Statically Inferring Usage Bounds for Infrastructure as Code

No Author Given

No Institute Given

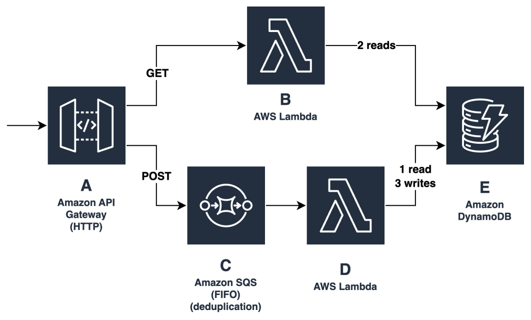
Abstract. Infrastructure as Code (IaC) has enabled cloud customers to have more agility in creating and modifying complex deployments of cloud-provisioned resources. By writing a configuration in IaC languages such as CloudFormation, users can declaratively specify their infrastructure and CloudFormation will handle the creation of the resources. However, understanding the complexity of IaC deployments has emerged as an unsolved issue. In particular, estimating the cost of an IaC deployment requires estimating the future usage and pricing models of every cloud resource in the deployment. Gaining transparency into predicted usage/costs is a leading challenge in cloud management. Existing work either relies on historical usage metrics to predict cost or on coarse-grain static analysis that ignores interactions between resources. Our key insight is that the topology of an IaC deployment imposes constraints on the usage of each resource, and we can formalize and automate the reasoning on constraints by using an SMT solver. This allows customers to have formal guarantees on the bounds of their cloud usage. We propose a tool for fine-grained static usage analysis that works by modeling the inter-resource interactions in an IaC deployment as a set of SMT constraints, and evaluate our tool on a benchmark of over 1000 real world IaC configurations.

Keywords: Infrastructure as Code, Static Analysis, Cost Estimation, FinOps

1 Introduction

One of the most pressing issues with IaC deployments is in the difficulty of estimating pricing [1]. Despite cloud providers' thorough documentation of pricing models and strong tool support, challenges remain in understanding the cost of large deployments. In a recent industry survey, it was found that 49% of IT executives find it difficult to get cloud costs under control, and 54% of those believe a primary challenge is a lack of visibility into cloud usage [1]. The issue of IaC analysis has also been recognized by the academic community to be of critical significance [3, 5, 6].

We identify two categories of existing tools for IaC cost estimation - those that rely on dynamic analysis and those that rely on static analysis. Dynamic analysis tools fall short in capturing topological changes to infrastructure, and existing static analysis tools require too much guesswork from the user.



(a) An example CloudFormation stack topology

$$A.POST + A.GET = 1000000$$
$$0 \leq w \leq 3 * A.POST$$
$$0 < r \leq A.POST + 2 * A.GET$$

(b) Constraints on usage

Fig. 1: Motivating example

AWS's Cost Explorer is one example of a dynamic analysis tool for IaC pricing. This tool allows users to track the ongoing usage and costs incurred of all resources in a live CloudFormation deployment. While dynamic analysis of IaC is helpful for existing infrastructure, such tools cannot be used for new deployments or topological modifications to existing infrastructure. Even a small change to the topology of an infrastructure might redirect user requests through a different path of the IaC resource graph, rendering past resource specific usage patterns irrelevant as topological changes induce a change in the dataflow.

AWS's Pricing Calculator, on the other hand, is one example of a static analysis tool for IaC pricing. This tool allows users to load a CloudFormation file, provide usage estimates, and see the anticipated cost of the overall infrastructure. However, estimating usage is extremely difficult and a regular pain point for customers. If the user is creating a new infrastructure, how can they estimate the usage? If there is a significant change to the topology of an existing infrastructure, how does the user know the extent to which past usage data can be extrapolated to the new infrastructure? Resource usage bounds can be also a useful analysis strategy for other system inquires, such as in the style of taint analysis, where a user may want to discover how one resource propagates data through the infrastructure, and which other resources are will see this data (have their usage impacted).

