Unearthing Semantic Checks for Cloud Infrastructure-as-Code Programs

Yiming Qiu Patrick Tser Jern Kon Ryan Beckett[†] Ang Chen

University of Michigan [†]*Microsoft*

Abstract

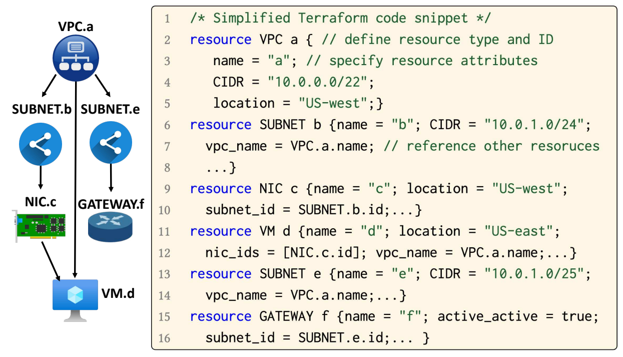
Cloud infrastructures are increasingly managed by Infrastructure-as-Code (IaC) frameworks (e.g., Terraform). IaC frameworks enable cloud users to configure their resources in a declarative manner, without having to directly work with low-level cloud API calls. However, with today's IaC tooling, IaC programs that pass the compilation phase may still incur errors at deployment time, resulting in significant disruption. We observe that this stems from a fundamental *semantic gap* between IaC-level programs and cloud-level requirements—even a syntactically-correct IaC program may violate cloud-level expectations. To bridge this gap, we develop Zodiac, a tool that can unearth IaC-level semantic checks on cloud-level requirements. It provides an automated pipeline to mine these checks from online IaC repositories and validate them using deployment-based testing. We have applied Zodiac to Terraform resources offered by Microsoft Azure—a leading IaC framework and a leading cloud vendor—where it found 500+ semantic checks where violation would produce deployment failures. With these checks, we have identified 200+ buggy Terraform projects and helped fix errors within official Azure provider usage examples.

CCS Concepts: **• Networks → Cloud computing**; **• Software and its engineering → Orchestration languages**; **Software reliability**.

Keywords: Infrastructure as code, cloud management, program analysis, configuration mining

1 Introduction

Many enterprises host their IT infrastructures in the cloud, but configuring and provisioning the underlying cloud resources remains a challenging task. Cloud datacenters are built by providers like Amazon but intended for third-party use (e.g., tenants like Electronic Arts). This user/owner split means that cloud tenants have limited visibility and control beyond the cloud-level APIs exposed by providers to tenants for resource management. Until recently, tenants relied on ad-hoc API scripts to manage resources (e.g., virtual machines, firewalls, gateways), which is a cumbersome process due to the myriad API calls and complex inter-call dependencies. Working directly with low-level APIs requires deep cloud expertise, and also creates burden for the tenant to track their cloud infrastructure state and updates. Hence, managing cloud resources with API-level scripting has been proven to be a daunting task.



```
/* Simplified Terraform code snippet */
resource VPC a { // define resource type and ID
   name = "a"; // specify resource attributes
   CIDR = "10.0.0.0/22";
   location = "US-west";}
resource SUBNET b {name = "b"; CIDR = "10.0.1.0/24";
  vpc_name = VPC.a.name; // reference other resoruces
  ...}
resource NIC c {name = "c"; location = "US-west";
  subnet_id = SUBNET.b.id;...}
resource VM d {name = "d"; location = "US-east";
  nic_ids = [NIC.c.id]; vpc_name = VPC.a.name;...}
resource SUBNET e {name = "e"; CIDR = "10.0.1.0/25";
  vpc_name = VPC.a.name;...}
resource GATEWAY f {name = "f"; active_active = true;
  subnet_id = SUBNET.e.id;... }
```

Figure 1. Example IaC cloud resource graph & code snippet.

Infrastructure-as-code (IaC) as a recent trend promises to simplify cloud management by shielding tenants from low-level APIs. Frameworks like Terraform [25], OpenTofu [20], and Pulumi [22] expose a higher-level interface that abstracts cloud resources as a set of configuration programs. As shown in Figure 1, tenants codify their desired infrastructure in a program: in this case, the Terraform snippet configures a VPC (virtual private cloud) with two subnets on Microsoft Azure—one for hosting a VM (virtual machine) and its NIC (network interface card), and a gateway. IaC frameworks then *automatically* construct the underlying cloud infrastructure by invoking cloud-level APIs based on the declared resources. This frees tenants from the tedium of working with API scripts, automating a large part of cloud management for ease of use. For this reason, IaC has seen significant uptake and gained wide popularity [2, 3, 20, 22, 25].

While IaC frameworks help *automate* provisioning tasks, mere automation does not beget *reliability*. In fact, IaC infrastructures are especially brittle, because IaC abstractions hide cloud-level complexities but fundamentally, do not remove

SOSP '24, November 4–6, 2024, Austin, TX, USA

ACM ISBN 979-8-4007-1251-7/24/11
https://doi.org/10.1145/3694715.3695974

them. Eventually, IaC frameworks execute provisioning tasks by handing them off to the cloud-level APIs, thereby subjecting them to the same range of potential errors as before. Worse, these errors are simply "out of sight" for the tenant at the development phase, only to manifest later as deployment-time issues. The Terraform snippet in Figure 1, for instance, compiles successfully but has two deployment errors. Microsoft Azure regulates that a) a VM and its NIC must not be located in different regions, and b) subnet CIDR ranges cannot overlap with each other. These issues go much deeper than the syntactic correctness of the IaC program.

We call this problem the **semantic gap**—even syntactically-correct IaC programs can induce unexpected cloud behavioral issues because, unbeknownst to the tenant, their IaC-level declarations conflict with cloud-level expectations; this is a fundamental problem caused by the tenant/provider split. The resulting deployment failures could leave the cloud infrastructure in an undesirable state, where the infrastructure provisioning process needs to be halted or rolled back for a fix. This is exacerbated by the fact that cloud provisioning is slow—even provisioning a single resource could take hours in the worst case [4, 5]; hence, fixing a buggy deployment is time-consuming as it may require destroying and/or recreating resources for the rollback. Rolling out a fix also requires deep expertise into cloud-level details, negating the benefits offered by declarative IaC platforms [27]. Furthermore, runtime deployment failures not only jeopardize the initial deployment of an IaC infrastructure, but also live updates to an existing deployment while it is serving user requests. Because of the potential damage of deployment errors, the wisdom of the DevOps engineering community is to perform reliability checks as early as possible, ideally at compilation phase, so that errors are detected and eliminated before any damage is done [24]. Because of all the above reasons, addressing this semantic gap is important to cloud reliability.

At first glance, the heart of the solution might seem clear—we just need to strengthen the IaC frameworks so that their compilation phase incorporates such **semantic checks** (e.g., on VM/NIC locations, and CIDR range requirements). While this approach is correct, the challenge lies in identifying the needed IaC checks in an automated manner. Cloud-level behaviors are opaque and poorly documented, so the required checks often need an intricate understanding of cloud operations; they could also evolve over time. Our investigations show that traces of such checks can be found in some of today's tools, but they are manually written by expert developers; this is a slow and tedious undertaking. Therefore, we wish to develop support for unearthing semantic IaC checks in an automated manner from public information.

Our roadmap is inspired by a line of work [60, 61, 70] that automatically discovers "invariants" for software configurations (e.g., MySQL) by mining open-source repositories to find configuration checks. Nevertheless, Zodiac faces a set of unique challenges in the IaC ecosystem. IaC programs are highly structural "configuration of configurations," which contain a myriad of interconnected resources each with distinct attributes. For instance, the inter-resource connectivity pattern between a gateway and its subnet and IP would influence the legality of their specific attribute values (e.g., their cloud regions and policies). Moreover, given the complex, blackbox nature of cloud backends, validating semantic checks also requires end-to-end deployment and observation. This stands in contrast to existing work that analyzes software code or online sources, or performs local testing to determine the validity of configuration checks [60]. Our tool, Zodiac, tackles these IaC-specific challenges in an automated pipeline for *mining* and *validating* semantic checks.

The **mining** phase of Zodiac captures the format of IaC semantic checks and automatically discovers check instances from online resources. We achieve this by designing a domain-specific *semantic knowledge base* and a check specification language. Jointly, they capture intra- and inter-resource constraints commonly seen in IaC programs, and give rise to a set of semantic check templates that can be used for mining. Zodiac generates hypothesized checks from these templates, uses online repositories for statistical filtering, and in certain cases, relies on large language models (LLMs) to fill in certain missing information. This produces a refined set of semantic checks, which we will proceed to validate in the cloud.

The **validation** phase of Zodiac reasons about the correctness of the checks by deploying IaC programs and observing their deployment outcome. Since semantic checks exhibit complex inter-resource correlations, Zodiac must minimize the interference across hypothesized checks when testing an IaC program. It relies on the semantic knowledge base and SMT solving to construct both positive and negative test cases for precise testing. A semantic check is validated if an IaC program that conforms to the check successfully deploys, and a similar program that violates the check fails to deploy. Zodiac then takes iterative passes over the hypothesized checks until each check is eventually validated or falsified.

We have applied Zodiac to a leading IaC framework, Terraform, for Microsoft Azure resources. It has extracted 500+ semantic checks for 52 types of popular cloud resources, many of which are not captured by state-of-the-art IaC tools. With these checks, Zodiac has found more than 200 buggy Terraform repositories and our results have been used for fixing four Microsoft Azure usage examples.

2 Motivation

In this section, we motivate our problem further both in its real-world relevance and the choice of our techniques.

2.1 Cloud Infrastructure-as-Code Programs

Cloud IaC frameworks are on the rise, with Terraform [25] leading the market. Terraform enables cloud users (e.g., the DevOps engineering teams in enterprise companies) to describe their desired infrastructure in a declarative language,

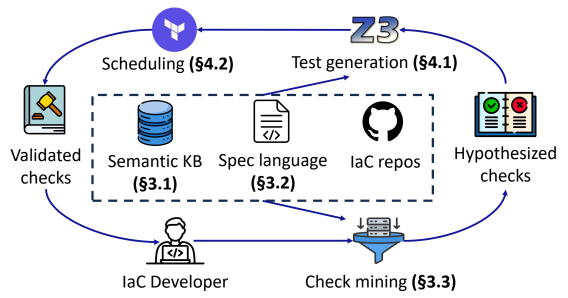


Figure 2. Overview of Zodiac pipeline.

akin to a configuration file for software entities (e.g., MySQL). Recall that Figure 1 shows a Terraform snippet and its compiled resource graph representation. The basic unit is a *resource block*, which defines a cloud infrastructure component (e.g., VM, NIC). Resource blocks are then composed to form a larger infrastructure encapsulating all needed cloud components. Their *dependencies* specify the relation across cloud components—e.g., a VM depends on a NIC in order to connect to the network. Thus, an IaC program not only requires that attributes of each resource are instantiated correctly, but also that dependencies across resources are correct. Further, attributes of a single resource also depend on inter-resource connectivity patterns—e.g., the maximum number of NICs that a VM could use depends on the specific VM sku [17].

