Automating Deployments in Azure using Resource Manager Templates (ARM)

Anil Kumar Manukonda
E-mail: anil30494@gmail.com

Abstract

Azure users provision cloud infrastructure through Infrastructure-as-Code (IaC) which ensures consistency in deployments. This study demonstrates the method of automating Microsoft Azure deployments by utilizing native Arm templates which serve as infrastructure-as-code solutions in Azure. This paper examines IaC principles in DevOps and ARM template development history alongside automated methods to deploy Azure resources. This section explores both the structure of ARM templates including parameters variables resources and outputs and shows how deployments are automated through Azure CLI along with Continuous Integration and Continuous Deployment pipelines. We present specific technical examples which use JSON templates to create virtual networks and storage accounts alongside role-based access deployments. ARM templates prove to be superior to manual provisioning by accelerating deployment times while minimizing errors and increasing scalability. This paper examines ARM templates alongside Terraform and Azure Bicep while evaluating their differences between learning approach and tooling complexity and multi-cloud capability. This paper examines template complexity and debugging challenges while proposing developments including enhanced tools and artificial intelligence solutions for deployment planning. The paper demonstrates ARM templates maximize Azure automation capabilities but recommends specific use cases based on workload requirements.

Keywords: Azure Resource Manager, ARM Templates, Infrastructure as Code (IaC), Azure CLI, Continuous Deployment, Azure DevOps, CI/CD Pipelines, JSON Templates, Bicep, Virtual Network Deployment, Storage Account Automation, Role-based Access Control (RBAC), Declarative Infrastructure, Azure Policy, Resource Provisioning, Automation, Template Linking, Modular Templates, Azure Bicep, HashiCorp Terraform, Pulumi, Multi-cloud Infrastructure, Cost Optimization, Deployment Orchestration, What-if Analysis, Deployment Validation, Version Control, DevOps, Azure Portal, Template Parameters, Error Handling, AI-assisted IaC, Governance, Cloud Scalability, State Management, Idempotency

Introduction

In modern DevOps deployments infrastructure serves as an essential component of lifecycle development that operates through automated code-based mechanisms instead of hand-driven procedures. Through Infrastructure-as-Code (IaC) developers define infrastructure with machine-readable configuration files to automate both environment creation and termination. Through IaC operations teams unite with developers to create infrastructure definitions which can be managed under version control for instant deployment of standard environments between development stages and production. The managed method decreases manual configuration requirements thus minimizing both scale-dependent errors and process duration.
ARM Templates operate as Microsoft's declarative infrastructure-as-code solution specifically built for

Azure deployments. ARM templates came with the 2014 Azure Resource Manager launch to replace “Classic” Azure deployments and utilize JSON for describing Azure resource definitions. The desired infrastructure state appears in ARM templates through resource definitions which describe cloud objects (VMs and networks and databases) with their respective properties but omit instructions for resource generation. Azure's Resource Manager service manages the deployment sequence of resources while automatically resolving dependencies between them. ARM templates enable Azure environments to establish repeatable deployment consistency thus meeting core requirements for DevOps CI/CD workflows.

The implementation of automation stands as a critical element for implementing effective cloud operations within Azure. Agile and cloud-native teams need to deploy intricate stacks (web apps, databases, networks, etc.) both intermittently and reliably. The deployment of Azure resources through manual methods is limited to poor scalability and error-proneness. Using ARM templates for automation generates benefits of fast deployment and consistent results while enabling version control of infrastructure side-by-side with application code. Multiple components of a web application environment containing an App Service for web applications alongside SQL databases in conjunction with VNet networking constructs and security rules can be specified through templates for single-step deployment. The deployment of network infrastructures (virtual networks, subnets, and network security groups) along with security configurations (Azure Policy assignments and role-based access control roles) can be choreographed through templates across subscriptions. These practical uses emphasize why automation methods matter to organizations. The deployment of multi-tier web applications and the establishment of hub-and-spoke virtual networks with routing configurations and runtime security policies across resources becomes possible through template deployment methods executed through CLI or CI/CD pipelines. This method removes repeated manual work and guarantees that the identical settings produce each environment (dev, test, prod) [2].

We examine the historical background and architectural structure of ARM templates in addition to workflow methods and practical use cases together with their resulting implications in the sections below. The research includes comparisons of ARM templates to other Infrastructure as Code (IaC) solutions along with an evaluation of existing challenges and upcoming advancements.

Background

Before resource management through Microsoft Azure operated under a "Classic" deployment model (Azure Service Manager) that managed resources one by one. In 2014 Azure brought the Azure Resource Manager (ARM) together with resource groups to introduce its contemporary deployment framework. ARM delivers Azure with a single management framework while using JSON templates as the language for defining infrastructure. The configuration of multiple Azure resources within a single deployment unit exists through ARM templates that function as JSON files. Users define what resources need to get created rather than describing how the creation process should work through this declarative method.

The development of ARM templates originated from requirements to achieve deployment consistency and repeatability. ARM templates deliver essential capabilities through JSON-based elements that perform the following functions: Users obtain lifecycle management through resource grouping in addition to deploying dependent resources in synchronized environments while benefiting from Azure RBAC and tagging implementation for governance control. The Microsoft team expanded ARM template capabilities by introducing additional resource support and extending template language functions beyond string and array operations up to conditionals and loops through copy functionality.

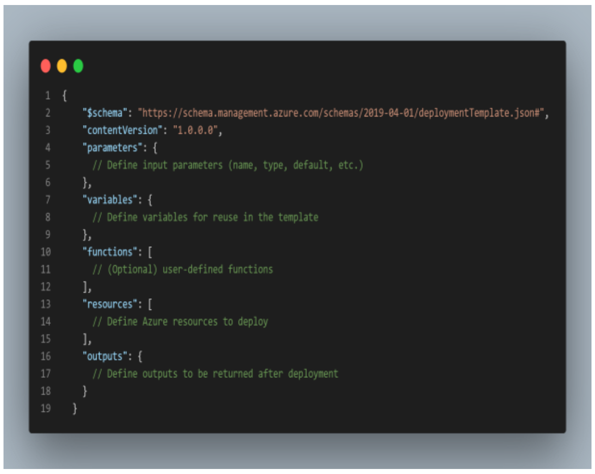
ARM templates became the core component of Azure automation strategies when organizations started treating infrastructure deployment through code. The ARM JSON code could be checked into source control systems where organizations would deploy it during application releases to synchronize infrastructure adjustments with software deployments. The new method outperformed traditional manual portal configuration by eliminating configuration errors and drift problems [2]. ARM templates enable idempotent deployments because Azure will update the environment to match the template repeatedly without creating redundant resources. Robust automation depends heavily on this essential property.

The process of manually creating raw JSON ARM templates proved to be verbose while also presenting user-unfriendly features. The Azure Bicep platform emerged following community requests as it provided better template abstractions over ARM templates during 2020. Bicep provides a domain-specific language which simplifies complex JSON expressions yet produces equivalent ARM JSON templates. The core deployment format for Azure remains ARM JSON templates even though Azure Bicep templates have emerged [7]. Any practitioner working with Azure must have a solid understanding of ARM templates.

The use of ARM templates has evolved into Azure IaC's established best practice standard. All deployments made through templates must pass through Azure Resource Manager's control plane system. A detailed examination of template structures accompanies an overview of practical deployment workflow operations in the following sections.

Architecture & Workflow

ARM Template Structure: Typically, an ARM template is a JSON document with several top-level sections: $schema, contentVersion, parameters, variables, functions, resources, outputs. Here is the simplified skeleton of an ARM template file displaying these sections:



```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
      // Define input parameters (name, type, default, etc.)
    },
    "variables": {
      // Define variables for reuse in the template
    },
    "functions": [
      // (Optional) user-defined functions
    ],
    "resources": [
      // Define Azure resources to deploy
    ],
    "outputs": {
      // Define outputs to be returned after deployment
    }
  }
```

Code 1: Structure of an Azure Resource Manager (ARM) Template

Each section serves a purpose:

- **$schema:** URL of the JSON schema where the provisions about the ARM template language are specified. This assists tooling in validating the template. For instance, when using the 2019-04-01 schema URL, which is typically used when making resource group deployments, as demonstrated above.