At a high level, we propose a static analysis method for IaC configuration files that assists users in making correct usage estimates. Our approach is, at its core, a modeling of the graph of the IaC and propagating local constraints across edges and nodes to be able to check global constraints. The key contributions of this work are:

1. a formal model of a subset of infrastructure-as-code that focuses on serverless architectures and message flow;
2. Cloudcap, a tool that models the resource usage of cloud infrastructures as a set of composable SMT constraints, allowing for user queries about validity of usage estimates and usage bounds of the overall IaC stack;

3. an evaluation on a benchmark dataset of over 1000 real world IaC files.

2 Motivating Example

As an illustrative example, consider the infrastructure depicted in Figure 1a. Assuming that this infrastructure is new, there will be no available historical usage data. When a user wants to estimate the cost associated with this infrastructure, they must provide estimates for the utilization of each resource. Generating these estimates requires the user to manually infer the relationship between resources, such as the correlation between (A) and (E), based on the infrastructure's topology.

For instance, if we envision 1 million requests on resource (A), then the number of read requests (r) and write requests (w) on resource (E) must adhere to the system of inequalities listed in Fig. 1b. Correctly inferring these bounds requires an understanding of how every resource propagates requests through the graph, while accounting for alterations in the quantity and types of requests. Inferring this set of constraints is a non-trivial task and requires significant experience and familiarity with all the configuration options of every resource. This process only becomes more difficult with larger and more complex IaC configurations.

3 A Formal Model of IaC

Infrastructure as Code is a powerful computational model, with many different resource types for compute, storage, load balancing, etc. In order to formalize our analysis technique, we need a formal model of IaC. Since supporting the full set of resource types for any one cloud provider is an engineering effort outside the scope of this work, we present a model of a subset of an IaC system that focuses on resource usage. This is a typical strategy used in the analysis of IaC, where a formal model captures a subset of the relevant behavior and resources for the application at hand [11].

Usually, a cloud infrastructure is comprised of multiple resources (or services) that pass messages over the network to communicate with each other. Formally, an infrastructure forms a graph, where the nodes are the resources and the directed edges are the message channels from a resource to another. The resources may send messages to and receive messages from the rest of the world, thus we can regard the "world" also as a node within the graph. Upon receiving a message over an incoming edge, a node can react by sending zero or more messages over each of its outgoing edges.

To formalize this notion of IaC, let M denote the set of all messages. In a most general setting, for every node n and for each of its outgoing edges e, there exists a function $f_e : M \to \mathbb{N}$, such that upon receiving a message m, n sends $f_e(m)$ messages over e. Given a predetermined finite set of message types MT, each message m has a message type $\mathit{mtype}(m) \in MT$. Then for all message type t and outgoing edge e, there exists a function $g_e : MT \to \mathbb{N}$ such that for all

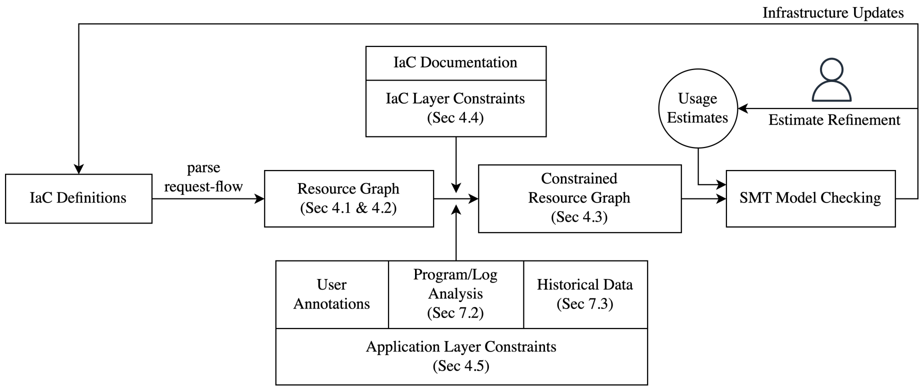

Infrastructure Updates
IaC Documentation
IaC Layer Constraints (Sec 4.4)
Usage Estimates
Estimate Refinement
IaC Definitions
parse request-flow
Resource Graph (Sec 4.1 & 4.2)
Constrained Resource Graph (Sec 4.3)
SMT Model Checking
User Annotations
Program/Log Analysis (Sec 7.2)
Historical Data (Sec 7.3)
Application Layer Constraints (Sec 4.5)

