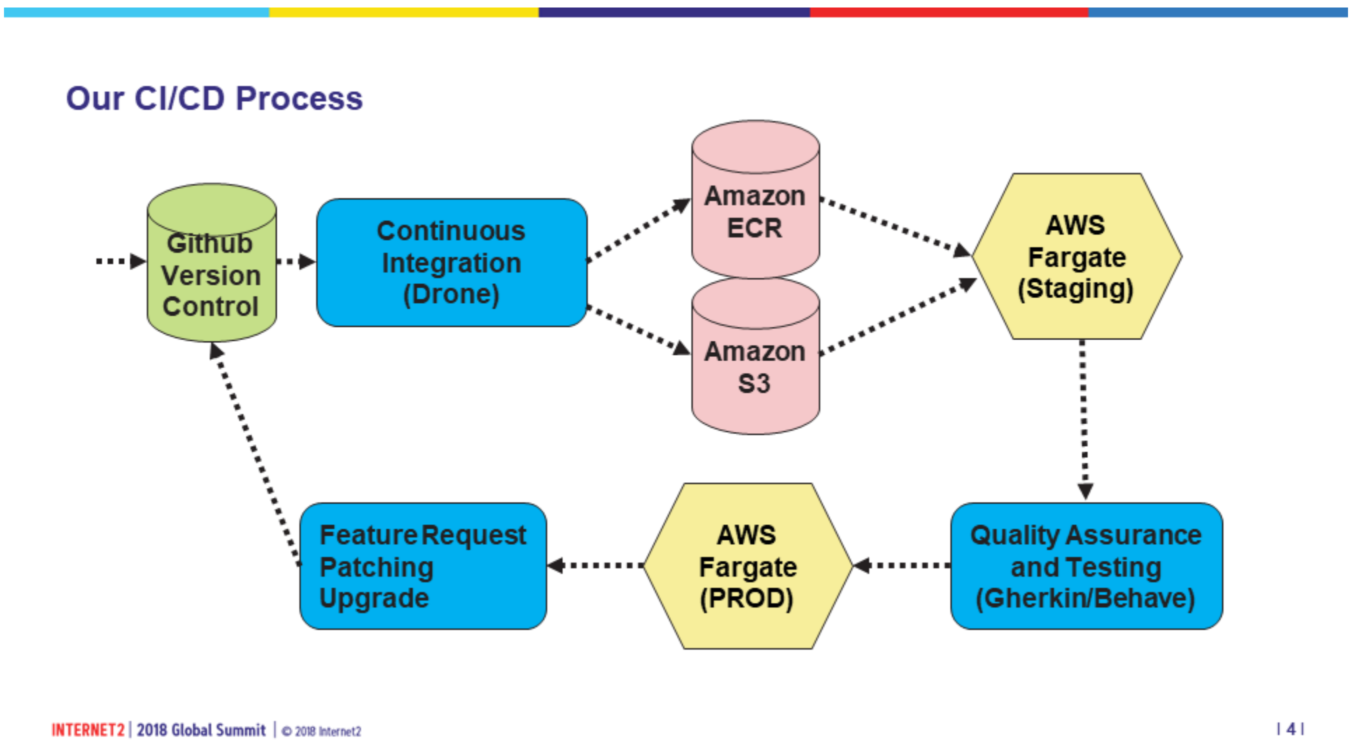
Our Successful Deployment of TIER Packages in AWS

In our previous blog post (see Chop Down the Beanstalk, Open Up the Fargate), we examined our path to using AWS Fargate as our container deployment host. I'm pleased to say, Illinois has achieved success in deploying the TIER Docker images in AWS Fargate, in particular the Grouper and Shibboleth images, so far in a testing environment, but with hopes of moving to production as soon as July.

Many of you may have seen our demo at the Internet2 Global Summit TIER Showcase, showing our prototype Grouper installation. That was the initial milestone of success, but we have continued to build on that by fine-tuning the deployment, adding the back-end database using AWS RDS, and adding an endpoint to the Application Load Balancer (ALB). In addition, we have repurposed the same code and CI/CD methods to deploy our first Shibboleth test instance in AWS. Here's a quick overview of the components that helped us achieve successful deployment.

Our software development team are more than just developers, they have been the pioneers of our DevOps process of continuous integration and continuous delivery (CI/CD) using AWS, in conjunction with tools such as Github, Terraform and Drone. Here's a look at a simplified version of our process, as shown on a slide during the TIER Showcase demo:


Our CI/CD Process
Github Version Control
Continuous Integration (Drone)
Amazon ECR
Amazon S3
AWS Fargate (Staging)
Quality Assurance and Testing (Gherkin/Behave)
AWS Fargate (PROD)
Feature Request Patching Upgrade
INTERNET2 | 2018 Global Summit | © 2018 Internet2
| 4 |

Github (Version Control)

Github has become our repository for all AWS code. Github has been in use by our software developers for some time, but in the DevOps world, with infrastructure as code, our service admins have no become Github Repo contributors. In our design, a particular service is derived using two repos: One containing the re-usable central Terraform modules that our software development team built, and then a separate repo that contains our Docker configurations and software configs. As a new service is launched in AWS, a new branch of the central service terraform module is created with a new folder in the region of deployment (i.e., **`us-east-2/services/<servicename>`**) containing small number of files for the specific Terraform declarations needed by the service infrastructure, with a reference back to the main modules, leveraging Terraform's modular capabilities to re-use code.