- **contentVersion:** A template user-defined version (e.g., "1.0.0.0"). It doesn't affect deployment behavior, but it may be used for versioning your templates.

- **parameters:** Enter values that can be accepted by the template at the time of deployment. Parameters make templates reusable by externalizing the values that are specific to given environments (e.g., names of resources, their sizes etc.). The name parameter distinguishes parameter, the type parameter identifies whether it's a string, an integer, a bool, an array, object, secureString, etc., and optional is the provider of a defaultValue and some metadata/description.

- **variables:** Local values calculated using parameters or constants that can be reused in the template. According to variables, one avoids repetition, and complex expressions are made easier. They are not externally provided; they are calculated once during deployment.

- **functions:** Local values calculated with parameters or constants that can be used to reuse within the template. With regards to variables, there is no repetition, and the simple expressions are made easier as well. They are not externally provided; they are computed once upon deployment.

- **resources:** A collection of objects that describe an Azure resource to deploy (or update). An object of a resource contains a resource type (e.g., "Microsoft.Compute/virtualMachines"), apiVersion, an indication of the REST API version to utilize for the specified object of a resource, name, location, and a block of properties with the type-specific settings. It can also contain dependencies, tags and child resources. The template may have several resources and by default Azure Resource Manager will try and create them in parallel as much as it is possible while serializing those that have specific dependencies on other ones.

- **outputs:** Values to be returned from the deployment when the resources are all provisioned. The outputs can reference already deployed resources (for instance, you can output an ID of generated resource or connection string). These outputs may be utilized by deployment scripts, greater templates provide they are deploying to, or may be plugging into other templates in the case of linking deployments.

Sample Template (VM Deployment): For instance, we can examine a template to deploy an Azure Web App (App Service) or Virtual Machine. We describe a VM example to shorten for brevity. VM deployment usually uses several resources, such as a network interface, a VM, disks, etc., while ARM template can explicitly define all needed fragments. For instance, a stripped-down VM template's resources may include:

- A virtual network (to connect the VM to a network).

- A subnet within that VNet.

- The VM should have a public IP address.

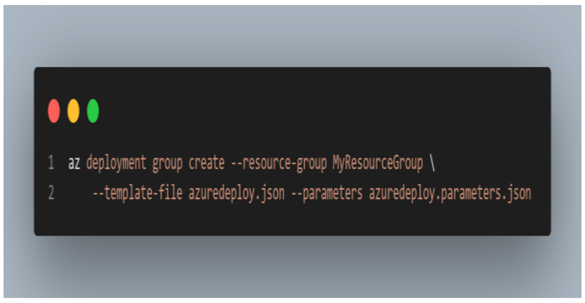
- A network interface connecting the subnet and IP.

- The virtual machine resource itself which has properties such as VM size, image, OS settings, admin credentials and etc, and the dependency on the NIC (the VM also demands the NIC ready).

Each of these would be a template's resource object, and the VM resource's dependsOn would include the NIC resource. Parameters could be used for a name of the VM, admin username, or size of a VM (e.g., Standard_DS1_v2). Because of space, we do not include a full VM template here, but the important aspect is that ARM templates allow defining complex multi-resource deployments from a single file and Azure Resource Manager arrange creating items in the correct sequence.

Deployment Workflow via Azure CLI: Once you have an ARM template and an optional parameters file or in-line parameters value, the template can be deployed using different tools. A popular way of doing it is through the Azure CLI. The flow of deployment is as follows:

1. **Invoke Deployment Command:** Utilize the command azure cli az deployment group create (for resource group scope deployments) with the target resource group, utilizing the template file, and also parameter values. For example:



```
az deployment group create --resource-group MyResourceGroup \
    --template-file azuredeploy.json --parameters azuredeploy.parameters.json
```

Code 2: Azure CLI Command to Deploy ARM Template to a Resource Group

With this CLI command, the template (and parameters) is packaged and a deployment request is sent to Azure Resource Manager.

2. **Authentication and Validation:** The CLI (already authenticated through az login or other means) invokes the ARM API. Azure Resource Manager (the control plane) validates the request (using Azure AD credentials), and validates whether the user or service principal has the right to deploy to given resource group. Then ARM carries out a

syntax and schema validation of the template. If there are errors (such as unknown resource types or malformations of JSON), deployment is refused prior to any changes.

3. **Resource Manager Orchestration:** On validation, the Azure Resource Manager goes ahead and creates the resources mentioned in the template. ARM determines the correct order because of dependencies. It will arrange the deployment provided they are done so that dependents are likely to be developed in order but independent it can be developed in parallel. This implies that the speedy deployment as compared with step manual process because Azure can spin up many assets at the same time possible. All Azure resource providers (Compute, Network, Storage, etc.) are responsible for resource creation calls. Figure 1 illustrates this concept: no matter whether they are done through the CLI, PowerShell, or the portal, azure's Azure Resource Manager takes all of them in which then communicates with the resource providers to deliver the services themselves.

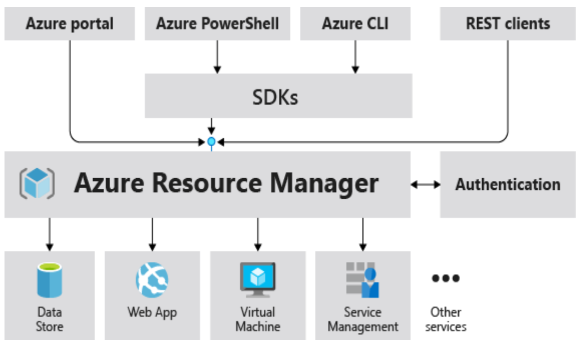


Figure 1: Azure Resource Manager (ARM) sits between client tools (Portal, CLI, PowerShell, SDKs) and the Azure services. It authenticates and authorizes requests (through Azure AD) and then forwards them to the correct resource providers. This pervasive layer of management guarantees that the deployments through ARM template or other methodologies result in similar outcomes.

4. **Deployment Monitoring:** The CLI command will stream progress to the console. Azure Resource Manager logs a deployment record in the chosen resource group. You can check the deployment status (e.g., using az deployment group show). If there are errors while deploying (say, one resource fails to create), ARM shall stop or roll back to increment or complete mode and return the error details.

5. **Completion:** In Azure CLI, a summary will be displayed on success, and any output which has been defined in the template will be shown. The resources are provisioned in Azure now. In the entire process, including the template submission and the processes it is followed up with, deployed infrastructure can occur within minutes, and furthermore, it could be applied again in another resource group or subscription using the same CLI command, merely by changing parameters.

This workflow illustrates one of the most significant benefits of ARM templates because One CLI command for example can deploy multiple interconnected resources with one declarative specification while one might need to click through the Azure Portal or specify multiple imperative commands to deploy.

Deployment Workflow via Azure DevOps (CI/CD Pipeline): In enterprise scenarios ARM templates deployments are commonly incorporated in CI/CD pipelines using Azure DevOps or GitHub Actions. The flow with Azure DevOps (using Azure Pipelines) consists of:

1. **Source Control:** ARM templates (JSON files), are present in source repository (e.g., Azure Repos or GitHub). When there are any changes made on templates, it causes a pipeline.

2. **Pipeline Trigger:** The continuous integration pipeline could lint or validate a template (such as the usage of the ARM Template Toolkit – arm-ttk – validating best practices). Then, a release pipeline or a stage in the pipeline is accountable for deployment to Azure.

3. **Azure Resource Manager Deployment Task:** Azure DevOps has integrated tasks for ARM template deployment. In a pipeline YAML or classic release, the Azure Resource Manager Template Deployment can be utilized. This task needs a service connection (service principal credentials) to Azure. This service principal is then used by the pipeline to authenticate to the Azure after which it

will run an ARM deployment (as the CLI command underneath). This task receives the template and parameters files from the source repo.

