canarie

Technical Guide:

Introduction to Cloud Independence

Contents

Introduction

Public clouds such as Amazon AWS, Google Cloud, and Microsoft Azure are increasingly impacting how IT decision-makers evaluate their application design and deployment options. They offer a wide range of services that accelerate common tasks such as the deployment of web applications, and remove costly maintenance and outage risks with flexible and dynamic infrastructure.

However, this acceleration and flexibility comes at a cost. Many of these services use technologies, APIs, and interfaces that are unique to the cloud provider. This makes them incompatible with similar offerings from other cloud providers, increasing the risk of your project being locked into a single cloud provider.

Cloud independence is a method of designing your technology solution to be cloud agnostic so that it's not locked to a single provider. By designing with cloud independence in mind, you can make informed decisions about when to use a cloud provider's unique features and when to opt for a more generic or portable solution. Your projects will still be able to leverage the benefits of a public cloud but will remain flexible enough to easily migrate to a different provider in the future.

Topics Covered

This tutorial will take you step-by-step through the process of designing a modern web application infrastructure hosted in the cloud. We won't dive too deeply into the application itself, but you will learn about the different services available in public clouds, their advantages, disadvantages, how to decide which services to use, and when to manage a component yourself.

The tutorial will introduce you to an example web application (a demo online polling system) and the cloud infrastructure components that make it up. We'll compare leveraging a public cloud provider's service for these components versus managing them yourself.

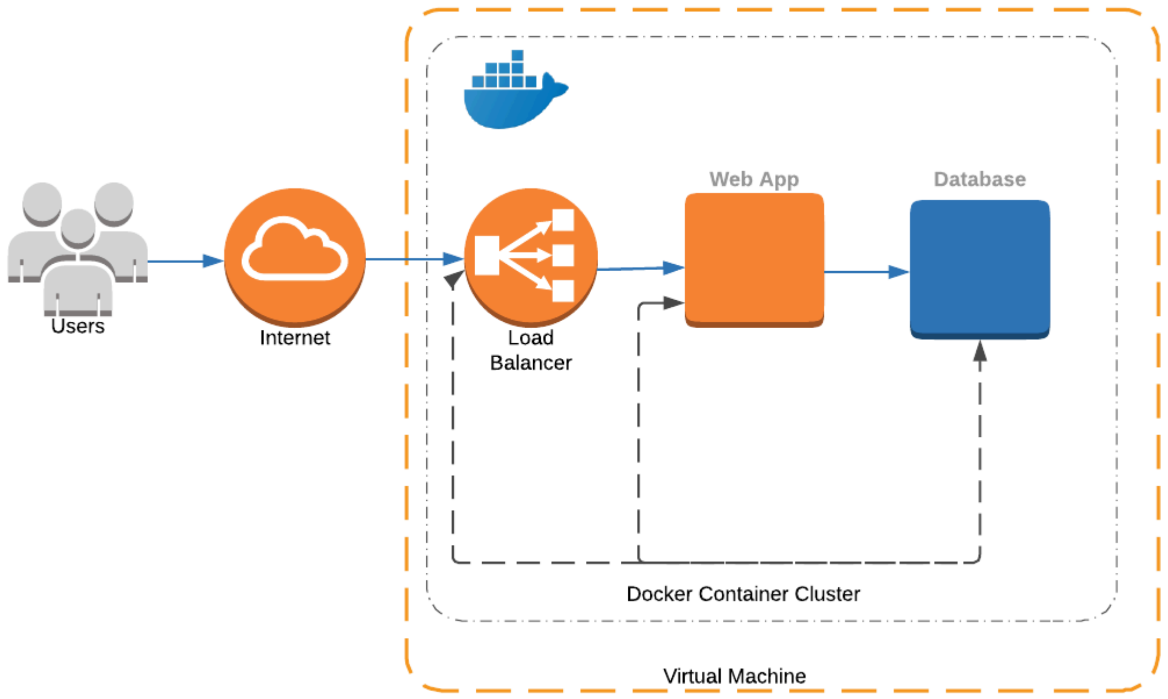
What You'll Need

- Some cloud and app building experience
- Credentials for a cloud provider – e.g. DAIR Cloud, OpenStack, Microsoft Azure, etc. (details in the example app's repository readme)
- A Docker Hub account
- Docker (download from your Docker Hub account)
- Git
- Terraform

Components of a Basic Web App

The example web application you will be deploying is a clone of the Django Tutorial application. **Django** is a popular python-based web application framework (learn more about Django Project here).

