Roadmap to Infrastructure as Code for Automated Self-Service Environments with AWS

Discover how DevOps ideals and infrastructure as code can help align and accelerate application delivery in your Amazon Web Service (AWS) environments.

Written by Sean Davis

Date: November 17th, 2020

An Ideal Approach for Success

Founding principles drive excellence by establishing a shared understanding of the spirit of what you are trying to achieve. The Infrastructure as Code (IaC) journey is reinforced through the following DevOps ideals to support the goal of relentlessly accelerating developer productivity. Understanding how each of these principles helps accomplish this goal will set you and the organization up for success. Some of these principles may not appear to be essential to achieve IaC, but IaC is not just about automation. It is about creating a declarative approach to solving a specific problem without defining a particular process. Automation is at the heart of IaC, but the actual value is not achieved only through scripts and definitions, but by taking a declarative approach to how teams deliver value to the service consumers. For example, one quick value delivery activity from IaC is managing all infrastructure via source control, creating an audit trail, and the ability to fix forward or rollback faster.

By paying close attention to efforts that can reduce complexity and centralize development tooling specific to IaC, technical debt management will directly correlate to the speed of adopting the chosen platform and tooling. Therefore, increasing focus on the right processes and tooling will increase delivery flow due to reduced handoffs and less technical debt, resulting in more satisfied developers and engineering teams. Leading to improvements that will begin to permeate daily work life and drive overall higher value delivery. Naturally, amplifying this inertia by installing feedback loops where developers, engineers, QA, security, and operations can share what is working and not working without the fear of being viewed as grousing or reprimand will yield more practical implementations. In establishing this shared understanding, teams can turn their efforts toward more activities driven by customer feedback, reinforcing the partnership between the customer and the organization while driving cohesion and cross-functionality between disparate groups and expertise domains.

Infrastructure as Code Benefits Beyond the Obvious

Infrastructure as code can significantly reduce the time to delivery for infrastructure and other resources and deliver transformational agility and efficiency through some of the paradigms below:

Focus

Unifying development code and infrastructure definitions in source control allow developers to maintain focus in their native tools to design, build and iterate on the application much faster than was capable in the days of handoffs and silos for infrastructure provisioning. Security focus groups will find deeper insights through change control and audit logs of repositories. Simultaneously, operations will have the ability to see all components in a single location through self-documenting definitions and templates and spend less time manually provisioning or scaling resources.

Collaboration

Generating shared bonds between development, security, and operations through open and honest feedback loops creates learning opportunities and builds cross-functional discipline and understanding. Working in the same native tooling developers, engineers, and security develops stronger bonds and experiences relating to infrastructure code management and security.

Process

Process refinement that supports application building blocks, known as definitions, will enable eventual evolutions into templates and culminate in full self-service catalogs that allow users to select and deploy resources uniformly across the organization. Combining development, security, and operational practices create a homogenous amalgam of service delivery that drives exceptional cross-functional capabilities.

Separation of Duties

Idempotent and declarative infrastructure templates prevent configuration drift, distinctly separate areas of responsibilities for each expertise domain, and build cross-organizational role redundancy. Simple resources at scale provide additional benefits for next-generation application designs, including containerization and serverless through predefined and reusable templates that increasing traceability and auditing throughout the entire lifecycle of the resource.

Time

Automation reduces the time to deploy, release, and scale infrastructure, shifting from reactive to proactive resource management. Cloud Formation templates and full-service AWS Service Catalogs supporting multi-tier application architectures further enhance delivery by consolidating manual steps into automated processes. Additionally, value is realized more quickly by a uniform understanding and continuous connection between teams with visibility into these processes' runtimes.

Cost

Resource templates enable spot and reserved instance planning to drive cost-effective design and architectures. Utilizing API hooks through services such as Cost Explorer or Trusted Advisor drives deeper savings and strategic deployment and scale of resources based on IaC cadences, operational needs, and customer peak and valley utilization. Empowering teams to leverage cost control processes directly in the pipeline enable resource provisioning to be more narrowly defined based on use case and application footprint preventing costly mistakes such as overprovisioning or improper tiering of resources.

Anatomy of Infrastructure as Code

Infrastructure as Code is a composite of many areas working in concert to provide a comprehensive solution to automation and team collaboration in resource provisioning and management. An increasingly thin line between the code that runs applications and the code that configures infrastructure means that development, security, and operations have an increasingly shared set of job responsibilities. Each role is critical in the overall ecosystem of IaC. Below, concepts and domains are broken down by core considerations and the impact of their roles on each area.

Source Control

IaC starts with a source code repository. Source code repositories provide additional capabilities over static files version control, rollback capabilities, auditing, and testing. By combining development best practice and engineering approaches, you achieve best in breed code management, storage, and security. AWS Code commit provides a secure, highly scalable private Git repository, allowing for collaboration on code. As we dive deeper into source control, we'll identify the functional areas and break them down further:

- **Versioning:** By versioning definitions, templates, and state files, the resource configurations can be incrementally improved, rolled back, and tracked through each iteration. A simple LINT defect or improperly placed comment could break the definition, script, or template. If a mistake in a file modification is made, very control can easily revert it. New commits automatically fail when properly leveraging a build pipeline, ensuring your environment's continued operation.