Fig. 2: System diagram

messages m of type t, $g_e(t) = f_e(m)$. This restricts each node to always send the same number of outgoing messages for each incoming message type. In other words, the resources have a predetermined finite set of message-sending patterns that do not change over time. Note that the granularity of the model depends on what one chooses the set of message types to be.

In the current version, Cloudcap focuses on AWS serverless cloud infrastructures, supporting popular resources including (but not limited to) APIGateway, DynamoDB, SQS, SNS, S3 and Lambda. The set of message types includes a generic message type `request` that represents all messages, and also additional resource-specific message types for certain resources. For example, APIGateway has additional message types that corresponds to HTTP methods (`GET`, `POST`, `PUT`, etc.); DynamoDB has `dynamodb_read` and `dynamodb_write` message types. This set of message types is generic enough to be applied to all infrastructures using the supported resources. For future work, one can also imagine that user-defined filters in the IaC code can be utilized to populate a customized set of message types for each IaC stack.

4 System Overview

As shown in Fig. 2, our system's input is an IaC configuration provided by the user. Cloudcap uses this to build a resource graph, instantiates variables that represent the resource usage measurements, and generates constraints that relate the variables and model the message-flow behaviors within the infrastructure. It is currently implemented for the AWS CloudFormation IaC platform targeting the AWS cloud platform, but the process can be similarly applied to other IaC languages (e.g. Terraform) and cloud platforms (e.g. GCP).

4.1 Building the resource graph

The major cloud platforms provide a large variety of resource types. We will use RT to denote the set of resource types.

For each resource type, they receive messages of various message types. For example, an AWS DynamoDB database can receive reads and writes, and an AWS APIGateway can receive GETs and POSTs. Following § 3, we denote MT to be the set of all message types (e.g. `dynamodb_read`), and use $\mathit{mtypes} : RT \to 2^{MT}$ to denote the assignments from a resource type to its set of message types.

From the topology defined in the IaC definitions, we can build a resource graph, where the nodes N are the resources and the directed edges $E \subseteq N \times N$ are direct message channels. We note that the resource graph is a dataflow graph instead of a dependency graph. Unlike a IaC dependency graph in which a directed edge corresponds to a partial order of deployment [11], a directed edge in the resource graph models how a resource induces requests to specific other resources. For later use, we also define the function $\mathit{rtype} : N \to RT$ that retrieves each node's resource type.

4.2 Node variables and edge variables

After building the resource graph from the dataflow analysis, we annotate the resource graph with variables. To model the quantitative details of the dataflow behaviors within the infrastructure, we instantiate *node variables* and *edge variables.*

A *node variable* represents the total count of a single message type received by a resource. The set of all node variables NV has a one-to-one correspondence to $\{(n, m) | n \in N, m \in \mathit{mtypes}(\mathit{rtype}(n))\}$. In other words, for each node, we instantiate a node variable for each message type associated with its resource type.

An *edge variable* represents the count of a message type that a node receives over a specific edge. The set of all edge variables EV has a one-to-one correspondence to $\{(s, t, m) | (s, t) \in E, m \in \mathit{mtypes}(\mathit{rtype}(t))\}$. In other words, for each edge, we instantiate an edge variable for each message type associated with the destination node.

To facilitate the constraint generation step next, for each node n_0, we let

- $NV(n_0) = \{v | v = (n, m) \in NV, n = n_0\}$ denote the set of node variables for n_0;
- $EV_{in}(n_0) = \{v | v = (s, t, m) \in EV, t = n_0\}$ denote the set of *incoming edge variables*, i.e. the edge variables for which n_0 is the source node;
- $EV_{out}(n_0) = \{v | v = (s, t, m) \in EV, s = n_0\}$ denote the set of *outgoing edge variables*, i.e. the edge variables for which n_0 is the destination node.

