Simplifying Cloud Management with Cloudless Computing

Yiming Qiu, Patrick Tser Jern Kon, Jiarong Xing[†], Yibo Huang, Hongyi Liu[†]
Xinyu Wang, Peng Huang, Mosharaf Chowdhury, Ang Chen

University of Michigan [†]Rice University

ABSTRACT

Cloud computing has transformed the IT industry, but managing cloud infrastructures remains a difficult task. We make a case for putting today's management practices, known as "Infrastructure-as-Code," on a firmer ground via a principled design. We call this end goal *Cloudless Computing*: it aims to simplify cloud infrastructure management tasks by supporting them "as-a-service," analogous to serverless computing that relieves users of the burden of managing server instances. By assisting tenants with these tasks, cloud resources will be presented to their users more readily without the undue burden of complex control. We describe the research problems by examining the typical lifecycle of today's cloud infrastructure management, and identify places where a cloudless approach will advance the state of the art.

CCS CONCEPTS

• **Networks** → **Cloud computing**; • **Software and its engineering** → **Orchestration languages**;

KEYWORDS

Infrastructure as code, cloud management

1 INTRODUCTION

Cloud computing has transformed the IT infrastructure, with 94% enterprises relying on cloud services of some form [1, 28]. However, cloud resources remain difficult to configure and manage [14]. Cloud infrastructure management is necessary because cloud workloads and applications are diversifying beyond a few broad categories of Software-as-a-Service (SaaS) products [52]. Each workload on the long tail of diverse cloud applications requires different infrastructure support; therefore, their management tasks are also customized. As a result, cloud tenants (e.g., enterprises) employ extensive cloud/DevOps engineering teams, who traditionally worked with cloud-specific APIs [65] to configure infrastructures or manually constructed infrastructures through a "ClickOps" approach using cloud portals [63].

Modern cloud automation frameworks arose as an attractive alternative to abstract away many of the complexities of cloud management tasks. They follow the "Infrastructure-as-Code" (IaC) paradigm [48] that enables users to define their desired infrastructures (e.g., VMs, subnets, and VPN gateways) by writing high-level IaC programs, while shielding them from low-level details about how the underlying resources are instantiated. Widely-used IaC frameworks include Terraform [32], OpenTofu [20], and Pulumi [24], all of which work across cloud providers. Additionally, there are cloud-specific counterparts such as Amazon CloudFormation [4] and Azure Bicep [6]. Given an IaC program, these frameworks reason about deployment plans and map them to cloud-level APIs to create, modify, or destroy cloud resources.

However, today's IaC frameworks lack a principled design to fully unleash their benefits, and IaC-level abstractions of cloud-level behaviors are often "leaky." There exists a significant gap between what cloud users perceive (i.e., the IaC-level configuration) and what they actually receive (i.e., the cloud-level infrastructure). We observe that many DevOps hours are spent to address this gap, often in a manual, trial-and-error fashion across the cloud infrastructure lifecycle. The industry ideal of DevOps engineering, which promises to streamline developer and operator teams, is severely hindered by these complexities. Developers without cloud operation expertise find it hard to manage infrastructure by themselves, while operators with that knowledge have to spend significant amount of time handling all types of bugs and failures, rather than providing easy-to-use interfaces to developers.

In this position paper, we argue for a vision that we call *Cloudless Computing*. Our view is that IaC-style management is the right direction forward, but we must rethink its constituent components, identify missing pieces from today's practices, and create a coherent roadmap. Like serverless computing, which aims to reduce the burden of cloud users in managing server VMs, cloudless computing aims to support cloud infrastructure management on behalf of users by handling "cloudy" management tasks in a principled manner. This will reduce the friction in managing cloud-based infrastructures, so that developers and operators can work in a concerted fashion for better control over their infrastructure. We articulate design gaps and needed tools for overseeing the entire lifecyle of cloud infrastructures, in order to meet the constantly-evolving demands of tenant workloads.

