# IT Project Guidance –

## On Environment as Code (EaC)

**Version:** 0.2

### Purpose

This document provides guidance on the use of declarative and functional programming techniques for defining and managing modern cloud environments. It introduces the concept of Environment as Code (EaC) and explores the practical, architectural, and organisational implications of using tools such as Azure Bicep, Terraform, and Pulumi. It also challenges common misconceptions around tool portability and simplicity.

### Synopsis

As cloud platforms expand beyond basic infrastructure, Environment as Code (EaC) has emerged as the necessary evolution of Infrastructure as Code (IaC). This document examines the practical and architectural implications of defining cloud environments using declarative and functional models. It compares native and cross-platform tooling across Azure and AWS, including Bicep, ARM, CloudFormation, CDK, Terraform, and Pulumi. By analysing idempotence, state, abstraction, and team workflows, it challenges assumptions about portability and simplicity. The guidance concludes that platform-native, declarative-first, pipeline-governed approaches provide the most resilient and maintainable model for modern cloud service delivery.

# Contents

## Purpose and Audience

This guidance is aimed at solution architects, cloud engineers, DevOps specialists, and project teams planning or managing environment provisioning in the cloud. It is designed to support strategic tool selection, architectural consistency, and long-term maintainability of system deployments.

## Scope

This guidance primarily focuses on Azure and AWS, the two dominant cloud platforms, and analyses their native and third-party provisioning models. It evaluates declarative and imperative toolsets including Bicep, ARM, CloudFormation, CDK, Terraform, and Pulumi. The discussion covers tool selection, real-world integration into pipelines, operational trade-offs, and long-term architectural considerations relevant to single-cloud and hybrid strategies.

## Background

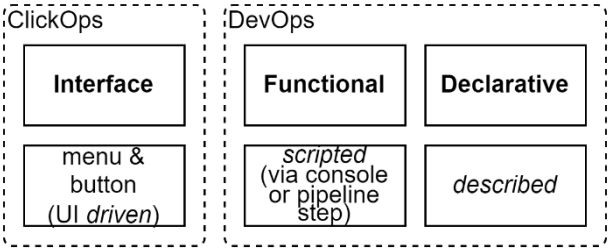


*Figure 1: Three approaches to configuring cloud environments*

There are three primary methods for programming cloud environments:

First is through the cloud provider's interface—typically a web portal with graphical configuration and provisioning workflows. These allow manual drag-and-drop or form-based setup of environments, helpful for exploration but unsuitable for repeatable deployment.

Second is functional scripting, where APIs are directly called using tools like PowerShell, Azure CLI, or other SDKs. These scripts tend to mirror the UI actions programmatically and are often embedded in pipeline steps. They offer flexibility and are the only way to do some imperative control flow operations.

Third is the declarative model, which describes desired state and lets the provisioning engine work out the execution. This model underpins tools like Azure ARM templates, Bicep, Terraform, and Pulumi. Declarative specifications are critical to achieving consistent, testable, and environment-specific deployments.

# Pipelines and Provisioning Roles

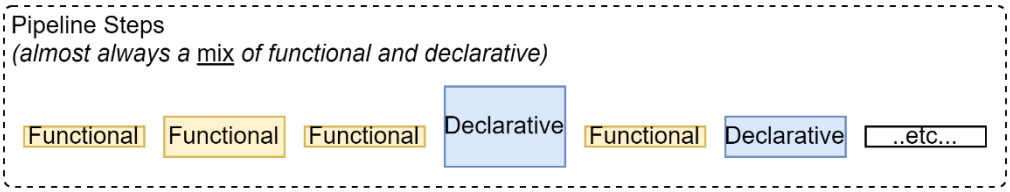


*Figure 2: Pipeline steps are a mix of functional and declarative*

However, the real-world use of these models does not occur in isolation. They exist as part of deployment pipelines—composed of discrete stages, validations, sequencing, and branching logic.

Functional scripting and declarative modelling have fundamentally different roles in this process. Declarative definitions describe what the environment should become. Functional steps orchestrate how and when declarations are applied, how outputs are validated, and what conditional logic governs deployment gates.

While functional scripting might appear more portable, this is largely illusory. Once systems begin using advanced platform-specific features—storage accounts, networking, security policies, gateways, WAF, caching, monitoring, AI services, etc.—the functional scripts that interact with those systems are no more portable than declarative specifications. *Both* are deeply tied to their target platforms.

Efforts to collapse functional and declarative concerns into a single language or framework tend to produce disappointing outcomes. Declarative languages are not optimised for procedural flow (including loops, etc.), and procedural languages cannot ensure convergence or idempotence without external tracking and complexity.