2.2 Syntactic vs. Semantic Checks

IaC frameworks do apply standard compilation checks to reject erroneous IaC programs (e.g., programs that declare a non-existent cloud resource type due to incorrect naming). However, this process hides a vast range of complexities that nevertheless still surface at deployment time.

IaC frameworks. IaC frameworks like Terraform only check for syntactic correctness during compilation, but cloud-level correctness requires deeper semantic checks. Incorporating additional checks into the compilation phase requires intricate expertise with a particular cloud, and the checks often need to be identified manually. The engineering practices of IaC frameworks also do not make semantic checks easy to incorporate. The core compiler in Terraform is a shim layer that is independent of cloud providers. If a specific provider desires to supply additional checks, they must do so via IaC "plugins," which only allow for constrained check formats on individual attributes—e.g., the `VM.priority` field must be either `Regular` or `Spot`. These check formats cannot capture more sophisticated, inter-resource checks—e.g., if a `gateway`'s `active_active` attribute is set to `False`, then it can connect to at most 1 `IP` resource. If an IaC program triggers a cloud-level violation, the resulting infrastructure will be buggy and may require manual fixes and redeployment.

Ancillary IaC checkers. There exist various ancillary IaC tools [11, 28, 30, 35, 52] that aim to capture more sophisticated checks. Checkov [11] and TFSec [30], in particular, allow DevOps engineers to add customized checks into their tools. These tools apply additional checks to an IaC program after the IaC compilation phase. However, their checks are manually handcrafted by developers with deep cloud expertise, and they primarily target security/policy compliance rather than deployment failures. TFLint [29], another related tool, provides some automation for check generation through cloud API specification analysis. But its checks only support per-attribute validation (e.g., available VM skus) and cannot capture inter-resource semantics.

2.3 Inspiration: Configuration mining

We are inspired by a line of work on automated invariant discovery via *configuration mining*, which uses online configuration repositories to identify likely invariants, and derive configuration checks. Configuration mining treats the underlying software as either opaque, with details of internal behavior unavailable, or transparent.

Works that fall into the first class [70] use association rule mining algorithms to discover regulations for application configurations (e.g., MySQL). While this fits the cloud setting, whose internal behaviors are unknown to Zodiac, existing techniques target systems whose configurations have a "flat" structure, representing correlations between a set of attributes; in contrast, IaC programs are complex, hierarchical configurations of configurations. Moreover, existing work in this direction also does not consider automated validation. Validation is often manual (e.g. going through Stack Overflow posts or GitHub issues)—while this works at the scale needed for regular software (e.g., dozens of checks for a specific software entity like MySQL), manual validation does not scale to the number and diversity of cloud resources. Works that fall into the second category [69] jointly analyze software internal with their configurations, allowing them to capture more complex dependencies, and identifying misconfigurations that blackbox analyses cannot capture. For example, years of efforts on network control plane analysis have made it possible to take router configurations as input and simulate the expected routing behavior (e.g., the BGP protocol) [36]. Conversely, cloud resources are hard to model formally, and their implementation details are opaque to tenants, making whitebox analysis less applicable.

2.4 Zodiac: Zero-Touch Discovery of IaC Checks

We believe effective mining and validation of semantic checks for cloud IaC frameworks has the following requirements:

- *Automated:* Both mining and validation phases should be fully automated, without human intervention.
- *High coverage:* The mining phase should reason across resources and attributes to unearth a large set of checks.
- *High fidelity:* The validation phase should test each check via actual cloud deployment observations.

Figure 2 shows the overall workflow of Zodiac. It starts by ingesting IaC repositories crawled from online sources. Based on a curated set of check templates using a semantic knowledge base (KB) and a specification language, it generates a set of hypothesized checks. Next, it performs statistical filtering and interpolation to reduce false positives and fill in missing details with the help of LLMs. The hypothesized checks are then fed into the validation phase. For each such check, Zodiac identifies conforming instances that could be used as positive test cases, and it further mutates them to obtain corresponding negative test cases. A check is validated if the positive test case succeeds to deploy but its negative counterpart does not. To further resolve conflicts across different checks, Zodiac plans the order of negative test case generation and deployment via a validation scheduler. We further discuss these two components in §3 and §4.

3 Mining Cloud IaC Semantic Checks

We start by discussing how Zodiac mines semantic checks across cloud resources. Its inputs are open-source Terraform repositories on Github, and its outputs are a set of hypothesized semantic checks to be further validated.

3.1 Semantic knowledge base

To bootstrap this process, Zodiac first constructs a *semantic knowledge base* (KB) that contains "base facts" for building semantic checks. We draw inspiration from projects that construct *semantic type systems* [44] and define three classes of IaC type information. Each class of information is programmatically collected from different online sources, in an automated manner. The KB entries are themselves useful rules, many of which are not enforced by IaC frameworks. Table 1 shows some examples.

The first class of Zodiac semantic information are *IaC native constraints*. Zodiac extracts this from IaC provider schema files, which contain precise information about these properties. Consider the following examples: the IaC resource attribute `SUBNET.name` is usually a *required* field with a *string* type whose value is supplied by the developer. IaC attributes may also be *optional* (i.e., do not have to be specified); *nested* (i.e., list or dict blocks with nested sub-attributes); or *computed* (i.e., values only known after deployment).

The second class are *provider-specific constraints*, which are regulated by individual cloud providers. For instance, although `subnet.name` is a regular string from the perspective of IaC frameworks, they have special reserved values for each cloud provider. As another example, only a subnet named "FWSubnet" can be used to host firewalls; thus, our KB states that "FWSubnet" is a provider-specific `Enum` value instead of a generic string. Similarly, the KB would encode whether an attribute is an IP CIDR range or a port number, or whether it has a default value. Zodiac gathers this information from the crawled Terraform repositories, which contain common usage patterns for resource attributes.

The last but most interesting class of semantic information are *resource references* in IaC programs. Consider the following cases: `SUBNET.name = VPC.name`, and `SUBNET.vpc_name = VPC.name`. At first glance, they look very similar to each other—both are references to an attribute of another resource—but they have different semantics. The first case simply states that the subnet and the VPC have the same name, but the second case specifies a deployment order when constructing the infrastructure. Specifically, the subnet is attached to the VPC, so it must be deployed after the VPC. Zodiac constructs reference semantics from IaC provider registry examples.

Attribute	Class 1	Class 2	Class 3
SUBNET.name	required, string	[GWSubnet, ...]	[VPC.name]
SUBNET.CIDR	required, string	IPv4; IPv6	N/A
VM.priority	optional, string	[**Regular**, Spot]	N/A
VM.nic_ids	required, list	N/A	[**NIC.id**]
SG.rule[i]	optional, dict	N/A	N/A
SG.rule[i].dir	required, string	[In, Out]	N/A

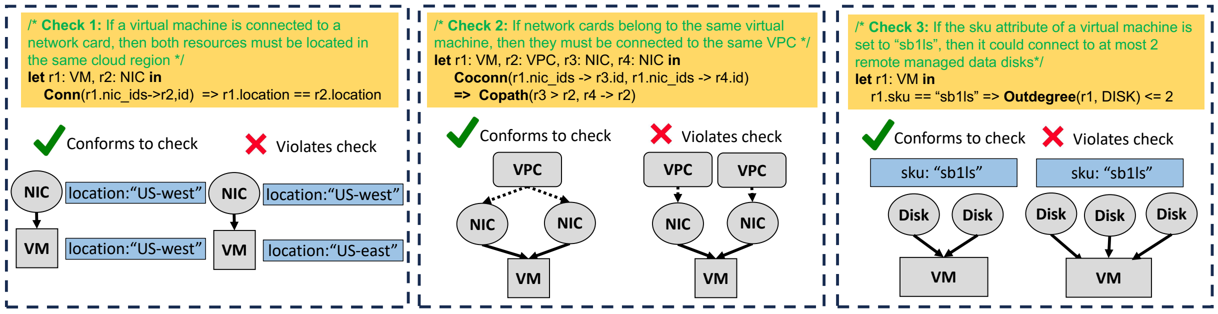
Table 1. Sampled semantic knowledge base entries. Default in class 2 and legality in class 3 are marked with bold fonts.

3.2 Semantic check specification

When checking software configurations [70], the templates are typically governing the relation across several attributes in the same configuration. For instance, EnCore [70] contains checks such as "uploaded file sizes for a PHP application should be smaller than `upload_max_filesize`," a relation between two size attributes. However, IaC programs are "configuration of configurations," because an overall cloud infrastructure contains myriad resources (e.g., VMs, NICs, gateways) in a hierarchical structure. Therefore, IaC semantic checks must incorporate this structure, not only capturing checks that govern an individual resource (e.g., a VM), but also how resources relate to each other (e.g. "if VM connects to NIC, then they must reside in the same region.")