4.3 Constraint generation

A constraint ϕ is an SMT formula such that $FV(\phi) \subseteq NV \cup EV$, where $FV(\phi)$ denotes the set of variables that occur free in ϕ. For each variable $v \in NV \cup EV$, there may be zero or more *basic* constraints that describe basic properties of the variables. At the moment, all variables have just the basic constraints that they are greater or equal to 0.

Next, for each node $n \in N$, the tool generates zero or more constraints. Each of these constraints $\phi \in \mathcal{C}$ is categorized as one of the following:

1. **incoming**: for a node variable $nv \in NV(n)$, the constraint ϕ relates nv to the incoming edge variables, i.e. $FV(\phi) \subseteq \{nv\} \cup EV_{in}(n)$
2. **intrinsic**: the constraint ϕ relates node n's node variables to each other, i.e. $FV(\phi) \subseteq NV(n)$
3. **outgoing**: for an outgoing edge variable $ev \in EV_{out}(n)$, the constraint ϕ relates ev to its node variables, i.e. $FV(\phi) \subseteq NV(n) \cup \{ev\}$.

Intrinsic constraints capture intra-resource behavior. For example, in the motivating example Fig. 1a, all incoming requests to (A) should be equal to the sum of the POST and GET requests to (A). Written as an SMT constraint, `A.requests` $=$ `A.GETs` $+$ `A.POSTs`.

Incoming and outgoing constraints capture inter-resource behavior. For the motivating example from § 2, the number of POST requests to (A) should equal the number of requests to (C). Written as an SMT constraint, the outgoing constraint from A to C is `A.GETs` $=$ `A_C.requests`. These are more difficult to derive as they can arise from both the IaC level configuration as well as the application layer (discussed in § 4.4 and § 4.5).

These three types of constraints allow the tool to model the system globally with only node-local knowledge. When generating constraints for a node, there is no need for graph traversal of any kind. The global set of constraints can be collected by simply iterating through the nodes.

4.4 IaC Layer Constraints

For many modern cloud serverless resources (e.g. AWS SQS and APIGateway), Cloudcap is able to generate all the necessary constraints using the information within the CloudFormation templates. Take our motivating example from § 2. The node (C) is an SQS service with FIFO and deduplication enabled. By consulting the AWS documentation, we know that with these configurations, the requests that come out of (C) (and in this graph are sent to (D)) will be less than or equal to the requests sent to SQS. This induces an outgoing constraint `C.requests` $>=$ `C_D.requests`. This constraint is an *IaC layer* constraint and can be derived purely from the IaC code. Every cloud resource is given a SMT constraint template, as manually derived from the documentation, that is used to generate these constraints.

4.5 Application Layer Constraints

For cloud resources with programmable behaviors, additional knowledge about the application program allows us to generate constraints that are not discoverable with just the IaC definitions. We call these *application layer* constraints. For example, in the motivating example, we have a constraint on the edge from (D) to (E) that the Lambda will induce 1 read and 3 writes to the DynamoDB (E) for every request to the Lambda. However, the information needed to generate such a constraint is beyond the scope of the IaC configuration, and thus Cloudcap is currently not able to infer application layer constraints. Cloudcap instead relies on user-provided custom SMT constraints for application layer constraints. In § 7.1, we discuss some approaches one may take to automate the inference of application layer constraints.

5 Evaluation

Methodology We draw our benchmark set from the PIPr dataset of public IaC programs [15]. PIPr contains 7104 public repositories of Programming Languages Infrastructure as Code (PL-IaC) projects written with Pulumi, AWS CDK or Terraform. With automated scripts (provided in our GitHub repository[1]), we identify the AWS CDK projects implemented in JavaScript or TypeScript, and run '`npm install; cdk synth`' in each project to synthesize a CloudFormation template for each. The result is our benchmark set containing 1062 CloudFormation templates.