- **Commit History:** Leveraging commit history, you increase your infrastructure code's security posture with historical records of changes that have been made. Commit history also provides insight into the frequency of change to detect anomalies, or a non-breaking alteration of a file may introduce security concerns. Commit history also ensures accountability to the person who made the change, human or machine. If defects, such as a service account overwriting a developer created definition, are introduced, the history allows for easy identification of the faulty process.

- **Branching / Merging:** Through branching and merging techniques, multiple changes can be made to infrastructure code, tested locally, and pull requested back into the main branch kept in lockstep with the latest version of application code it supports. As it deploys alongside the application code, branching provides critical capabilities to develop future components and merge them back into the deployable branch based on a point-in-time change, providing opportunities for more advanced techniques such as hotfix and patching merges. Best of all, each branch can trigger its actions based on the infrastructure code deployed on it.

- **Collaboration:** Standard processes such as pull requests allow team members to collaborate and provide context during the code review process. The extension of Identity and Access

Management also allows for additional permission and collaboration at various contribution levels. Carefully defining viewer to admin access and code review approvers and template visibility are among the security features that should be considered during setup to take full advantage of IaC control in the source repository.

Governance

- **Regulatory Compliance:** Organizations that leverage the cloud must ensure compliance with many regulatory standards, for example, SOC2, PCI, and GDPR. When building infrastructure as Code, guardrails should be implemented based on these standards to ensure compliance. For instance, SOC2 requires an IAM password policy to exist to ensure compliance. Infrastructure as Code solves this by leveraging a template such as the one below to set strong password policies:

```
resource "aws_iam_account_password_policy" "strict" {
  minimum_password_length = 12
  require_lowercase_characters = true
  require_numbers = true
  require_uppercase_characters = true
  require_symbols = true
  allow_users_to_change_password = true
}
```

- **Resource Policies:** There are hundreds of resources leveraged in the cloud, and each requires specific areas and details to pay attention to prevent exposure or a breach. For example, database and storage services are particularly vulnerable to data exposure, primarily when provisioned without encryption enabled or made public, intentionally or unintentionally. EC2 instances can be prone to leaving behind EBS volumes or unattached IP's when deleted. Resource policies drive more efficient use and establish baseline hygiene. As security is a layered approach, it's always a good idea to assess and identify related security configurations and remediate them early in the development cycle as possible.

- **Secrets Management:** Hardcoded secrets is a widespread mistake that consists of putting plain text passwords, secret keys, or other sensitive data in source code. Attackers can leverage these credentials to perform privilege escalation and lateral movement throughout an organization. Due to virtually no trackability with plain text, it's challenging to trace hardcoded credentials in runtime. As with most scripts, IaC can lend itself to easily hardcoding credentials in definitions and templates as well. Ensure policies are defined that scan your definition and template files with TFSec or a similar solution to identify hardcoded and privileged credentials.

- **Auditing:** Leveraging logging is essential with IaC to ensure visibility into resource risk, threats, remediation, and forensics. CloudTrail and CloudWatch are best in breed AWS services that assist with providing greater visibility and audit control. Don't forget to include the proper configuration items on AWS resources to turn these services on. The small option below is "optional" but will make a significant difference in service logging. As seen below, you can turn on CloudTrail to publish events of global services to the logs:

```
resource "aws_cloudtrail" "foobar" {
  name                          = "trailname"
  s3_bucket_name                = aws_s3_bucket.foo.id
  s3_key_prefix                 = "bucket_"
  include_global_service_events = false
}
```

- **Untrusted Sources**: Image registries are the primary area IaC pulls from when deploying templates that leverage compute. Untrusted sources can introduce significant security risk as the industry sees an uptick around malware, crypto mining, and public open-source repo compromises. When scanning IaC templates, ensure every image is from a trusted source, and ideally is hashed and digitally signed to prevent tampering.

Templates (Definitions)

- **Scope:** Dynamic infrastructure can be a challenging concept that results in templates that slowly introduce inefficiencies into your IaC template library and service catalogs. By keeping templates as simple as possible and purpose-driven to reduce complexity. There is an art to the balance as there needs to be enough of a template that users can derive architecture patterns instead of creating unique templates for every application. However, it should not be so diverse that it covers every use case but carries unnecessary components or references not needed by the users deploying the template.

- **Limits:** Limits on definition files prevent template typos and inefficient resource deployments from causing business interruptions. Specifying the upper and lower limits of resource provisioning also ensures that templates are purpose-built for the applications they support. Limiting by source, destination, size, quantity, and environment are great places to start when setting definition boundaries.

- **Lifecycle:** Limits on definition files prevent template typos and inefficient resource deployments from causing business interruptions. Specifying the upper and lower limits of resource provisioning also ensures that templates are purpose-built for the applications they support. Limiting by source, destination, size, quantity, and environment are great places to start when setting definition boundaries.

- **Consistency:** IaC delivers on the ability to create consistent and predictable environments. When resources follow the same pattern, support, operations, and security become more straightforward and repeatable while infrastructure becomes more dispensable. Treat infrastructure code like a container, don't allow access but to the bare minimum, keep it all as immutable as possible and keep out of band scripts and one-off controls out of the equation. Building upon native idempotency by creating supporting patterns such as environment variables vs. new templates will ensure maximum consistency.