In describing this vision, we are faced with the fact that today's cloud management is a set of loose, under-documented

HotNets '23, November 28–29, 2023, Cambridge, MA, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 979-8-4007-0415-4/23/11...$15.00
https://doi.org/10.1145/3626111.3628206

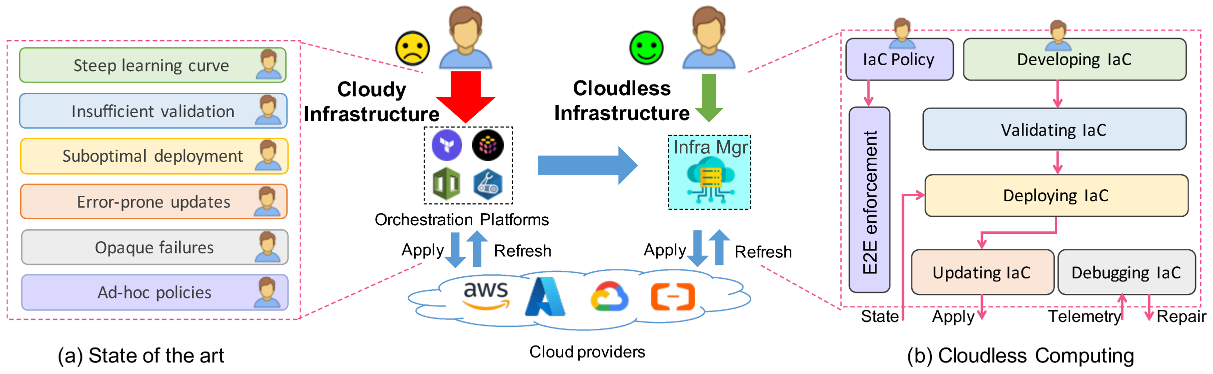

Figure 1: We call the end goal of an ideal management scheme "cloudless computing." (a) Today's practices, known as "Infrastructure-as-code" (IaC) suffer from a range of limitations, which undercut the benefit they bring and make the management process foreign and "cloudy." (b) We believe that a principled design will result in a better framework for cloud infrastructure management, providing a correct and streamlined process for the entire cloud lifecycle.

practices rather than an exact science. We anchor our discussion by embedding the research problems into the typical lifecyle of today's cloud infrastructure, including developing the IaC configurations, validating configuration correctness, scheduling the deployment steps, performing infrastructure updates and handling failures, as well as enforcing user-defined policies on the cloud infrastructure throughout all stages. Figure 1 is an overview of the full stack for cloud infrastructure management, and we examine each component in detail in this paper.

2 INFRASTRUCTURE AS CODE (IAC)

Infrastructure-as-Code (IaC) frameworks, such as Terraform, OpenTofu, and Pulumi, have gained wide popularity.

2.1 Existing IaC frameworks

IaC frameworks allow cloud users to codify their desired infrastructure as a high-level configuration file, removing the need to understand the underlying provisioning steps to achieve their desired end-state. This could be done with imperative or declarative designs. Some IaC frameworks are developed outside the major cloud providers. In Pulumi, IaC programs are written using existing imperative programming languages (e.g., Pulumi's Python SDK [21] package). In Teraform/OpenTofu, IaC programs are written in a declarative style using the HCL language [16], which is an expressive language with many constructs for modularity. Other IaC frameworks are supported by individual cloud providers directly, such as Azure Bicep and AWS CloudFormation, offering analogous functionalities. They either have their own languages [11] or use JSON/YAML as configuration formats.

Consider a simplified IaC program in HCL, as shown in Figure 2. It first uses a Terraform data source to obtain the current AWS region being used (line 2). Next, it declares a variable (lines 4-7) describing a configuration value to be defined later. Resource blocks are another important element in HCL, and they declare instances of specific infrastructure

```
1  /* Simplified Terraform code snippet */
2  data "aws_region" "current" {}
3
4  variable "vmName"{
5   type = string
6   default = "cloudless"
7  }
8
9  resource "aws_network_interface" "n1"{
10  name = "example-nic"
11  location = data.aws_region.current.name
12 }
13
14 resource "aws_virtual_machine" "vm1"{
15  name = var.vmName
16  nic_ids = [aws_network_interface.n1.id]
17 }
```

Figure 2: A simplified Terraform IaC program.

objects, such as network interfaces (lines 9-12) and virtual machines (lines 14-17). Each such resource is further configured with various attributes (e.g., name, location).

After the user describes the infrastructure in the IaC program, the rest is handled by the IaC frameworks for deployment. As the first step, basic validation will be performed to check for format and grammatical correctness [40]. Next, the user-provided IaC program (i.e., the user's desired cloud state) will be automatically compared with the user's current cloud state (if any), resulting in a resource dependency graph where some nodes are marked as to be added or deleted [26, 38]. In the case of Pulumi, IaC programs are embedded directly in application code [23], so its language runtime observes code execution to extract resource registrations (e.g., create an AWS S3 bucket) in order to construct the graph. From there, an execution plan [34] is created, which specifies what resources need to be updated in what dependency order. Finally, resources are deployed or destroyed [33], by instructing the appropriate resource providers [27, 36] to construct required contexts and execute appropriate cloud API calls.