5.1 Quantitative Analysis

Cloudcap currently supports 12 AWS CloudFormation resource types. The tool also flags 22 resource types as non-dataflow related and omits these in the construction of the resource graph. A resource type is non-dataflow related if it is not an infrastructure resource (for instance, IAM policies and Lambda permissions). Each supported resource type typically has between 1 to 3 message types.

As a tool designed to be integrated into an existing user workflow, Cloudcap completes in a short amount of time, spending only 0.5 ± 0.15 seconds for all samples on a 32GB M1 Max MacBook Pro. As a proxy measure of complexity, we looked at the 457 templates that have at least 3 supported resource nodes, and found that the average graph degree (i.e. number of edges directed into a node) is 0.9 with a standard deviation of 0.6. This means that IaC resource graphs are usually very sparse and thus gives us confidence that Cloudcap will scale well, even with larger IaC configurations. Combined with the finding that the number of resources is usually small (benchmark set averaging 21.9 with standard deviation of 21.1), we don't expect performance to be an issue even on IaC configurations beyond our benchmark set.

We ranked the samples by the number of constraints generated by Cloudcap, and show the top 100 samples in Fig. 3. Each bar is also broken down into the four constraint types: basic, incoming, intrinsic and outgoing. This chart shows that a small number of IaC configurations have significant complexity (benchmark #1 in Fig. 3 has 529 constraints), and there is a long tail of IaC configurations

[1] https://github.com/anonymized/anonymized

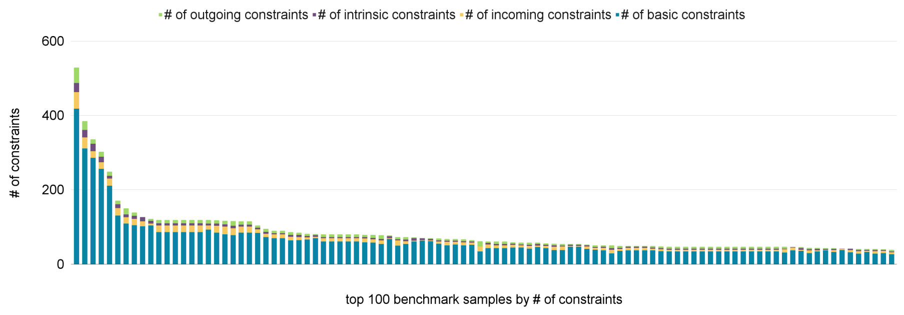


Fig. 3: Number of constraints for the benchmark samples (top 100).

that still have enough complexity such that there is value in automating this reasoning task (benchmark #100 in Fig. 3 has 39 constraints).

5.2 Sample Benchmark

To concretely demonstrate the workflow of Cloudcap, we walk through an illustrative example of a developer tasked with implementing and deploying a simple project. The project is one of our benchmarks from the PIPr dataset (ID 501027421), and is a public GitHub repository named "HektorCyC/url-shortener-app". Written in TypeScript and AWS CDK, the project is a URL shortener application, in the form of a chain from APIGateway to Lambda to DynamoDB. When run through Cloudcap, this IaC configuration has 28 constraints (ranking #138 in the benchmarks, sorted by number of constraints).

After coding up the project, the developer synthesizes a CloudFormation template named `cfn.yaml` and runs `cloudcap estimates-template cfn.yaml`. This command provides them with a template for usage estimates. After being filled in, the template looks as following:

```
apigateway:
  GETs: 100000000
  PATCHs: 100000000
  DELETEs: 100000000
dynamodb_table:
  dynamodb_reads: 3000000
```

The Lambda application receives GET, PATCH and DELETE requests from the APIGateway. For a GET request, it reads the DynamoDB once; for a PATCH or DELETE request, it reads once and writes once to the DynamoDB. The developer writes these application layer constraints as custom SMT-LIB constraints in a file named `app_constraints.smt2`, and runs `cloudcap check cfn.yaml app_constraints.smt2`. In the output, the tool reports that the user-provided estimates are invalid. The developer reviews the estimates again, discovering that they made a mistake and left out some zeros in the `dynamodb_read` usage estimate.

