International Journal of Emerging Trends in Engineering and Management Research (IJETEMR)

|ISSN: 2455-7773 | www.ijetemr.com| A Double-Blind Peer-Reviewed and Bimonthly Scholarly Journal|

|Volume 9, Issue 4, July-August 2024|

DOI:10.15662/ijetemr.2024.0904005

Normalize the Inputs, Not the Clouds: Schema-Validated Profile Generation and Collision-Free Address Planning for Multi-Cloud Infrastructure as Code

Ramakrishna Kommineni

Senior DevOps Engineer, o9 Solutions, Dallas, TX, USA

ABSTRACT: A platform that runs on several public clouds must decide where to absorb the differences between them. A common resource model across providers tends to reduce each cloud to the features all of them share, while separate per-cloud code leaves operators with a different procedure, input format, and address plan for every provider. This paper presents an arrangement summarized as *normalize the inputs, not the clouds*, designed by the author for a multi-tenant enterprise SaaS platform on Amazon Web Services (AWS), Microsoft Azure, Google Cloud, and a fourth, regional cloud provider. Provider-specific Terraform modules use each cloud's native mechanisms, such as OpenID Connect federation for pod identity on AWS, managed identities on Azure, and Workload Identity Federation on Google Cloud. Above them, a provider-neutral generator with per-cloud subcommands validates operator inputs against JSON Schema, plans address space with a recursive CIDR splitting algorithm and a shared reservation ledger that prevents overlaps when environments are provisioned concurrently, and renders Terragrunt profiles from Jinja2 templates. A converter brings legacy Ansible-managed environments into the same form by reading current sizes, network settings, and cost tags from live cloud APIs, and a pipeline runs generated profiles in plan and apply modes with per-build workspace isolation. We describe the design, its production use and lessons, and a methodology for evaluating it.

KEYWORDS: Infrastructure as code, multi-cloud, Terraform, Terragrunt, code generation, JSON Schema, IP address management, CIDR allocation, brownfield migration

I. INTRODUCTION

Infrastructure as code (IaC) manages networks, clusters, and identities through versioned definitions instead of manual procedures [1]. Research on IaC has grown around frameworks, usage, defects, and testing [2], and practitioner interviews show that adopting it raises its own maintenance and tooling challenges [3]. These sharpen when one product runs on several clouds: the same environment, with a network, a Kubernetes cluster, node pools, workload identities, and firewall policy, must be built from different primitives on each provider.

Two responses are common. The first defines a provider-independent resource model and translates it to each cloud. Such a model can express only what every provider supports, so cloud-specific capabilities are either dropped or reintroduced as exceptions. The second keeps separate code for each cloud and lets each team provision in its own way. That preserves native capabilities but leaves operators with different input formats, different checks, and address plans that are coordinated by convention, which fails once environments are created concurrently.

This paper describes a third arrangement, designed by the author for a multi-tenant enterprise SaaS platform: *normalize the inputs, not the clouds*. The Terraform modules remain provider-specific and use each cloud's native mechanisms. What is made uniform is the path from operator intent to deployable code: one generator, one validation discipline, one address planner, and one execution pipeline. The paper makes four contributions. (1) *Schema-validated generation:* operator inputs are validated against JSON Schema and rendered with Jinja2 templates into Terragrunt profiles by a command-line interface (CLI) with per-cloud subcommands, so malformed inputs are rejected before any profile exists. (2) *Collision-free address planning:* a recursive CIDR splitting algorithm carves each environment's range into public, private, service, and cluster subnets, and a shared reservation ledger prevents overlaps between environments provisioned concurrently. (3) *Brownfield migration:* a converter turns legacy Ansible profiles into generator invocations by reading current sizes, network settings, and cost tags from live cloud APIs. (4) *Isolated execution and an evaluation plan:* a

|ISSN: 2455-7773 | www.ijetemr.com| A Double-Blind Peer-Reviewed and Bimonthly Scholarly Journal|

|Volume 9, Issue 4, July-August 2024|

DOI:10.15662/ijetemr.2024.0904005

pipeline runs generated profiles in plan and apply modes with per-cloud credentials and per-build workspaces, and we define how the design should be evaluated in production.

Section II reviews related work. Section III states the problem and requirements. Section IV presents the design, Section V its production deployment, and Section VI the evaluation methodology. Section VII discusses limitations, and Section VIII concludes.