- **The Case for Imperative Versus Declarative Approach:** Imperative is excellent when you need to specify the exact way something needs to be done; unfortunately, this makes it brittle for reuse. However, there are great examples of why you would use imperative over declarative such as a one-time migration activity. If you know without a doubt the state before, the imperative approach is a great option. On the other hand, declarative is great for reusability and doesn't require explicit instructions as it's able to inspect and adapt its activity to support variations in states. It doesn't know, and it doesn't care; this makes declarative ideal for day to day IaC but may be overkill for something like one time and special case deployments.

Testing

Good infrastructure as code hygiene starts by applying the same rigorous testing principles to definitions and templates as you apply to the code that runs on it. To check for errors and inconsistencies in your infrastructure, ensure all environments are the same, prevent configuration drift; various testing techniques are used. If a mistake occurs, the tests will stop or assist in building in self-healing t your infrastructure pipelines. I will outline some useful tools in the Terraform suite as it is one of the most popular IaC platforms. Best of all, it's free so that you can get started with zero investment. If you're looking for the oldest IaC tool, that nod goes to CFEngine, automating since 1993.

- **Validation:** The easiest step to take to safeguard your IaC deployments and ensure your definitions aren't laden with bad code is validation. Tools like Terraform validate are perfectly suited to identify these problems and bring them to the forefront quickly as shown below:

```
→  terraform git:(master) ✗ terraform validate
Success! The configuration is valid.
```

- **Code Quality:** Another common challenge is platform drift; as our environments age, so do the platforms' versions; writing code with one version doesn't guarantee compatibility with future versions. TFEnv is a great tool to help manage the multiple potential versions of Terraform in your pipelines. This can be a time-saver, especially when you have multiple providers who have required version dependencies. An example of TFEnv at work is shown below:

```
→  02-tfenv git:(master) ✗ tfenv use min-required
Detecting minimal required version...
Min required version is detected as 0.12.21
[INFO] Switching to v0.12.21
[INFO] Switching completed
→  02-tfenv git:(master) ✗ terraform --version
Terraform v0.12.21
```

- **Static Checks:** When it comes to code quality, you want to start with a simple static analyzer like TFLint. TFLint looks at your TF code and identify things like typo's or invalid values. One of the most underrated things about this static analysis tool is that if you provide it with your AWS credentials, it can check and validate fields used by the AWS provider to ensure you ask for something legitimate. Seen below is an example of where a user-provided the AWS TF provider for several "BOGUS" pieces of information:

```
→  03-tflint git:(master) ✗ tflint --deep
4 issue(s) found:

Error: "BOGUS" is invalid AMI ID. (aws_instance_invalid_ami)

  on terraform.tf line 8:
   8:   ami                    = "BOGUS"

Error: "BOGUS_TOO" is an invalid value as instance_type (aws_instance_invalid_type)

  on terraform.tf line 9:
   9:   instance_type          = "BOGUS_TOO"

Error: "123456789" is invalid security group. (aws_instance_invalid_vpc_security_group)

  on terraform.tf line 10:
  10:   vpc_security_group_ids = ["123456789"]

Error: "BOGUS" is invalid key name. (aws_instance_invalid_key_name)

  on terraform.tf line 11:
  11:   key_name               = "BOGUS"
```

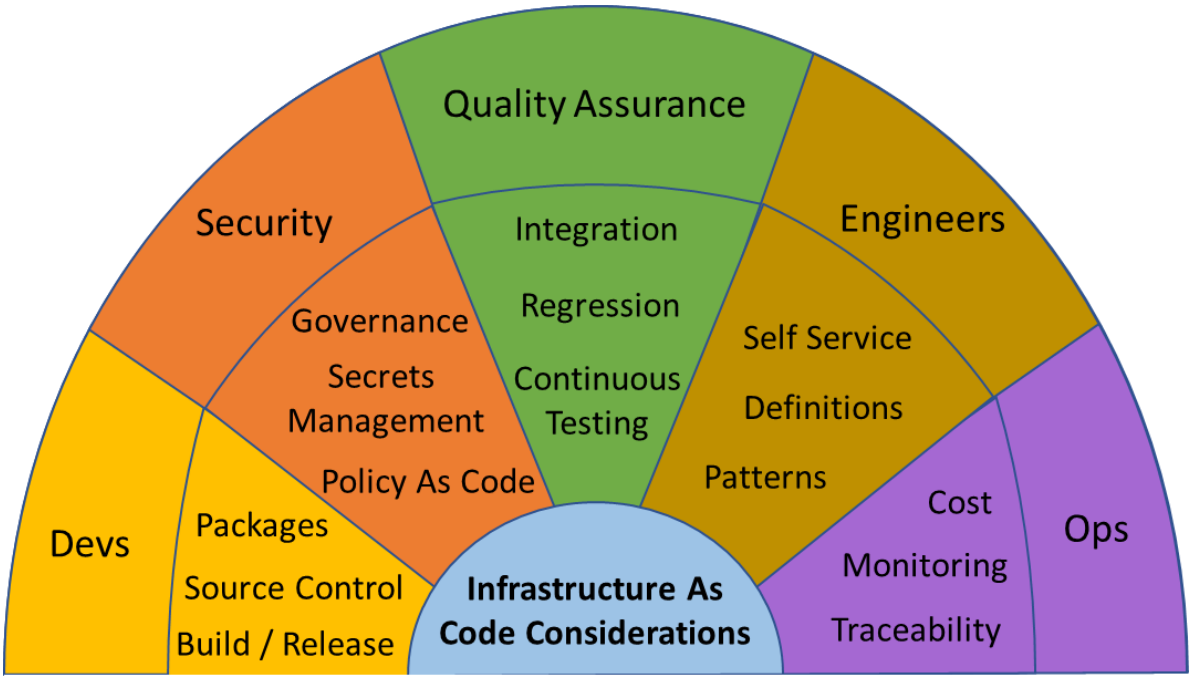
Another great feature of the TFLint tool is it will also do some “smart” things such as recommend better versions of updated instances as seen below:

```
Warning: "t1.micro" is previous generation instance type. (aws_instance_previous_type)

  on terraform.tf line 18:
  18:   instance_type = "t1.micro"
```