4. **Orchestration and Deployment:** Azure Resource Manager takes the deployment from the pipeline and goes to the step of provisioning the resource, as with a manual invocation of CLI. As far as ARM is concerned, the fact that the request came from a pipeline does not make any difference, the validation and the orchestration work the same way.

5. **Continuous Deployment and Iteration:** The pipeline can be configured to run in various environments (e.g., Dev, QA, Prod) with some parameter-fashion files for each environment to provide environment-specific values (e.g., names or sizes). Once deployed, tests may take place (integration tests on the newly deployed infrastructure), and if the tests pass, the pipeline can also promote the same template into the next environment or region etc. If an update to a resource is required later, a modification of the template in code is done, and the pipeline is provided more resources again, ARM can make incremental changes to the existing resources. The pipeline in Azure DevOps provides traceability – each run is noted and any ARM deployment error would be seen in the pipeline logs.

A similar strategy can be applied using the official GitHub actions with Azure ARM Deploy action or the Azure CLI action. The high-level architecture for CI/CD with ARM templates is displayed in Figure. 2.

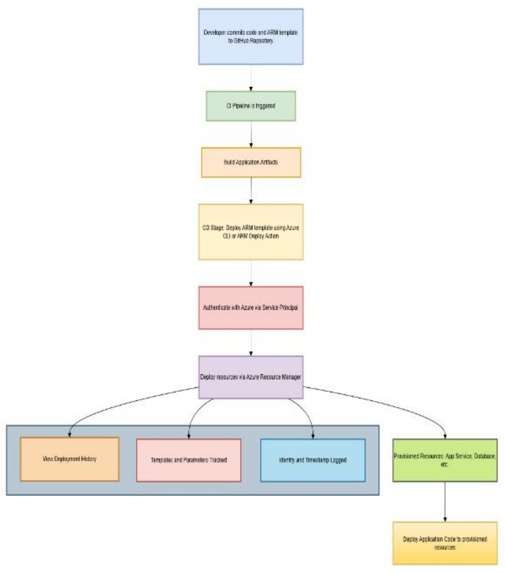

Developer commits code and ARM template to GitHub Repository
CI Pipeline is triggered
Build Application Artifacts
CD Stage: Deploy ARM template using Azure CLI or ARM Deploy Action
Authenticate with Azure via Service Principal
Deploy resources via Azure Resource Manager
View Deployment History
Templates and Parameters Tracked
Identity and Timestamp Logged
Provisioned Resources: App Service, Database, etc.
Deploy Application Code to provisioned resources

Figure 2: CI/CD Pipeline using ARM Templates

Imagine that a developer commits code and an ARM template into a Git repository; a CI pipeline provisions the app and an Azure Pipeline CD stage utilizing an ARM template deploys azure resources (App Service, DB etc.) using a service principal connection. The ARM template deployment stage communicates with Azure Resource Manager, which is responsible for creating or modifying the resources within the Azure subscription. The application code is then deployed by the pipeline to those provisioned resources.

With the use of these workflows, ARM templates bring infrastructure-as-code in real life: application changes including code and infrastructure changes undergo the same review and deployment process. Cloud-based Storage in Azure is integrated with Azure's logging and auditing. In the Azure Portal, it is possible to check the history for the deployment of each resource group, it is possible to see which template and parameters debuted, at which time, and by what identity. This is one of the benefits of ARM – the Azure platform remembers the deployments of the template, which facilitates the troubleshooting and carrying out of compliance.

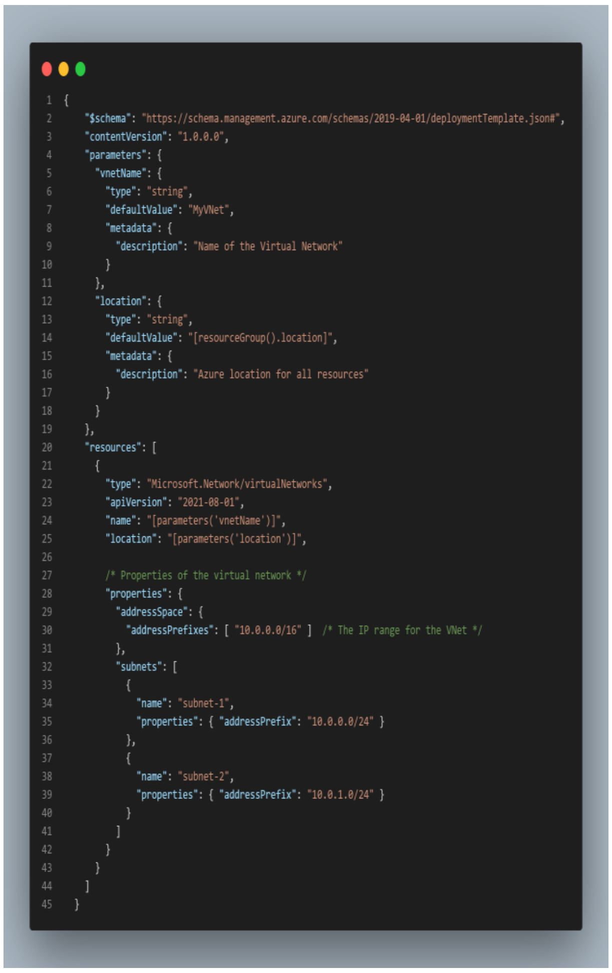
ISSN: 2583-6129
DOI: 10.55041/ISJEM01502

Implementation & Use Cases

We now give concrete examples of ARM templates in action after a discussion of ARM template structure and deployment flows. Examples below include but not limited to the most common use cases: networking, storage, and access control. Each dropped snippet is an abridged JSON template (part of said template), with inline comments for clarification.

Example 1: Virtual Network and Subnets

Establishing a virtual network (VNet) is very commonly the first step before the creation of cloud infrastructure for an application. An ARM template snippet to deploy an Azure VNet with two subnets is below. We parameterize the VNet name and location to achieve the reuse capability.



```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
      "vnetName": {
        "type": "string",
        "defaultValue": "MyVNet",
        "metadata": {
          "description": "Name of the Virtual Network"
        }
      },
      "location": {
        "type": "string",
        "defaultValue": "[resourceGroup().location]",
        "metadata": {
          "description": "Azure location for all resources"
        }
      }
    },
    "resources": [
      {
        "type": "Microsoft.Network/virtualNetworks",
        "apiVersion": "2021-08-01",
        "name": "[parameters('vnetName')]",
        "location": "[parameters('location')]",

        /* Properties of the virtual network */
        "properties": {
          "addressSpace": {
            "addressPrefixes": [ "10.0.0.0/16" ]  /* The IP range for the VNet */
          },
          "subnets": [
            {
              "name": "subnet-1",
              "properties": { "addressPrefix": "10.0.0.0/24" }
            },
            {
              "name": "subnet-2",
              "properties": { "addressPrefix": "10.0.1.0/24" }
            }
          ]
        }
      }
    ]
  }
```

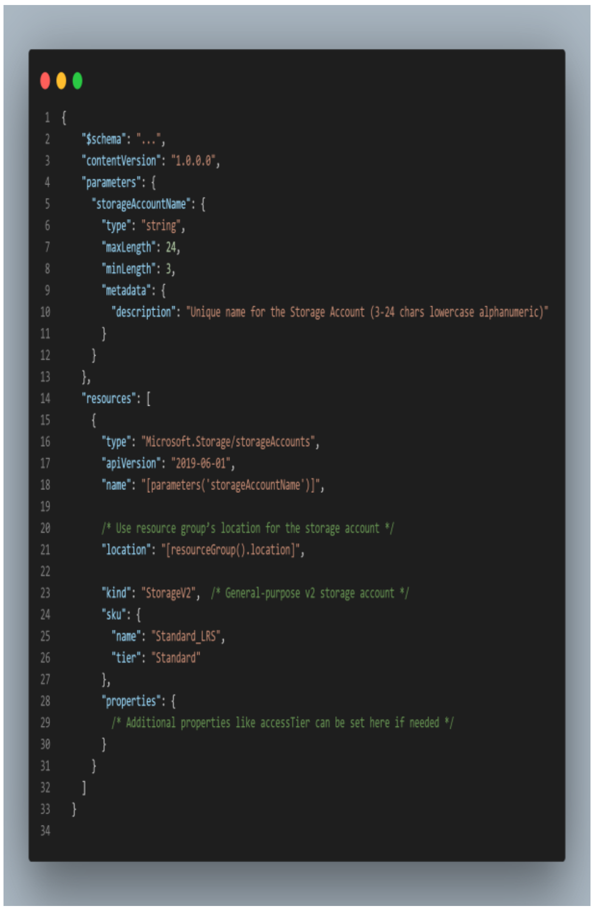
Code 3: ARM Template Example for Deploying an Azure Virtual Network with Subnets