*They serve different purposes*.

The solution is not unification, but disciplined separation: use declarative languages for environment state, and functional logic for orchestration and control.

This distinction is not a limitation—it is a design principle. It reflects a mature understanding of how cloud provisioning actually works.

## Modelling Approaches

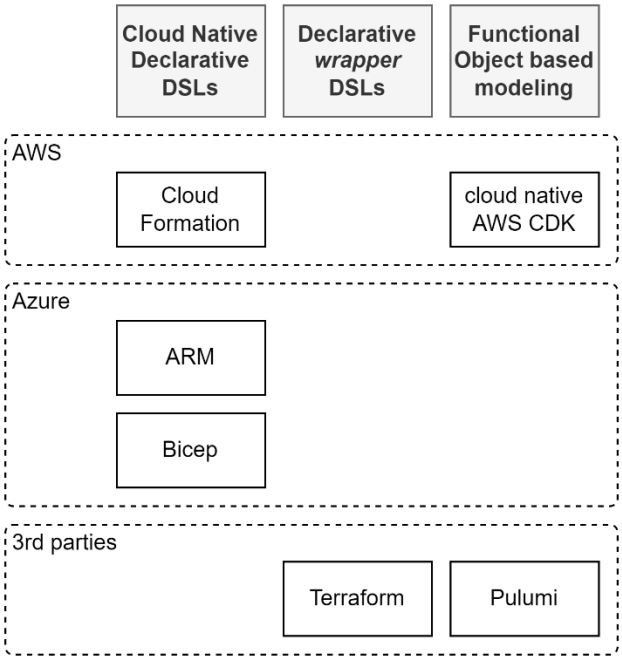


*Figure 3: Modelling Approaches*

There are three dominant approaches to modelling cloud environments in code. Each reflects different priorities, audiences, and trade-offs:

First is the cloud-native declarative approach. Azure ARM templates and Bicep, and AWS CloudFormation, represent Domain Specific Languages (DSLs) that directly describe the desired state of cloud resources. ARM and CloudFormation are powerful but verbose,

often challenging for authorship or readability.  Bicep improves on ARM with a more compact and readable syntax – but not much. These tools suit environments where consistent deployment is critical and the platform's idempotent behaviour is leveraged directly.

Second is the use of third-party declarative wrappers such as Terraform. These tools offer a common DSL that abstracts over multiple cloud providers. Additionally, they introduce features such as state tracking, module reuse, and preview execution. Terraform appears simpler and more approachable to operations staff due to its readable syntax and visual planning stages, but brings baggage in the form of external state management and licensing overhead.

Third is the language-native functional modelling approach exemplified by Pulumi and AWS CDK. These tools allow developers to model infrastructure using general-purpose programming languages such as Python, TypeScript, C#, or Go. These tools appeal to developers familiar with software design, enabling abstractions, testing, and dynamic logic. Pulumi’s model is particularly aligned with object-oriented thinking, similar to ORM patterns, making it attractive to those seeking composability and expressiveness. However, these models are often opaque to operations teams and introduce runtime complexity and language binding dependencies.

AWS CDK (Cloud Development Kit) aligns closely with Pulumi in philosophy and structure. It enables developers to write infrastructure code in general-purpose languages like TypeScript or Python, which it then synthesises into native CloudFormation templates. This mirrors Pulumi’s approach of converting high-level language objects into deployable infrastructure plans. CDK is particularly well-adopted within AWS-native teams due to its tight service integration and native toolchain support. However, like Pulumi, it tends to alienate operations teams unfamiliar with software development practices and increases dependency on complex application logic and language runtime environments.

At present, Azure does not offer a directly equivalent: while Azure supports PowerShell, CLI, and ARM/Bicep for scripting and declarative provisioning, it lacks a fully supported language-native modelling framework akin to CDK or Pulumi. Efforts to use general-purpose programming languages in Azure often rely on community projects, wrappers, or hybrid automation tooling, rather than a first-party modelling abstraction layer.

This division mirrors the roles within most teams: developers favour language-native expressiveness, operations teams lean towards structured declarative models, and both groups must navigate between clarity, flexibility, and platform compliance.