However, mapping from the IaC-level code constructs to the cloud-level APIs is quite involved [25, 36]. Mapping rules are usually crafted by a collaboration between teams from the IaC vendors and cloud providers, which is a manual and labor-intensive process [3, 15]).

Besides cloud providers and IaC framework vendors, the IaC ecosystem also includes another third-party community consisting of IaC contributors, who create open-source IaC modules [37] that can be composed in various deployment scenarios by cloud users.

2.2 Limitations of today's IaC management

While existing IaC frameworks lower the barrier of using the cloud, they have notable limitations across all stages of infrastructure management, presenting significant challenges.
Developing IaC infrastructure (§3.1). For enterprises that have not adopted IaC, transitioning to IaC-style management will be a steep learning curve. This not only involves learning a new paradigm and ecosystem, but also migrating an existing cloud deployment already in use to IaC programs.
Validating IaC infrastructure (§3.2). IaC-level programs do not have visibility into cloud-level API behaviors. Thus, a seemingly correct IaC program (i.e., one that compiles successfully) may still cause deployment errors. We need stronger mechanisms to validate IaC correctness.
Deploying IaC infrastructure (§3.3). Existing IaC frameworks suffer from inefficiencies when deploying cloud resources. Some common reasons are the lack of scheduling optimizations and dependency pruning.
Updating IaC infrastructure (§3.4). Once an IaC infrastructure is in deployment, it goes through myriad updates over its lifecycle. In existing frameworks, concurrent updates are subject to race conditions which could lead to inconsistency, and their rollback processes are not sufficiently robust.
Diagnosing IaC infrastructure (§3.5). During the lifecycle of the cloud infrastructure, many things can go wrong. Accidental drift from the desired state, or infrastructure bugs, require additional support from an IaC-level cloud debugger.
Policing IaC infrastructure (§3.6). The cloud lifecycle requires changes to IaC programs based on user intentions. Various policies are often required to govern each IaC program snapshot and how the infrastructure evolves. Existing frameworks and policy languages do not adequately support cloud user-level policies.

3 TOWARD CLOUDLESS COMPUTING

To achieve cloudless computing, we must systematically address all of these issues in the lifecycle of IaC-style cloud infrastructure management.

3.1 Developing IaC infrastructure

The cloud infrastructure lifecyle starts with the development phase. Cloud users must configure their desired infrastructure correctly by providing an IaC program. This is not an easy task, especially for IaC tools like Terraform [32] and OpenTofu [20] that involve learning a new language. Furthermore, for many enterprises that already have cloud-based infrastructures, their current deployments are not created and managed by IaC frameworks. We need better support for IaC development.
Automated IaC synthesis. Recent developments in LLMs (large language models) put us at an inflection point where program synthesis is ever closer to practical use. Unfortunately, existing LLM-based tools [22, 30, 31] frequently generate invalid IaC code, even for small-scale templates involving widely used resources (e.g., AWS EC2). Not only do LLMs hallucinate basic syntax, but they are also liable to introduce security vulnerabilities, representing a risk for production environments. Thus, one research direction is to tailor ML-assisted synthesis techniques [50, 58] specifically for IaC program generation, with the goal of generating reliably correct IaC programs that would improve the productivity of existing IaC users, while simultaneously lowering the barrier for new users to adopt IaC tools. A potential solution to this open problem is to decompose the infrastructure into its component elements to simplify synthesis, while jointly applying formal and textual specifications (e.g., type-guided and ML-based search) for multi-modal synthesis to improve reliability. Yet another approach could consider injecting relevant portions of the user's existing infrastructure as additional context in a retrieval augmented generation [60] fashion to guide the LLM in generating personalized code or suggestions.
Porting non-IaC infrastructures to IaC. The ability to port an existing, non-IaC cloud infrastructure to IaC frameworks is essential to wider adoption. Today, many enterprises already construct their infrastructures directly using cloud-level APIs or cloud portals outside IaC frameworks. Porting these deployments to IaC requires high-fidelity translation of low-level cloud infrastructure state to an equivalent IaC program, which is a challenging task. Industry practitioners have recognized this need, and tools like Aztfy [7] and Terraformer [41] resort to porting with static, pre-defined templates. The resulting IaC programs usually lack clear structures and require the DevOps engineers to manually analyze and refactor them.