Explanation: Our two parameters are vnetName and location. The location default value refers to an ARM template function resourceGroup(). location to default to the same location as the resource group (making the template easy to reuse in any region). There is one entry in resources array: a Microsoft.Network/virtualNetworks resource (under the version 2021 API). The name of the Vnet is set based on vnetName parameter. In the properties, we give addressSpace (the IP range block for the VNet), as well as an array of two subnets. Every subnet has a name and address prefix. In a full deployment, one may also provide a Microsoft.Network/networkSecurityGroups resource, and may associate it to subnets, or define other network properties, but that is not the core that this simple example is about.

When this template is used, it will create a virtual network called "MyVNet" and two subnets. Through changing the parameter values, we would be able to deploy several VNets (for example – one per environment) from same template, naming them differently or from different regions.

Example 2: Storage Account

Azure Storage Accounts are a basic resource for storing of data (blobs, files, queues, tables). Some of common template patterns are illustrated by them: with a generated name and giving SKU (pricing tier). Follow is an example of how one can deploy a storage account:

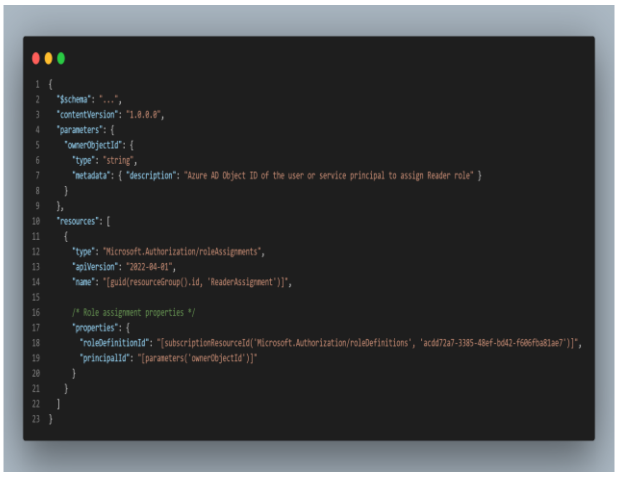

```
{
  "$schema": "...",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "storageAccountName": {
      "type": "string",
      "maxLength": 24,
      "minLength": 3,
      "metadata": {
        "description": "Unique name for the Storage Account (3-24 chars lowercase alphanumeric)"
      }
    }
  },
  "resources": [
    {
      "type": "Microsoft.Storage/storageAccounts",
      "apiVersion": "2019-06-01",
      "name": "[parameters('storageAccountName')]",

      /* Use resource group's location for the storage account */
      "location": "[resourceGroup().location]",

      "kind": "StorageV2",  /* General-purpose v2 storage account */
      "sku": {
        "name": "Standard_LRS",
        "tier": "Standard"
      },
      "properties": {
        /* Additional properties like accessTier can be set here if needed */
      }
    }
  ]
}
```

Code 4: ARM Template for Deploying an Azure Storage Account

Explanation: The storageAccountName is a parameter due to the fact that the storage accounts must be unique across all of Azure. We leave the user to enter a name that would fit Azure's requirements (therefore the minLength/maxLength and a description). The resource has the type of Microsoft.Storage/storageAccounts. We set kind: StorageV2 for a general-purpose v2 account. The sku is configured to Standard_LRS (locally-redundant storage, standard performance). The SKU tier "Standard" is supplied by the name but given for clarity. We rely on the resource group's location to deploy the storage account in the same region. We could add tags or other settings (enabling blob public access or default network rule sets, for instance) by either extending the properties or adding child resources (blobServices, for example). This template would create a new storage account. On practice, one might "mix" this with other resources. for instance, a build-deploy template that is provisioned that may contain storage account as well as an App Service, as well as, write out a storage account connection string for the app to consume.

Example 3: Role Assignment (Access Control)

Infrastructure as code is more than just deployable resource(s) such as: compute or network; it can also create security and access policies. It is handy to use ARM templates to deploy Azure RBAC role assignments for automating governance. Following is an example of resource group template assignment of the Reader role to a user for a resource group:


```
{
  "$schema": "...",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "ownerObjectId": {
      "type": "string",
      "metadata": { "description": "Azure AD Object ID of the user or service principal to assign Reader role" }
    }
  },
  "resources": [
    {
      "type": "Microsoft.Authorization/roleAssignments",
      "apiVersion": "2022-04-01",
      "name": "[guid(resourceGroup().id, 'ReaderAssignment')]",

      /* Role assignment properties */
      "properties": {
        "roleDefinitionId": "[subscriptionResourceId('Microsoft.Authorization/roleDefinitions', 'acdd72a7-3385-48ef-bd42-f606fba81ae7')]",
        "principalId": "[parameters('ownerObjectId')]"
      }
    }
  ]
}
```

Code 5: ARM Template for Assigning Reader Role in Azure

Explanation: Reader role that comes within Azure has a definite GUID of its role definition Id (Acdd72a7-3385-48ef-bd42-f606fba81ae7) built in. We use the well known GUID in roleDefinitionId, and we have the whole resource ID by using the subscriptionResourceId function (which appends the subscription ID automatically). The principalId refers to the object ID of the user or service-principal which will be assigned the Reader-role – passed as ownerObjectId parameter to the template. The name of the role assignment resource must be unique and by convention we usually use a GUID made from stable identifiers. Here we use the ARM function guid(resourceGroup().id, 'ReaderAssignment') to generate a deterministic GUID from the resource group's id and a string; This avoids rerunning the template from trying to create duplicate assignments (same GUID will be calculated and ARM will know the assignment resource already exists).

When deployed to a resource group, this template (using the appropriate objectId parameter) will assign that principal the Reader role on the resource group. This approach is handy for automating the provisioning of access control – say you provision a new application environment, you could automatically pass the dev team's Azure AD group contributor role on the resource group, etc, all this through templates. It's more secure and less error prone than clicking in the portal, particularly when repeating across many environments.

Parameterization and Linking Templates: The above shows parameterization from inside single templates. ARM templates also support linked, or nested, templates, which supports modular deployments. As an example, one template can reference another template (stored externally such as within storage account or GitHub) via the Microsoft.Resources/deployments by using a templateLink or inline template definition. This enables splitting a huge deployment to smaller more focused templates (e.g. one template for "network infrastructure", one for "app infrastructure", and one for "database") and then re-assembling them by linking. When working with linked templates, the sub-templates have to exist at a URI (public GitHub URL, or Azure Storage SAS URL, etc.). This comes at the cost of increased complexity as a price for better organization. Alternatively, templates nested (embedding template JSON in the parent template) can also be used for modularity without references to other files outside of templates. In practice, numerous Azure architects use either linked templates or address complexity with the use of tools such as Bicep or Terraform (to be discussed later) for better modularization. Now, ARM templates do support these scenarios, natively — parameters can actually be passed to the child templates from the parent ones, making for a value re-use.

In conclusion, above are the implementation examples on how ARM templates capture different Azure deployment scenarios in code. By running these templates (manually or via pipelines) one can automate the provisioning not of merely virtual machines and networks, but resignation of higher-level constructs such as the entire app environment and their surrounding security constructs.

Results & Discussion

Automating Azure deployments with ARM templates yields significant improvements in consistency and efficiency. We compare manual and ARM-based deployments, present empirical and hypothetical results on deployment speed and cost, and discuss the benefits and limitations observed.