We develop a domain-specific assertion language to address the topological nature of IaC programs and resource dependencies, beyond attribute-level checks developed in existing work. Our observation is IaC semantic checks are assertions over a graph, where nodes represent cloud resources and edges represent resource-level composition. The primitives of our language likewise articulate common graph patterns and graph-based assertions, categorized in two classes. We start with **topological** predicates:

- **conn**(r_1.in → r_2.out): Resource r_1 is connected to resource r_2 over a directed edge in the IaC resource graph, via two attributes r_1`.in` and r_2`.out`. Attributes where inter-resource connections are established are *inbound and outbound endpoints*. For instance, **conn**(`NIC.b.subnet_id` → `SUBNET.a.id`) means that an inbound endpoint of `NIC.b` is connected to an outbound endpoint of `SUBNET.a`. Each outbound endpoint (e.g., `SUBNET.a.id`) could connect to many inbound endpoints (e.g., 8 different `NIC.subnet_id`), while inbound endpoints usage is usually restricted (e.g., `NIC.subnet_id` must connect to a single `SUBNET.id`). Connections could also be built among IaC resources with the same type, e.g., **conn**(`DISK.b.source_id` → `DISK.a.id`).



```
/* Check 1: If a virtual machine is connected to a
network card, then both resources must be located in
the same cloud region */
let r1: VM, r2: NIC in
   Conn(r1.nic_ids->r2,id)  => r1.location == r2.location
```

```
/* Check 2: If network cards belong to the same virtual
machine, then they must be connected to the same VPC */
let r1: VM, r2: VPC, r3: NIC, r4: NIC in
   Coconn(r1.nic_ids -> r3.id, r1.nic_ids -> r4.id)
   =>  Copath(r3 > r2, r4 -> r2)
```

```
/* Check 3: If the sku attribute of a virtual machine is
set to "sb1ls", then it could connect to at most 2
remote managed data disks*/
let r1: VM in
   r1.sku == "sb1ls" => Outdegree(r1, DISK) <= 2
```

Figure 3. Three example semantic checks: on single resources, connectivity patterns, and aggregation properties, respectively. Paths and direct connections are represented by dashed and solid lines, respectively.

$r \in \text{Var}, t \in \text{ResourceType},\ v \in \text{Value},\ \alpha \in \text{Attribute}$

$$
\begin{array}{lcll}
\mathit{check} & ::= & \textbf{let}\ \mathit{bind}\ \textbf{in}\ \mathit{exp}_1 \Rightarrow \mathit{exp}_2 & \textit{semantic check} \\
\mathit{bind} & ::= & r_1 : t_1, \ldots, r_n : t_n & \textit{bindings} \\
p & ::= & r.\alpha & \textit{endpoint} \\
\tau & ::= & t \mid !t & \textit{type specifier} \\
\mathit{val} & ::= & v & \textit{base value} \\
 & \mid & p & \textit{endpoint value} \\
 & \mid & \textbf{indegree}(r, \tau) & \textit{indegree value} \\
 & \mid & \textbf{outdegree}(r, \tau) & \textit{outdegree value} \\
\mathit{exp} & ::= & \textbf{conn}(p_1 \longrightarrow p_2) & \textit{connected expr} \\
 & \mid & \textbf{path}(r_1 \longrightarrow r_2) & \textit{path expr} \\
 & \mid & \textbf{coconn}(p_1 \longrightarrow p_2, p_3 \longrightarrow p_4) & \textit{coexist expr} \\
 & \mid & \textbf{copath}(r_1 \longrightarrow r_2, r_3 \longrightarrow r_4) & \textit{coexist expr} \\
 & \mid & \mathit{op}(\mathit{val}_1, \mathit{val}_2) & \textit{conditional expr} \\
 & \mid & !\,\mathit{op}(\mathit{val}_1, \mathit{val}_2) & \textit{negated expr} \\
\mathit{op} & ::= & \texttt{==} \mid \texttt{!=} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{<} \mid \texttt{>} & \textit{comparison} \\
 & \mid & \textbf{overlap} \mid \textbf{contain} \mid \textbf{length} & \textit{function}
\end{array}
$$

Figure 4. The grammar for semantic checks.

- **path**($r_1 \longrightarrow r_2$): Resource r_1 is reachable to another resource r_2 over a path in the IaC resource graph. This is recursively defined over a set of **conn** relations with an unspecified path. For instance, if `VM.c` is connected to `NIC.b`, which in turn is connected to a subnet entity `SUBNET.a`, we derive **path**(`VM.c` $\longrightarrow$ `SUBNET.a`).
- **coconn**(r_1.in $\longrightarrow$ r_2.out, r_3.in $\longrightarrow$ r_4.out): A correlation predicate defined over two edges, where $r_1, \ldots, r_4$ are resource nodes. It evaluates to true when both edges (r_1.in $\longrightarrow$ r_2.out) and (r_3.in $\longrightarrow$ r_4.out) coexist in the IaC graph—e.g., a `VPC.a` has two subnets `SUBNET.b` and `SUBNET.c` can be expressed as **coconn**(`SUBNET.b.vpc_name` $\longrightarrow$ `VPC.a.name`, `SUBNET.c.vpc_name` $\longrightarrow$ `VPC.a.name`).
- **copath**($r_1 \longrightarrow r_2$, $r_3 \longrightarrow r_4$): A recursive predicate defined over two paths that evaluates to true when the two paths ($r_1 \longrightarrow r_2$) and ($r_3 \longrightarrow r_4$) co-exist in an IaC graph. For instance, **copath**(`NIC.b` $\longrightarrow$ `VPC.a`, `NIC.c` $\longrightarrow$ `VPC.a`) states that `VPC.a` has two NICs configured in the IaC graph.

The above predicates constrain topological patterns of the IaC resource graph. While we could design additional primitives to model a larger graph region, in practice we find that these primitives are enough to capture complex interactions among IaC resources. In addition to these topological primitives, we introduce two counting expressions that capture **aggregation** properties of the IaC graph.

- **indegree**(r, τ): The number of incoming edges, of a specific type τ, to resource r. For instance, a semantic check might state that any network card is only attached to one virtual machine, or **indegree**(`NIC.a`, `VM`) == 1.
- **outdegree**(r, τ): Analogous to the above aggregation property, only with a different edge direction.

Semantic checks by examples. Our domain-specific assertion language can describe a wide range of semantic checks over an IaC graph, and Figure 3 shows three concrete examples. *(i) left:* This check states that if a VM is connected to a NIC, then they must be instantiated in the same cloud region. As shown in the figure, if the NIC is located in "US-East" instead of "US-West," this would lead to a deployment error. *(ii) middle:* It states that if two NICs are connected to the same VM, then they must be contained in the same VPC resource. *(iii) right:* Any virtual machine whose `sku` attribute is set to `sb1ls` must have $\leq$ 2 data disks attached to it—an aggregation property. The figure also shows example IaC topologies that conform to or violate each check.

3.3 Semantic check mining

The specification language and semantic KB enable us to narrow down the search space of IaC semantic checks. We curated 84 templates based on our grammar via a mixture of manual effort and automated generation. We first manually added semantic constraints to each expression (e.g., if the right side of "==" is an attribute value, then it must be an Enum type defined in the KB rather than a string) to restrict the template search space, and automatically combine constrained expressions as template conditions and statements (e.g., combine conn with "==" to generate template for check 1 in Figure 3). We then manually pruned some trivial templates that add little information. The resulting templates are reusable across IaC providers and user repositories and this curation only needs to be done once for each cloud provider.

Association rule mining. The mining algorithm then examines crawled IaC repositories under the lens of templates, and instantiate all witnessed checks. Consider a simple check template that specifies intra-resource attribute relations:

```
for resource r in C:
  find(r.attr1 == Enum => r.attr2 != null):
```

For this template, the mining engine iterates through all IaC programs, identifying the contained resource types where the Enum value of an attribute is positively associated with the existence of another. As an example, this template might find the following check, stating that a spot VM must be configured with an eviction policy:

```
let r:VM in
r.priority == 'Spot' => r.evict_policy != null
```

For inter-resource templates, the mining engine iterates through groups of resources to instantiate specific instances:

```
for resource r1, r2 in C:
  find(Conn(r1.in, r2.out) => r1.attr1 == r2.attr2)
```

Our running example that constrains `VM` and `NIC` locations would be such an instance. Since these identified checks may be incorrect or incomplete, next we further refine them.

Statistical filtering. For each identified check, Zodiac computes *confidence* and *lift* values for statistical filtering. Confidence is the conditional probability that a given check is satisfied when it occurs in the dataset, or Confidence(X⇒Y) = P(Y|X); this metric prefers checks with fewer counterexamples. Lift indicates whether the predicate and assertion of a check are independent or correlated: Lift(X ⇒ Y) = $\frac{\mathrm{P(Y|X)}}{\mathrm{P(Y)}}$, where a value of one indicates that X and Y are independent. a value higher than one is stronger evidence that the condition and statement are positively correlated. Zodiac filters out semantic checks with low confidence or lift.

Large language model reasoning. Next, we handle a common phenomenon that quantitative properties, ranges, and Enum types could lead to incomplete or inaccurate checks. For example, "VM with sf2 sku can be attached to 2 NICs" might be witnessed in some repositories, but the actual cloud requirements could exist in a more general form—e.g., "for a given VM type, the maximum number of NICs is t." Since there are 100+ VM types and they could have 1-64 attached NICs, such specific information may not be directly observed in online repositories. We thus leverage LLMs for interpolation to retrieve specific information about potentially ambiguous check instances. An "interpolation query" might state, for instance, "for a sf2 sku VM, what is the maximum number of NICs allowed?" and the model may answer "4"; Zodiac then includes this refined check.

Zodiac performs a series of prompt engineering steps, which perform fact-checking on candidate checks, since LLMs excel at such tasks [37, 49, 50]. Specifically, Zodiac translates interpolation checks into natural language description, and generates an LLM prompt for few-shot learning—i.e., providing several pairs of input-output examples, where an input is an interpolation query, and an output is a concrete answer (e.g., NIC count). By doing so, Zodiac relies on LLMs to mitigate data scarcity issues that are an important limitation of configuration mining work. Zodiac requires that the LLM refer to reliable online sources (e.g. cloud provider documents) to obtain up-to-date information outside of the mining dataset. The intuition is that these documents typically contain detailed usage descriptions (e.g. sku tables [17]); therefore, LLMs can effectively interpret this context and produce reliable answers.

4 Validating Cloud IaC Semantic Checks

Next, Zodiac constructs test cases for each hypothesized check for deployment-based validation. We write the set of validated checks as R_v, which initially is empty, and the set of candidate checks as R_c, which is yet to be validated. In each iteration, Zodiac picks a candidate check c from R_c, and attempts to validate or falsify it. c is then removed from R_c and if it passes validation, will be added to R_v. This repeats until R_c becomes empty. To validate a check c, Zodiac finds an IaC program t_p that conforms to c and validates that t_p can be deployed successfully—this is called the *positive* test case for c. It also obtains a *negative* test case t_n by mutating t_p to violate the check c, and further confirms that t_n produces a deployment error (e.g., IaC plugin errors, cloud service errors, or inconsistent IaC states). Satisfying both conditions will validate the check; otherwise, the check is falsified.