We believe that porting from existing cloud infrastructures to IaC must be assisted with a program optimizer that provides structural guidance. Further, the main objective is code "quality" in terms of ease of understanding and maintenance rather than just correctness or performance goals. This raises two interesting research questions: (1) how should we formally define and quantify these code metrics? and (2) how should we devise automated refactoring techniques to achieve these objectives? For instance, if the cloud-level state contains many resources of the same type, the corresponding IaC program should use compact structures such as `count` and `for_each` in Terraform instead of a straight enumeration of all resources one by one; as another example, nested modules in Terraform are another way to wrap sets of resources with the same structure. For an individual resource, many of its cloud-level attributes could be removed when porting to the IaC level, because they will be automatically constructed when lowering an IaC program to the cloud level.

3.2 Validating IaC infrastructure

Even a grammatically-correct IaC program could exhibit undesired behaviors—akin to "configuration errors" in other systems where the problem does not stem from the IaC program itself but rather the parameter values [67, 70]. For cloud deployments, such errors are exacerbated by the fact that (1) configuration correctness is eventually decided at the cloud level, not the IaC level; (2) cloud providers have differing expectations on correct behaviors; and (3) cloud behaviors evolve over time due to feature changes. Thus, we believe that cloudless computing requires a powerful IaC validation phase to catch potential deployment issues as early as possible, preferably before deploying any resources into providers, to reduce the amount of DevOps engineering effort and time.

Semantic validation with stronger IaC types. Current IaC languages are weakly typed. For instance, in Terraform, resource attributes are treated as generic "strings" although they carry much richer semantic information—e.g., one "string" may specifically represent a virtual machine and another specifically a subnet. With today's types, composing resources into dependency graphs is error-prone. As a concrete example, Azure requires that a virtual machine resource must reference its network interface by the resource ID; however, at the IaC level, this reference could be easily misused (e.g., by referencing the ID of a different resource type).

Thus, one interesting research direction is to augment the IaC frameworks with semantic types [57], to make resource composition easier by design. This follows existing work [57] that performs type discovery within a restricted vocabulary, but IaC frameworks support a much larger set of cloud resources, each with different attributes, which are further constantly evolving due to cloud feature changes. To address these additional challenges, one possible solution is to rely on IaC usage examples, IaC documentations, as well as cloud-level API specifications, to derive stronger validation checks either using analytical or NLP-based methods. If we could automatically extract a graph representation of resource types and dependencies from online sources, then we can derive a knowledge base about resource types and update it as cloud features evolve at the IaC level.

Deeper, cloud-specific validation. Further validation is required beyond just typing, often in a cloud-specific manner. Consider a concrete example: Azure requires that VMs and their attached network interface cards (NICs) must be in the same cloud region. If a configuration violates this rule, it will error out during deployment. However, at the IaC level, a program may specify VMs and their NICs to be in different regions while still passing all IaC-level syntax check. As additional examples, Azure VMs could specify a password only if another `disable_password` attribute is explicitly set to false; Azure virtual networks cannot have overlapping address spaces if they are connected with each other through peering or gateway connection. These cloud-level constraints usually involve interactions among multiple different resources and their parameters, which are often under-specified at the IaC level, because the IaC-level compiler is not fully aware of the cloud-level expectations, which could further change over time. Today, cloud users are also caught by surprise due to deployment errors, and fixing these problems increases DevOps engineering cost and time. Instead of leaving this burden to users at deployment time, we believe that these surprises should be eliminated at compile time via stronger, cloud-level validation. Our insight is that IaC-style management offers an opportunity to transform cloud-level constraints into IaC-level program checks, e.g., through domain-specific customization to existing techniques such as specification mining [54, 66, 70].

3.3 Deploying IaC infrastructure

After validation, IaC frameworks hand off the execution to the cloud by invoking various cloud-level APIs to update resources based on the dependency graph. Today's IaC frameworks, however, suffer from long deployment times due to suboptimal planning and "best effort" graph walks, constraining the velocity of incorporating needed features.

Accelerating IaC deployment. Deploying an IaC program to the cloud could take a long time, sometimes on the order of hours or even days [56, 62]. Current IaC frameworks only perform basic dependency analysis on the resource dependency graph [33], missing out potential acceleration opportunities with optimized deployment plans. The resource dependency graph is a DAG (directed acyclic graph), with multiple "parallel" subgraphs that can be deployed concurrently. Further, resources on "non-critical paths" could make way for "critical paths" to expedite the completion of the deployment. We believe that further analyses will not only lead to faster deployment speeds, but also help to locate potential errors quickly when debugging an IaC program. However, such analyses would require taking into account domain-specific constraints that dictate how IaC deployments can or cannot be parallelized—e.g., cloud API rate limiting, estimated deployment times for various cloud resources, retries in case of resource hanging or failure—to achieve this goal.