- **SecurityTests:** TFSec is the security brother of the TFLint tool. Where TFLint looks at common usage, TFSec focuses more on security based static checks. TFSec is great for finding security issues such as passwords in templates. It can also catch defined attempted policy violations such as a public S3 bucket or missing tags.

Orchestrating Infrastructure Better Together



One of the biggest hurdles will be **how** do we get Developers, Ops, and Security working together to define and approve the environments, standards, and processes?

The answer is simple; we give them instruments to do their job. In a band, every instrument plays a crucial role in the production of the right sound.

Every player has a role, much like every instrument has. Each player must be attuned with the others to see the cues and know when to carry and when to collaborate at the right times.

Another thing to note is that just like every player, there is still a clear concept of separation of duty; just as there are multiple instrument chairs in an orchestra, many people can fulfill any given role within the IaC pipeline to ensure integrity is maintained.

One of the most familiar songs IaC plays is CI/CD as a Service. This is where a company chooses to commoditize services and provide them as a consumable self-service pipeline.

Because of the massive amount of integrations/middleware/ script glue required to wire an entire pipeline together, most companies have chosen to create a standardized platform that every dev team can leverage when developing applications. Code is pushed to a shared repo, built with a standard tool, tested with a common set of tools, deployed with a standardized set of tools, and monitored with a standard suite of tools. The process is always the same and simplifies collaboration and use.

Just as an orchestra has sections and must be aware of the instruments around them, each technology section must be working in unison with each group around them to achieve orchestration.

Source control of your infrastructure templates, security policies that provide guardrails to your infrastructure definitions, and application patterns will go a long way towards a more mature IaC ecosystem.

IaC often starts with big dreams and plans but ends with barebones delivery. This is the typical pitfall most find themselves in when implementing IaC. Someone offers the world and shares how it'll change everything, but then the experience is often the opposite, lackluster and underwhelming. Once again, we must remember that our goal is to relentlessly accelerate developer productivity. And now that we know how to work as a team, here's how we will change the paradigm to under promising and overdelivering.

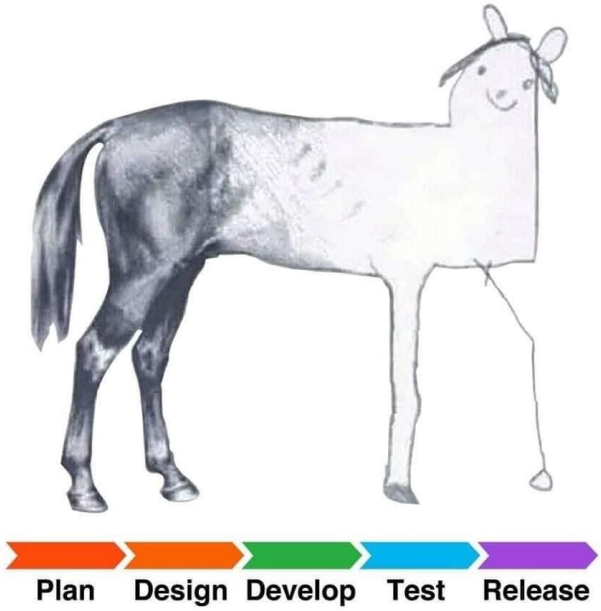


Figure 1 The true lifecycle of IaC

Our primary tool in accomplishing this will be through useful measurement and introspection. This will tell us where we are winning and where we are still learning. It'll also help us refine the areas we need to tweak, such as approval flows and impact the delivery of services; the goal is to get better, the approvals to get less restrictive, and increase flow.

As we look towards these questions, everyone should have shared responsibility in the input, measurement, and accountability of the process.

Here are some introspective questions we should be asking ourselves.

- What is the outcome as compared to the plan?
- Do we have end-to-end visibility of application footprints?
- Can we track and manage microservices dependencies or deploy apps impacting other apps or creating unnecessary dependencies?
- Are our users educated on the application patterns and template definitions and know how to use them appropriately, or are developers leveraging custom-made definitions for every application?
- Did approval flows save us from making costly mistakes, or were they too heavy handed that it canceled out some of the advantages of IaC?
- Did we consider upstream and downstream dependencies, or did we break the infrastructure deployments companywide with one change to a master template?
- Do we see increasing adoption of standard definitions, or do we see an uptick in unique templates?
- Are we tracking new or emerging needs, or are we only focused on current existing applications?