4.1 Encoding the mutation search space

For a candidate check c, there must exist some program P that satisfies this check. Assume for now that we will directly use P as the positive test case t_p, and our goal is to find a negative test case t_n by mutating t_p.

An example: Consider the following check c, which states that a VPC cannot host more than one gateway:

```
let r1:GW, r2:VPC in
path(r1 → r2) => outdegree(r2, GW) == 1
```

Zodiac iterates through its corpus to identify a program that declares a VPC with exactly one gateway; this is t_p. To construct t_n, Zodiac mutates this program so that it contains a VPC with multiple gateways. Note that this may not be simply adding gateways to this existing VPC, because we need to ensure that the mutated program does not violate other checks. Otherwise, t_n would have violated multiple checks, and even if it fails to deploy, Zodiac cannot conclude that c is the root cause for the failure. For instance, Zodiac may realize that there is another check c' stating that a subnet cannot have more than one gateway:

```
let r1:GW, r2:SUBNET in
conn(r1.subnet_id, r2.id) => outdegree(r2, GW) == 1
```

In this case, when mutating the program, Zodiac will add a new gateway as well as a new subnet, so that c' will not be violated. Recursively, this requires Zodiac to examine additional checks when adding the subnet—e.g., a third check

c'' stating that subnets under the same VPC cannot have overlapping CIDR ranges:

```
let r1:SUBNET, r2:SUBNET, r3:VPC in
coconn(r1.subnet_id → r3.id, r2.subnet_id → r3.id)
      => !overlap(r1.CIDR, r2.CIDR)
```

Our goal is to ensure that only c is violated for precise testing.

Solver-aided mutation. We encode these constraints as SMT formulas so that our t_n violates a check c, but conforms to all other checks in R_v and R_c as well as base facts in our semantic KB. Given a positive test case t_p, Zodiac first encodes potential mutation strategies by instrumenting its attributes and connections with symbolic values, and asks an SMT solver to identify concrete assignments.

Encoding mutations. Consider the following check, which states that a sf2 sku VM cannot host more than 2 NICs:

```
let r:VM in
r.sku == 'sf2' => indegree(r, NIC) <= 2
```

Suppose t_p contains a `sf2` `VM.a` with two network interfaces `NIC.d` and `NIC.e`, then Zodiac would create t_n by adding a third NIC and attaching it to `VM.a.nic_ids`. It converts part of t_p into symbolic variables denoted by ?? as shown below:

```
/* instrumented existing resources */
resource VPC b, SUBNET c, NIC d, e ...
resource VM a {
  sku = "sf2"; location = "eastus";
  nic_ids = [NIC.d.id, NIC.e.id, ??]}
/* instrumented virtual resources */
resource NIC v0 {
    location = ??; subnet_id = ??}
resource VPC v1, SUBNET v2
```

A symbolic endpoint is put into `VM.a.nic_ids` as a third NIC that would violate the candidate check. The actual NIC is `NIC.v0`, also with symbolic values. Zodiac also adds `VPC.v1` and `SUBNET.v2`, as the new NIC might require its own VPC or subnet; these additional resources may be instantiated or skipped depending on the SMT solving results.

Encoding requirements. Next, Zodiac encodes the semantic KB. For example, it may assert that all newly added connections must be legal (e.g. `NIC.v0.subnet_id` cannot directly connect to `VPC.a.id`)—e.g., based on Class 3 semantic KB entries. Zodiac also encodes that if a new resource is added, all its required endpoints must be correctly connected to existing resources, based on Class 1 semantic KB entries. As an example, if a new `NIC.v0` is connected to a `VM.a` at its outbound endpoint, then it must be connected to some subnet at its inbound endpoint. Likewise, Zodiac asserts that `NIC.v0.location` must be a valid cloud region name based on the semantic KB. Zodiac then encodes checks in R_c and R_v as SMT constraints. For instance, to assert a check "VM and its NIC must be in the same location", the solver will make sure that the assigned `NIC.v0.location` is equal to `VM.a.location`, which in this case is `eastus`. Solving all constraints above thus results in the following t_n:

```
/* SMT generated negative test case */
resource VPC b, SUBNET c, NIC d, e ...
resource VM a {
  sku = "sf2"; location = "eastus";
  nic_ids = [NIC.d.id, NIC.e.id, NIC.v0.id]}
resource NIC v0 {
    location = "eastus"; subnet_id = SUBNET.c.id}
```

Immutable resources. IaC frameworks comprise an evolving ecosystem of service providers, so Zodiac may find resources that it does not yet support (e.g., the KB does not contain relevant information). Zodiac leaves such "unattended" resources unchanged when performing mutations since it does not know what valid mutations look like.

Minimizing changes. Finally, Zodiac minimizes the difference between t_n and t_p by adding SMT optimization objectives. We require that attribute values and topology connections should remain unchanged to the extent possible, preferring original values and connections used in t_p. We also require that value mutations should minimize the distance from the original values—e.g., mutating a CIDR value to its adjacent range with the same prefix length). The final t_n is expected to compile successfully in the IaC framework but produce a deployment failure.

Pruning IaC programs. Thus far, we have assumed that t_p is some existing IaC program P in our corpus. In practice, Zodiac simplifies P by removing two types of resources before generating t_p and t_n. First, Zodiac prunes away resources that are not reachable from resources that trigger the candidate check c, so that the resulting programs have fewer resources. This not only leads to smaller SMT encodings but also lower cloud deployment cost. For example, when validating a check that only regulates VM attributes, Zodiac will remove gateways and peerings from P and only preserve the minimal set of resources (e.g., a single VM). In addition to unreachable resources, Zodiac also removes child resources that are only deployed after the candidate check takes effect—e.g., the disk association of a VM. This pruning is achieved by identifying a single instance that conforms to c in the original program—i.e., a set of resources that witness a check—and keeping only this instance and its ancestor resources that are required for deploying this instance (e.g., a VM alongside its NIC, subnet and VPC). We call this a *minimal deployable configuration* (MDC) which is our t_p.

4.2 Scheduling check validation

Next, we discuss how to schedule test cases and deploy them into the cloud to validate the corresponding semantic checks. The most naïve solution would be to test a randomly chosen candidate check against a randomly chosen program, then repeat this process until all checks are validated or falsified. However, in reality, it is not always possible to generate negative test cases for each candidate check. A check may be "untestable" because any attempts to mutate its attributes or topology to violate a candidate check always has the side effect of violating other checks in R_c and R_v. As an example, consider three candidate checks below, assuming they are the only checks in R_c—that is, R_v is empty:

```
/* Candidate check (1) */
let r1:NIC, r2:VPC in
path(r1 → r2) => r1.location == r2.location
/* Candidate check (2) */
let r1:VM, r2:NIC in
path(r1 → r2) => r1.location == r2.location
/* Candidate check (3) */
let r1:VM, r2:VPC in
path(r1 → r2) => r1.location == r2.location
```

Consider the case where we start by validating check (2), which asserts that the VM and NIC locations must be the same. If we mutate the location of NIC, then (1) and (2) will be violated simultaneously, and if we mutate the location of VM, then (2) and (3) will be violated at the same time. These check conflicts could lead to a stalemate.

Validation scheduling algorithm. Hence, the ordering of testing is important, and Figure 5 shows our scheduling algorithm to resolve check conflicts that may occur. At a high level, the algorithm *iterates* (◁O1) over R_c until it becomes empty (line 5). Each iteration is further composed of a *false positive removal* pass and a *true positive validation* pass. The intuition is that removing false positives could make it easier to validate true positives (because fewer candidate checks will be involved), and the same applies in reverse—validating true positives make it easier to remove more false positives, as validated checks become part of the ground truth.

In the false positive removal pass (lines 6-14), a candidate check is classified as false positive if one of the following cases occur: 1) no negative test case can be generated because the solver always returns UNSAT due to conflicts with checks in R_v (line 11); 2) a negative test case exists but does not result in deployment failures (line 13). When generating negative test cases for false positive removal passes, the algorithm ensures that the test cases conform with all the checks in R_v, but allows other checks in R_c to be violated. This is because R_c has not yet been validated and its violation should not stop Zodiac from making progress. In fact, if a deployment succeeds for a test case containing multiple R_c violations, then it means all violated checks are false positives. As shown on line 10, checks in R_v are encoded as hard constraints in the SMT solver, while checks in R_c are encoded as soft constraints through SMT minimization primitives (◁O2).

In the true positive validation pass (lines 17-24), if a negative test case fails to deploy, we further examine whether 1) it only has one check violation (line 21), or 2) it violates multiple checks simultaneously (i.e., $size(R_n) > 1$), but these checks are *indistinguishable* from each other across all their test cases (line 23). If either is true, Zodiac places such target check(s) into R_v and marks them as validated. We calculate indistinguishable check groups G_i (◁O3) before the beginning of each true positive validation pass (line 16). The algorithm

```
1:  function ValidationScheduling($R_c$, $R_v$, $P$)
2:      // Calculate evaluation partial order among checks
3:      EvalPartialOrder($R_c$)                                  ◁ O4
4:      // Iterate until candidate check set becomes empty
5:      while $R_c$ != $\emptyset$ do                                 ◁ O1
6:          for $c$ in $R_c$ do // False positive removal pass
7:              // Find positive test case in user repos
8:              $t_p \leftarrow$ FindCheckInstance($c$, $P$)
9:              // Calculate negative test case and its violations
10:             $t_n, R_n \leftarrow$ SMTMinimize($t_p$)                   ◁ O2
11:             if $t_n$ == $\emptyset$ then
12:                 $R_c \leftarrow R_c - \{c\}$
13:             else if Deployment($t_n$) == Success then
14:                 $R_c \leftarrow R_c - \{c\}$
15:         // Find groups of indistinguishable checks
16:         $G_i \leftarrow$ GroupIndistinct($R_c$)                      ◁ O3
17:         for $c$ in $R_c$ do // True positive validation pass
18:             $t_p \leftarrow$ FindCheckInstance($c$, $P$)
19:             $t_n, R_n \leftarrow$ SMTMinimize($t_p$)
20:             if Deployment($t_n$) == Failure then
21:                 if $size(R_n)$ == 1 then
22:                     $R_c \leftarrow R_c - \{c\}$; $R_v \leftarrow R_v \cup \{c\}$
23:                 else if $R_n$ in $G_i$ then
24:                     $R_c \leftarrow R_c - \{c\}$; $R_v \leftarrow R_v \cup \{c\}$
```

Figure 5. End-to-end validation scheduling algorithm.

comprises two steps. First, it generates a negative test case t_n for each check c in R_c. If the test case for c also violates another check c' and vice versa, c and c' are put into a candidate group. Second, for each candidate group, the algorithm searches through each available positive test case t_p to generate a t_n that violates one of those checks yet conforms with all other checks in the same group. If this process fails across all t_p with the solver reporting UNSAT, then they are indeed indistinguishable from each other. For instance, if Zodiac cannot find a t_n that only violates one of the checks in (2) and (3), then they are marked as indistinguishable.