The Docker build is stored in the other repo and contains our sets of Dockerfiles, along with the configs and secrets that are to be added to the images. Although the repos go hand-in-hand to deploy a service, it is important to observe the distinction: the Terraform codes builds the AWS infrastructure that defines the service (the virtual hardware), while the Docker code builds the images that are pushed to the Amazon Elastic Container Registry (ECR) (the operating system and software). That is, once the infrastructure is built, if there are no changes to the networking or container definitions, the service images themselves can be quickly updated using the ECR and the containers restarted, without redeploying infrastructure.

Terraform (Infrastructure)

The Terraform code is executed using a useful wrapper known as TerraGrunt by GruntWorks, that preserves the Terraform state in an AWS S3 bucket automatically. Once we have our Github repo of our service branch of the Terraform modules, we can first preview the components being built using **`terragrunt plan`**, and check for any errors. Once this looks good, we simply issue a **`terragrunt apply`** to execute the code. Typically there will be a dozen or so components "constructed" in AWS, including load balancer endpoints, clusters, networks, service names and tasks.

Docker (Images)

As mentioned, the service configuration of Grouper is based on the published TIER Grouper Docker images. Shibboleth follows the same path using TIER Shibboleth images. Custom Dockerfiles were built using a two-layer model of a customized "base" image, and then further customizations for each Grouper role being deployed. More on that in "The Finished Product" section below.

Drone (Pipelining)

As of this writing, we have not yet implemented the Drone components, but the main purpose of Drone is to "watch" for a new commit/push of the infrastructure configuration in Github, and instigate a build/rebuild of the infrastructure in a staging area of AWS using a docker push. We will update you more on Drone configuration in a future blog post.

In Drone's place, we have basically scripted a similar behavior that logs in to aws, grabs the docker login command, builds the Docker images locally, tags them and pushes them up into the Amazon ECR. With the infrastructure already built, it's simply a matter of STOPping the running container, so that it restarts with the newest image, tagged "latest".

Behave (QA Testing)

Like Drone, we still have work to do, but have chosen Behave for our QA testing suite. Once this process is refined, we will be sure to describe this in a follow-up post.

The Finished Product

Using the TIER Demo packaging as a starting point, we defined our base infrastructure for Grouper to have three active containers: a UI, a Webservice, and a Daemon node. This was basically accomplished with three separate Terraform configurations and three different task.json files, to allow unique customizations of network port exposure and memory sizes needed by each Grouper role. As mentioned before, this was stored in a branch of our central modules code.

Following the same three-node model, the Docker configuration followed in a similar way. First we built a customized "Grouper Base" image, which derived from the original TIER image (**`FROM tier/grouper:latest`**), but added our custom configs pointing to the RDS instance of the grouper database, configs for connecting to the campus Active Directory (which happen to have domain controllers as EC2 instances in AWS), SSL certificates, credentials and secrets, etc. that were common to all three nodes. Then each node had its own individual Dockerfile that derived from our Grouper base image, to add additional specifics unique to that image. Once all the images were built, they were tagged using the AWS ECR repo tag and Docker Push'd up to the ECR.

Once the images were uploaded to the ECR, we were able to run the Terraform (using Terragrunt) to launch the service. Within minutes, we had running containers answering on the ports defined in the task.json file, and we could bring up the Grouper UI.

More to Do

Still more integration needs to be done. Besides the Drone and Behave processes, we have to conquer a few more things.

1. **Converting our RDS instance as a separate code**. Generally, with a persistent database, you don't want to continuously burn down and redeploy, so we have to treat this special, but we do want it in code so that the configuration is self-documenting and reproducable, for example to bring up a cloned instance for testing. For now we just "hand-crafted" the RDS instance in the AWS console.
2. **Tackling DNS** One issue with cloud-hosted solutions is the DNS nameservers. While we have our own authoritative nameservers for all things illinois.edu, we must have our DNS configuration for AWS in their Route 53 DNS resolvers. This requires us to delegate a particular zone, so that we can associate our ALB endpoints with static, resolvable hostnames in our namespace. In turn, we can define friendly service CNAME records in the campus DNS to point to the Route 53 records.
3. **Shift to blue/green deployment methods** We have to shift our thinking on what a "test" environment really is, and move toward the blue/green concepts of DevOps. This changes the way we think of changes to a service and how it is deployed, but the CI/CD model.
4. **Autoscaling Nodes** We one day hope to configure AWS to just add more UI nodes or more Daemon nodes or more WS nodes if/when the load gets to a certain level that we need to spread the work. Lots of testing and evaluating how that behaves in the Grouper architecture.
5. **Grouper Shell Node** We have settled on the idea of a separate and longer-living EC2 instance that contains Grouper and SSH to allow us remote access and to execute GSH scripts against the AWS instance of Grouper. It would be interesting to experiment with the idea of converting some of the oft-run GSH scripts into individual launched Fargate instances that just run and die on their own, perhaps on a schedule.

We plan to demo much of this at the June 19th TIER Campus Success meeting. Bring your questions and feedback!