II. BACKGROUND AND RELATED WORK

A. Terraform and Terragrunt

Terraform modules package related resources behind input variables [4], and those variables can carry validation rules that Terraform evaluates when it processes the configuration [5]. Terragrunt organizes deployments as units, each a directory whose configuration deploys one piece of infrastructure and can be operated on its own [6]. Terraform's plan command previews the changes a configuration would make without applying them [7], and remote state backends lock state to prevent concurrent writes [8]. The design uses all of these unchanged.

B. Multi-Cloud Abstraction

Pulumi components group resources behind a single interface so that consumers create complex infrastructure through a simple one [9]. Crossplane compositions act as templates that create several resources as one composite resource, and a platform team can back one composite API with different compositions [10]. A uniform interface over provider-specific implementations is therefore an established idea, and this paper does not claim it. Here, normalization happens at generation time, outside any control plane, and produces ordinary Terragrunt code that is reviewed through a plan before it is applied. The generator also plans address space against a shared ledger and converts legacy environments, two concerns these frameworks leave to the user.

C. Validation and Templating

JSON Schema describes the structure of JSON documents and the constraints on them [11], and Jinja2 is a general-purpose text template engine [12]. Terraform's variable validation applies to one module at a time and runs when Terraform evaluates that module. A schema over the generator's inputs checks an environment's inputs as a whole and before any profile exists, which moves the error to the point where the operator is still entering values.

D. Address Management

Private networks draw on the address blocks reserved by RFC 1918 [13], and classless prefixes allow those blocks to be divided at arbitrary prefix lengths [14]. Cloud providers offer address management services: Amazon VPC IP Address Manager organizes address space into pools and allocates CIDRs to VPCs [15], and IP address management in Azure Virtual Network Manager assigns nonoverlapping CIDRs from pools [16]. Each service manages space within its own provider. Terraform state locking does not address the problem either, because every environment has its own state, and two runs that write different states can still choose the same range.

E. Bringing Existing Infrastructure Under Code

Terraform can import existing resources and generate configuration for them [17], and Terraformer generates configuration and state files from existing infrastructure [18]. Both work resource by resource and produce code that mirrors what exists. The converter described here works one level higher: it reconstructs the generator's inputs, so a migrated environment is expressed through the same modules and profile structure as a new one.

III. PROBLEM AND REQUIREMENTS

The platform runs isolated tenant environments, each a set of cloud resources on one provider: a virtual network and subnets, a managed Kubernetes cluster and its node pools, workload identities, firewall and egress configuration, and supporting storage. Environments are created regularly, sometimes concurrently, by operators who should not need deep knowledge of every cloud. Some environments predate the IaC platform and were provisioned with Ansible.

We derive six requirements. (R1) *Native fidelity:* each cloud uses its own mechanisms for identity, network filtering, egress, and node capacity, with no reduction to a common subset. (R2) *Uniform operation:* operators use one tool and one input discipline on every cloud. (R3) *Early rejection:* malformed inputs are rejected before any profile is produced. (R4) *Collision-free addressing:* no two environments receive overlapping ranges, including when they are provisioned concurrently. (R5) *Brownfield continuity:* existing environments can be expressed in the generated form without changing

International Journal of Emerging Trends in Engineering and Management Research (IJETEMR)

|ISSN: 2455-7773 | www.ijetemr.com| A Double-Blind Peer-Reviewed and Bimonthly Scholarly Journal|

|Volume 9, Issue 4, July-August 2024|

DOI:10.15662/ijetemr.2024.0904005

their sizes, networks, or cost attribution. (R6) *Reviewable, isolated execution:* every change can be previewed before it is applied, and concurrent builds do not interfere.

IV. DESIGN

A. Two Layers

Fig. 1 shows the design. The lower layer is a library of provider-specific Terraform modules (R1). The upper layer is a provider-neutral generator that turns operator inputs into Terragrunt profiles, the per-environment set of units that call those modules with concrete values (R2). Legacy environments enter through a converter that produces ordinary generator invocations, and one pipeline executes every profile.