In theory, the scheduler could run into a "reasoning loop" where all negative test cases violate multiple semantic checks and fail during deployment. This situation could again lead to a stalemate. In practice, the hierarchical structure of IaC programs helps alleviate this problem. Consider the three candidate checks earlier in this subsection, and assume they are all true positives. At first glance, it seems they all have check conflicts with each other. However, since VM only gets deployed after its NIC, we may construct a test case that contains a NIC but does not have a VM, so the solver can ignore check (2) and (3) when evaluating (1). This ordering between semantic checks forms an *evaluation partial order*, which naturally resolves reasoning loops among inter-resource semantic checks. This ordering also reduces the amount of iterations, as interference among checks with different partial orders are minimized. As shown in line 3, the scheduler reorders all hypothesized checks so that those with higher partial order will always be evaluated first (◁O4). Although the above approach does not resolve intra-resource

Semantic check templates	Example mined by Zodiac	Category
A.attr$_1$ == Enum ⇒ A.attr$_2$ == Enum	"If GW.sku is Basic, GW.active_active is False"	intra-resource
A.attr$_1$!= A.attr$_2$ ⇒ A.attr$_3$!= A.attr$_4$	"Different direction SG rules have diff. priority"	intra-resource
copath(A → B, A → C) ⇒ !**overlap**(B.attr$_1$, C.attr$_2$)	"Two tunneled VPCs have exclusive IP CIDR"	inter w/o agg
conn(A.in$_1$ → B.out$_1$) ⇒ B.attr1 == Enum	"IP associated with NAT must use standard sku"	inter w/o agg
conn(A.in$_1$ → B.out$_1$) ⇒ **coconn**(A.in$_2$ → C.out$_2$, B.in$_3$ → C.out$_3$)	"Route table and its routes must in same VPC"	inter w/o agg
coconn(A.in$_1$ → B.out$_1$, A.in$_2$ → C.out$_2$) ⇒ B.attr$_1$!= C.attr$_2$	"VM os_disk and data disk have different name"	inter w/o agg
conn(A.in$_1$ → B.out$_1$) ⇒ **outdegree**(B, τ) == 1	"A NIC could only be attached to one VM"	inter w/ agg
A.attr$_1$ == Enum ⇒ **indegree**(A, τ) == 0	"VPC2VPC type tunnels can't use HA GW "	inter w/ agg
conn(A.in$_1$ → B.out$_1$) ⇒ **outdegree**(B, τ) == 0	"No other resource can share subnet with GW"	inter w/ agg
A.attr1 == Enum ⇒ **outdegree**(A, τ) == int	"Basic sku GW can have at most 10 tunnels"	interpolation
A.attr1 == Enum ⇒ **indegree**(A, τ) == int	"sf4 sku VM can be attached to at most 4 NICs"	interpolation
A.attr1 == Enum ⇒ A.attr2 != Enum	"Premium sku SA prohibits GZRS redundancy"	interpolation

Table 2. Some representative check formats Zodiac has currently validated. From top to bottom, they are intra-resource checks, inter-resource checks without and with aggregation, and checks enhanced by LLM interpolation.

reasoning loops, we note that the latter does not appear in our semantic check templates.

Validation examples. Next, we discuss how the scheduling algorithm works in action, by observing how it handles different scenarios around the motivating candidate checks.

I. All checks are true positives. The validation scheduling algorithm orders checks by their partial order, so check (1) will be validated first without any conflicts. Check (2) and (3) are then deemed as a group of indistinguishable checks because they are always conformed or violated at the same time. The true positive validation pass should be able to resolve this case, and put both checks into R_v.

II. Some checks are false positives. Suppose check (2) is the only true positive. In this case, check (1) will be evaluated first according to partial order, and get removed as a false positive. Check (2) and (3) in this case do not have conflicts any more, e.g., check (2) could mutate NIC and check (3) could mutate VPC, without triggering other violations. They can thus be easily resolved during upcoming passes.

5 Evaluation

Prototype. We have implemented Zodiac [31] in ~11,000 lines of code in Python: ~3,100 for data processing and semantic KB construction, ~4,100 for the mining engine, and ~3,800 for the validation engine. The KB construction and mining steps are implemented using Rego [18] queries, and the test case generation uses the Z3 SMT solver. Interpolation queries are performed using GPT-4.

Corpus and pipeline. We applied Zodiac to 52 popular resource types in Microsoft Azure, crawling 26,000 IaC repositories from GitHub. This yielded ~6,000 projects with ~3.8 million lines of Terraform code after preprocessing. We then compiled these IaC programs into deployment plans, which serves as the basis for both Zodiac mining and validation steps. The mining phase completed in under 2 hours. We further filtered out projects incompatible with our SMT solver implementation and fed the remaining ~4200 projects to the validation phase, which finished within 3 days. Our evaluation focuses on several key research questions:

- How effective is Zodiac at discovering semantic checks, and what are the implications of these checks?
- How effective is Zodiac compared to baseline systems?
- How effective are the design techniques in the mining and validation phases in discovering semantic checks?
- How effective are Zodiac checks at finding real bugs?

5.1 Discovered semantic checks

The most important metric is the quantity and quality of the semantic checks that Zodiac is able to discover. In the mining phase, Zodiac discovered ~9,800 hypothesized checks, and filtered out ~5,600 based on confidence and lift. The validation phase produced 510 validated checks (indistinguishable checks are counted as one). We present several examples below, and Table 2 summarizes the key templates.

(1) Premium storage account (SA) users might expect advanced replication support [6]—for instance, geo-zone redundancy (`GZRS`) which provides both datacenter (zone) and secondary region (geo) failovers. However, surprisingly, Zodiac finds that `GZRS` is not available to `Premium` but only `Standard` SAs. This is because `Premium` is in fact optimized for latency requirements instead of failover.

```
let r:SA in
r.sku == 'Premium' => r.replica != 'GZRS'
```

(2) VMs have an `os_disk` attribute, which is the storage for the OS image, and it also has "data disk" resource type for the main storage. At first glance, they do not appear correlated, but Zodiac finds an inter-resource check stating that their names must be different. Our study reveals that although the IaC program uses different names for these disks, at the Azure level both are instantiated in the same way and cannot have naming conflicts.

```
let r1:VM, r2:DISK, r3:ATTACH in
coconn(r3.vm_id → r1.id, r3.disk_id → r2.id) =>
 r1.os_disk != r2.name
```

(3) We discussed earlier that Azure has reserved subnets—e.g., only "GWSubnet" can host a gateway GW. Zodiac is able to further find that such subnets are also quite exclusive.

Error Phase	Consequence	Example mined by Zodiac	Share
Plugin checks	Target resource fails before requests are sent to providers.	"Standard IP use static allocation"	9.00%
Pre-deploy sync	Target resource fails as provider claims "already exists".	"Disks have different names"	5.84%
Sending request	Target resource fails during initial creation attempts.	"Peering VPC CIDR can't overlap"	74.94 %
Polling request	Target resource fails during async. polling attempts.	"FW subnet can't use delegation"	7.79%
Post-deploy sync	Target resource completes but IaC/cloud states are inconsistent.	"subnet only attach to 1 route table"	2.43%

Table 3. IaC programs that violate Zodiac semantic checks could produce several classes of deployment errors.

If a subnet is hosting a GW, then it cannot host other types of resources (e.g. a NIC):

```
let r1:GW, r2:SUBNET, in
conn(r1.subnet_id → r2.id) => outdegree(r2, !GW) == 0
```

Deployment failure scenarios. Table 3 further shows the failure scenarios due to semantic check violations. After compilation, the IaC program will go through several steps. First, IaC frameworks will perform a set of plugin checks outside the core compiler—using checks from individual plugin providers. Violations against these checks account for 9.00% of deployment failures within Zodiac test cases. This result indicates that some Zodiac checks are already considered by today's IaC plugin developers. However, these plugin checks are not static analyses; they are performed as resources are being deployed—recall that the core IaC compiler does not expose proper interfaces for implementing such semantic checks. As a result, deployment will proceed normally at first until a violation halts or disrupts the infrastructure.

After plugin checks, the IaC framework queries the current cloud state and synchronizes that state with the IaC program to be deployed. 5.84% of the failures occur at this step, typically due to resources with conflicting identifiers. If this passes, the IaC framework sends creation requests to cloud providers, which initiates the actual deployment phase. Most test cases (74.94%) fail here for myriad reasons, such as invalid attributes, conflicting CIDR ranges, "resources not found" errors, invalid connections, "features not supported for sku." If no errors occur here, then the IaC framework initiates a polling phase to retrieve the eventual cloud state asynchrously, on resources that are slow to create (e.g. FW); 7.79% failures happen at this step. Finally, the IaC framework performs another round of synchronization on the successfully-deployed resources to determine whether the deployed states are as expected, and this captures the remaining 2.43% errors that are silent across the deployment attempt, typically because some created resources are overridden by subsequent ones.

Impact of failures. We now discuss the impact of deployment failures when an IaC program violates Zodiac semantic checks, denoting the "blast radius" of a check as the number of resource types that could be affected if that check is violated. Consider an example check "two tunnelled VPCs cannot have overlapping CIDR ranges." In this case, before the tunnel is created, the two VPCs could be deployed successfully, along with their child resources. Later on when the tunnel fails to deploy, users need to change VPC CIDR ranges for a fix. Since most direct changes to VPC-level CIDR ranges are not allowed by Azure, we would need to recreate the VPCs as well as all their child resources from scratch. In other words, these resources (36 types among 52 that Zodiac covers) are impacted by the *rollback radius* due to deployment failure.