Let's first take a look at the high-level architecture that you will be building towards through this tutorial.

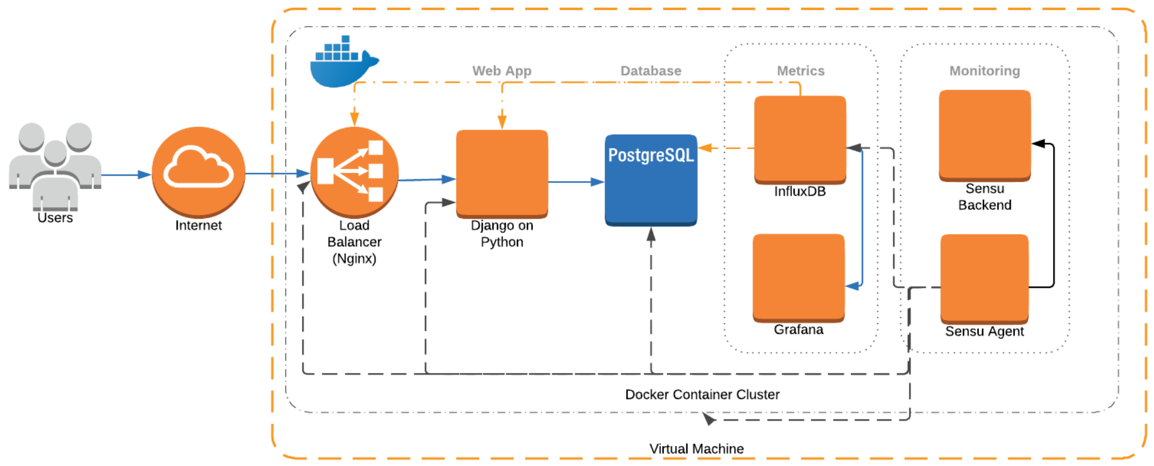


You can see that our example solution is made up of the following three main components:

- **Load Balancer:** Accepts incoming network traffic to distribute among application servers.
- **Web Application:** Running code that users interact with through a web browser.
- **Database:** A service that stores data in a reliable and persistent manner.

Most public cloud providers have variations of each of these components. You can also use an open-source solution that is agnostic to any public cloud or a mix of both.

For example, if you wanted to deploy a python-based web application that uses a Postgres database with an open-source load balancer such as NGINX, the end result with monitoring and metrics collection may look like this:



You could decide to leverage both public cloud services and cloud independent components by swapping one or more of the services listed above with a specific public cloud offering. The diagram stays the same, but with a different block replacing one or more components. We will go over a few of those options below.

Web Application

The first component to look at is the example web application written in Python. Python is a robust and easy to use language with a lot of flexibility in how it can be hosted in the cloud. Our example application deployed in a Docker container will listen for client connections on port 8000.

Docker allows you to easily pack, ship, and run any application as a tiny and portable container that can run almost anywhere. Millions of applications have adopted Docker since its release in 2014, so you'll be able to find lots of help, documentation, and support if needed. All major cloud providers support a wide array of Docker-compatible services; thus you'll be able to easily move your “containerized” application from one provider to another.

One advantage of Docker containers is that there are a variety of ways to host them in the cloud. The most basic and flexible way is to create a **virtual machine (VM) instance** and manually install Docker within the instance to be able to host container applications. This method will work across every cloud provider but it requires considerable systems administration knowledge as well as a large time investment in maintenance throughout the application's lifecycle.

To make management of Docker easier, cloud providers have Docker-enabled services such as **AWS Fargate** and **Azure Container Instances**.

All of these solutions fall in different areas of the spectrum as shown below. Generally, the more managed a solution is the more expensive and rigid it is in terms of implementation; lower maintenance can, however, compensate in some situations.

Fully Managed	**Partially Managed**	**Complete Control**
Google App Engine	AWS Fargate	Docker on Instance
Heroku	Azure Containers	
Serverless (AWS Lamba, AWS Functions)		

Running the container cluster in a **container orchestration service**, such as AWS Fargate or Azure Containers, will be more expensive but can result in less maintenance work for IT staff. This is because the service provider takes care of the underlying container stack, along with the management of the container cluster and associated VMs. There is a risk container templates for one cloud provider can't easily be ported to another provider, so be prepared when choosing your solution.

Load Balancer

Having users access the voting application directly is possible, but a better long-term solution is to place a load balancer in front of it. Interacting with the voting application by way of the load balancer provides scaling assistance down the line, and immediately provides a buffer between the internet at large and your application. This network separation provides an additional security layer over a direct connection as NGINX (or similar). Load balancers have a hardened network layer typically not found by default in most web application.