Accelerating deployment updates. IaC deployment is not a one-time effort; rather, deployment "deltas" are frequently incorporated to a live cloud infrastructure. Today's IaC frameworks unfortunately treat them similarly as a deployment from scratch—even a single resource update will trigger expensive queries on all cloud-level resource state and recomputation of the deployment plan from the ground up [38]. This results in high turnaround time. Cloudless computing should provide optimizations that enable incremental updates to accelerate cloud deployments. Our observation is that modifications to individual resources have a limited impact, affecting only a small subset of successor and predecessor nodes in the resource dependency graph. By identifying the "impact scope" of a deployment change, we can confine the changes to a significantly smaller resource subgraph, like in other contexts [55, 69]. This will reduce the overhead on resource state queries and redeployment, and lead to cost savings.

3.4 Updating IaC infrastructure

IaC infrastructure updates raise a set of challenges that go beyond accelerating deployment speeds.

Concurrent updates and mutual exclusion. For a large enterprise, multiple DevOps engineers or teams share the same cloud infrastructure and may submit updates concurrently. This further requires IaC frameworks to detect and avoid operation conflicts during infrastructure updates. Existing tools simply lock the entire cloud infrastructure for modifications at any scale [35], restricting the potential for parallel updates. Partitioning the cloud infrastructure into smaller segments managed by different DevOps engineers is not practical either, since the infrastructure is fundamentally a shared resource. Cloudless computing should provide granular locking mechanisms for concurrent updates while guaranteeing isolation. For instance, if we provide *per-resource* locks, mutual exclusion needs only arise when the same resource is being updated by different DevOps teams. Furthermore, a per-resource lock still allows them to execute updates on other resources without having to wait for all concurrent updates to settle. In general, we need a lock manager backed by an IaC database that reflects the "golden state" of the cloud infrastructure, as well as transaction mechanisms for atomic updates while guaranteeing isolation. Updates are scheduled based on the logical state and locks in the database, and only later applied to the physical infrastructure. Different lock scheduling strategies can be developed for different update goals.

IaC rollbacks during updates. In reality, any update might fail due to runtime errors, or the cloud users themselves may request a rollback for other reasons. One might think this is as simple as retrieving the previous state, analyzing the delta from the current deployment, and modifying it back [39]. However, this is not sufficient, as resource modifications may not be reversible in the same manner in which they are performed. Simply applying a previous configuration doesn't always roll back the infrastructure to its intended previous state. For instance, consider the case where a virtual machine instance has been modified with custom network settings that are not captured in the configuration files. Rolling back to a previous version does not mean these modifications will be automatically reversed simultaneously—as a matter of fact, they are often ignored by IaC workflow. In such cases, one viable solution is to identify resource modifications that are not easily reversible, and then destroy them with a new deployment from scratch. We want to minimize the amount of resource redeployment in the rollback process, and also guarantee a reliable identification of rollback plans before any updates are performed. Similarly, better version control systems that track the mapping between past configurations and their corresponding states—i.e., a "time machine"—would be a significant help to checkpointing resource states and generating precise rollback plans.

3.5 Diagnosing IaC infrastructure

An IaC debugger for cloud infrastructures is essential for cloudless computing, as failures happen frequently and are opaque to cloud users. The debugger should consist of an observability component that monitors runtime failures, as well as a repair component that reflect the cloud-level errors to the IaC-level program and suggest possible fixes.

IaC drift detection and reconciliation. "Resource drift" is a common class of runtime problems in IaC deployments. It refers to cloud infrastructure changes that occur outside of the control of cloud IaC [17]—e.g., when the infrastructure is managed by IaC frameworks but also legacy cloud-level API scripts. Without timely mitigation, the hybrid tooling could produce conflicting operations and result in failures or other vulnerabilities. Existing IaC frameworks cannot easily capture drifts caused by operations outside their control. Industry tools like driftctl [12] attempt to bypass the IaC frameworks and directly use cloud-level API to scan the deployment state, which incurs significant time overhead due to cloud API rate limiting [46]. Frequent scanning is also expensive if API calls have quotas or paywalls [45]. Cloudless computing should support drift detection natively within its own stack, by an observability component that relies on cloud activity logs [8, 13] to detect "drift events." If any unexpected event is reported in the log, the IaC frameworks should either regenerate the IaC-level program to reflect the latest deployment, or notify corresponding parties for further reconciliation.