- Are templates partitioned by groups, tags, regions, domains, or are templates universally used?

This list lays the groundwork for future conversations where more purposeful and impactful decisions will drive increased adoption, reduce complexity creep, and most importantly help our teams work better together.

Simplistic Design That Scales with Your Organizations Needs

Simplistic design is a foundational concept, and the areas below explain the considerations in each leg of the journey you should consider. As you explore each of these domains, they provide milestones in the process and build upon each other to create something more significant that cannot be accomplished without the one before it.

Plan

Today pays for tomorrow, and good IaC hygiene begins with proper planning. Before jumping right into a new deployment or buying a bevy of new tools, take the time to plan for improvements to existing services and pipelines. Please keep it simple, define the value before the work begins, lay the groundwork first, and consider your limitations to go a long way towards paying down current tech debt before accruing new debt.

- Complexity (Gall's Law)
  It can also be useful to think about IaC through Gall's Law's lens, which states: A complex system that works is invariably found to have evolved from a simple system that worked. A complex system designed from scratch never works and cannot be patched up to make it work. You must start over with a working simple system. Start simple, and over time you will find the complexity grows but start with a complex design, and you will find a very challenging road ahead. Each decision needs to be weighed against the business's objectives, and its benefits to prevent traveling down a costly road built on technical debt.

- Consider Cost Control
  Build in cost control where possible, through tags or AWS API into AWS services, ensure that automation isn't running unchecked by cost, even in proof of concept and lab environments. Cost Explorer and Trusted Advisor go a long way towards assisting in cost control of IaC initiatives.

- Operational Limitations
  Know your operational limits and balance them against your value, ambitions, and principles to make sure you don't set yourself up for failure to automate things for more money than the revenue or staff can support once created. Stay aware of global early adopters and laggards; each region will have different patterns.

- Upstream/Downstream Impacts
  When creating IaC across expertise domains, think about all its upstream and downstream providers and keep things contained where possible and establish communication where necessary. Both humans and automation have impacts on one another and don't ignore dependencies between the two. Finally, ensure approval flows enhance and scale with the process and tooling, not impede it.

- Consider the Entire Supply Chain

When creating IaC, think about delivery to the entire supply chain; if security or operations are left out, you will develop rifts and erode trust with each "next new thing" or initiative. Instead, focus on how each discipline can dross educate and expose challenges to each other to create learning opportunities and cross-discipline experiences.

Map

Leveraging tools such as value stream and dependency mapping are crucial steps in the IaC journey. Without mapping and defining what you are trying to accomplish will make it difficult to identify value in new initiatives. Take the time to focus on what's working and what's not and determine what steps can provide the most significant value to your team, organization, and customers. Outlined below are a few steps to get started on the right path.

- Manually Define Process
  Most are tempted to dive right in but always take the time to manually define the process first, then move on to combining and refining the steps into automated actions. Defining the intended path first ensures you capture the whole picture and not just the squeakiest wheels.

- Measure Feature Value
  Via value stream or perceived business value, how do we measure the importance of creating a new widget, script or "provider" to deliver our resources? While it may sound nice to have a utopia and the freedom to build anything at will, the technical debt will drown you before you make any significant progress. Instead, focus on measuring value and defining the road ahead based on business, technology, security, and operational team overlap. When we drive value for everyone, everyone wins.

- Establish Name & Tag Standard
  Establish baselines of how you will name and manage the resources deployed to ensure you can quickly identify, map, and determine connectivity and dependencies as well as track cost. Then build independent applications or microservices that don't need to map dependencies and leverage shared services based on templates and standards to simplify auditing.

- Define App Patterns
  Application patterns help developers drive towards the best way to build the application in IaC vs. manual construction. It also provides visual insight into how the infrastructure is built and scaled for operational support. Creating application patterns that follow architectural standards using templated building blocks go a long way towards a simplistic infrastructure that is easy to maintain and understand.

- Policy as Code Guardrails
  Codified guardrails through security policies that govern what's allowed and not allowed when creating the resource will ease management of the infrastructure provisioned. Setting the standard and identifying what needs auditing will help identify issues that resolve themselves without the need for tools. Remember, security IaC tooling is not a catchall. Building flexible

policies and practices into the way patterns are designed, tested, and deployed will reduce the need for overbearing security controls and speed IaC delivery.

Automate

Define what you will and won't automate or perhaps what is and is not valuable to automate. Simplistic design starts with simplistic code and simplistic environments; too much code spoils the application deployment. Leave one off's, infrequent, and highly variable tasks to be optimized later. Focus on the quick wins that can be gained from defined value, not from technical grousing.

- Practice A.U.T.O. Script
  Practice A.U.T.O. scripting, which stands for "Anywhere Used to Operate." It means you automate in any scenario that the process is used operationally in the organization. It makes it dead simple to know when to spend time writing scripts versus just manually doing the task, and it's a policy that management can easily understand and back.

- Version Control IaC
  Version control is the starting block in the marathon we call automation. It's important to consider how the organization will version control their infrastructure, configurations, and supplementary systems such as images and sidecars. It's also important to standardize what branching and merging strategies you will employ and that all code, not just application code, is stored in version control.

