Toggle Blog Navigation

Automating Pulumi Import with Manually Created Resources

Posted on Monday, Jan 30, 2023

Josh Kodroff

A few weeks ago, I was speaking with a consultant at one of the big firms who asked me how he could introduce Pulumi into a client's organization when that client had created many infrastructure resources manually through the AWS console and was running production workloads on those resources.

Introducing modern cloud infrastructure tooling and automation is relatively simple (or at least more straightforward) when organizations decide to adopt IaC from the start of their cloud journey, but what about organizations who have gone far enough down the route of manually created cloud infrastructure to see the perils of that approach? Many teams come to this realization only when they've deployed too many production workloads to start over from scratch. If your organization is looking at Pulumi as an IaC solution, it's worth bringing these resources under management because of the low effort and high value of having a single pane of glass to manage all of your resources.

Bringing resources under IaC (and Pulumi specifically) gives an organization an immediate return on investment. Note that while we list some benefits specific to the Pulumi Service, Pulumi supports a number of backends for both state management and secrets management and the Pulumi Service is not required in order to adopt Pu

- **Version control/change history:** With IaC placed under v easily roll back to previous versions of infrastructure, see when, which provides critical visibility into how infrastruc The Pulumi Service provides a great experience here, wi version history and deployment logs along with git contextual data in a single pane of glass.

- **Automation and pipelines:** Because IaC is code, it enjoys the same main benefits from automated pipelines as application code: a repeatable and reliable process. We can also enjoy significant security benefits: When IaC and automated pipelines become the standard way to deploy infrastructure, organizations can significantly reduce the number of users with direct access to critical infrastructure (i.e., via the cloud console or CLI) and the level of privileges needed (e.g., read-only rather than full admin). Once we get our pipeline established so that we have repeatable automated delivery of our infrastructure, we can add automated quality assurance to Pulumi programs via unit testing and policy as code.
- **Drift detection and correction:** Most IaC tools will detect changes made out of band to resources, e.g., changes made manually in the console, and attempt to reconcile the resource back to its declared configuration. Thus, by bringing manually created resources under IaC, organizations can put a definitive end to the slow and error-prone practice of manually managing cloud resources. In Pulumi, the `pulumi refresh` command will refresh all resources in the state file. For more information on strategies to address configuration drift, check out Patterns of Drift Detection with Pulumi in the Pulumi blog.

About Pulumi Import

The `pulumi import` command is executed within the context of a Pulumi stack. The user supplies the type of resource to be imported, a name with which to identify the resource in the Pulumi program and state file, and a resource-dependent ID. For example, we might run the following command to import an AWS VPC (this command is sourced from the API docs for the AWS Classic provider in the Pulumi Registry):

```
pulumi import aws:ec2/vpc:Vpc imported-vpc vpc-0b0a6ad0766eccf3b
```

When we run the command, Pulumi queries the AWS API for the VPC's attributes. The resource's attributes are added to the stack's state file and the command prints out code to be added to our Pulumi program:

```
$ pulumi import aws:ec2/vpc:Vpc imported-vpc vpc-0b0a6ad0766eccf3b
Previewing import (dev)

View Live: https://app.pulumi.com/jkodrofftest/pulumi-import-blog-sample/dev/pr

Type                     Name                         Plan
 +   pulumi:pulumi:Stack  pulumi-import-blog-sample-dev  create
 =   └─ aws:ec2:Vpc      imported-vpc                 import
```

```
Resources:
  + 1 to create
  = 1 to import
  2 changes

Do you want to perform this import? yes
Importing (dev)

View Live: https://app.pulumi.com/jkodrofftest/pulumi-import-blog-sample/dev/up

Type                          Name                           Status
 +   pulumi:pulumi:Stack      pulumi-import-blog-sample-dev  created (4s)
 =   └─ aws:ec2:Vpc           imported-vpc                   imported (3s)

Resources:
  + 1 created
  = 1 imported
  2 changes

Duration: 5s

Please copy the following code into your Pulumi application. Not doing so
will cause Pulumi to report that an update will happen on the next update comma

Please note that the imported resources are marked as protected. To destroy the
you will need to remove the `protect` option and run `pulumi update` *before*
the destroy will take effect.

import * as pulumi from "@pulumi/pulumi";
import * as aws from "@pulumi/aws";

const imported_vpc = new aws.ec2.Vpc("imported-vpc", {
  cidrBlock: "10.0.0.0/16",
  tags: {
      Name: "vpc",
  },
}, {
  protect: true,
});
```

Note that in the preceding example we have TypeScript output, but `pulumi import` will automatically detect the language our Pulumi project uses and will output code in the correct language.

Our resource also has the `protect: true` option specified by default, which means that Pulumi will not delete (or recreate) the resource if the program changes. This is an overridable default setting chosen for safety: If a resource is being imported in the first place, it's typically because there's an important workload using it, and we should be very careful before deleting or re-creating it.

Now that we've run the `pulumi import` command and added the code that the command outputs, our resource can now be managed by Pulumi. If we run the `pulumi preview` command, we will see that our resource has been added to our stack and that no changes have been detected, meaning that Pulumi has correctly imported all of our VPC's attributes:

```
$ pulumi preview
Previewing update (dev)

View Live: https://app.pulumi.com/jkodrofftest/pulumi-import-blog-sample/dev/pr

  Type                  Name                              Plan
  pulumi:pulumi:Stack   pulumi-import-blog-sample-dev

Resources:
  2 unchanged
```

Automated Batch Import Capabilities

For projects with many resources, `pulumi import` also has an option that allows it to take a list of resources in a specially formatted JSON file for batch import. We'll be using this batch import capability to add our manually-created resources into the Pulumi state. A basic sample JSON input follows:

```
{
 "resources": [
   {
     "type": "aws:ec2/vpc:Vpc",
     "name": "my-vpc",
     "id": "vpc-094958e4051c478c3"
    },
      {
        "type": "aws:ec2/vpc:Vpc",
        "name": "my-other-vpc",
        "id": "vpc-0f12a82357335a28f"
      }
    ]
  }
```

Because `pulumi import` gives us the full code for an imported resource with all attributes (as demonstrated in the previous section), and because `pulumi import` has an option for batch import from an external file source (both key differentiators between Pulumi and other IaC tools) we can use `pulumi import` to import existing resources at scale in an automated fashion.