IaC debugging and repair. Infrastructure deployments "error out" at the cloud level, but cloud users view their infrastructure at the IaC level. When a problem occurs, cloud providers generate error messages at the API level, which can make it difficult for users to understand the exact IaC resources involved and how to resolve the error. For example, an error message like "Linux virtual machine creation failed because specified NIC is not found" lacks precise correlation to the original IaC program itself—the above error message gives people the impression that NIC does not exist, while the root cause is that the NIC and VM were not configured in the same region. To make things worse, such error messages do not even pinpoint the specific "lines of code" as to which parameter is causing the anomaly. We need debuggers that correlate runtime cloud-level errors to the IaC program itself. This could be an analytical process or equipped by LLMs to translate natural language error messages into higher-level root causes and suggest fixes [47, 61, 68].

3.6 Policing IaC infrastructure

An IaC program describes a snapshot of the cloud infrastructure, but across the cloud lifecycle, there are user-specific policies that govern not only individual IaC programs but also their evolution over time. For example, an enterprise may require autoscaling policies while ensuring that their infrastructure does not exceed their budget; another may require that some specific resource types must be used (e.g., AWS database instances with the latest CPU features); infrastructures may also be subject to regulatory policies (e.g., GDPR, FedRAMP) as well as myriad security and privacy guidelines commonly practiced in their specific industry [19, 43, 44, 49, 53]. Thus, cloudless computing needs an "infrastructure controller" that enforces IaC policies across the lifecycle.

Enforcing policies with a controller. Analogous to an SDN controller, IaC policing tools could be viewed as the controller for the cloud infrastructure lifecycle, allowing users to enforce different policies as needed. Existing tools (e.g., Terrascan [42], Checkov [10]) either rely on an Open Policy Agent (OPA) [19] language (e.g., Rego [18]), or framework-specific languages (e.g., Sentinel [29] for Terraform), but these policy languages are hard to master. For instance, Rego is akin to Datalog, significantly different from languages that DevOps engineers are familiar with. We believe that a better controller would expose higher-level abstractions for authoring policies. Another angle is to support automated policy generation, e.g., inferring user-specific policies from their existing IaC programs. For instance, by adapting template extraction techniques [59], instead of writing exact policies, we can turn the problem into "outlier detection," which compares new IaC programs with templates extracted from existing repositories to detect deviations from common practices.

Policies as observations and actions. We believe that a better abstraction would clearly separate two aspects of the policy: the *observations*, and the *actions*. Consider autoscaling policies as a concrete example. Today, cloud autoscaling [64] targets certain services (e.g., for VMs [9] and Kubernetes [2]) and scale in/out events (e.g., CPU/memory utilization). However, users cannot easily define policies that are not explicitly supported by cloud providers, such as "scale out the number of VPN gateways and attached tunnels if traffic throughput is close to their capacity," or "scale out the number of VMs if their attached network interfaces are highly loaded." This is because the current IaC frameworks do not explicitly capture and expose enough metrics and events as "observations," while existing policy languages do not expose sufficiently rich "actions" to evolve the IaC program based on the observations. Allowing for a wider range of observations and actions would better support a broader variety of user policies.

Moreover, policies take effect at different phases of the infrastructure lifecycle. At each stage, different "observations" and "actions" would apply. For example, a policy that governs failure handling could take resource drifts as observations, but another policy that governs IaC updates may use autoscaling metrics for decision making. Thus, the policy language should be flexible enough to capture the evolving set of observations and actions throughout the cloud infrastructure lifecycle.

4 MANAGEMENT: THE FINAL FRONTIER

Much of cloud research has been directed to better designs for its software/hardware stacks and myriad use cases. Cloud infrastructure management as of today, although foundational to cloud usage, is defined by the set of tools and best practices rather than principled studies. To better enable innovation, we believe now is the time to focus on *management* issues as a top-level objective in cloud computing research.

We do view two lines of work as sharing similar goals with us, improving the manageability of cloud infrastructures, even if stated implicitly at times and to varying degree.

Serverless computing [5] aims to simplify cloud usage by allowing tenants to focus on their key business logic, without getting bogged down with the details of server management. This reduces management overhead compared to serverful computation. *Sky computing* [51] aims to simplify the use of multi-cloud resources, by resource scheduling via an inter-cloud broker. This reduces cross-cloud management burdens for cloud users. *Cloudless computing*, on the other hand, calls out cloud manageability as the center of attention, whether for serverless/serverful, single-cloud/sky infrastructures, throughout their deployment lifecycle.