Load balancers can operate in two modes: Layer 4 or Layer 7.

Layer 4 load balancers work at the **Transport Layer,** which deals with the delivery of network traffic, regardless of the content.

Layer 7 load balancers operate at the **Application Layer,** which deals with the actual content of the network traffic and can allow more intelligent decisions about where the traffic should go.

For example, with a Layer 7 load balancer requests for a specific page can go to a specific web application server. A Layer 4 load balancer meanwhile will redirect entire requests to application servers without concern of the request itself. Both load balancers offer the ability to monitor which application servers are best to send requests to.

Most load balancing options from cloud vendors are focused on Layer 4 balancing (e.g. AWS Elastic Load Balancer, Azure Load Balancer), but recently, Layer 7 load balancing has become more widespread and readily available (e.g. AWS App Mesh using Envoy, Azure Application Gateway) on top of DNS-based balancing options (e.g. AWS Route 53, Azure Traffic Manager).

Choices in open source and self-managed load balancers include HAProxy, NGINX, and Envoy.

As will be repeated throughout this tutorial, the value tradeoff between leveraging and paying for a vendor's service compared to managing your own will be dependent on your team's expertise and time, and its importance to your goal. A few thoughts on fully managed services vs. self-managed services are below:

Fully Managed Service	**Self-Managed**
Simplified design and improved application maintainability	Increased complexity but higher degree of control
Cost savings over hiring skilled IT professionals	Cost savings if skilled workers already in-house
Speed implementation but cloud vendor lock-in	Slower to implement but portable to other clouds
Leverage built-in high-availability (HA) or networking services, which translates to more robust and reliable solution	Typically requires more investment to develop and test; HA ready for scale in production

For this tutorial, we'll be using the self-managed, open source NGINX Docker container, which can be used as both a Layer 4 *and* Layer 7 load balancer. For simplicity, we'll be using it strictly like a Layer 4 load balancer. The NGINX Docker container is easy to use and provides an easy way to save costs in both operational effort and network bandwidth fees. In addition, we will be able to move our configuration to a different cloud provider proving its portability.

By choosing the open source, NGINX Docker container, we lose the ability to leverage cloud provider-specific networking features such as integrated high availability and identity access controls.

Database

In order for the voting application to track all the votes that users cast, the voting data needs to be stored in a database. Just like with the web application and load balancer, you can choose to manage a database yourself or use a database service provided by a public cloud provider.

Running your own database gives you flexibility and control over your database deployment but it can prove to be a large task with respect to administration and maintenance. Issues with performance, resilience, disk space, backups, or access can take up a great deal of time better spent working on your project. If your database isn't maintained properly, it can create significant problems for the responsiveness of your application, can result in application failure or outage, or make it a vulnerable target for data breaches.

Cloud providers offer a variety of "production grade" **Database as a Service (DBaaS)** options, with different levels of management and additional high availability or resilient features. For example, Amazon Web Services offers most standard database backends as part of their **Relational Database Service (RDS)**, including their own system, **Aurora**. Both offer similar features but Aurora is exclusive to Amazon and boasts significant performance improvements over comparable counterparts.

Database pricing with cloud vendors is also complicated. Cost planning involves usage charges for storage, bandwidth, quantity of queries, data exports, and more. Additionally, while leveraging a cloud provider's database service will save you the time and effort required to manage a database, you run the risk of locking your data into a single cloud vendor's solution.

Retaining data ownership should be a priority, so this tutorial will focus on managing our own database service. Data sovereignty is another important consideration for many and if so, you need to be aware of how to ensure your data resides exclusively in a Canadian datacentre. For this tutorial we will use a PostgreSQL Docker container. Just like with the web application and load balancing choices, using a Docker container will ensure we can easily move these components to any other public cloud. Below is a brief summary of fully managed vs. self-managed database solutions.

Fully Managed Service	**Self-Managed DB**
– Could be proprietary and result in vendor lock-in	– Direct control and shell access to DB
– Amazon Aurora is a native HA solution with optimized multi-region failover and replication.	– Can be used for specific formats or storage engines that are not supported by a fully managed service. For example, Amazon Aurora only supports innoDB
– Fully dependent on cloud provider for bug fixes and upgrades	– Time and resource(s) required to maintain DB and OS patching/upgrades
– Does not require a dedicated DBA	– Requires a DBA
– Automatic backups and scaling – Continuous backup to object storage with no performance impact	– Requires extra effort and planning to handle backups, logging, scaling for growth and

– Eliminates need for backup windows – No need for capacity planning in advance	performance monitoring
– Little to no control over data sovereignty	– Full control over data sovereignty

Cloud Independent Tools

In the previous section we reviewed the components of our cloud-enabled web application in detail. We discussed solutions available from public cloud providers, solutions that you can manage on your own, and the trade-offs of each.

