HLA Curation Interface Infrastructure

December 10th, 2024 | Liam Mulhall

This document can be viewed here.

Introduction

Trying to decide what infrastructure to use for the HLA Curation Interface (HCI) has been difficult. I am not a DevOps engineer. I'm just a regular software engineer. Prior to working on this project, my system administration expertise was limited to SSHing into a server and manually configuring and deploying things. I had no experience with infrastructure as code tools like Ansible or Terraform. I had to learn about DevOps and the tools used in DevOps on my own from scratch. In this document, I will try to summarize what I have learned, and I will try to justify the decisions I made about the HCI's infrastructure.

Background

The gene curation interface (GCI) and the variant curation interface (VCI) are developed and deployed using a framework called Serverless. The Serverless framework is not to be confused with the idea of serverless computing, which Wikipedia defines as "cloud service category in which the customer can use different cloud capabilities types without the customer having to provision, deploy and manage either hardware or software resources, other than providing customer application code or providing customer data." The Serverless framework and other similar frameworks like SST abstract away the messy details of creating infrastructure. In theory, all the developer needs to do is provide the application code, and have a basic understanding of the cloud services being used. In practice, when something goes wrong with the infrastructure that the framework can't resolve, the developer is stuck because they don't know how the framework works under the hood. Another limitation faced by developers who delegate the management of infrastructure to a framework is being stuck with whatever options the framework provides. In the case of Serverless, you're required to build on top of AWS Lambda, which has limitations. Another downside to the Serverless and SST frameworks is that they are backed by venture capital firms. This means that they will eventually try to squeeze money out of their users. I took all of these downsides into consideration when thinking about what infrastructure I wanted to use for the HCI.

Options Considered

The table below shows the main options I considered.

Option	What's the dollar cost?	What's the maintenance cost?	Understand "under the hood"?	Is it flexible?	Does it follow industry best practices?	Does it allow for zero-downtime deployments?	Score = 3*green + 2*yellow + 1*red
On-prem server configured with Ansible	low	high	yes	yes	no	no	12
Lightsail configured with Ansible	low	high	yes	yes	no	potentially	13
AWS App Runner & RDS	medium	low	no	no	yes	yes	13
AWS Copilot CLI Load Balanced Web Service & RDS	medium	low	no	no	yes	yes	13
Serverless	medium	low	no	no	yes	yes	13
SST	medium	low	no	no	yes	yes	13
Hand-Rolled ECS & RDS	medium	medium	yes	yes	yes	yes	16

I'll discuss each of these options in more detail below.

On-prem server configured with Ansible

This is what I initially wanted to do. It's close to what I was familiar with from my previous job. However, there wasn't much buy-in from my fellow engineers and my supervisor. This is understandable because it's a relatively old-fashioned way of deploying software.

Lightsail configured with Ansible

This was my second idea. If I couldn't use our on-prem server, I could rent a server from AWS that I could SSH into and configure and deploy software the way I'm familiar with. If I wanted to avoid managing a database, I could use RDS. If I wanted to have zero-downtime deployments, I

might be able to use two different Lightsail instances to do blue-green deployments (or something along those lines). The problem with this approach was that it's kind of a weird hybrid of doing things the old-fashioned way and doing things the new way. Lightsail is a managed service, so there are some things abstracted away from you. There's also the issue of needing to get the code working regardless of the underlying operating system. If I wanted to use the on-prem server as a test server and a Lightsail instance as a production server, I'd probably want to containerize the application code. This made me consider using the AWS services that are specifically meant for containers.

AWS App Runner & RDS

App Runner is similar to Serverless and SST in that it's a serverless computing service. In theory, it abstracts away the messy details of creating and managing the infrastructure underlying an application. In practice, I found that it spits out cryptic error messages when something isn't working as expected. I can't get to the bottom of what went wrong because the error messages from the deeper layers don't bubble up to the top layer. Based on what I've read on various forums, this is a common complaint. Another issue I found with App Runner is that it takes a long time to deploy services.

AWS Copilot CLI Load Balanced Web Service & RDS

AWS Copilot is an open source command line interface (CLI) that aids the engineer in creating and managing AWS infrastructure. It operates at a level higher than App Runner. It can be used to create an App Runner service. It can also be used to create other services, including the one I would have liked to use for the HCI, a load balanced web service. In my initial experiments with it, I liked it. It took a long time to deploy services, but it abstracted away a lot of the stuff I didn't want to deal with, e.g. networking, security groups, etc. It also provided a way to set up code pipelines, which would have been convenient. Unfortunately, AWS Copilot is going to be put into maintenance mode soon and it will eventually be deprecated. That's not necessarily a bad thing. It would have had the same problems as Serverless and SST, but it wouldn't have been VC-backed.

Serverless

After some frustration with the other AWS services, I considered using the most up-to-date version of Serverless. When I went to set up the Serverless app, I was prompted to sign in to Serverless in order to use the framework. This is a huge turnoff for me. I really don't like the idea of having to pay to use open source software. At some point, the people who make Serverless are going to try to make a profit, and I don't want Stanford ClinGen to have to pay for both an intermediary framework and cloud computing services. I'm especially wary of Serverless because it leverages proprietary AWS services, which can lead to vendor lock-in in addition to framework lock-in.

SST

I briefly considered using SST because it seems to be a bit more flexible than Serverless, but it has problems similar to Serverless and the other serverless computing options I considered.

Hand-Rolled ECS & RDS

I came to the conclusion that there is no free lunch when it comes to creating and managing the infrastructure for the HCI. If I use a serverless computing option that abstracts away the messy details, I gain some convenience at the cost of a lot of control and transparency. I'd prefer to have the control and transparency, so my plan is to use Terraform to create and manage the AWS infrastructure for the HCI. I'm going to roughly follow this tutorial to create infrastructure around AWS's Elastic Container Service (ECS) and their Relational Database Service (RDS).