ACKNOWLEDGMENTS

Many cloud architects, cloud engineers, and DevOps engineers have taken time to share their practices with us, especially Mark Tinderholt at Microsoft, for which we are grateful. We also thank the anonymous reviewers for their helpful feedback. This work was partially supported by NSF grants CNS-1942219, CNS-2106751, CNS-2107147, CCF-2123654, CNS-2214272, CNS-2317751, CNS-2317698, a Google PhD Fellowship, and a VMware Early Career Faculty Grant.

REFERENCES

[1] 26 cloud computing statistics, facts & trends for 2023. https://www.cloudwards.net/cloud-computing-statistics/#Sources.
[2] AKS Autoscaler. https://learn.microsoft.com/en-us/azure/aks/cluster-autoscaler#about-the-cluster-autoscaler.
[3] AWS Cloud Control API. https://aws.amazon.com/cloudcontrolapi/.
[4] AWS CloudFormation. https://aws.amazon.com/cloudformation/.
[5] AWS Lambda. https://aws.amazon.com/lambda/.
[6] Azure Bicep. https://learn.microsoft.com/en-us/azure/azure-resource-manager/bicep/.
[7] Azure Export for Terraform. https://github.com/Azure/aztfexport.
[8] Azure Monitor Activity Log. https://learn.microsoft.com/en-us/azure/azure-monitor/essentials/activity-log.
[9] Azure VM Scale Set. https://learn.microsoft.com/en-us/azure/virtual-machine-scale-sets/overview.
[10] Checkov: ship code that's secure by default. https://bridgecrew.io/checkov/.
[11] Comparing JSON and Bicep templates. https://learn.microsoft.com/en-us/azure/azure-resource-manager/bicep/compare-template-syntax.
[12] Driftctl. https://driftctl.com/.
[13] GCP Cloud Audit Logs. https://cloud.google.com/logging/docs/audit.
[14] Hashicorp State-of-the-Cloud Survey. https://www.hashicorp.com/state-of-the-cloud.
[15] HashiCorp Terraform on Azure. https://azure.microsoft.com/en-us/solutions/devops/terraform/.

[16] HCL: the HashiCorp configuration language. https://github.com/hashicorp/hcl.

[17] Infrastructure Drift and Drift Detection Explained. https://snyk.io/blog/infrastructure-drift-detection-mitigation/.

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

[19] Open Policy Agent. https://www.openpolicyagent.org/.

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

[21] Pulumi & Python. https://www.pulumi.com/docs/languages-sdks/python/.

[22] [Pulumi] AI. https://www.pulumi.com/ai/.

[23] [Pulumi] Automation API. https://www.pulumi.com/docs/using-pulumi/automation-api/.

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

[25] [Pulumi] Packages. https://www.pulumi.com/registry/.

[26] [Pulumi] pulumi stack graph. https://www.pulumi.com/docs/cli/commands/pulumi_stack_graph/.

[27] [Pulumi] Resource Providers. https://www.pulumi.com/docs/concepts/resources/providers/.

[28] RightScale 2019 State of the Cloud Report from Flexera. https://resources.flexera.com/web/media/documents/rightscale-2019-state-of-the-cloud-report-from-flexera.pdf.

[29] Sentinel integration with terraform. https://docs.hashicorp.com/sentinel/terraform.

[30] STRUCTURA's AI Assistant. https://www.structura.io/resources/build-terraform-code-using-structuras-ai-assistant.

[31] [styra] AI-Generated Infrastructure-as-Code: The Good, the Bad and the Ugly. https://www.styra.com/blog/ai-generated-infrastructure-as-code-the-good-the-bad-and-the-ugly/.

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

[33] [Terraform] Command: apply. https://developer.hashicorp.com/terraform/cli/commands/apply.

[34] [Terraform] Command: plan. https://developer.hashicorp.com/terraform/cli/commands/plan.

[35] Terraform Locking. https://developer.hashicorp.com/terraform/language/state/locking.

[36] [Terraform] Providers. https://registry.terraform.io/search/providers?namespace=hashicorp.

[37] Terraform Registry Modules. https://registry.terraform.io/browse/modules.

[38] Terraform: Resource Graph. https://developer.hashicorp.com/terraform/internals/graph.

[39] Terraform Rollback. https://developers.cloudflare.com/terraform/tutorial/revert-configuration/.

[40] Terraform validation. https://developer.hashicorp.com/terraform/cli/commands/validate.