- Test Templates
  Test your templates just as rigorously as you do application code; the sooner you find the problems before the build or commit, the better. Leverage tools like TFLint, TFSec, and TFEnv to catch these issues before build time. Never write definitions and hope that the orchestrator or scanning tools will see it; it's essential to discipline teams to adopt acceptable coding practices. The tools are there to catch the mistake, not identify them.

- Secure Templates
  Make sure to call out security requirements for each template. Identifying these needs early will help highlight some of the challenges that a vendor or an internal team may be uniquely suited to solve. Finally, build in your infrastructure and configuration security policies into the tools and pipelines that consume these templates to ensure compliance and standardization across any application or environment.

- Leverage Immutability
  Immutability in templates is as vital as immutability in infrastructure. Enforce restricted access to both; in the same way, you don't want an engineer ssh'ing into a container, don't allow direct modification of a template. Each change should go through the same process a developer would go through for application code. Make the change, create a pull request, require at least 1-2 reviewers to approve, and then commit the change.

Scale

The final leg of the journey is iterative and focuses on the scale of the design and infrastructure that drives IaC. This is also where the model of IaC begins to considerably mature into GitOps, operating more resilient and intelligent systems capable of supporting ecosystems of infrastructure, not just resource deployment. Outlined below are necessary steps to take in consideration of the scaling journey as it pertains to IaC.

- Simple Resource Deployment
  Keep your application patterns and design patterns as simple as possible, review them often, and make the necessary changes to conform to multiple use cases. Stay away from snowflakes. Finally, deployment from CLI, pipeline, or self-service should be the same experience and produce the same result.

- Auto Scaling
  Build auto scale into each template that you create to don't have to do manual scaling operations. The template should also limit the flexibility to keep mistakes and typo's from impacting any environment. Each application and environment should always define minimum and maximum scale sets that are appropriate for the environment. Leveraging trusted Advisor and Cost Explorer here will help you identify where there are opportunities to tighten up limits.

- Spot and Reserved Instances
  Leverage tooling to explore plan where there are opportunities to leverage spot and reserved instances. Identify patterns that can drive cost management and savings based on the specific application and infrastructure patterns and profiles. Then scale your code to take advantage of usage patterns that can be adapted to optimize their footprints.

- GitOps, Containers and Serverless
  Git, as a single source of truth, is key. What distinguishes GitOps is the use of a "pull" CD model, meaning that changes are no longer pushed through a pipeline, but rather states are pulled by agents and compared to determine how to get to the desired state. While there are many progressive environments that mimic some of this behavior, containers are the primary use case for GitOps implementation.

- Serverless Functions
  For the few that are driving serverless may think there is no future for IaC due to the perception there is no infrastructure, but you'd be wrong. AWS Community hero, Shimon Tolts, explains it simply for those who are looking for stronger IaC postures with serverless (predominantly Fargate and Lambda). To learn more, visit: https://aws.amazon.com/blogs/aws/building-a-modern-ci-cd-pipeline-in-the-serverless-era-with-gitops/.

Evolution and The Road to GitOps

In the early stages of IaC your focus will lie on getting the right players involved, building your base pipeline, and installing the tools needed to get infrastructure programmatically created. Developers and engineers will be operating at the base standard of needs: to submit their code to a versioned repository, then code is pushed or pulled through an automation mechanism that allows for infrastructure to be programmatically deployed on-premise, in the cloud, or both.