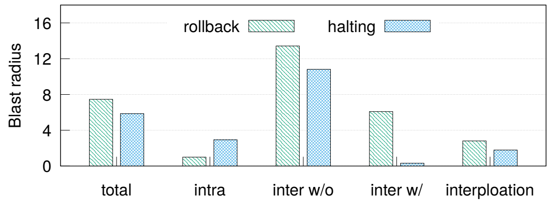


Figure 6. Violations against Zodiac semantic checks could lead to halted deployments or state rollback. Blast radius refers to the upper bound of impacted resource types.

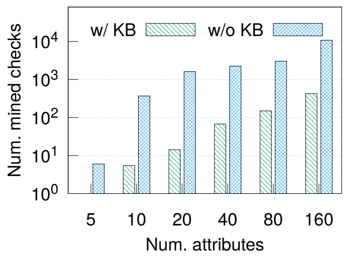
As Figure 6 shows, each semantic check violation would impact ~7 resource types that requires rollback actions in the worst case, which is the upper bound of resource types that must be recreated to fix a deployment. Another ~6 are within what we call the *halting radius*, because they could not be deployed at all before the violation is resolved. The blast radius changes across semantic check categories—e.g., intra-resource checks have a smaller rollback blast radius because only the failed resource itself needs to be changed; inter-resource checks (without aggregation) have the largest blast radius (both halting and rollback), as the graph patterns are more complex and have more resource types.

5.2 Zodiac vs. existing tools

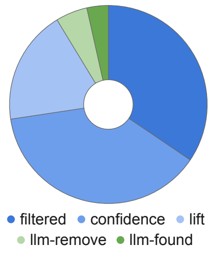
We compare Zodiac against several IaC checkers. First, the Terraform native `validate` command matches user IaC programs against provider schema JSON files, which contain basic syntax inspections and simple semantic checks (e.g. conflicting attributes). Furthermore, there are also several security checkers (TFSec, Checkov, TFComp, Regula) that are developed outside Terraform as ancillary tools. They could capture security incidents such as "Password authentication is insecure thus should not be used in VM," or "Public internet access to SSH ports is insecure thus should be disabled in SG." These tools typically operate on compiled IaC deployment files. TFLint is another popular static checker that captures invalid Enum values in resource attributes, and raises warnings when IaC programs deviate from best practices (e.g. whether single-line comments are used). TFLint does not reason across different attributes or resources, and is thus incapable of handling any checks mined by Zodiac.

Tool	Spec	Phase	Prevalence	Precision
Native	JSON	Config	11.74%	36.67%
TFSec	JSON	Plan	11.54%	- - -
Checkov	YAML	Plan	66.34%	- - -
TFComp	BDD	Plan	3.91%	- - -
Regula	OPA	Plan	13.31%	- - -
TFLint*	HCL	Config	- - -	- - -

Table 4. Semantic checks found by Zodiac are not present in other IaC static analysis tools. Prevalence denotes the percentage of Zodiac test cases marked as invalid/insecure, precision evaluates their overlapping with Zodiac checks.



(a) Zodiac knowledge base



(b) Zodiac filtering

Figure 7. Zodiac global knowledge base and filtering methods help with constraining the amount of candidate checks.

Table 4 shows the comparison. We randomly generated ~500 negative test cases for validated Zodiac checks as inputs to all checkers, and report their prevalence (percentage of inputs with reported issues) and precision (percentage of actual deployment problems among reported issues). Our first observation is that most sampled test cases could pass IaC native validation without triggering any errors: only 11.74% of them encountered compilation failures. It is worth noting that most of these failures are not violations against semantic checks, but rather generic syntax problems in sampled test cases. Only 36.66% among them (4.00% of total test cases) point to actual semantic violations, typically due to missing attributes in a resource block (e.g. neither password nor ssh_key is declared in a VM).

While ancillary security checkers (e.g. Checkov) do not aim at capturing deployment failures (shown as '- - -'), they actually reported some problems in our test cases for two reasons. First, the original IaC programs in our corpus could be insecure in the first place, triggering checker reports. Second, the pruning optimization of Zodiac removes resource that are not directly related to deployment success. For instance, security checkers might suggest a subnet should have a SG attached, but Zodiac will often remove SG during testing. We were unable to compare against TFLint directly because it only works against the HCL format, while Zodiac test cases support configurations and planning files in JSON.

5.3 Effectiveness of the mining phase

Zodiac applied a set of domain-specific techniques for semantic check mining and filtering, to capture more high quality rules while discarding obvious false positives early on. In Figure 7(a), we demonstrate how Zodiac's knowledge base helps reduce the total number of candidate checks. The x-axis shows resource types with varying number of attributes. Simpler resource types (e.g., peering) may have fewer than 10 attributes, whereas complex ones (e.g., VM) have more than 80. The y-axis denotes number of mined intra-resource checks per resource type, while each group of columns shows the result w/ and w/o KB involvement respectively. For both cases, the number of mined checks grows as number of attributes increases, but w/ KB reduces the number of mined checks by several orders of magnitudes. For instance, intra-resource check mining w/o KB generated more than 70,000 mined checks, which is almost 35 times higher than those generated by Zodiac.

Figure 7(b) shows the number of checks removed by statistical filtering (i.e., confidence and lift), as well as the number of checks completed by LLM-based interpolation. The confidence filter removed 38.3% of mined checks, which means that those checks are not always respected in existing user repositories. The lift filter further removed an additional 16.2% of mined checks, indicating that their conditions and statements did not have a strong correlation.

We further leveraged LLMs to test the filtering effectiveness using a randomly sampled set of mined checks. Out of these 400 checks, ~34% pass the statistical filtering and the rest were discarded by confidence and lift. We asked the LLM to assess whether these checks are true positives. For the checks that have passed statistical filtering, the LLM reports 18.80% as true positives; for checks that are filtered out statistically, this drops to 4.53%. Although LLMs may mistakes in these assessments, this difference is substantial and suggests that confidence and lift are effective in removing low-quality checks. Finally, the interpolation pass was able to generate more than 800 checks initially, 40% of which were supported by the LLM and added to the candidate rule list (shown as llm-found). The rest were discarded (llm-remove).

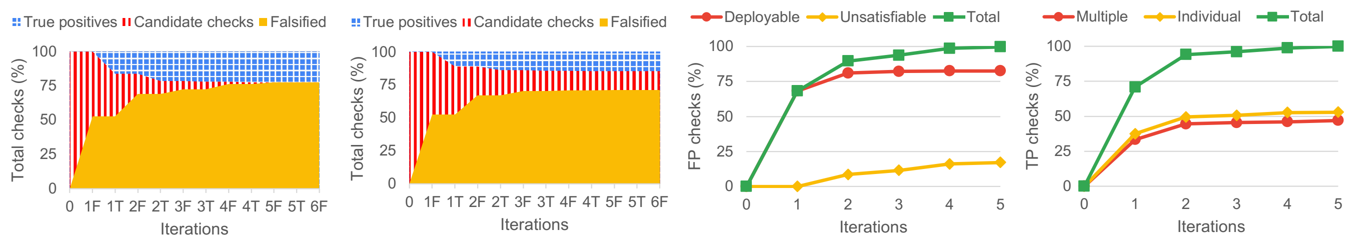
Check encoding strategy	TP Num.	FP Num.
Ignoring non-target checks	4.80	11.76
Zodiac (consider other checks)	0	4.04

Config mutation strategy	Attr. Num.	Topo. Num.
No constraints on changes	11.05	3.20
Zodiac (minimizing changes)	2.87	2.90

Table 5. Zodiac test case generation needs to consider interference from other checks, and minimize mutation impact.

5.4 Effectiveness of the validation phase

Zodiac's validation pipeline is carefully designed to eliminate false positives. In Table 5, we present our major design decisions on negative test case generation. The top half of the table shows that considering all checks in R_v and R_c helps find failure root causes. If we only test a single candidate check, without considering R_v and R_c, this results in an



(a) Zodiac scheduling. (b) Scheduling: no indistinct handling. (c) False positive removal process. (d) True positive validation process.

Figure 8. All scheduling components must be used for effective validation.

average of 4.80 true positive violations and 11.76 false positive violations when generating negative test cases for each program. This makes it difficult for Zodiac to draw definitive conclusions about the validity of a check. Our solution, which considers all known checks, ensures that no violations against R_v occur, while minimizing violations towards R_c. The bottom half shows that encoding attribute and topology changes as minimization constraints helps with negative test case generation. Without these constraints, the average number of attribute changes in each negative test case is as high as 11.05, reducing the reliability of validation results.

Figure 8 shows the importance of our validation scheduling algorithm. Figure 8(a) demonstrates the overall convergence process of the validation phase. False positive removal passes gradually mark checks as false positives, while true positive validation passes put them into R_v. After six iterations, the candidate rule set R_c becomes empty, ending the validation process. Figure 8(b) shows the scheduling process without handling indistinguishable checks. The validation engine converges to a stage where no new true or false positives can be found, yet R_c is still not empty. Figure 8(c) breaks down the false positive removal passes. In the beginning, most false positives are removed because their negative test cases could not trigger failures (shown by the 'Deployable' curve). As more checks are evaluated, the focus of removal shifts to cases where target checks do not have negative test cases due to conflicts with checks in R_v (shown by the 'Unsatisfiable' curve). Figure 8(d) breaks down the true positive validation passes. It shows the importance of handling indistinguishable checks, as almost half of the true positives are validated with more than 1 check violation in negative test cases (shown by the 'Multiple' curve), i.e., they are within certain indistinguishable check groups.

Finally, Table 6 demonstrates how the pruning methods used by Zodiac helps with minimizing test cases. It lists several example resource types and their average positive test case size—with MDC-based pruning (pruned) and without pruning (orig.). MDC helps remove both attended resources (att.) and unattended resources (unatt.), reducing the size of test cases by a magnitude of 3X to 9X. Most checks mined by Zodiac could be evaluated with fewer than 10 resources, except interpolation checks that require aggregation operators. This ensures the SMT solving time is typically within a second on a standard server, and cloud deployment cost per test

Type	pruned/att.	orig./att.	pruned/unatt.	orig./unatt.
FW	**6.50**	17.88	**1.00**	5.00
SG	**2.92**	18.33	**0.42**	5.58
GW	**5.60**	18.33	**0.40**	5.58
LB	**3.92**	22.50	**1.08**	9.92
RT	**4.57**	41.57	**1.14**	8.71

Table 6. Zodiac needs pruning during the scheduling phase to reduce overhead and avoid unattended resource types.

is under a dollar, both affordable even for individual developers. It is also evident that IaC programs typically come with multiple unattended resources (orig./unatt.), which could post threats to validity if not mitigated by MDC (though not completely removed, shown as pruned/unatt.).