ARM Templates vs. Manual Deployment

Deployment Time: One of the obvious advantages of IaC automation is faster deployments. The manual resource deployment (e.g., clicking through the Azure Portal or imperative CLI commands for each resource), one-at-a-time, is slow and ordered. Each resource can be set up separately and the human operator must wait for each step. With ARM templates, Azure can parallelly deploy a lot of resources as long as their dependencies have been met. This translates to a complex environment that may take hours to setup manually can be deployed within minutes using a template. As well, once a template is written, deploying it into another environment (set up another identical test environment, for example) is simply a matter of running the deployment again, maybe with a different parameter file. Figure 3 represents the deployment time of one iteration of a hypothetical scenario. First, creation and deployment of an ARM template requires some effort, (approximately 40mins) when compared to 60mins to write and deploy the same template through manual means. The ARM deployment literally becomes one command and when used in subsequent iterations (deploying to new environment or repeating the deployment – e.g. 10 minutes vs. 60 minutes per each manual approach).

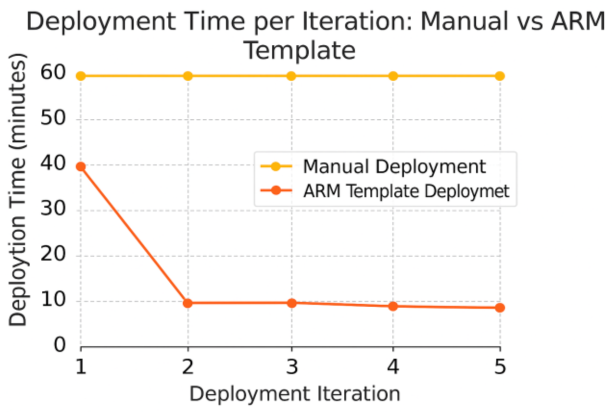
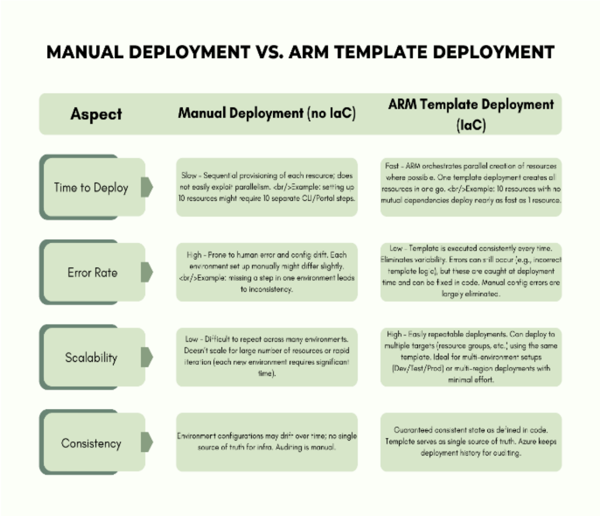


Figure 3: Deployment time per iteration for a complex environment – manual vs. ARM template. After the initial template authoring, reusing the ARM template drastically cuts provisioning time in later iterations. Manual deployments consistently take longer due to step-by-step setup for each environment (hypothetical example).

This speed advantage is reflected in industry practice. For example, such usage of IaC involves spinning up test environments demand basis and tearing them down, which brings agility. The ability of Azure Resource Manager to perform the parallel deployments makes this process further faster.

Error Rate: Manual processes introduce errors into the field – the engineers might not configure the setting correctly, or miss a step and the result will be inconsistencies (e.g., the staging environment was forgotten in the config/setup and therefore differs from production). Through ARM templates, the deployment process has been automated and repeatable thus significantly lowering such human errors. The template succeeds or fails entirely, which means if it works, it should work again the next (in case the input parameters are right). As AWS says (in relation to IaC in general): "Manual configuration is error-prone…. By comparison; IaC minimizes errors and facilitates error checking" [2]. Validation is part of ARM templates – if you specify an incorrect property the deployment should fail fast. Even though a failed deployment is never a good thing, it is usually better than a silent misconfiguration that occurs with a manual set up. Teams have come to a higher level of confidence in the setup of their infrastructure with templates due to the fact that every change is tracked and intentional and mistakes are caught before they are deployed through code review processes.

Scalability and Consistency: Manual deployment simply fails to scale when scaling up to multiple environments or complex systems. IaC glows in such situations by enabling easy replication of environments. For instance, in the case where an organization would need to deploy the same set of resources with each new customer or region, a parameterized ARM template could be deployed multiple times with various parameters, and the identical stacks would be created. This was conventionally a very painful task to complete manually. As mentioned within the context of general IaC, one can "use IaC to duplicate the exact same environment and quickly make the new deployment operational. IaC eradicates the redundant manual steps and checklists that were necessary in the past." [2]. In Azure ARMA Templates facilitate this kind of scalability – you can launch multiple resource groups with the same infrastructure definition simultaneously. Consistency is thus enforced: each deployment based from the same template produces identical resource configuration (unless inputs are different). Table 1 summarizes these comparisons:



MANUAL DEPLOYMENT VS. ARM TEMPLATE DEPLOYMENT

Aspect	Manual Deployment (no IaC)	ARM Template Deployment (IaC)
Time to Deploy	Slow - Sequential provisioning of each resource; does not easily exploit parallelism. <br/>Example: setting up 10 resources might require 10 separate CLI/Portal steps.	Fast - ARM orchestrates parallel creation of resources where possible. One template deployment creates all resources in one go. <br/>Example: 10 resources with no mutual dependencies deploy nearly as fast as 1 resource.
Error Rate	High - Prone to human error and config drift. Each environment set up manually might differ slightly. <br/>Example: missing a step in one environment leads to inconsistency.	Low - Template is executed consistently every time. Eliminates variability. Errors can still occur (e.g., incorrect template logic), but these are caught at deployment time and can be fixed in code. Manual config errors are largely eliminated.
Scalability	Low - Difficult to repeat across many environments. Doesn't scale for large number of resources or rapid iteration (each new environment requires significant time).	High - Easily repeatable deployments. Can deploy to multiple targets (resource groups, etc.) using the same template. Ideal for multi-environment setups (Dev/Test/Prod) or multi-region deployments with minimal effort.
Consistency	Environment configurations may drift over time; no single source of truth for infra. Auditing is manual.	Guaranteed consistent state as defined in code. Template serves as single source of truth. Azure keeps deployment history for auditing.

Table 1. Manual Deployment vs. ARM Template Deployment

In practice, implementation of ARM templates can significantly reduce the time needed to provision infrastructure for the new projects as well as eliminate the "works on my machine" syndrome, as infrastructure variances result in application problems. By introducing ARM templates to CI/CD, teams get ongoing release of infrastructure – e.g. a change to a template (changing VM size, adding new resource) is code reviewed, and put through the pipeline of care, just like application code, thus making the updates traceable and controlled.

Impact on Deployment Speed and Cost

Putting aside the qualitative gains, there are quantitative benefits, including frequency of deployment and cloud cost. More times it is deployed as the process become automated (it's easier and safer to automate). This will lead to more iterative improvement processes and rapid value delivery. Cost optimization is also one of the considerations; despite the fact that the main goal is speed and error minimization:

Cost Optimization: ARM templates can, in several ways, indirectly result in ease of optimizing cost. First, templates render use of best practice such as right sizing of resource and absent or lack of deployment of unnecessary components possible. Infrastructure is therefore code and engineers can then more easily review and discuss the need for each resource within a template (as in code review) which (it is hoped) will return an over-provisioned SKU before the cost is actually spent. Second there is automation possibility exists to tear down and recreate environments on demand rather than resources running (and anyway paying costs) in anticipation for the manual effort to put them back together. For example, if a dev/test environment is described by a template, it might be automatically predictable in the morning and automatically destroyed at night, and that can save hundreds of hours of cloud runtime cost. This is not mechanizable to do manually but ease of doing it for an automated template deployment in a scheduled pipeline.

