeline

IaC deployment pipelines are responsible for performing the codified changes to the infrastructure resources. These deployment pipelines enforce the state of the infrastructure and therefore must also enforce the state of the infrastructure for each environment as well. In order to accomplish this, deployment pipelines should be run often to ensure that infrastructure remains in the correct configuration.

WORKFLOW & TOOLS FOR IAC

With the model of defining infrastructure as code, all changes to infrastructure should be approached similar to application code changes. The workflow will follow the same pattern as making normal changes in software delivery, including local development, testing, and integration, and will leverage similar tools including Git, pull requests, and pipelines.



Local Development

Pick up the work item

As with all code changes, work begins at the ticket-level. In Azure DevOps, engineers will pick up the work item. Following ticket-based engineering principals, code changes and work items are linked to provide maximum visibility and traceability. The engineer should then create the branch representing that unit of work and check it out locally.

Branching patterns and workflow for IaC Deploy Repository

Recall that for the IaC Deploy Repository, the branching pattern should follow a promotion path with one branch per environment. To minimize merge conflicts and encourage better flow, changes should be made at the lowest level and merged through pull requests to higher environments. Even for changes to the production environment, the code change should start on the **dev** branch and be propagated through the **staging** and **prod** branches through pull requests.

Write infrastructure changes and tests

Once their branch is checked out locally, the engineer will then make the infrastructure code change. After writing the IaC, the engineer should perform some basic checks. With Terraform, this involves three commands:

- **terraform validate** — ensure Terraform has all the correct configurations present
- **tflint** — ensure the correct API definition for cloud provider resource definitions
- **terratest** — run automated tests for your infrastructure code

Info box: Terratest is used as an example integration test framework with Terraform. For scaling considerations, Terraform should be refactored into consumable modules. For mature teams, Terratest can be moved from running locally to running only in the PR validation. See appendix on Terraform considerations.

Validate infrastructure changes

Before committing code and integrating changes, the engineer should validate the changes, similar to application developers running unit tests. With Terraform, engineers run a plan against the Azure subscription to see the expected outcome of the infrastructure definition.

Note: To execute any Terraform against a cloud provider, the engineer will need to define credentials to their Azure subscription. Ideally this would live within their local environment.

- **terraform plan** — compare the current infrastructure with the proposed infrastructure changes

Although developers should be free to deploy and delete their own dev infrastructure, deploying shared infrastructure should always only happen in the pipeline (similar to how applications should always be built and deployed from a pipeline and not from local).

If infrastructure is deployed locally, a clean up should follow. This enables cost-efficiency and promotes the concept of immutable infrastructure.

Commit and push

Once the infrastructure changes are validated locally, it is time for the engineer to get ready to integrate their changes back to the main branch. With Terraform, engineers should run **terraform fmt** before committing to include proper indentations and spacing. Code should be committed and their branch should be pushed back up to source control for a pull request and code review.

Pull Request and Code Review

Since infrastructure changes are viewed similar to application code changes, the IaC should be developed on a branch and merged back to the main branch through the pull request process. As mentioned earlier, all changes should start as a pull request into the dev branch, and then are promoted to higher environments through pull requests from one environment branch to the next.

With an IaC Deploy Repository, the dev, staging, and prod branches should all be protected branches with proper branch policies within ADO Repos. This includes required reviewers and build validation.

Build validation pipeline run

Since the dev, staging, and prod branches have a build validation required, there will be a pipeline run for the pull request. However, the PR pipeline will not be deploying any infrastructure (except infra deployed/torn down via Terratest). Instead, it will perform the following steps:

- Validations like **terraform fmt** and **tflint** are run once on the code base.
- **terraform validate** should be run for each environment, not just the target branch of the PR.
- **terratest** should be used to run automated tests for your infrastructure code.
- **terraform plan** should be run for the target branch of the PR.

We recommend having the **terraform plan** output attached to the pull request as a comment that requires review.