6 Related Work

Our work is related to cost analysis of cloud deployments and workload characteriztion. Workload characterization has been an active research area with a focus on understanding resource utilization, performance, and cost implications [4]. Traditional approaches to workload characterization often involve profiling applications or services to understand their resource requirements and performance characteristics. These studies typically rely on statistical analysis of historical data [2, 16]. Some existing works aim at dynamic cost estimation by considering real-time resource utilization metrics and billing information. These approaches often leverage machine learning techniques to predict costs based on historical usage patterns [7, 8, 14]. However, they struggle to capture the intricate relationships and dependencies between resources defined in IaC deployments. Recent research has explored topology-based cost models for understanding and optimizing cloud deployments [10, 12, 13]. These approaches consider the arrangement and relationships between different components in an IaC deployment. However, they do not explicitly model the constraints imposed by the deployment topology and the component characteristics on resource usage. Our proposed system instead builds a topology-based resource graph to enforce constraints inherent in the component (intrinsic contraints) and the relationship between components (incoming and outgoing constraints), providing end users with formal guarantees on usage bounds.

Unlike existing approaches that predominantly rely on accurate usage estimates to provide cost estimation in cloud deployments, our system focuses on evaluating the feasibility of such estimates by modeling the intricate relationships between infrastructure components in the form of a resource graph. By employing SMT constraints to encapsulate these inter-resource interactions, our approach allows for a more rigorous examination of the deployment's constraints, informing end users whether the estimated resource usage aligns with the inherent limitations and relationships defined within the infrastructure components. This novel perspective does not only enable existing approaches to predict costs more accurately but also to ascertain the validity and appropriateness of initial usage estimates given the constraints imposed by the deployment's topology.

7 Discussion

A limitation of the evaluation is the lack of usage estimates in the benchmark. The PIPr dataset provides sufficient IaC definition samples, but does not provide corresponding resource usage estimates. Therefore, Cloudcap is not benchmarked in its capability to check them against the generated constraints. In fact, usage estimates are highly proprietary data, so they are generally not publicly available. However, we don't expect validating the usage estimates against the generated constraints to be an expensive problem.

A few software engineering issues also still need to be resolved for the tool to be adopted in industrial workflows. Often, especially in enterprise environments, IaC deployments are modularized. Developing an application from multiple stacks is a common and even best practice in modern IT operations [9]. However, Cloudcap requires a full view of the application's architecture to provide a comprehensive set of constraints. One workaround is to create a merged template strictly for running cost estimates. Another is to constrain individual templates with Cloudcap and manually supply constraints that would connect the models together. As a novel approach to cloud resource usage analysis, a user study would help to find the to most effective ways to integrate the tool into existing usage analysis workflows.

7.1 Application Layer Constraints from Program Analysis

For resources with programmable behaviors (e.g. AWS Lambda), the IaC definitions are usually paired with the application source code. To extract the application layer constraints, an option is to run symbolic execution on the application source code. This might be tractable if the application is a relatively small code snippet, but will be more difficult if the program logic is complex. In the case that code analysis is not feasible, if we are updating an existing infrastructure and have log data (e.g. from the AWS Cost Explorer), we can infer some relation between the number of requests from one resource to the next. In this case, we can check the logs and see the timestamped relations between incoming requests on the neighboring nodes. The local relations on usage can be more reliably pulled from log data than global constraints, as the global relations depend on the topology of the infrastructure.

In the current version, Cloudcap focuses on the Iac layer and allows user to submit the application layer constraints as custom SMT-LIB assertions. This allows the tool to remain useful for infrastructures that utilize unsupported resources. Take the motivating example. If we replace resource (B) with an EC2 instance, Cloudcap cannot generate estimates for the entire architecture because that would require understanding the application runtime behavior. However, the user may understand that behavior and can insert corresponding constraints. Even in this case, the tool still greatly reduces the manual work and room for human error by taking care of the rest of the architecture.