Also ARM templates also fit nicely together with Azure's governance tools, including tags and Azure Policy. Showback/chargeback is possible with the help of tags applied using template (e.g., tagging of resources with project or environment ID's). ARM itself is a part of Azure's cost governance, by standardizing setups, not ending up in "snowflake" installations that lead to nasty surprises. As outlined by Infracost, the (FinOps tool vendor) ARM delivers a consistent layer that is supportive of standardized resource deployments, policy adherence or better visibility of the infrastructure that is all important for cloud cost optimization. In other words, there is 'less work' by every resource deploying through ARM (with known templates) to be able to guarantee that it conforms to cost saving configurations (for example using reserved instance, dev environments would use lower cost SKUs, etc.).

To understand in a very simple manner what a cost weighing exercise may look like consider a situation where manual deployment could leave some inefficiencies (say default SKUs or services left active longer) and where on the hand codified deployment could have used optimal SKUs and turned services off not required off hours. Figure 4 shows a supposition difference in the monthly cost:

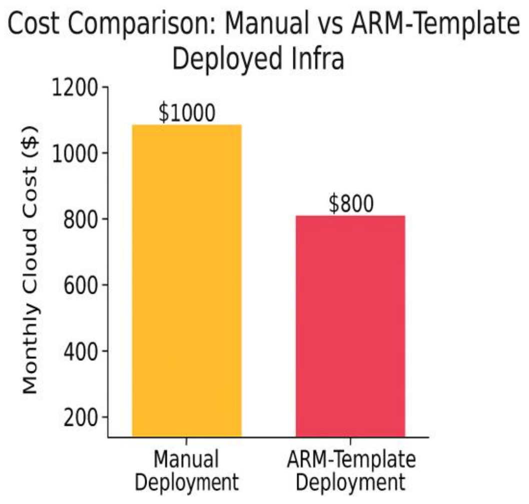


Figure 4: Hypothetical cost comparison for an application infrastructure deployed manually vs. via ARM templates. In this scenario, the ARM template deployment uses optimized resource choices (Standard tier, minimal sizing for non-prod, automated schedules), costing less per month than the manually deployed counterpart. Tags and policies applied through ARM also avoid unexpected costs.

In this instance, 20 % of the costs were saved by the approach delineated by ARM. It is natural, however, that IaC itself does not ensure cost savings – but it does provide mechanisms (repeatability, parameterization, policy integration) that enable the simplification of the roll out of cost controls. For instance, someone may be able to add to a template a policy assignment that blocks deployment of expensive VM sizes, or simply ensure that all dev resources are S, not M by default. Such consistency is hard to impose on using ad-hoc approach.

Limitations and Challenges (Discussion): Despite the profit that the ARM templates bring, the users have met some challenges including:

- **Verbosity and Complexity:** ARM JSON syntax can be quite verbose with large numbers of deployments. Managing and creating a large JSON file requiring complex nesting is risky. Even once using Visual Studio Code extensions and tooling, many found the learning curve of getting to grips with the functions and syntax of the ARM template to be steep. This was a major contributing factor to Azure Bicep's development, since Bicep provides a more concise syntax, yet compiles down to ARM JSON. For instance, an item that is 200 lines of JSON is written in 50 lines of Bicep. In pure ARM JSON complex expressions (string concatenations for names or conditions) are very hard to look at and debug.

- **Debugging and Error Handling:** When a template deployment of an ARM fails, sometimes the error messages from Azure can be confusing. For example, an error can state that a specific resource provisioning failed but cannot clearly state which template line it resulted from. There is no interactive template debugger – error output during deployment, unless – Activity logs in Azure Portal. Utilities such as ARM-TTK will pick up some things if you use them properly before deployment (e.g., naming conventions, property best practices), but it doesn't validate the logic. The appearance of the what-if deployment preview goes some way – Azure's what-if operation is able to demonstrate which resources would be created/modified/deleted by the template deployment without doing actual changes. This is like Terraform's "plan" step. However not all failures of what-if can be caught – for example if the template refers in a place where there is no resource – then what-if will flag it, but unfortunate for the logic not accurate slightly (say a string concatenation led to a name that isn't valid), you may only learn about it only on actual deployment. To put it briefly, testing ARM templates is difficult; there is no dry run built in that guarantees success so you often must deploy to a test resource group to fully test out a complex template.

- **Lack of Native Modular Structure in JSON:** As talked about, you can reference templates, but that requires hosting the sub-templates and introduces external dependencies. The lack on flat JSON structure lacks the concept of modular includes (outside of it utilising nested deployment resources). Consequently, decomposing a solution into reusable components is not a straight forward process in raw ARM. It's again rectified by Bicep, using modules (even you can call one bicep file from another easily), but in plain arm json, one either evolves into maintaining one giant file or manages a set of templates manually.

- **State and Idempotency Issues:** ARM deployments are also implicitly idempotent, and there is no need to track separate state files (Azure understands how it has the resources). Generally this is good, but one limitation is that in the case one needs to, say, destroy resources, ARM templates at resource group scope don't have a way to directly "delete everything not in this template" except in Complete mode, which can be dangerous if not careful. In complete mode, it will delete the resources in the target scope that are not defined in the template, which will make the real state on the exact (same) copy of the template. Terraform users occasionally wonder why in ARM templates there's no simple delete workflow – you utilize either Azure CLI/PowerShell to delete resources, or you go with complete mode templates or Azure Blueprints for cleanup. Also, some sophisticated scenarios (e.g creating an resource, retrieving its output, base on that decision, another resource within the same template) can't be done in one ARM template – you may have to chain deployments or use scripts that a general-purpose language (or Pulumi) could manage that logic.

- **Testing and Simulation:** However, there is no "official unit test" framework for ARM templates except by deploying them and trying to get them to work. As infrastructure code, this is area of growth – there are some third-party tools that enable local simulation at least, or verification of templates to the Azure schemata. What-if by Microsoft is useful but sometimes not 100% true for complex changes. For this reason, practitioners tend to keep separate test Azure subscriptions or resource groups with which to regularly test deployments (sometimes automated nightly deployments to check that templates still work as Azure evolves).

One mitigation on error handling is, of course, to divide deployments into smaller units e.g. deploy the network first and then VMs rather than one massive template so that debugging can be done in more isolated segments. Azure also supports deployment calls both incremental mode (the default) and complete mode and this is variable with the installation process. However, normally incremental mode is safer for updates (won't modify existing resources not in the template) but if you remove resource from the template it will not be removed from Azure. Such nuances have to be harnessed as part of governance processes.

These restrictions notwithstanding, many Azure practitioners do manage huge infrastructures with the templates but still resorting (now often) to Bicep: for authoring. The constraints have driven alternatives and enhancements (which will be discussed next) but there is a distinct impression that for Azure-centric deployments, ARM templates offer a degree of native integration (deployment history in Azure Portal, no external state management, immediacy of new Azure services) which third party tools cannot possibly match.

Challenges & Alternatives

Although ARM templates are powerful in terms of Azure automation, engineers have identified some alternatives to cover the challenges of these ones:

Challenges Recap: It tends to be burdensome to debug large JSON templates and to deal with complex deployments. As the infrastructure and the teams grow, it is error prone to maintain dozens of JSON files with thousands of lines. In addition, various organizations might choose to use a single IaC tool on several clouds while the counterpart of ARM templates, which is Azure-specific.