5.5 Real world misconfigurations

To evaluate Zodiac's capability of detecting real world semantic check violations, we applied our checks to inspect all repositories used in the validation phase. Overall, Zodiac detected misconfigurations in 85 of these repositories, accounting for 2.0% of our dataset. As another test, we manually encoded the top-3 checks with the most amount of violations into Github API search queries [8–10]. Scanning through Github, these three checks identified 200+ other repositories (outside our dataset) that violate these constraints.

Moreover, we were able to identify four incorrect usage examples within the Terraform Azure provider documentation. We have submitted bug reports and suggested fixes as Github issues [13–16] to the Azure provider plugin developers, who responded quickly and fixed all of them. As an example, one such buggy usage [26] consists of the following resources:

```
/* Associate NIC with APPGW address pool */
resource VPC a, SUBNET b, c ...
/* Violation 1: IP of APPGW must have Standard sku */
resource IP d { sku = "Basic"; allocation = "Dynamic"}
resource APPGW f { ip_id = IP.d.id;
  subnet_id = SUBNET.b.id}
/* Violation 2: The subnet of APPGW is exclusive */
resource NIC e { subnet_id = SUBNET.b.id }
resource Association g ...
```

This program passes IaC validation and compilation in Terraform, but violates two Zodiac checks simultaneously.

Violation 1: Azure requires that if an IP address resource is used for an APPGW (application gateway), then it has to use `Standard` sku rather than `Basic`:

```
let r1:APPGW, r2:IP, in
conn(r1.subnet_id → r2.id) => r2.sku == 'Standard'
```

A naïve fix seems to be changing the sku value from `Basic` to `Standard`, but in fact, doing so would result in another semantic check violation within the IP resource:

```
let r:IP in
r.allocation == 'Dynamic' => r.sku == 'Basic'
```

The check states that if IP resource does not use `Basic` SKU, it cannot apply `Dynamic` allocation. A complete fix must therefore also change the allocation from `Dynamic` to `Static`.

Violation 2: A second semantic violation arises from incorrect subnet usage. APPGW, like GW mentioned in Section 5.1, requires exclusive usage of its subnet. However, the NIC in this program shares the same `subnet.b` with APPGW, which goes against cloud requirements. It is also worth noting that the developers of the usage example did declare two distinct subnets (`subnet.b`, `subnet.c`) earlier in the program, but later code only made use of one subnet. To fix this, we could instead correct the NIC to the other `subnet.c`. This example shows that mistakes could easily occur even for expert Terraform programmers.

5.6 False positives

Like other configuration mining projects, Zodiac is subject to an open-world assumption [19]—there may be invariants that the mining phase cannot find, either because they do not appear in the crawled dataset, or because they go beyond our curated mining templates. As such, the validation phase cannot offer soundness guarantees. Indeed, we have found checks that align with Zodiac's definition of true positives (i.e. t_p could be deployed while corresponding t_n fail to deploy), but are in fact false positives. This is because, when the mutated test case t_n fails to deploy, the root cause may lie in some other checks that Zodiac is not aware of, instead of the candidate check that is being validated. As a concrete example, Zodiac's validation engine believes that a check stating "if a VM is reachable to a VPC, then it must specify a source image reference block" is correct:

```
let r1:VM, r2:VPC, in
path(r1 → r2) => r1.source_image_ref != null
```

However, this rule is not a complete semantic check. In fact, if `VM.create` attribute is set to `Image`, then the above statement is indeed true. Instead, if the create option is set to `Attach`, then VM can be deployed without a source image reference:

```
let r:VM in
r.source_image_ref == null => r.create == 'Attach'
```

The false positive occurred because cloud users rarely use the `Attach` option, to a point that `Image` appears to be the only available create option within our dataset. Zodiac henceforth failed to unearth the correct check regarding `Attach`. Consequently, Zodiac cannot generate a meaningful negative test case to falsify the incorrect claim.

Initially, our validation engine outputs 539 semantic checks, but 29 of them have been identified as false positives, accounting for 5.4% of all validated checks. Among them, 17 false positives (3.1%) are identified through an automated counterexample testing pass. Concretely, Zodiac looks for additional repositories that violate each check and observes whether they actually fail to deploy. If some of them are in fact deployable (i.e., counterexamples of the check exist), then Zodiac marks the check as false positive. The remaining 12 (2.2%), on the other hand, are identified by manually examining all checks. Specifically, we cross-reference each check with Terraform and Azure documentations. If a check is recognized as a possible false positive, then we manually craft test cases that violate the check and observe deployability.

6 Discussion

Different IaC frameworks. While our current prototype targets Terraform, there exist several other IaC frameworks, such as AWS CDK [1], CloudFormation [2], CDKTF [12], and Pulumi [22], and Bicep [3]. *(i) Declarative vs. imperative.* frameworks like Pulumi use imperative code (e.g., Python and TypeScript) for configuring resources, in contrast to the declarative approach in Terraform. In Terraform, the intra- and inter-resource relations are directly encoded into the configuration, whereas an imperative IaC program might require more advanced software analysis techniques, e.g., for mining semantic checks and mutating programs to obtain negative test cases. *(ii) Framework architectures.* IaC frameworks also differ in their software architectures. For instance, Terraform has a core compiler that is cloud-agnostic, with plugin extensions that can be integrated by individual cloud providers. AWS CDK, on the other hand, only targets Amazon's cloud; it does not expose similar interfaces, and its core compiler might have a smaller semantic gap since it is developed by the cloud provider itself. Extending Zodiac to other IaC platforms, therefore, may require additional manual curation of the semantic KB, check templates, and new program analysis techniques for deployment-based testing.

One potential roadmap is to analyze the IaC deployment plans instead of the original IaC programs. Typically, an IaC program is first compiled into a JSON-like format before the deployment phase, and across IaC frameworks their deployment plans have similar formats which could serve as the common denominator. For instance, CDKTF and Terraform share the same JSON plan format; AWS CDK compiles into CloudFormation [2] which also supports JSON. Some of Zodiac's components (e.g., the mining engine) already operate on the JSON deployment plans, which should be reusable in specific cases. We leave this exploration to future work.

Handling IaC/cloud-level changes. Cloud services may introduce new resources or modify existing ones, so Zodiac needs to periodically update its semantic checks by rerunning the automated mining/validation pipeline. Zodiac may also need to incorporate new check templates over time, and constructing additional templates is currently a manual process. However, the templates only comprise 400 LoC for our current version, and we expect that template changes would occur more infrequently than service changes.

Different cloud providers. Our prototype targets Microsoft Azure, but many other cloud providers use IaC-style management. To generalize Zodiac's techniques to other providers (e.g., AWS, GCP), assuming that IaC programs are written against Terraform, the mining and validation pipelines should be reusable. We would need additional effort to curate check templates and KB entries for the new cloud provider. Recall that for Azure, we manually curated the templates which account for about 400 lines of code; for AWS and GCP, we expect a similar amount of manual work and a similar template library size. Beyond the three major providers, additional challenges will arise if Zodiac needs to support "low-resource" providers (i.e., smaller clouds). Zodiac relies on open-source IaC repositories, so the data scarcity issue would be amplified when supporting a less popular cloud provider. This is also an interesting avenue of future work.

Unsupported constraints. As a limitation, there are two classes of constraints that Zodiac currently does not capture. (i) Region-specific: Each cloud provider may have multiple regions, each with certain service differences—e.g., some VM skus may not be supported in all regions [23]. (2) Subscription-specific: Each cloud account may come with distinct resource quota, as users can request more capacity for their subscription on demand [7]. Extending Zodiac to capture these semantic checks is an interesting avenue of future work.

Use cases. Apart from finding violations in IaC programs, Zodiac could also enable other IaC tasks. For instance, some IaC frameworks are introducing LLM-powered program synthesis workflows [21], but the generated programs often suffer from bugs due to hallucination. Zodiac semantic checks, in this case, could serve as a RAG (retrieval-augmented generation) knowledge base [53] to provide additional context to LLMs. By explicitly asking LLM models to conform with these checks, users could potentially improve the quality of their generated programs and accelerate development. Another use case of Zodiac is to systematically bolster IaC provider documentation. Zodiac could transform unearthed semantic checks into natural language format, and offer them to IaC users as documented deployment insights.

7 Related work

Association rule mining. Program analysis techniques have been used to discover domain-specific checks/patterns/errors, typically configuration invariants [60, 61, 64, 70] and guidance on correlated changes on cloud services [57]. They focus on domain-specific inputs dissimilar to cloud IaC, and do not consider automated validation in their designs.

API fuzzing. Another line of work performs *RESTful API fuzzing* [34, 47, 63, 66] to test service API calls, or uses *Configuration Error Injection Testing (CEIT)* [54, 71] to test application configurations. These projects only aim to find defects in API/application implementations, while Zodiac aims to not only find problems, but also pinpoint their root causes.

Verification. LLM-based filtering methods have been developed to validate the query correctness [46, 55, 59, 65], but they are prone to hallucinate, thus cannot be used directly for correctness critical validation. Zodiac instead uses LLMs to fill in missing check details. Formal verification techniques have been applied to other domains such as network configuration verification [36, 56] and synthesis [38, 58], performance analysis [33], and specific components of cloud computing [39, 41, 62]. Zodiac instead applies formal reasoning to generate and prune semantic check test cases.

Invariant mining. Apart from configuration mining, there are many works focusing on program invariant mining [32, 40, 42, 43] distributed protocol invariant mining [45, 67, 68], and trace invariant mining [48, 51]. Their targeted invariants and mining methods are different from those of Zodiac.

8 Conclusion

Cloud IaC frameworks are gaining popularity. However, today, IaC programs that pass the compiler could still fail during the actual cloud deployment. These problems are hard to identify and fix, and could result in disruption to a deployed cloud infrastructure. Zodiac is a tool that mines and validate additional semantic checks as reliability guardrails for IaC. Our evaluation shows that Zodiac can identify many useful checks whose violation can trigger deployment failures. This outperforms baseline techniques and existing IaC static checkers. It has also identified bugs in real-world Terraform repositories and documentation.

Acknowledgments

We thank our shepherd, Rebecca Isaacs, for her insightful and detailed feedback that helped improve this paper. We also thank the anonymous reviewers for their helpful reviews. This work was partially supported by NSF grants CNS-1942219, CNS-2106751, CNS-2106388, CNS-2214272, and a VMware Early Career Faculty Grant.