> Note
>
> If you are looking to import resources that were created with a non-Pulumi IaC tool as opposed to manually in the console, the approach described in this article will still work, but you may also want to consider the tools tf2pulumi and cf2pulumi for resources created with Terraform and CloudFormation, respectively. For a comprehensive guide on the various options for importing existing resources into Pulumi, see Migrating to Pulumi.

Designing an End-to-End Solution

Now that we've explained the benefits of bringing manually created resources under IaC, and the capabilities of the `pulumi import` command, we need to design a solution that will allow us to leverage `pulumi import` 's batch import capabilities so that we can bring all resources in our cloud environment under Pulumi management.

Our solution for an automated batch import of resources into Pulumi comprises the following steps:

1. Create an account scraper using our cloud provider's SDK (or CLI) to query for the resources to be imported.
2. Take the output from the cloud provider's SDK and transform it into a JSON file suitable as an import for a batch `pulumi import` operation.
3. Run `pulumi import` to bring the resources under Pulumi control.

Writing an Account Scraper

In our account scraper, we will use the AWS Python SDK, boto3, to query resources created in the AWS console, but a similar approach can be applied to the many cloud and SaaS providers supported by Pulumi. The full code for the sample solution can be found on GitHub at https://github.com/pulumi/pulumi-import-aws-account-scraper. If you find the tool useful, please submit issues and/or PRs to support additional resource types!

Because the ID attribute (or multiple attributes in the case of some resources) vary based on the resource type (in AWS at least, your cloud provider may vary), we need to write custom code for each resource type that we want to import. (Note that the account scraper code only queries a subset of AWS resource types, but the code can be easily modified to accommodate additional resource types.) For example, the following code shows a query for associations between route tables and VPC subnets:

```python
def import_route_table_associations(ec2_client):
    pulumi_resources = []

    route_tables = ec2_client.describe_route_tables()['RouteTables']

    for route_table in route_tables:
        for association in route_table["Associations"]:
            if 'SubnetId' not in association:
                continue
            pulumi_resources.append({
                "type": "aws:ec2/routeTableAssociation:RouteTableAssociation",
                "name": f"import-{association['RouteTableAssociationId']}",
                "id": f"{association['SubnetId']}/{route_table['RouteTableId']}"
            })

    return pulumi_resources
```

We can run the entire account scraper program to output our resources to a single JSON

```
python account_scraper.py > /path/to/file.json
```

Batch Import

Now that we've generated our JSON file, we can use the `pulumi import` command to import all of our resources in a single command and output our code to its own file:

```
pulumi import -f /path/to/file.json -o imported-resources.ts -y
```

And with that, our resources are now managed by Pulumi!

Next Steps

Now that we have our resources under Pulumi management, we might want to consider some additional steps to build upon our IaC solution:

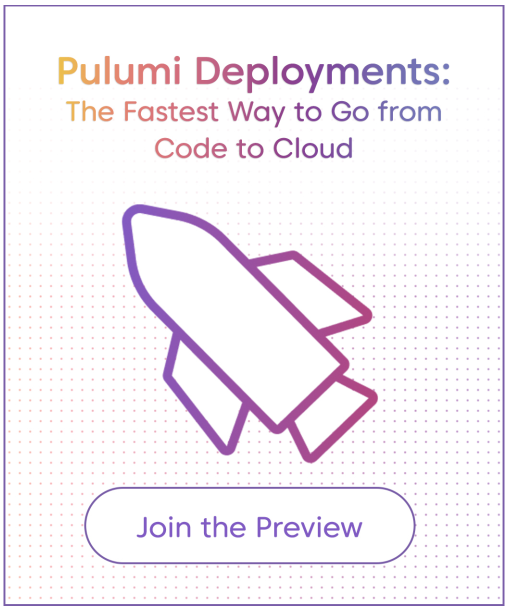
- **Set up a CI/CD pipeline:** By adding our Pulumi code to a delivery pipeline, we can help ensure that all infrastructure changes go through an automated process. Pulumi has helpful integrations and documentation for many popular tools. See Continuous Delivery in the Pulumi docs for more information.
- **Add policy as code:** By adding policy as code to our IaC pipeline, we can ensure that our infrastructure is in compliance with any security or regulatory requirements before it's ever provisioned in the cloud, as well as checks on any existing Pulumi resources. For more information on Pulumi's policy as code capabilities, see CrossGuard (Policy as Code) in the Pulumi docs.

Conclusion

We've shown how `pulumi import` enables organizations to quickly move from slow and error prone manual management of cloud resources to a modern, automated, code-centric approach, even when we need to preserve those manually created resources because they are running critical workloads. It's never too late for an organization to transition to managing infrastructure resources as code, and Pulumi makes this transition as painless as possible!

aws import

Share this post



Get Started

Install

Documentation

Registry

Public Roadmap

Security

Enterprise

AWS

Azure

Google Cloud

Containers

Serverless

Kubernetes

About Us

Request a Demo

Resources

Slack Archive

Contact Us

Support

Careers

Case Studies

Awards & Recognitions

Brand Resources

Trademark Usage

Acceptable Use Policy

Terms & Conditions

Privacy Policy

Professional Services Agreement