[41] Terraformer: CLI tool to generate terraform files from existing infrastructure. https://github.com/GoogleCloudPlatform/terraformer.

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

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

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

[45] Throttling Resource Manager requests. https://learn.microsoft.com/en-us/azure/azure-resource-manager/management/request-limits-and-throttling.

[46] Tools for Infrastructure Drift Detection. https://snyk.io/blog/tools-infrastructure-drift-detection/.

[47] T. Ahmed, S. Ghosh, C. Bansal, T. Zimmermann, X. Zhang, and S. Rajmohan. Recommending root-cause and mitigation steps for cloud incidents using large language models. In *ICSE*, 2023.

[48] M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero, and D. A. Tamburri. Devops: introducing infrastructure-as-code. In *ICSE-C*, 2017.

[49] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV*, 2011.

[50] B. Berabi, J. He, V. Raychev, and M. Vechev. Tfix: Learning to fix coding errors with a text-to-text transformer. In *ICML*, 2021.

[51] S. Chasins, A. Cheung, N. Crooks, A. Ghodsi, K. Goldberg, J. E. Gonzalez, J. M. Hellerstein, M. I. Jordan, A. D. Joseph, M. W. Mahoney, A. Parameswaran, D. Patterson, R. A. Popa, K. Sen, S. Shenker, D. Song, and I. Stoica. The sky above the clouds, 2022.

[52] M. Cusumano. Cloud computing and SaaS as new computing platforms. *Communications of the ACM*, 53(4):27–29, 2010.

[53] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, 2008.

[54] J. Eberhardt, S. Steffen, V. Raychev, and M. Vechev. Unsupervised learning of api aliasing specifications. In *PLDI*, 2019.

[55] W. Fan, C. Hu, and C. Tian. Incremental graph computations: Doable and undoable. In *SIGMOD*, 2017.

[56] B. Grubic, Y. Wang, T. Petrochko, R. Yaniv, B. Jones, D. Callies, M. Clarke-Lauer, D. Kelley, S. Demetriou, K. Yu, and C. Tang. Conveyor: One-tool-fits-all continuous software deployment at Meta. In *OSDI*, 2023.

[57] Z. Guo, D. Cao, D. Tjong, J. Yang, C. Schlesinger, and N. Polikarpova. Type-directed program synthesis for restful apis. In *PLDI*, 2022.

[58] J. He, C.-C. Lee, V. Raychev, and M. Vechev. Learning to find naming issues with big code and small supervision. In *PLDI*, 2021.

[59] S. K. R. Kakarla, A. Tang, R. Beckett, K. Jayaraman, T. Millstein, Y. Tamir, and G. Varghese. Finding network misconfigurations by automatic template inference. In *NSDI*, 2020.

[60] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[61] J. Li, B. Hui, G. Qu, B. Li, J. Yang, B. Li, B. Wang, B. Qin, R. Cao, R. Geng, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *arXiv preprint*, 2023.

[62] Z. Li, Q. Cheng, K. Hsieh, Y. Dang, P. Huang, P. Singh, X. Yang, Q. Lin, Y. Wu, S. Levy, and M. Chintalapati. Gandalf: An intelligent, end-to-end analytics service for safe deployment in large-scale cloud infrastructure. In *NSDI*, 2020.

[63] J. Lloyd. Cloud foundations and landing zones. In *Infrastructure Leader's Guide to Google Cloud: Lead Your Organization's Google Cloud Adoption, Migration and Modernization Journey*, pages 239–244. Springer, 2022.

[64] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 12:559–592, 2014.

[65] F. Petrillo, P. Merle, N. Moha, and Y.-G. Guéhéneuc. Are rest apis for cloud computing well-designed? an exploratory study. In *ICSOC*, 2016.

[66] M. Santolucito, E. Zhai, R. Dhodapkar, A. Shim, and R. Piskac. Synthesizing configuration file specifications with association rule learning. In *OOPSLA*, 2017.

[67] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *SOSP*, 2011.

[68] T. Yu, Z. Li, Z. Zhang, R. Zhang, and D. Radev. Typesql: Knowledge-based type-aware neural text-to-sql generation. In *NAACL*, 2018.

[69] E. Zhai, A. Chen, R. Piskac, M. Balakrishnan, B. Tian, B. Song, and H. Zhang. Check before you change: Preventing correlated failures in service updates. In *NSDI*, 2020.

[70] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou. Encore: Exploiting system environment and correlation information for misconfiguration detection. In *ASPLOS*, 2014.