References

[1] AWS cloud development kit. https://github.com/aws/aws-cdk.
[2] AWS CloudFormation. https://aws.amazon.com/cloudformation/.
[3] Azure Bicep. https://learn.microsoft.com/en-us/azure/azure-resource-manager/bicep/.
[4] Azure site-to-site VPN connection. https://learn.microsoft.com/en-us/azure/vpn-gateway/tutorial-site-to-site-portal.
[5] Azure SQL Managed Instance management . https://learn.microsoft.com/en-us/azure/azure-sql/managed-instance/management-operations-overview?view=azuresql.
[6] Azure Storage Account Redundancy. https://learn.microsoft.com/en-us/azure/storage/common/storage-account-overview.
[7] Azure subscription and service limits, quotas, and constraints. https://learn.microsoft.com/en-us/azure/azure-resource-manager/management/azure-subscription-service-limits.
[8] Checking for IaC Programs where APPGW uses Basic IP Address with Dynamic Allocation Method. https://github.com/search?q=

azurerm_public_ip+allocation_method++Dynamic+NOT+static+ azurerm_application_gateway+resource+NOT+each+language%3A HCL+&type=code.

[9] Checking for IaC Programs where APPGW with non-WAF v2 sku uses Web Application Firewall. https://github.com/search?q=waf_ configuration+NOT+WAF_v2+azurerm_application_gateway++NOT+ variable+NOT+output+language%3AHCL+&type=code.

[10] Checking for IaC Programs where the Request Routing Rule of Standard v2 APPGW does not Specify Priority. https://github.com/search?q=+azurerm_application_gateway+request_routing_rule+NOT+ priority+Standard_v2+language%3AHCL&type=code.

[11] Checkov: ship code that's secure by default. https://bridgecrew.io/checkov/.

[12] Cloud development kit for terraform. https://developer.hashicorp.com/terraform/cdktf.

[13] Example Usage within azurerm_application_gateway Documentation Cannot be Deployed Successfully. https://github.com/hashicorp/terraform-provider-azurerm/issues/27065.

[14] Example Usage within azurerm_dedicated_hardware_security_module Documentation Cannot be Deployed Successfully. https://github.com/hashicorp/terraform-provider-azurerm/issues/27078.

[15] Example Usage within azurerm_mssql_database Documentation Cannot be Deployed Successfully. https://github.com/hashicorp/terraform-provider-azurerm/issues/27194.

[16] Example Usage within azurerm_network_interface_application_gateway _backend_address_pool_association Documentation Cannot be Deployed Successfully. https://github.com/hashicorp/terraform-provider-azurerm/issues/27222.

[17] Microsoft Azure Fsv2-series VM. https://learn.microsoft.com/en-us/azure/virtual-machines/fsv2-series.

[18] Opa's native query language rego. https://www.openpolicyagent.org/docs/latest/policy-language/.

[19] Open world assumptions. https://www.sciencedirect.com/topics/computer-science/open-world-assumption.

[20] OpenTofu: The open source infrastructure as code tool. https://opentofu.org/.

[21] Pulumi ai. https://www.pulumi.com/ai.

[22] Pulumi: Infrastructure as code in any programming language. https://www.pulumi.com/.

[23] Regions for virtual machines in azure. https://learn.microsoft.com/en-us/azure/virtual-machines/regions#special-azure-regions.

[24] Shift testing left with unit tests. https://learn.microsoft.com/en-us/devops/develop/shift-left-make-testing-fast-reliable.

[25] Terraform by Hashicorp. https://www.terraform.io/.

[26] Terraform Resource: Manages the association between a Network Interface and a Application Gateway's Backend Address Pool. https://registry.terraform.io/providers/hashicorp/azurerm/3.97.0/docs/resources/network_interface_application_gateway_backend_address_pool_association.

[27] Terraform v.s. alternatives. https://developer.hashicorp.com/terraform/intro/vs/cloudformation.

[28] Terrascan: Detect compliance and security violations across Infrastructure as Code to mitigate risk before provisioning cloud native infrastructure. https://runterrascan.io/.

[29] TFLint: A Pluggable Terraform Linter. https://github.com/terraform-linters/tflint.

[30] TFSec: Security Scanner for Your Terraform Code. https://github.com/aquasecurity/tfsec.

[31] Zodiac: Unearthing Semantic Checks for Cloud Infrastructure-as-Code Programs. https://github.com/824728350/Zodiac.

[32] Glenn Ammons, Rastislav Bodík, and James R Larus. Mining specifications. *ACM Sigplan Notices*, 37(1):4–16, 2002.

[33] Mina Tahmasbi Arashloo, Ryan Beckett, and Rachit Agarwal. Formal methods for network performance analysis. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023.

[34] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: Stateful REST API fuzzing. In *IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019.

[35] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Checking security properties of cloud service rest apis. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020.

[36] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2017.

[37] Giannis Bekoulis, Christina Papagiannopoulou, and Nikos Deligiannis. A review on fact extraction and verification. *ACM Computing Surveys (CSUR)*, 55(1):1–35, 2021.

[38] Eric Hayden Campbell, William T Hallahan, Priya Srikumar, Carmelo Cascone, Jed Liu, Vignesh Ramamurthy, Hossein Hojjat, Ruzica Piskac, Robert Soulé, and Nate Foster. Avenir: Managing data plane diversity with control plane synthesis. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021.

[39] Claudia Cauli, Meng Li, Nir Piterman, and Oksana Tkachuk. Pre-deployment security assessment for cloud services through semantic reasoning. In *Computer Aided Verification (CAV 21)*, 2021.

[40] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.

[41] Alexandros Evangelidis, David Parker, and Rami Bahsoon. Performance modelling and verification of cloud-based auto-scaling policies. *Future Generation Computer Systems*, 87:629–638, 2018.

[42] Grigory Fedyukovich and Rastislav Bodík. Accelerating syntax-guided invariant synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2018.

[43] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. Quantified invariants via syntax-guided synthesis. In *Computer Aided Verification (CAV 19)*, 2019.

[44] Zheng Guo, David Cao, Davin Tjong, Jean Yang, Cole Schlesinger, and Nadia Polikarpova. Type-directed program synthesis for restful apis. In *43rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 22)*, 2022.

[45] Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. Finding invariants of distributed systems: It's a small (enough) world after all. In *18th USENIX symposium on networked systems design and implementation (NSDI 21)*, 2021.

[46] Jean Kaddour, Joshua Harris, Maximilian Mozes, Herbie Bradley, Roberta Raileanu, and Robert McHardy. Challenges and applications of large language models. *arXiv preprint arXiv:2307.10169*, 2023.

[47] Myeongsoo Kim, Saurabh Sinha, and Alessandro Orso. Adaptive REST API Testing with Reinforcement Learning. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.

[48] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. Automatic mining of specifications from invocation traces and method invariants. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 14)*, 2014.

[49] Nayeon Lee, Yejin Bang, Andrea Madotto, Madian Khabsa, and Pascale Fung. Towards few-shot fact-checking via perplexity. *arXiv preprint arXiv:2103.09535*, 2021.

[50] Nayeon Lee, Belinda Z Li, Sinong Wang, Wen-tau Yih, Hao Ma, and Madian Khabsa. Language models as fact checkers? *arXiv preprint arXiv:2006.04102*, 2020.

[51] Caroline Lemieux. Mining temporal properties of data invariants. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE 15)*, volume 2. IEEE, 2015.

[52] Julien Lepiller, Ruzica Piskac, Martin Schäf, and Mark Santolucito. Analyzing infrastructure as code to prevent intra-update sniping vulnerabilities. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 21)*, 2021.

[53] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[54] Wang Li, Zhouyang Jia, Shanshan Li, Yuanliang Zhang, Teng Wang, Erci Xu, Ji Wang, and Xiangke Liao. Challenges and opportunities: an in-depth empirical study on configuration error injection testing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 21)*, 2021.

[55] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with Alphacode. *Science*, 378(6624):1092–1097, 2022.

[56] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. P4v: Practical verification for programmable data planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on data communication (SIGCOMM 18)*, 2018.

[57] Sonu Mehta, Ranjita Bhagwan, Rahul Kumar, Chetan Bansal, Chandra Maddila, Balasubramanyan Ashok, Sumit Asthana, Christian Bird, and Aditya Kumar. Rex: Preventing bugs and misconfiguration in large services using correlated change analysis. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020.

[58] Yiming Qiu, Ryan Beckett, and Ang Chen. Synthesizing runtime programmable switch updates. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023.

[59] Laria Reynolds and Kyle McDonell. Prompt programming for large language models: Beyond the few-shot paradigm. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021.

[60] Mark Santolucito, Ennan Zhai, Rahul Dhodapkar, Aaron Shim, and Ruzica Piskac. Synthesizing configuration file specifications with association rule learning. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA 17):1–20, 2017.

[61] Mark Santolucito, Ennan Zhai, and Ruzica Piskac. Probabilistic automated language learning for configuration files. In *Computer Aided Verification: 28th International Conference, (CAV 16), Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28*. Springer, 2016.

[62] Alireza Souri, Nima Jafari Navimipour, and Amir Masoud Rahmani. Formal verification approaches and standards in the cloud computing: a comprehensive and systematic review. *Computer Standards & Interfaces*, 58:1–22, 2018.

[63] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. Resttestgen: automated black-box testing of RESTful APIs. In *IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020.

[64] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, 2004.

[65] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

[66] Huayao Wu, Lixin Xu, Xintao Niu, and Changhai Nie. Combinatorial testing of RESTful APIs. In *Proceedings of the 44th International Conference on Software Engineering (ICSE 22)*, 2022.

[67] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. DuoAI: Fast, automated inference of inductive invariants for verifying distributed protocols. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

[68] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. Distai:data-driven automated invariant learning for distributed protocols. In *15th USENIX symposium on operating systems design and implementation (OSDI 21)*, 2021.

[69] Jialu Zhang, Ruzica Piskac, Ennan Zhai, and Tianyin Xu. Static detection of silent misconfigurations with deep interaction analysis. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA 21):1–30, 2021.

[70] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. Encore: Exploiting system environment and correlation information for misconfiguration detection. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems (ASPLOS 14)*, 2014.

[71] Sai Zhang and Michael D Ernst. Proactive detection of inadequate diagnostic messages for software configuration errors. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 15)*, 2015.