Alternative IaC Tools:

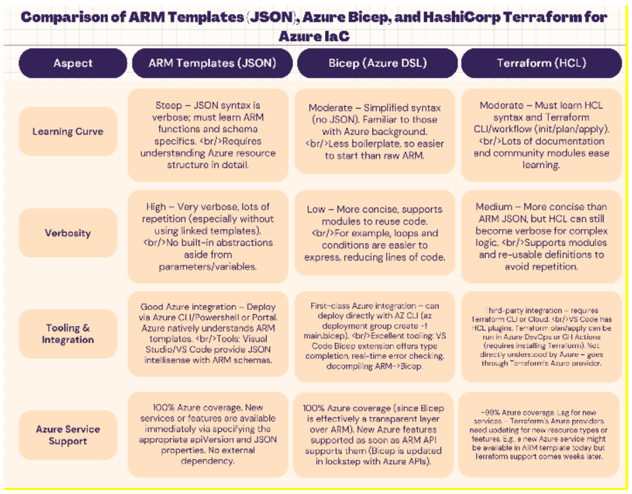
- **HashiCorp Terraform:** An immensely popular open source IaC tool, the tool describes the infrastructure using its exclusive language HCL (HashiCorp Configuration Language). Terraform is not cloud-agnostic – one Terraform deployment can be used to provision Azure, AWS, GCP, etc., with a plugin architecture of providers. A lot of organizations prefer Terraform due to multiple-cloud or homogeneity reasons. When comparing with ARM: Terraform has a more extensive language for abstractions (modules, loops, conditional resources), a vibrant ecosystem and matured and a plan command that explains what will change, prior to applying. However, Terraform demands management of state (which is typically done in a backend such as Azure Storage or Terraform Cloud), which adds its own complexity. Terraform also, sometimes, falls short in supporting the latest Azure features since the Azure provider needs an update. On the other side, ARM templates can take any new Azure resource/property as soon as it is released in the Azure REST API without a need to have an update on plugin. The learning curve of Terraform is moderate – all one needs to learn HCL and the Terraform CLI, and HCL is usually considered closer to concise than raw JSON.

- **Pulumi:** Members of a brand new entrant that allows you to code infrastructure code in generic programming languages (TypeScript, Python, C#, Go, among others). Pulumi, then, provisions resources through cloud SDKs. For instance, using Pulumi, one could write a Python script that will create an Azure VNet and VM courtesy of Pulumi's Azure Native provider (which behind the scenes uses Azure's REST API directly). It gives us the entire strength of programming (loops, conditions, complex logic, external package importations) to IaC. It's very flexible and easy to integrate with existing developer workflows (because you can use the same language as your app). However, it demands that developers must learn both programming language and cloud SDKs. Pulumi operates with state like Terraform. It is a suitable one for strong software engineering background teams, that want to treat infrastructure as software.

- **Azure Bicep:** As mentioned Bicep is basically "ARM Templates 2.0" in terms of authoring experience. It was created by Microsoft to make ARM template creation easier. Bicep has a criptic syntax and closer to C# or JavaScript without the JSON quotes and braces overhead. It supports modules, loops, conditions, full support for all the Azure resource types (again, because it directly maps to ARM). One large advantage – the lack of a state file to manage (it uses Azure's inherent state), and that is one less thing to be concerned about with Terraform [7]. Bicep files are transpiled into ARM JSON at deployment time (either when using the command line interface, or programmatically when deploying a .bicep file via the Azure CLI/PowerShell). Therefore, you receive the advantages of a cleaner syntax, but continue utilizing the powerful ARM engine for deployment. Bicep is Azure-specific (unlike non-Azure resources it cannot deploy), but for the teams using Azure, it has become much more popular than raw ARM JSON now. Bicep has a relatively easy learning curve for those that are already familiar with ARM concepts – many can learn it faster than learning Terraform, in part because Bicep is designed to feel natural to Azure users.

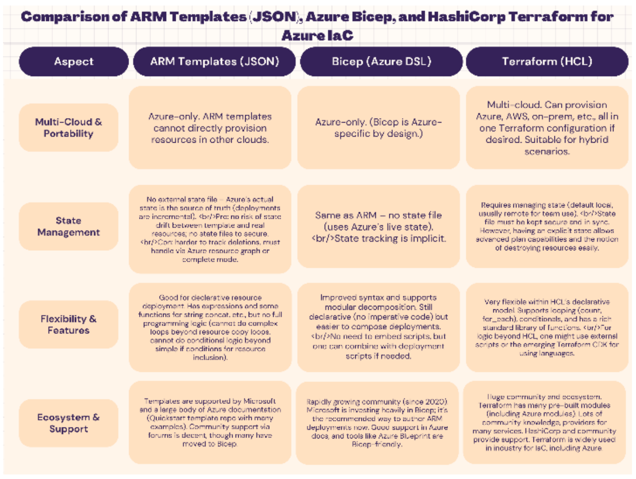
- **AWS CloudFormation (for context):** CloudFormaion (JSON/YAML based) is AWS's equivalent to ARM templates. Although not relevant to Azure, it's interesting to note that the industry trend began when such template-driven IaC came in (CloudFormation pre-dates ARM templates by a couple years). Azure ARM is for Azure what similar in spirit is. Later, came Terraform to standardize cloud IaC and now each cloud owns its DSLs (Azure's Bicep, AWS CDK etc.) trying to learn from it.

For Azure deployments, most of the major decision points narrow down to ARM/Bicep vs Terraform (Pulumi is on the rise, but less common). Table 2 makes comparisons between ARM templates, terraform, and Bicep against a few dimensions.



Comparison of ARM Templates (JSON), Azure Bicep, and HashiCorp Terraform for Azure IaC

Aspect	ARM Templates (JSON)	Bicep (Azure DSL)	Terraform (HCL)
Learning Curve	Steep – JSON syntax is verbose; must learn ARM functions and schema specifics. <br/>Requires understanding Azure resource structure in detail.	Moderate – Simplified syntax (no JSON). Familiar to those with Azure background. <br/>Less boilerplate, so easier to start than raw ARM.	Moderate – Must learn HCL syntax and Terraform CLI/workflow (init/plan/apply). <br/>Lots of documentation and community modules ease learning.
Verbosity	High – Very verbose, lots of repetition (especially without using linked templates). <br/>No built-in abstractions aside from parameters/variables.	Low – More concise, supports modules to reuse code. <br/>For example, loops and conditions are easier to express, reducing lines of code.	Medium – More concise than ARM JSON, but HCL can still become verbose for complex logic. <br/>Supports modules and re-usable definitions to avoid repetition.
Tooling & Integration	Good Azure integration – Deploy via Azure CLI/Powershell or Portal. Azure natively understands ARM templates. <br/>Tools: Visual Studio/VS Code provide JSON intellisense with ARM schemas.	First-class Azure integration – can deploy directly with AZ CLI (az deployment group create -f main.bicep). <br/>Excellent tooling: VS Code Bicep extension offers type completion, real-time error checking, decompiling ARM->Bicep.	Third-party integration – requres Terraform CLI or Cloud. <br/>VS Code has HCL plugins. Terraform plan/apply can be run in Azure DevOps or GH Actions (requires installing Terraform). Not directly understood by Azure – goes through Terraform's Azure provider.
Azure Service Support	100% Azure coverage. New services or features are available immediately via specifying the appropriate apiVersion and JSON properties. No external dependency.	100% Azure coverage (since Bicep is effectively a transparent layer over ARM). New Azure features supported as soon as ARM API supports them (Bicep is updated in lockstep with Azure APIs).	~99% Azure coverage. Lag for new services – Terraform's Azure providers need updating for new resource types or features. E.g., a new Azure service might be available in ARM template today but Terraform support comes weeks later.



Comparison of ARM Templates (JSON), Azure Bicep, and HashiCorp Terraform for Azure IaC

Aspect	ARM Templates (JSON)	Bicep (Azure DSL)	Terraform (HCL)
Multi-Cloud & Portability	Azure-only. ARM templates cannot directly provision resources in other clouds.	Azure-only. (Bicep is Azure-specific by design.)	Multi-cloud. Can provision Azure, AWS, on-prem, etc., all in one Terraform configuration if desired. Suitable for hybrid scenarios.
State Management	No external state file – Azure's actual state is the source of truth (deployments are incremental). <br/>Pro: no risk of state drift between template and real resources; no state files to secure. <br/>Con: harder to track deletions, must handle via Azure resource graph or complete mode.	Same as ARM – no state file (uses Azure's live state). <br/>State tracking is implicit.	Requires managing state (default local, usually remote for team use). <br/>State file must be kept secure and in sync. However, having an explicit state allows advanced plan capabilities and the notion of destroying resources easily.
Flexibility & Features	Good for declarative resource deployment. Has expressions and some functions for string concat, etc., but no full programming logic (cannot do complex loops beyond resource copy loops, cannot do conditional logic beyond simple if conditions for resource inclusion).	Improved syntax and supports modular decomposition. Still declarative (no imperative code) but easier to compose deployments. <br/>No need to embed scripts, but one can combine with deployment scripts if needed.	Very flexible within HCL's declarative model. Supports looping (count, for_each), conditionals, and has a rich standard library of functions. <br/>For logic beyond HCL, one might use external scripts or the emerging Terraform CDK for using languages.
Ecosystem & Support	Templates are supported by Microsoft and a large body of Azure documentation (Quickstart template repo with many examples). Community support via forums is decent, though many have moved to Bicep.	Rapidly growing community (since 2020). Microsoft is investing heavily in Bicep; it's the recommended way to author ARM deployments now. Good support in Azure docs, and tools like Azure Blueprint are Bicep-friendly.	Huge community and ecosystem. Terraform has many pre-built modules (including Azure modules). Lots of community knowledge, providers for many services. HashiCorp and community provide support. Terraform is widely used in industry for IaC, including Azure.