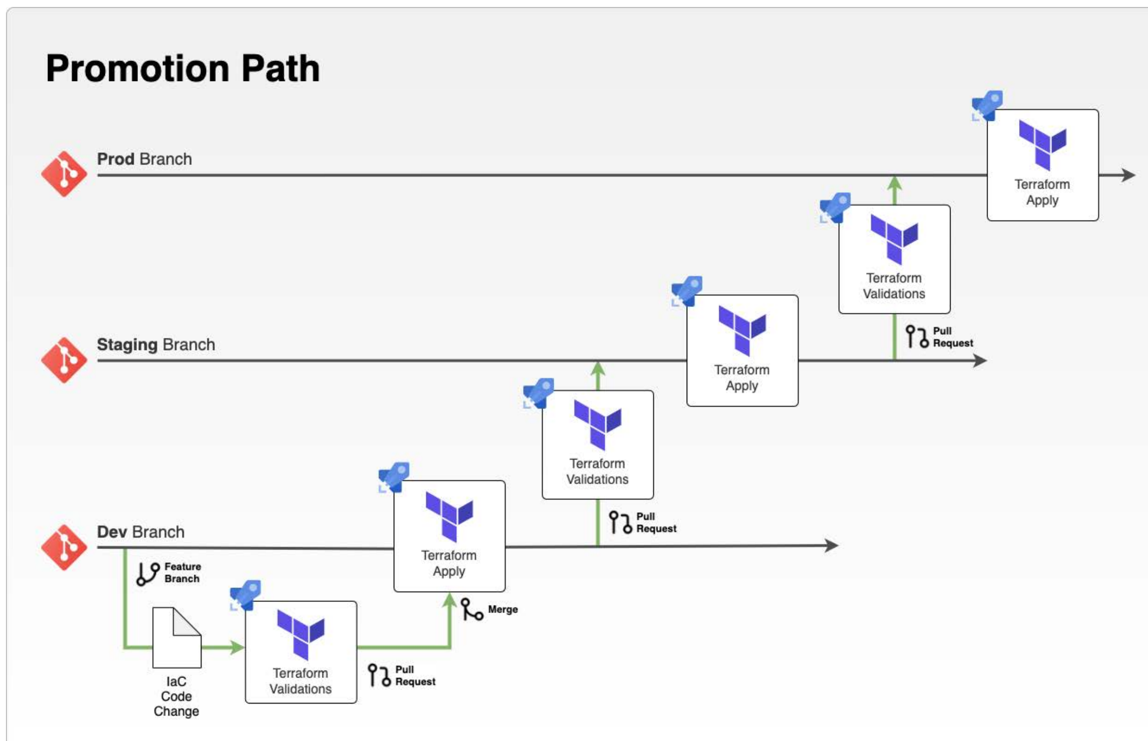
IaC code review

An IaC pull request is similar to application code pull request because both elements are code and therefore changes are reviewed in a similar manner by looking at the delta. Within ADO pull requests, this view can be easily seen and maintained through comments, suggestions, and status updates.

However, the IaC pull request is different from an application code pull request because IaC engineers should be concerned about the pipeline logs. Even with a successful pipeline run, they need to analyze the expected outcome of the proposed infrastructure. With Terraform, this means reviewing the actual environment configuration (i.e. the **terraform plan**) in the pipeline logs, unless it was attached to the pull request for easier review.

Pipeline Deployment

When the pull request has been approved, the engineer can then complete the PR and merge their change into the target branch (dev). This pipeline run will perform the same validations as the pull request build validation, but then perform an apply instead of plan in order to apply the desired infrastructure changes.



Testing. If a platform team is not supporting infrastructure, teams may want to perform additional integration tests. This can be in the form of a manual validation before deploying the application, or it can be utilizing a testing tool like Terratest, which will spin up and teardown test infrastructure. See the appendix for our views on IaC SDLC with Terraform or platform teams.

Approvals. With Terraform, the **terraform apply** requires manual approval by default but can be by-passed for lower environments. This approval can be centrally managed through Terraform Enterprise (see appendix).