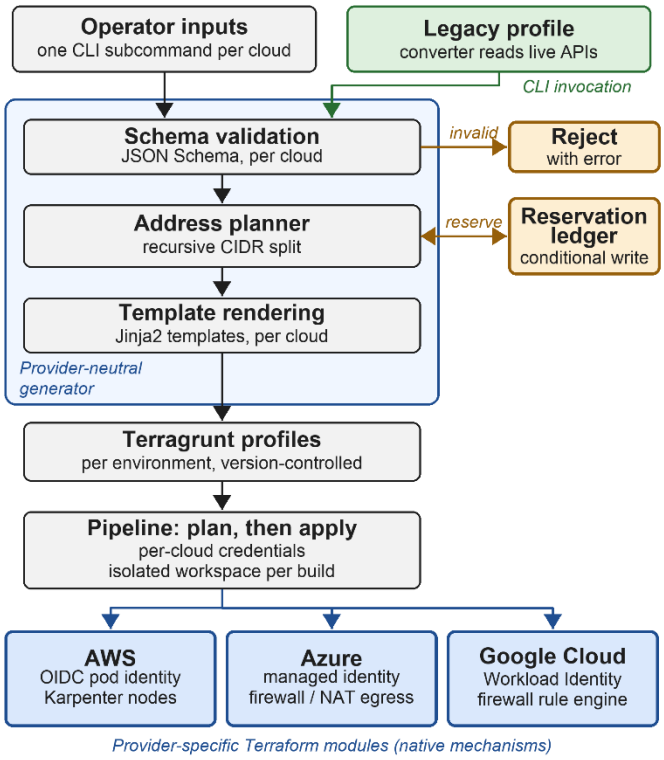


Fig. 1. Provider-neutral generation over provider-specific modules. Inputs from operators or from the migration converter are validated, assigned address space against a shared reservation ledger, and rendered into Terragrunt profiles that a pipeline plans and applies. The fourth, regional provider follows the same pattern and is omitted.

B. Provider-Specific Modules

Table I lists the native mechanisms the modules use. On AWS, the cluster module creates the EKS cluster and its OpenID Connect identity provider and chains IAM role assumptions, so that pods assume IAM roles without static credentials [19]; Karpenter provisions nodes matched to the requirements of pending pods. On Azure, AKS clusters use managed identities [20]. The virtual network module supports two egress modes, a NAT gateway or Azure Firewall, selected per environment, and the firewall module configures the Premium tier with intrusion detection and prevention, TLS inspection, and FQDN filtering [21]. Network security group rules are produced from templates expanded against environment parameters, giving a common baseline with per-environment overrides. On Google Cloud, private GKE clusters use Workload Identity Federation [22], Calico network policy, and release channels. Firewall rules [23] come from a data-driven engine: rules are declared as maps of source and target tags, protocols and ports, and priorities, and environment-specific maps are deep-merged over the defaults, so security posture can differ by environment without changing the module.

International Journal of Emerging Trends in Engineering and Management Research (IJETEMR)

|ISSN: 2455-7773 | www.ijetemr.com| A Double-Blind Peer-Reviewed and Bimonthly Scholarly Journal|

|Volume 9, Issue 4, July-August 2024|

DOI:10.15662/ijetemr.2024.0904005

TABLE I. NATIVE MECHANISMS USED BY THE PROVIDER-SPECIFIC MODULES

Concern	AWS	Azure	Google Cloud
Workload identity	IAM roles for service accounts via OIDC federation	Managed identity	Workload Identity Federation
Cluster	EKS with OIDC provider	AKS with managed identity	Private GKE, release channels, Calico policy
Node capacity	Karpenter provisioning	AKS node pool sizing	Autoscaled node pools, spot or preemptible, taints
Network	VPC with cross-account peering	VNet with firewall or NAT egress; templated NSG rules	VPC with data-driven firewall rules

The identity row shows why the modules are not forced into one interface. All three mechanisms let a workload obtain cloud credentials without a stored secret, but they rest on different trust relationships: a federated OIDC provider and role chain, an identity attached to the resource, and a binding between a Kubernetes service account and a cloud identity. A shared identity abstraction would have to hide these differences or expose all of them. The design instead lets the generator, which knows the target cloud, supply each module with the inputs that module expects.

C. Schema-Validated Profile Generation

The generator is a CLI with one subcommand per cloud. Table II lists the stages an invocation passes through. Inputs are validated against a JSON Schema [11] that declares required fields, types, and allowed values; an invocation that fails validation stops with an error message and produces nothing (R3). Valid inputs pass to the address planner and then to rendering, where Jinja2 templates [12] produce the Terragrunt units of the environment [6], each referencing a provider-specific module and supplying its inputs.