Table 2. Comparison of ARM Templates (JSON), Azure Bicep, and HashiCorp Terraform for Azure IaC

The considerations in practice may boil down to these considerations.

- If an organization is Azure-only and uses official tooling – Bicep (and thus ARM) is a wonderful choice – without external dependencies, build on Azure's native deployment engine.

- If the organization has multi-cloud needs or existing Terraform tendencies, Terraform presents one workflow to rule them all at the cost of inserting another layer on top of Azure.

- Even some teams use a mix: For example, use Bicep/ARM for some things, and terraform for others, although that requires careful coordination to prevent conflicts.

It should be noted though that these tools are not mutually exclusive with ARM templates – Bicep compiles to ARM templates and Terraform's Azure provider finally invokes the Azure Resource Manager's APIs (i.e. just like an ARM template deployment but in a procedural fashion). The Azure Native provider by Pulumi also relies, under the hood, directly on ARM. So, ARM (as platform capability) is always on the game. these tools just constitute a different authoring or orchestrating experience.

Emerging Trends – AI and Advanced Tooling: In terms of the future, we observe attempts to make further cloud automation simpler. For instance, projects, such as IaC from higher level designs and natural language, are being explored. One could picture an AI powered service where you specify the infrastructure (or it monitors your running infra) and it creates an ARM template or a Bicep file for you. Even Microsoft's own Azure Quickstart Center and template exporters can deconstruct templates from resources already present (the "export template" capability of the Azure Portal supports resource groups). In the future AI could help to optimize these templates – for example suggest more optimal configurations or detect anomalies in template definition before deployment. Coupling ARM deployments with AI planning can, for example, imply the identification of an appropriate SKU by virtue of past use or advice on the addition of an auto scaling configuration, for example. Such AI integrations may, while speculative, reduce the manual effort in writing IaC and planning capacity further. Azure's deployment what-if analysis is already a step towards wiser deployment planning (basically the platform is "predicting ahead" of what will change). We may be winners in terms of more automated rollback strategies or self-healing deployments which in case a deployment fails, the system may analyze and correct (if you tried a deployment and for whatever reason the deployment failed, the system may have attempted to analyze and correct the issue). as future improvements.

Conclusion

Cloud architects and DevOps engineers are offered clear benefits when automating deployments in Azure through the use of ARM templates. With the introduction of Infrastructure-as-Code using ARM templates, teams get consistent, repeatable and auditable infrastructure deployment. This paper described how ARM templates, as Azure's native IaC solution, provide the possibility of the entire cloud environment (compute, networking, storage, security) definition via declarative JSON files which can be under version control and within CI/CD pipelines. We showed how using ARM templates results in faster time to deploy, less error, simpler scalability in comparison with manual provisioning, that fits DevOps objectives of agility and reliability [2].

They are particularly suitable for Azure-native workloads – the cases where all that's needed are in Azure and full use of the Azure Resource Manager can be made. They shine when you need to launch complex Azure services (anything from a simple web app to a full AKS cluster with supporting resources) and desire first-party support and on-demandness (e.g., ability to deploy any new Azure service the day it is launched). ARM templates make sure that all these resources are deployed as a single unit and that they are managed by Azure's access and control policies. Moreover, deployment history and what-if analysis are some of the features that ensure that ARM is a strong option for enterprise deployments.

However, we also thought that ARM JSON, in its raw form, has its drawbacks – majorly verbosity and user un-friendliness. Azure Bicep provides a welcome and useful improvement for organizations/projects that are pain points in that by making template authorship simpler while still using ARM underneath. Where Azure forms part of a bigger multi-cloud strategy or a more standardized IaC tool is desired then third party solutions such as Terraform or Pulumi may be opted for, as an either/or or complementary to ARM-based templates. Each comes however with trade offs such as in complexity, flexibility and the support ecosystem.

After all, ARM templates (which are further reflected in Bicep) is a strong tool for Azure automation. They make possible a DevOps-centric infrastructure lifecycle: from design (in code), to continuous deployment (through pipelines), to maintenance (with incremental updates and tracking) even up to message (in complete mode or with scripts). In addition to acceleration and standardization of deployment, benefits of using ARM templates would include more appropriate coordination of infrastructure changes with software development process (in review and testing). For any Azure project of average or above-average complexity, we suggest using ARM templates or Bicep, and we advise treating the IaC as seriously as application code.

For future improvements, we foresee even more intimate integration of ARM deployments with intelligent tooling. Types of service like what-if in constitutes ongoing investment into project bicep by Microsoft and shows the path to more accessible and secure azure iac. There could be thoughts of AI assisted template authoring, or more advanced deployment orchestrators which can test and check the change before going live (further reducing risk). Perhaps, gradually, a manual change management could evolve into an automated version of such a process. In addition, options to simulate deployments in offline or improved error diagnostics would significantly improve the flow of the ARM template workflow.

Ultimately, the ability to automate Azure deployments through use of Azure Resource Manager templates is a best practice for the cloud architecture and DevOps engineers who are out to create reliable and scalable cloud infrastructure management. It uses all the power of the platform offered by Azure while maintaining order using code. Having a wide selection of IaC tools to choose from nowadays, teams can be flexible as to which approach suits them best, but broad knowledge of ARM templates serves a fundament on which other tools are placed. Organizations can deploy software and infrastructure changes faster with greater confidence while using IaC on Azure, which is a competitive advantage in the cloud-driven world.

References

1. S. Kuehn, "An Intricate Look at ARM Templates – Part 1 – Background and History," Aug. 2019. Referred From: https://www.refactored.pro/blog/2019/8/15/arm-templates-part-1
2. Amazon Web Services, "What is Infrastructure as Code? – IaC Explained," AWS whitepaper, 2023. Referred from: https://aws.amazon.com/what-is/iac/#:~:text=Reduce%20configuration%20errors
3. Element Digital, "Infrastructure as Code vs Manual Deployment: Why IaC Isn't Always the Answer," Element Digital Blog, 2023. [Online]. Available: elementdigital.com.au.
4. Spacelift, "What is an Azure ARM Template? Overview, Tutorial & Examples," Spacelift Blog, 2024. Referred from: https://spacelift.io/blog/arm-template
5. E. Borzenin, "ARM template design choices and deployment time," Infrastructure as Code – Borzenin Blog, August 17, 2020. Referred from: https://borzenin.com/arm-templates-deployment-time/
6. Microsoft Azure, "Comparing Terraform and Bicep," Microsoft Docs – Azure DevOps/Infrastructure, Mar. 2023. Referred from: https://learn.microsoft.com/en-us/azure/developer/terraform/comparing-terraform-and-bicep?tabs=comparing-bicep-terraform-integration-features
7. Reddit user discussion, "Bicep vs Terraform," r/Azure on Reddit, 2022. Referred from: https://www.reddit.com/r/AZURE/comments/yoy1in/bicep_vs_terraform/?rdt=37514#:~:text=The%20biggest%20advantage%20of%20Bicep,the%20current%20resources%2C%20to