7.2 Reusing Historical Usage Data

The static analysis and usage estimate procedure described above gives user the ability to put constraints on their infrastructure usage. However, for updates to existing infrastructure, historical usage may be available and valuable. We can incorporate this valuable data into our procedure by identifying subgraphs of the infrastructure that are minimally impacted by the IaC topological changes.

References

1. Anodot: 2022 state of cloud cost report (2022), https://go.anodot.com/2022-cloud-cost-report
2. Azmandian, F., Moffie, M., Dy, J.G., Aslam, J.A., Kaeli, D.R.: Workload characterization at the virtualization layer. In: 2011 IEEE 19th annual international symposium on modelling, analysis, and simulation of computer and telecommunication systems. pp. 63–72. IEEE (2011)
3. Böhme, L., Beckmann, T., Baltes, S., Hirschfeld, R.: A penny a function: Towards cost transparent cloud programming. In: Proceedings of the 2nd ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments. pp. 1–10 (2023)
4. Calzarossa, M.C., Massari, L., Tessera, D.: Workload characterization: A survey revisited. ACM Computing Surveys (CSUR) **48**(3), 1–43 (2016)
5. Cito, J., Leitner, P., Fritz, T., Gall, H.C.: The making of cloud applications: An empirical study on software development for the cloud. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 393–403 (2015)
6. Cito, J., Piskac, R., Santolucito, M., Zaidman, A., Sokolowski, D.: Resilient Software Configuration and Infrastructure Code Analysis (Dagstuhl Seminar 23082). Dagstuhl Reports **13**(2) (2023). https://doi.org/10.4230/DagRep.13.2.163
7. Dezhabad, N., Ganti, S., Shoja, G.: Cloud workload characterization and profiling for resource allocation. In: 2019 IEEE 8th international conference on cloud networking (CloudNet). pp. 1–4. IEEE (2019)
8. Gao, J., Wang, H., Shen, H.: Machine learning based workload prediction in cloud computing. In: 2020 29th international conference on computer communications and networks (ICCCN). pp. 1–9. IEEE (2020)
9. Koskelin, J.: Modular Infrastructure as Code in Azure PaaS. Master's thesis, University of Helsinski (2023)
10. Leitner, P., Cito, J., Stöckli, E.: Modelling and managing deployment costs of microservice-based cloud applications. In: Proceedings of the 9th International Conference on Utility and Cloud Computing. pp. 165–174 (2016)
11. Lepiller, J., Piskac, R., Schäf, M., Santolucito, M.: Analyzing infrastructure as code to prevent intra-update sniping vulnerabilities. In: Tools and Algorithms for the Construction and Analysis of Systems: 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27–April 1, 2021, Proceedings, Part II 27. pp. 105–123. Springer (2021)
12. Lin, C., Mahmoudi, N., Fan, C., Khazaei, H.: Fine-grained performance and cost modeling and optimization for faas applications. IEEE Transactions on Parallel and Distributed Systems **34**(1), 180–194 (2022)
13. Obetz, M., Patterson, S., Milanova, A.: Static call graph construction in {AWS} lambda serverless applications. In: 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19) (2019)
14. Saxena, D., Kumar, J., Singh, A.K., Schmid, S.: Performance analysis of machine learning centered workload prediction models for cloud. IEEE Transactions on Parallel and Distributed Systems **34**(4), 1313–1330 (2023)
15. Sokolowski, D., Spielmann, D., Salvaneschi, G.: Pipr: A dataset of public infrastructure as code programs (Nov 2023). https://doi.org/10.5281/ZENODO.8262770, https://zenodo.org/doi/10.5281/zenodo.8262770
16. Wolski, R., Brevik, J.: Using parametric models to represent private cloud workloads. IEEE Transactions on Services Computing **7**(4), 714–725 (2013)