Cloud-specific logic stays in the subcommands. The AWS subcommand generates SSH key pairs, stores them in Amazon S3, and configures EKS node pools. The Azure subcommand plans VNet subnets, sizes AKS node pools, and integrates managed identities. The Google Cloud subcommand configures GKE, designs the VPC, stores keys in Cloud Storage, and sets up Workload Identity. Operators see one consistent interface, while each cloud receives its own implementation.

TABLE II. STAGES FROM INPUT TO DEPLOYED CHANGE

Stage	Action	Errors caught
Validate	Check inputs against the subcommand's JSON Schema	Missing, mistyped, or disallowed values
Plan addresses	Split the environment range; reserve it in the ledger	Overlap with any reserved range
Render	Fill per-cloud Jinja2 templates to produce Terragrunt units	Template errors

|ISSN: 2455-7773 | www.ijetemr.com| A Double-Blind Peer-Reviewed and Bimonthly Scholarly Journal|

|Volume 9, Issue 4, July-August 2024|

DOI:10.15662/ijetemr.2024.0904005

Stage	Action	Errors caught
Plan	Pipeline runs Terraform plan through Terragrunt	Provider-side rejections; unexpected changes
Apply	Pipeline applies the reviewed change	Runtime failures, reported per build

D. Collision-Free Address Planning

Each environment receives one block from the platform's private address space [13]. The planner carves the block into public, private, service, and cluster subnets with a recursive splitting algorithm [14]. A block larger than the subnet to be placed is divided into two halves of the next longer prefix; one half continues to be divided for the current subnet and the other is kept for the remaining ones, with larger subnets placed first. Because every subnet is a node of one binary tree of halvings rooted at the environment block, any two subnets are disjoint, and none can fall outside the block. Overlap inside an environment is therefore excluded by construction.

Overlap between environments is excluded by the reservation ledger, a DynamoDB table shared by all generator runs (R4). Environment blocks are aligned to a fixed prefix length within the platform's address pool, so two candidate blocks either coincide or are disjoint. The planner reserves a candidate with a conditional write that succeeds only if no reservation for that block exists [24]. When two runs choose the same block at the same time, exactly one write succeeds; the other run receives a condition failure, selects the next free block, and retries. The reservation is made at generation time, before any cloud resource exists, and one ledger serves the subcommands of all clouds, so the address plan holds across providers rather than within one provider's address management service.

E. Brownfield Migration

Environments provisioned before the platform existed were described by Ansible profiles. Those profiles record what was requested, not necessarily what is running, so the converter treats them as a starting point and reads the current state from the cloud: instance types and VM sizes from the AWS EC2, Azure Compute, and Google Compute APIs, network settings from the provider network APIs, and cost tags from the resources themselves. It then emits generator CLI invocations rather than profiles (R5). Table III summarizes how attributes are treated.

TABLE III. ATTRIBUTE HANDLING IN THE MIGRATION CONVERTER

Attribute	Read from	Treatment
Instance type, VM size	Live compute APIs	Mapped to node pool sizes; where two legacy roles share a pool, the larger type is kept
Windows service type	Legacy profile	Mapped to the matching node pool
Network settings	Live network APIs	Passed to the generator so the profile describes the existing network
Cost tags	Live resources	Carried into the generated profile unchanged

Emitting invocations instead of profiles means that a migrated environment passes through the same validation, address planning, and rendering as a new one, and ends up with the same structure. Because the converter reconstructs inputs rather than state, the pipeline's plan mode gives a direct check of its fidelity: a plan for a migrated environment that proposes replacing running resources points to an input the converter did not reconstruct faithfully.

F. Plan and Apply Pipeline

Generated profiles are kept in a version-controlled repository with one directory per environment and are executed by a Jenkins pipeline. A build targets one environment in one of two modes: plan mode runs Terraform's plan through Terragrunt and reports proposed changes without applying them [7], and apply mode applies them (R6). Before either step, the pipeline sets up credentials for the target cloud and for that cloud's Terraform state backend. Credentials are set

|ISSN: 2455-7773 | www.ijetemr.com| A Double-Blind Peer-Reviewed and Bimonthly Scholarly Journal|

|Volume 9, Issue 4, July-August 2024|

DOI:10.15662/ijetemr.2024.0904005

up per cloud environment rather than per provider, so a sovereign Azure cloud has its own. Each build runs in a workspace isolated by build number, so concurrent builds for different environments do not share working files. The pipeline also configures platform integrations that vary by cloud and environment type, such as a runtime security agent and connectivity to central log collection.