## The Portability Myth

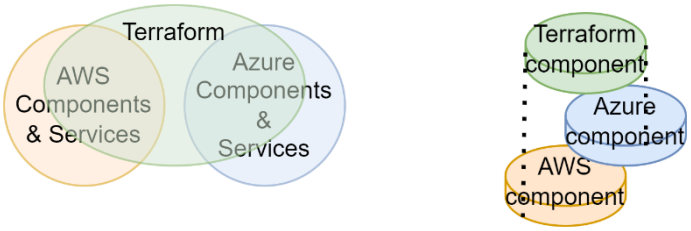


*Figure 4: Incomplete overlay at both the whole and component level*

The marketing claim of cloud portability through tools like Terraform is not borne out in practice. While it's technically feasible to use the same DSL to target multiple clouds, the overlap between provider-specific services is minor. Attempting portability results in *both* narrowing down to legacy components *and* discourages the use of modern, vendor-specific capabilities.

Worse, it imposes an abstraction cost without avoiding the need to learn the provider-specific implementations.

No modern service environment should be limited to IaaS-level primitives in order to achieve notional portability.

## The Distraction of State

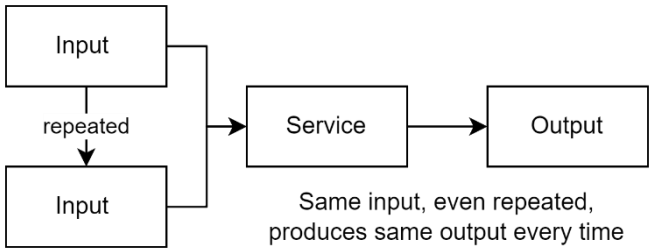


*Figure 5: The reliability of idempotency.*

Cloud-native provisioning engines—such as Azure ARM, Azure Bicep, and AWS CloudFormation—are designed to be idempotent by default. That is, they aim to converge the environment to the declared state without unnecessary redeployment or duplication. This design exists for a reason: it allows infrastructure descriptions to be safe, repeatable, and auditable across multiple executions without requiring side-channel memory.

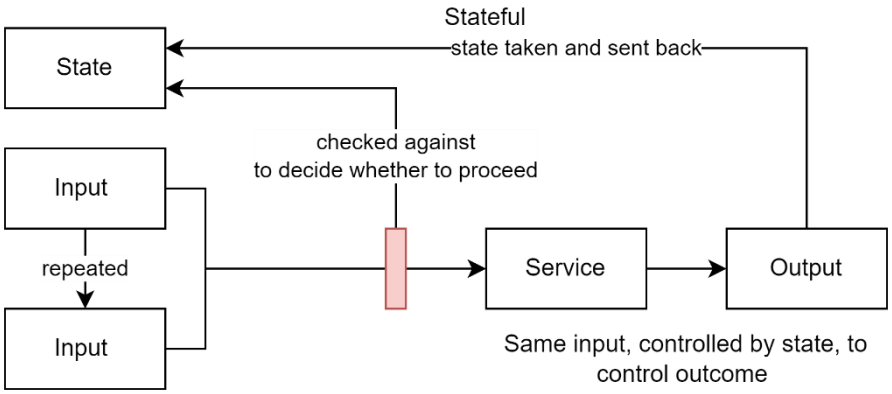


*Figure 6: State persistence required to manage outcome*

In contrast, Terraform introduces its own layer of state tracking, managing a local or remote file that represents the last known state. This design allows Terraform to preview changes and detect drift, but it also introduces a new source of truth—outside the platform—that must be maintained, secured, and reconciled. While this model emerged to compensate for limitations in early vendor tooling, it has become a distraction in its own right.

Pulumi and AWS CDK generate resource graphs that are ultimately rendered by idempotent backends—CloudFormation in AWS, or Pulumi's own engine. However, Pulumi retains an explicit state model akin to Terraform. Although more flexible and developer-friendly, it still introduces sidecar logic and runtime coupling that shifts responsibility from the platform into the tool. These designs reflect an effort to generalise and abstract the provisioning process, but the outcome often compromises simplicity, determinism, and platform fidelity.

The continued centrality of state in many third-party tools is not an architectural requirement—it is a workaround for gaps that no longer exist at the platform level. Change detection and validation belong in pipelines, using explicit comparison, environment introspection, or policy evaluation. Provisioning tools should be focused on expressing the desired state—not maintaining memory of prior executions.

To be clear: external state tracking is not inherently harmful. In some complex, multi-stage, or hybrid environments, it may offer useful checkpoints or overlays. But it should not be the default, nor the foundation. It is a tool of last resort when native idempotence and pipeline orchestration cannot meet the need.

If the tool insists on managing state centrally, it is not improving your cloud—it is replacing it.

## Terraform

Terraform remains popular with decision makers and novices of EaC for good reasons:

- Rich ecosystem of providers and modules
- Clear change previews before apply
- Extensive community documentation and examples
- Powerful conditionals and loops for complex setups
- Mature support for remote backends and team workflows
- Rich ecosystem of integrations (e.g. Sentinel policy enforcement)

However, these strengths come with operational overhead, such as state file management, increased pipeline complexity, and security burdens.