Promotion. When the **dev** pipeline successfully deploys infrastructure, changes will be promoted to the next environment through a pull request with **dev** as the source branch and follow the similar pattern as described.

Teardown. Teams can also set up a separate pipeline to destroy infrastructure resources. This should exist within the same IaC deploy repository, but should only run on a defined schedule or be manually triggered. This may be useful in situations where you want to tear down an environment's infrastructure to save costs, for example nightly or over weekends. Guardrails should be implemented to protect higher environments from accidental destruction.

APPENDIX

Tools Index

A wholly integrated toolset provides visibility and flow of the work in progress for a delivery team. From chat/group messaging tools, to issue tracking, to build pipelines, all tools should work together to provide feedback, audit, and communication on the work item. A subset of tools have become industry standard for implementing IaC within the Azure ecosystem. Some of those tools are defined below.

- **Azure Repos** provides a fully featured git-based implementation for version control of code that helps teams track changes to code over time. Proper IaC requires that code lives in a version control system where changes to the code can be reviewed, versioned, and tracked.
- **Azure Pipelines** provides the ability to build and test code by having all of the necessary steps defined in a pipeline. Pipelines are triggered automatically when code changes are made and include logic to perform only the necessary steps based on the trigger. Examples of triggers include new code being pushed to a branch, a pull request being created, or a pull request being merged.
- **Terraform** acts as the orchestration tool for Infrastructure as Code. Terraform allows infrastructure to be expressed as code and allows us to track and maintain state changes to infrastructure so that changes can be made only as needed and configuration drift is minimized or eliminated.
- **TFLint** is a Terraform linter focused on possible errors, best practices, naming conventions, etc. TFLint looks for provider-specific invalid states in your code and should be run early and often in the SDLC,both locally and from within pipelines.
- **Terratest** is a GO library that allows us to create and automate tests for infrastructure as code. Terratest runs integration tests on actual infrastructure that is provisioned with Terraform.

Other Infrastructure Considerations

This whitepaper has focused on producing cloud infrastructure on Azure using Terraform. Similar SDLC patterns exist to create Golden Image virtual machines using Packer or container images using Docker. Look for another article from Liatrio for these patterns.

Terraform Considerations

This whitepaper used vanilla Terraform as a use-case for infrastructure as code. For a thorough implementation of IaC with Terraform, consider our opinionated views on Terraform Enterprise, Terragrunt, and SDLC for Terraform Modules.

Platform Teams for Producer/Consumer Model

Enterprises may form a separate platform team to manage and support underlying infrastructure images or modules. In this pattern, the platform team is responsible for following and maintaining an SDLC around building Terraform modules, container images, or VM golden images, while the AppDev leverages these offerings. This producer / consumer model can work well provided the platform team follows proper versioning and deprecating patterns and an innersourcing model for AppDev teams to contribute. Look for a separate article from Liatrio with our recommendation on this structure.

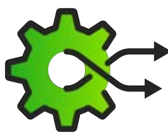
ABOUT LIATRIO

We Help Your Business Deliver Faster And Safer

Liatrio guides complex organizations through their digital transformation journey, enabling them to deliver value faster and safer with Enterprise DevOps and Cloud Native delivery.

We partner with large, successful enterprises to drive systemic change and transformation that helps you scale your entire approach to software delivery.

Our Core Capabilities



Enterprise DevOps Transformation	Cloud Native Delivery	Modern Platform Engineering	DevSecOps
Accelerate business results and scale your organization with a lean, value-driven approach to software delivery and IT operations.	Empower your teams to build scalable apps in dynamic environments and make high-impact changes frequently and predictably with little toil.	Reliable applications are built on modern self-service platforms that reduce engineering friction.	Speed of delivery while always staying safe and secure in an automated way. Remove untimely, manual, last gate siloed approvals and validations.

DEVOPS AND CLOUD TRANSFORMATIONS

LIATRIO.COM

©2021 Liatrio, Inc. All Rights Reserved.