V. PRODUCTION DEPLOYMENT

A. Environment

The design runs on a multi-tenant enterprise SaaS platform whose isolated tenant environments are spread across AWS, Azure, Google Cloud, and a fourth, regional cloud provider. The module library covers compute, networking, security, identity, storage, and managed services on all four providers, and the profile repository holds the generated profiles for every environment.

B. Operation

A new environment starts with an operator running the subcommand for the target cloud. The invocation is validated, the environment block is reserved and split, and the profiles are rendered into the repository. A plan build then shows the resources to be created, and an apply build creates them. Migration follows the same path: the converter emits the invocation, and the remaining steps do not change.

C. Lessons Learned

Normalize where operators work. The pressure in a multi-cloud estate is to make module interfaces identical. The design spends that effort on a single entry point instead: operators gain one way of working, and the modules keep the native mechanisms that make each cloud worth using.

Address allocation is a coordination problem. Any allocation step that is not atomic can give the same range to two runs that proceed concurrently. Planning alone cannot exclude this; an atomic reservation in a shared ledger can.

Read the running system, not its old description. A legacy profile records what was requested when the environment was built, which need not match what runs today. Reading sizes, networks, and tags from the cloud APIs anchors each migrated environment to what is actually deployed.

Migration must preserve capacity and cost attribution. When two legacy roles collapsed into one node pool, the converter kept the larger instance type so that neither workload lost capacity, and cost tags were carried over so that cost reporting continued across the migration.

VI. EVALUATION METHODOLOGY

A. Data Sources and Baseline

The evaluation draws on the version-control history of the profile repository, generator output including validation errors, the reservation ledger, pipeline build records with plan output, and the provider APIs, which report the ranges and tags actually in use. The baseline is the provisioning practice that preceded the generator: Ansible-managed environments and manually prepared configurations.

B. Metrics

Table IV defines the metrics. Early rejection and plan failure rate test R3; address overlaps, ledger coverage, and reservation conflicts test R4; migration fidelity and tag continuity test R5. The hand-edit rate shows whether the generator remains the source of profiles, and lead time measures operator effort. Collecting these measurements over a sustained production period is ongoing work.

International Journal of Emerging Trends in Engineering and Management Research (IJETEMR)

|ISSN: 2455-7773 | www.ijetemr.com| A Double-Blind Peer-Reviewed and Bimonthly Scholarly Journal|

|Volume 9, Issue 4, July-August 2024|

DOI:10.15662/ijetemr.2024.0904005

TABLE IV. EVALUATION METRICS

Metric	Definition	Source
Early rejection	Share of input defects caught by schema validation rather than at plan or apply	Generator output; build records
Plan failure rate	Share of plan builds that fail, generated versus baseline profiles	Build records
Address overlaps	Pairs of environments with overlapping ranges	Provider APIs; ledger
Ledger coverage	Ranges in use that have no reservation in the ledger	Provider APIs; ledger
Reservation conflicts	Conditional-write failures resolved by retry	Ledger; generator output
Migration fidelity	Share of migrated environments whose first plan replaces no running resource	Plan output
Tag continuity	Share of migrated resources that keep their cost tags	Provider APIs
Hand-edit rate	Share of profile changes not produced by the generator	Version control
Provisioning lead time	Time from a validated request to an applied environment	Build records

VII. LIMITATIONS

Keeping modules provider-specific has a cost: adding a cloud requires a new subcommand, templates, and modules. Schema validation checks structure and allowed values but not provider-side conditions such as quotas or regional availability, which surface only at plan or apply. The ledger guarantees freedom from collisions only among ranges it records; address space used by environments that predate it, or by systems outside the platform, must be recorded for the guarantee to extend to them. The planner works on IPv4, and reclaiming the ranges of decommissioned environments is not addressed here. The converter reconstructs only the attributes it reads, so settings outside that set must be reviewed through the plan. The design has been used on one platform, and quantitative results are not yet reported.

VIII. CONCLUSION

Multi-cloud infrastructure does not have to choose between a common model that flattens every provider and per-cloud practices that diverge. Keeping the Terraform modules native and normalizing the path that produces their inputs gives operators one way of working while each cloud keeps its own mechanisms. The generator rejects malformed inputs early, reserves non-overlapping address space atomically, and accepts legacy environments through the same path as new ones. Future work includes reporting measurements with the methodology of Section VI, extending the planner to IPv6, and adding reclamation of released ranges to the ledger.