## Pulumi

Pulumi offers an alternative model. Instead of defining resources in a DSL, users write code in real programming languages (Python, C#, Go, etc.) to construct environment descriptions. These are compiled into resource graphs and then executed similarly to Terraform.

Pulumi's core idea—treating environment configuration as a language-native object model—is conceptually elegant. It offers testing and validation within the language, better debugging, and easier abstraction. For developers familiar with object-oriented or functional programming, it is more intuitive than declarative DSLs.

However, Pulumi still introduces its own layer of abstraction and lock-in, and its commercial model includes similar licensing concerns to Terraform.

## Vendor Specific Tradeoffs

This leaves a difficult impasse.

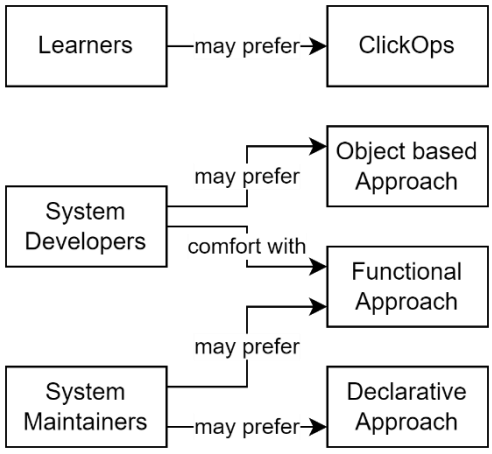


*Figure 7: Indicative (not fixed) stakeholder preferences*

Operations teams tend to prefer Terraform because it offers familiar patterns, approachable syntax, and a large base of examples. However, they often lack the development skills or tooling discipline to use Terraform's functional features effectively. This results in fragile templates, limited reuse, and poor lifecycle management.

Developers, by contrast, are more likely to gravitate toward Pulumi or AWS CDK, which better support object modelling, code reuse, and automated testing. Yet these tools require significant ramp-up and alignment with software development practices, making them inaccessible or opaque to many operations engineers.

Meanwhile, Azure Bicep—although a clearer DSL than ARM—still feels foreign to both groups. It remains low-level, limited in abstractions, and tied tightly to Azure’s resource model. As a result, many teams view Terraform as the least alienating option for cross-role collaboration, even if it is not the most capable.

The question is not which tool is most palatable. It is whether Terraform provides enough long-term value to justify locking in a decade or more of cost and complexity. In most cases, beyond the trivial and early training years, it does not.

Portability is a myth. The real choice is between the two dominant cloud vendors, and the best path is often to go directly to the vendor-native tooling. It offers the most alignment, least abstraction debt, and clearest lifecycle model. If the platform is already being committed to, then its provisioning model should be treated as part of the architecture—not bypassed in favour of a cross-platform abstraction that solves few real problems. Training is a reality. Teams must be prepared to invest in learning the vendor's native tooling, just as they would any other architectural component. Abstraction layers that claim to remove this need tend to shift complexity rather than eliminate it, often making systems harder to understand and govern over time.

## Bridging with AI

While stepping away from tools like Terraform or Pulumi may seem like abandoning a layer of convenience, the landscape has fundamentally shifted. The rise of AI copilots—such as GitHub Copilot and ChatGPT—has dramatically lowered the barrier to working directly with native cloud SDKs and APIs. These tools provide real-time code generation, contextual suggestions, and instant access to documentation, effectively removing the need for abstraction layers designed to simplify or "dumb down" infrastructure management.

In this new paradigm, the complexity that once justified wrappers is now manageable—even elegant—when paired with intelligent assistance. Rather than relying on external state management or DSLs, engineers can work closer to the metal, with AI as a fluent guide. This not only enhances transparency and control but also aligns with modern DevOps principles of clarity, reproducibility, and minimalism.

## Principles

The following are principles that summarise the architectural stance that emerges from the above analysis. They are not preferences, but tested heuristics—grounded in the realities of vendor ecosystems, operational patterns, and the long-term lifecycle of cloud-delivered systems:

- Declarative and functional steps benefit from remaining separated, each fulfilling a distinct role in pipeline orchestration.
- Vendor-native tooling should be embraced when platform commitment is made; portability is a false economy.
- Idempotence is foundational. Any additional state model must justify its operational and architectural overhead.
- Abstraction adds cost. Clarity, governance, and maintainability must guide tool selection.
- Abstraction adds risk. Keeping continuing and complete alignment between two vendors, over all environment components and services, over a full service lifespan cannot be guaranteed, or even realistically expected. This risk grows as both the cloud platform and the abstraction layer evolve independently, making divergence inevitable and maintenance brittle.