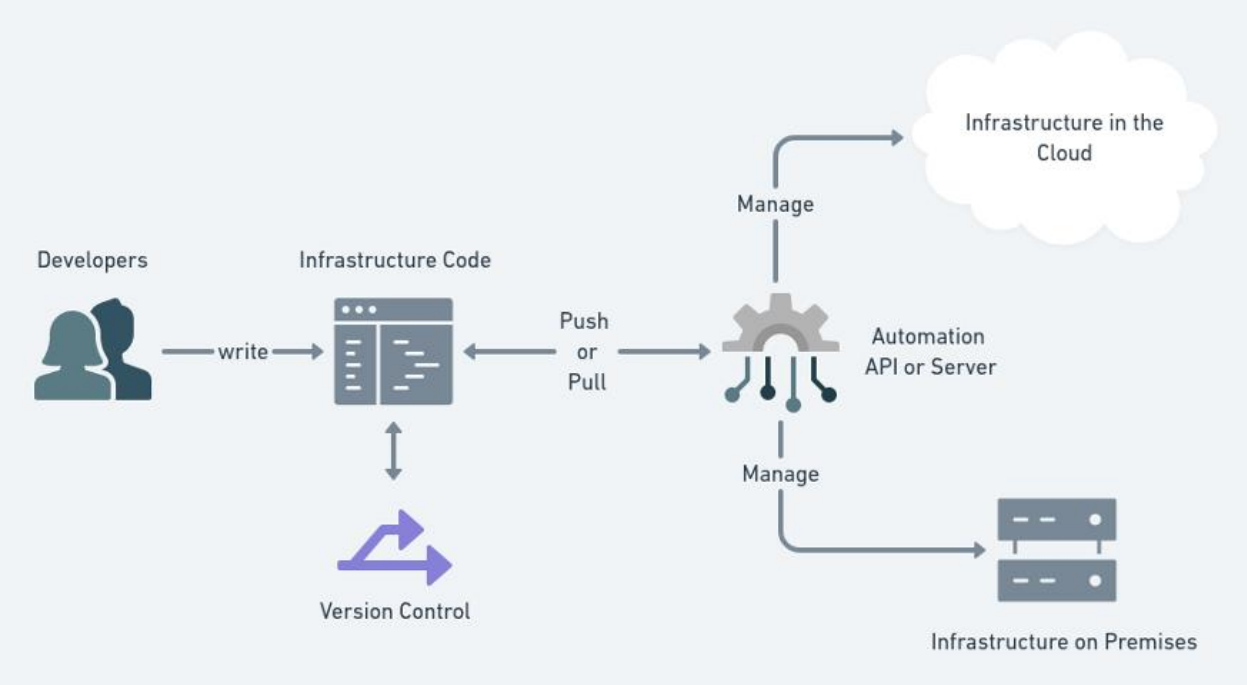


Figure 2: https://blog.stackpath.com/infrastructure-as-code-explainer/

As complexity grows, the need to connect development, infrastructure, and operations will arise, driving the adoption of more autonomous systems and next-generation technologies. Due to the increased diversity of components and capabilities, GitOps provides cohesion that flows through development, engineering, security, and operations. GitOps takes the necessary underpinnings of IaC and expands on them by enabling systems to begin orchestrating push and pull mechanisms that connect containerized builds from code commit to monitoring. Automation of triggers between components that understand context allows each piece to work together to deliver the end-to-end value of more significant complexity of systems. This enables development code, image pipelines, tool configurations, infrastructure, and even monitoring systems to push and pull activities together while leveraging the same versioning application code leverages. The ability to manage and compare the current state of your infrastructure and your applications so that you can test, deploy, rollback, roll forward with a complete audit trail, all from Git, is the foundation of the GitOps philosophy its best practices in motion. It's all possible thanks to Kubernetes being managed almost entirely through a declarative configuration and the immutability containers provide. As shown in the following example, GitOps supports much more complex workflows while automating what would typically span several teams and departments.