IX. ACKNOWLEDGMENT

The author designed the provider-specific module architecture, the profile generator, the address planner and reservation ledger, and the migration converter, and thanks the engineers who implemented, reviewed, and operated these

|ISSN: 2455-7773 | www.ijetemr.com| A Double-Blind Peer-Reviewed and Bimonthly Scholarly Journal|

|Volume 9, Issue 4, July-August 2024|

DOI:10.15662/ijetemr.2024.0904005

components. Portions of the text and Fig. 1 were prepared with the assistance of AI tools; the author reviewed and revised all content and takes responsibility for it.

REFERENCES

[1] K. Morris, *Infrastructure as Code: Dynamic Systems for the Cloud Age*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2020.
[2] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams, "A systematic mapping study of infrastructure as code research," *Inf. Softw. Technol.*, vol. 108, pp. 65–77, 2019.
[3] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba, "Adoption, support, and challenges of infrastructure-as-code: Insights from industry," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, 2019, pp. 580–589.
[4] HashiCorp, "Modules overview," Terraform documentation. [Online]. Available: https://developer.hashicorp.com/terraform/language/modules
[5] HashiCorp, "Use input variables to add module arguments," Terraform documentation. [Online]. Available: https://developer.hashicorp.com/terraform/language/values/variables
[6] Gruntwork, "Units," Terragrunt documentation. [Online]. Available: https://docs.terragrunt.com/features/units/
[7] HashiCorp, "terraform plan command reference," Terraform documentation. [Online]. Available: https://developer.hashicorp.com/terraform/cli/commands/plan
[8] HashiCorp, "State locking," Terraform documentation. [Online]. Available: https://developer.hashicorp.com/terraform/language/state/locking
[9] Pulumi, "Components," Pulumi documentation. [Online]. Available: https://www.pulumi.com/docs/iac/concepts/resources/components/
[10] Crossplane Authors, "Compositions," Crossplane documentation. [Online]. Available: https://docs.crossplane.io/latest/composition/compositions/
[11] A. Wright, H. Andrews, B. Hutton, and G. Dennis, "JSON Schema: A media type for describing JSON documents," IETF Internet-Draft draft-bhutton-json-schema-01, 2022. [Online]. Available: https://json-schema.org/draft/2020-12/json-schema-core
[12] Pallets Projects, "Template designer documentation," Jinja documentation. [Online]. Available: https://jinja.palletsprojects.com/en/stable/templates/
[13] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear, "Address allocation for private internets," IETF RFC 1918 (BCP 5), 1996.
[14] V. Fuller and T. Li, "Classless inter-domain routing (CIDR): The Internet address assignment and aggregation plan," IETF RFC 4632 (BCP 122), 2006.
[15] Amazon Web Services, "What is IPAM?," Amazon VPC IP Address Manager User Guide. [Online]. Available: https://docs.aws.amazon.com/vpc/latest/ipam/what-it-is-ipam.html
[16] Microsoft, "What is IP address management (IPAM) in Azure Virtual Network Manager?," Microsoft Learn. [Online]. Available: https://learn.microsoft.com/en-us/azure/virtual-network-manager/concept-ip-address-management
[17] HashiCorp, "Import resources overview," Terraform documentation. [Online]. Available: https://developer.hashicorp.com/terraform/language/import
[18] Google Cloud Platform, "Terraformer," GitHub repository. [Online]. Available: https://github.com/GoogleCloudPlatform/terraformer
[19] Amazon Web Services, "IAM roles for service accounts," Amazon EKS User Guide. [Online]. Available: https://docs.aws.amazon.com/eks/latest/userguide/iam-roles-for-service-accounts.html
[20] Microsoft, "Managed identities for Azure resources," Microsoft Learn. [Online]. Available: https://learn.microsoft.com/en-us/entra/identity/managed-identities-azure-resources/overview
[21] Microsoft, "What is Azure Firewall?," Microsoft Learn. [Online]. Available: https://learn.microsoft.com/en-us/azure/firewall/overview
[22] Google Cloud, "About Workload Identity Federation for GKE," Google Kubernetes Engine documentation. [Online]. Available: https://docs.cloud.google.com/kubernetes-engine/docs/concepts/workload-identity
[23] Google Cloud, "VPC firewall rules," Cloud Next Generation Firewall documentation. [Online]. Available: https://docs.cloud.google.com/firewall/docs/firewalls
[24] Amazon Web Services, "Working with items and attributes in DynamoDB," Amazon DynamoDB Developer Guide. [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/WorkingWithItems.html