## Conclusion

Environment as Code is the right strategic framing. Declarative infrastructure is necessary but insufficient. Real-world teams will continue to mix declarative and imperative logic. The goal must be to architect this mix clearly, limit imperative sprawl, and push validation and decision-making into pipelines—not into deployment tools.

Idempotent-by-design approaches like Bicep integrate better with pipeline execution and cloud-native governance. State-driven models like Terraform offer power but with baggage. Pulumi presents a promising hybrid, but its object-based abstraction is still vendor-controlled.

No tooling choice substitutes for architectural clarity and environmental discipline. Tooling must serve the system design, not constrain or abstract it into false promises of portability or simplicity.

Declarative-first. Pipeline-governed. Provider-specific. This remains the most truthful pattern today.

# Appendices

## Appendix A - Document Information

### Authors & Collaborators

- Sky Sigal, Solution Architect

### *Versions*

0.1 Initial Draft

0.2 Added Diagrams

### *Images*



### *Tables*

**No table of figures entries found.**

### *References*

**There are no sources in the current document.**

### *Review Distribution*

The document was distributed for review as below:

| Identity | Notes |
|---|---|
| Dries Venter | |
| Werner vd Merwe | |
| | |

## Audience

The document is technical in nature, but parts are expected to be read and/or validated by a non-technical audience.

## Structure

Where possible, the document structure is guided by either ISO-* standards or best practice.

## Diagrams

Diagrams are developed for a wide audience. Unless specifically for a technical audience, where the use of industry standard diagram types (ArchiMate, UML, C4), is appropriate, diagrams are developed as simple “box & line” monochrome diagrams.

## Acronyms

**API** : Application Programming Interface.

**ARM** : Azure Resource Management XML based template language. See *CloudFormation*.

**AWS** : Amazon Web Services

**Azure** : Microsoft Web Services

**CDK** : AWS Cloud Development Kit

**DSL** : Domain Specific Language

**DDD** : Domain Driven Design

**GUI**: Graphical User Interface. A form of UI.

**ICT**: acronym for Information & Communication Technology, the domain of defining Information elements and using technology to automate their communication between entities. IT is a subset of ICT.

**IT** : acronym for Information, using Technology to automate and facilitate its management.

**UI** : User Interface. Contrast with API.

## Terms

Refer to the project’s Glossary.

**Application Programming Interface** : an Interface provided for other systems to invoke (as opposed to User Interfaces).

**Bicep** : Azure’s DSL to simplify ARM.

**Capability** : a capability is what an organisation or system must be able to achieve to meet its goals. Each capability belongs to a domain and is realised through one or more functions that, together, deliver the intended outcome within that area of concern.

**ClickOps** : the pejorative description of setting up interface manually using an interface (such as Azure's Portal) rather than developing the functional or declarative description of an intended environment.

**CloudFormation** : uses JSON or YAML to declaratively describe intended environments. Comparable to Bicep or ARM.

**Cloud Service providers** : the two primary cloud service providers are Azure and AWS.

**Domain** : a domain is a defined area of knowledge, responsibility, or activity within an organisation or system. It groups related capabilities, entities, and functions that collectively serve a common purpose. Each capability belongs to a domain, and each function operates within one.

DevOps

**Entity** : an entity is a core object of interest within a domain, usually representing a person, place, thing, or event that holds information and can change over time, such as a Student, School, or Enrolment.

**Environment** : either a production or non-production standalone organisation of components and services. The default list of non-production environments are Built Test (BT), Dev Test (DT), System Test (ST), User Test (UT), Integration Test (IT), Training, PreProd, and the only production-data environment is (PR) Prod.

**Function** : a function is a specific task or operation performed by a system, process, or person. Functions work together to enable a capability to be carried out. Each function operates within a domain and supports the delivery of one or more capabilities.

**Person** : a physical person, who has one or more Personas. Not necessarily a system User.

**Persona** : a facet that a Person presents to a Group of some kind.

Pipeline

**Portal** : Azure's interface to their cloud environment services. Provides a clickops based approach to learning what is available.

**Quality** : a quality is a measurable or observable attribute of a system or outcome that indicates how well it meets expectations. Examples include reliability, usability, and performance. Refer to the ISO-25000 SQuaRE series of standards.

**User** : a human user of a system via its UIs.

**User Interface** : a system interface intended for use by system users. Most computer system UIs are Graphics User Interfaces (GUI) or Text/Console User Interfaces (TUI).