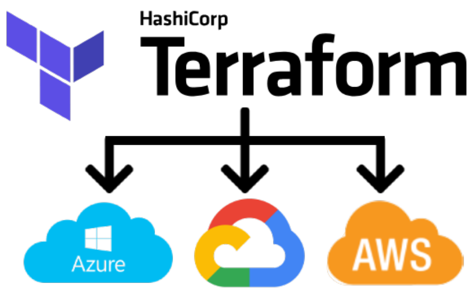
While understanding these components is important to successfully creating your solution, you also need to understand *how* to create and deploy your solution. To do that we will go over the different tools available to assist in creating, deploying, and managing your solution.

A common theme for all of our tutorials will be the idea of Infrastructure as Code. With infrastructure as code, the creation, management, and operations of our example web application infrastructure is all going to be defined as some sort of *code* along with the application itself. When the entire solution is defined as code, we can use our tools to read and execute the code resulting in the same predictable and reliable deployment outcome each time. Contrast this to deploying our example web application and infrastructure through a series of keystrokes and mouse clicks; there could be typos and missed buttons at every step.

Defining our solution as code also allows us to adopt other development best practices such as **version control**, **tagging, and releasing**. These kinds of best practices will help us when it comes time to perform change management, maintenance, upgrades, and testing.

Finally, defining our infrastructure as code will allow us to more easily move between public cloud providers as needed. With the core of our application defined in code, tools like Terraform can be used to deploy it to different locations or environments (e.g. development, staging, production) with minor modifications. Moving to different cloud providers will only require us to change names and functions of resources.

Let's take a look at the core tools that will be used to deploy our example web application:



Terraform

Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently through code. It can manage popular public cloud services as well as custom in-house solutions.

Terraform uses a common declarative language to create and manage resources across cloud providers. This is beneficial because otherwise, a user would either have to manually create and manage cloud resources by keystroke and mouse clicks, or by using the public cloud provider's native programming language (for example, AWS's CloudFormation), which would increase the likelihood of lock-in.

Terraform is also able to show you a preview or plan of your actions. This allows you to see if a change to your infrastructure will cause destructive outcomes before they happen.

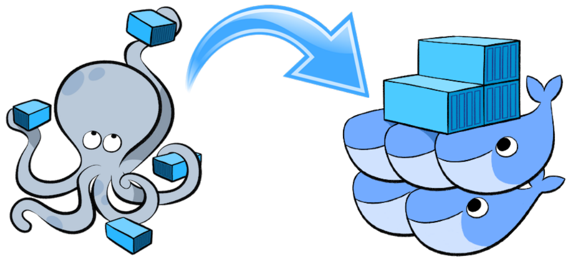
Finally, Terraform enables more advanced types of solution deployments, such as blue-green styles, and deployment of production and development environments in parallel.

Docker

Docker is a suite of tools that allow you to create and manage application code in the form of *containers*. A container holds a software's code, supporting services, resource requirements (such as CPU and memory requirements), and access requirements (such as administrative privileges and network access) bundled into a single portable package.

Docker is a popular solution for managing and deploying software applications, notably web applications. As mentioned, there are numerous Docker-compatible cloud services and cloud independent tools, along with a thriving community of Docker experts. By leveraging Docker we can guarantee our packaged application is able to be managed and deployed almost anywhere and that we can lean on the community for help when needed.

Related to the idea of Infrastructure as Code, Docker containers are defined by creating something called a Dockerfile. This is a text file which lists the steps to create the container. This Dockerfile can be managed just like any other piece of code.



Source: https://blog.emsa-cloud.com/docker-compose

Docker Compose

Docker Compose is a tool for defining and orchestrating multiple Docker containers. Docker Compose captures all aspects of the containers in a YAML file, allowing the relationship between multiple containers to be captured in code.

Deployment and Configuration

Now that we've covered the architecture of our web application and the tools we'll use to deploy our application, let's look at the process of deploying to the cloud.

As mentioned in the introduction, we have designed our web application to run in multiple Docker containers. While public cloud providers offer various services to host Docker containers directly, we will instead install Docker on a basic compute instance (virtual machine) ourselves.