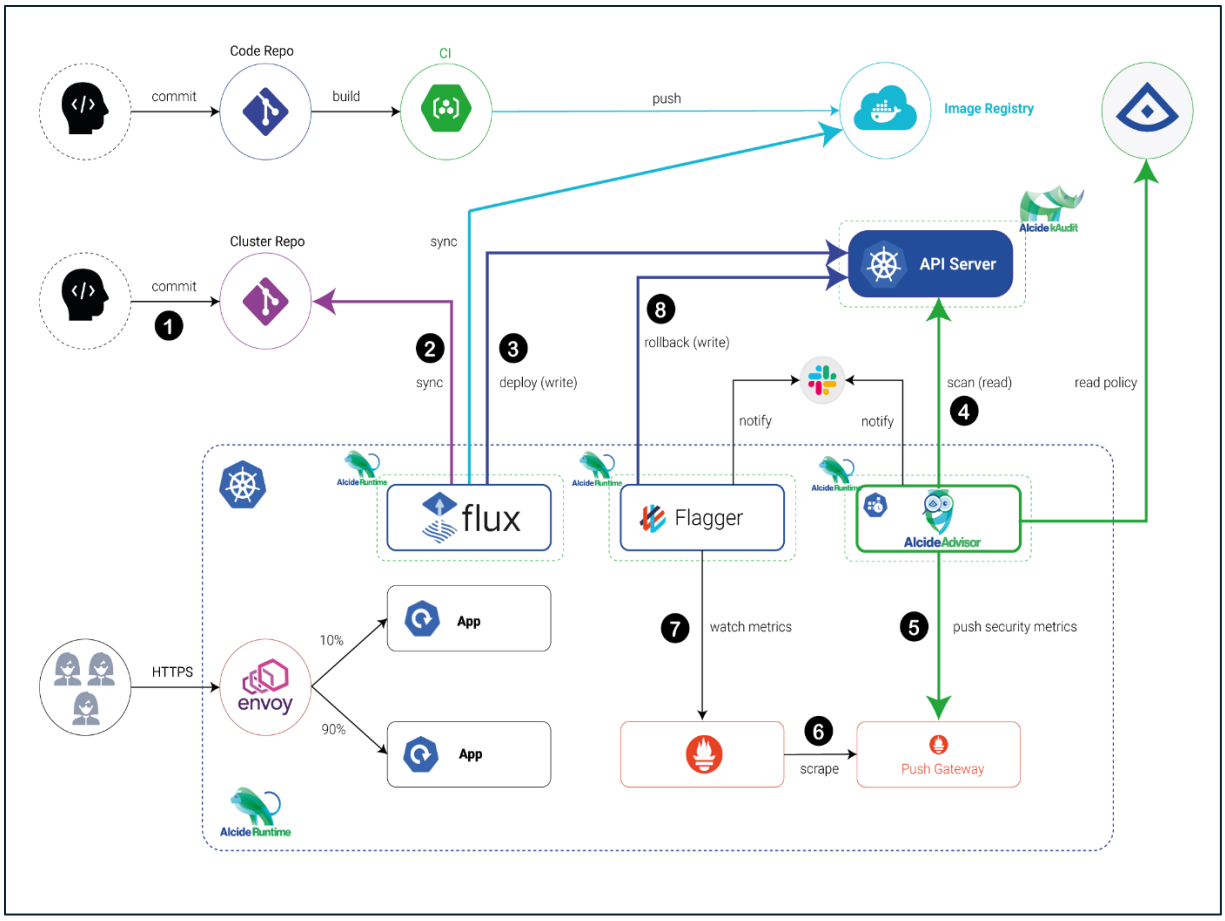


Figure 3 https://blog.alcide.io/gitops-a-security-perspective

In the GitOps world, our typical workflow for a developer will look the same but operate significantly different. GitOps operates primarily on a pull-based CD methodology. When the developer creates a pull request for a new feature, the code is reviewed and approved. A merge triggers the CI/CD pipeline to build and run a series of tests and drop the newly created image into an image repository. A deployment agent then watches the image repository and notices the change. It then pulls down the new image and updates its YAML in the configuration repository. Finally, a synchronization agent in the cluster detects the cluster is out of date, pulls the change manifest from the configuration repo, and deploys the new feature to production. While this is an oversimplified explanation, it shows you very quickly how a GitOps environment can drastically increase productivity and stability through connected automation of infrastructure and operations. By combining and automating communications between existing code repositories, image repositories and configuration repositories you can achieve greater visibility and interoperability through GitOps.

Road to Success Considerations

As a conclusion to our journey, the most important lesson is the road to success is paved with failure. Outlined below are some of the critical areas of consideration in your journey to IaC and beyond to prevent the same pitfalls of others who have made the journey before you. Each topic is instrumental in creating the proper balance of human interaction and technology automation. Alignment, purpose, and establishing a strong foundation of cultural cohesion is paramount to achieving technology goals. Remember, IaC is not just about automation but solving problems through a declarative approach.

- **Start with People**
  Set a baseline and ensure alignment to what you are trying to accomplish and ensure they know what value looks like. It's imperative to ensure each person understands their importance and how they contribute. Empower teams to experiment and learn from their failures. Leverage technical expertise that already exists and doesn't get bogged down by tool bias. Finally, give them a view of the bigger picture, and you'll find tremendous success.

- **Work Together**
  Consider every component in the end-to-end supply chain and toolchains required. Lack of communication and cross-functionality can erode trust and create friction when trying to implement new solutions. It's also important to note; everyone should understand how and where they connect and the upstream and downstream impacts of their domain decisions. Finally, identify whom to go to with questions for each domain. During times of significant change, documentation and communication fatigue from not knowing whom to talk to or who can answer a question end up causing significant problems and cultural and technical debt.

- **The Glue that Binds**
  People are equally effective glue as scripts can be. Because there are numerous integration points and teams involved in your IaC efforts, take special consideration of the entire supply chain and try to understand each connection point required to make those parts functions across domain boundaries. Ultimately, everything is a system of systems. Leveraging value stream mapping will ensure you gain the necessary insight and improve the overall IaC process instead of creating more work in the name of automation.

- **Plan, Map, Automate, Scale**
  As the saying goes, Garbage in, Garbage out. Plan for simplicity by focusing on the outcome, not the tools that will drive the outcome. Identify all the manual steps and automation required in a way that makes sense and brings the most value. Dependency mapping can be a vital tool that brings some peace to this very chaotic journey. Scale as

complexity increases. Track and adjust infrastructure usage based on capacity and criticality.

- **Simplify and Secure**
  Define as much as you can upfront and leverage programmatic ways to manage environments such as predefined application patterns or templates. Keep the snowflakes to a minimum. Also, ensure that you have the appropriate people involved in the environment approval flows. Most importantly, ensure healthy socialization, visibility, and collaboration of security practices and policies, by building policies and procedures only into the pipeline but as native as possible to the developer. Lint checkers or IDE security scanning plugins are worth their weight in gold. Finally, define environment requirements upfront, identify differences before building templates, and ensure approval flows are rigorous but not impeding.

- **Measure, Measure, Measure**
  Avoid vanity metrics, quantify your toolchains' impact, and share your progress with stakeholders and other teams. To keep value high and outcomes strong, focus on metrics that matter, not ones that look good on paper. Don't be afraid that the numbers will make you look bad. Every number is a baseline to a better outcome. Ensure you quantify each toolchain's value and, most importantly, be transparent about the progress and learning as you go. Open collaboration is essential for critical feedback loops, and nothing resonates with people more than authenticity.

- **Build for Reusability**
  Build your templates to be as self-documenting as possible and with the mindset that any team can easily understand, manage, and extend them for their use. This is the sole purpose of templates, reusability. Keep the template sprawl to a minimum; educated socialization and use of your template language will prevent you from rolling out snowflake images for every application—leverage naming and tagging standards and typical process flow to keep operations simple and straightforward.

About the Author

Sean Davis is a DevOps Institute Ambassador, flagship instructor, a board member for several technology advisory boards, brand ambassador for YouExec, and DevSecOps Advisor for Transunion. Before Transunion, Sean was the Chief Transformation Evangelist for Equifax. He focuses on creating opportunities between the market, customers, consumers, technology, and security teams to deliver exceptional experiences and exponential talent and market growth. He coaches technical groups and individuals to exceed their expectations and build secure, durable, and scalable business cultures and products. His primary areas of expertise are business transformation, technical leadership, Dev(Sec)Ops, Agile, and performance coaching.