Since we're leveraging Docker, we still retain the ability to move to a public cloud's Docker-specific service if we want to. This kind of balance ensures we don't become locked into a public cloud provider's native service which aligns to our philosophy to always strive to be cloud independent.

A Note on Container Orchestration Engines

Container Orchestration Engines (COE) such as Docker Swarm and Kubernetes provide an additional layer of automation and management on top of Docker. We will not be using a COE in this tutorial.

Code Repository

The code for our example web application can be found in the DAIR solutions repository. You can download a ZIP folder or clone it to your workstation with a Git client by running the following command in your terminal:

```
git clone https://solutions.cloud.canarie.ca:3000/DAIR/dair-cloud-independence-tutorial
```

As long as you are using a Mac or Linux-based workstation, the repository contains everything you need to deploy the application. If you are running Windows, you may consider launching a Linux VM, but that is outside the scope of this tutorial.

The Git repository has a detailed README file which you should read and follow. This file will describe how to obtain credentials to either AWS, Azure, or an OpenStack cloud. We recommend getting credentials for at least two clouds.

Deploying the Application

Our next steps assume that you have credentials for an OpenStack cloud and Microsoft Azure.

Deploying to OpenStack

Follow the instructions in the README file to authenticate to OpenStack and set your environment variables accordingly. Afterward, from your terminal, change directories into the Git repository you checked out locally and run the following command:

```
make apply ENV=openstack
```

When the command has finished, you'll see something like the following:

```
Apply complete! Resources: 11 added, 0 changed, 0 destroyed.


Outputs:

polls_url = http://111.203.33.44/polls

public_ip = 111.203.33.44
```

Where the IP: 111.203.33.44 will be something different on your screen. You can copy and paste the "polls_url" into your browser and see our example application running.

Before you move on to the next section, you can either leave the OpenStack deployment running or you can delete it using this command:

```
make destroy ENV=openstack
```

Deploying to Azure

Follow the instructions in the README file to authenticate to Azure and set your environment variables accordingly.

Next, run:

```
make apply ENV=azure
```

When the command has finished, you'll see something almost identical to when we deployed to OpenStack. And if you visit the "polls_url", you'll see the same output!

Congratulations, you've just deployed the same web application to two different clouds! You can now see the power of leveraging Infrastructure as Code and containerized applications for speed and portability of deployment.

Deploying to AWS

See if you can follow the instructions in the README file to authenticate and deploy to AWS. It should be the same as OpenStack and Azure, requiring you to set environment variables.

Making Changes to the Application

Being able to deploy the application is great and being able to deploy it to multiple clouds is even better. However, it's inevitable that a deployed application will need to be changed. Maybe someone wants a new feature added to the application, or you decided to change the design of the site, or someone reports a bug.

Being able to manage and deploy changes in a controlled way is a very important part of infrastructure development. It's so important that we'll have future tutorials dedicated to this topic. For now, we'll show a very basic way of change deployment.

The Git repository you downloaded from DAIR solutions repository is made up of several directories. The directory called "apps" contains all the files required to manage the applications that make up our total cloud infrastructure. Inside "apps", you'll find another directory called "docker_files", and within "docker_files", a directory called "app" which is our Django web application. To make changes to the Django application, do so from this directory.

To change the "polls" page to add a heading at the top that says, "Polls", open the file "apps/docker_files/app/polls/templates/polls/index.html" in the text editor of your choice.

Insert a new line below Line 3 that reads:

```
<h1>Polls</h1>
```

Save the file and close your editor.

Now that the change has been made, we need to deploy it to update our running application. At the root of the repository, in the same location where the Makefile is, run:

```
make deploy_app ENV=openstack

make restart_app ENV=openstack
```

After these commands have run, you will be able to visit http://111.203.33.44/polls and see the new header.

While the steps involved with making this change looked simple, there was a lot that happened behind the scenes. Terraform created the virtual machine, copied the Docker files, built the containers, and connected the services. This is only scratching the surface and managing or deploying changes is a large topic for another tutorial. In the meantime, for a bit of extra practice, look at the "deploy_app" and "restart_app" tasks in the Makefile and see if you can trace exactly what these commands do.

Summary

In this tutorial, we explained what it means to be cloud independent and why it's important. We illustrated the idea of cloud independence with a modern web application architecture and introduced you to many tools to support our independence goals. We touched on how to maintain a modern web application using tools that support Infrastructure as Code, and we walked through deploying the web application to three different cloud providers, proving our idea that cloud independence works.

Stay tuned for additional tutorials on this and other topics.