**System walks through.**

1. Clone the repository

* From gitbash, cd to C, mkdir Repos (capital R). Under Repos, mkdir code, under code, mkdir odin, under odin, mkdir sipir-mission-bootstrap.
* Cd into sipr-mission-bootsrap and do:

Git init (this will initialize an empty git repository. So in the sipr-mission-bootstrap folder, a new “.git” folder will be created). You will now see “(master)” in front of your path(/c/Repos/code/odin/sipr-mission-bootstrap) in gitbash

* Go to GitLab and create an access token
* Next do “git remote add origin https;//gitlab.odin.dso.mil/platform-one/devops/mission-bootstrap/sipr-mission-bootstrap.git”
* Next do “git remote -v” and then “git fetch origin” once prompted to add your credentials (use you username and access token from git lab).
* Next do “git pull origin master”
* Open it in visual studio code
* If you run into a ssl certificate issue, do this

git config --global http.sslbackend schannel

1. In the mission-manifests folder, you have all the deployment files for all the applications. In the integrations folder, you have all the configmaps and secrets for all the applications.
2. Go to AWS and bookmark (star) all the important services, including EC2, secrets manager, s3, VPC, wafd & shield, RDS, etc…
3. All the EC2 instances that have “general-agent” in their name are “worker nodes”, and those that have “server” in their names are “master nodes”. There is generally 3 master nodes and 3 or more worker nodes for each cluster. The IPV4 private addresses are important since you will sometimes use it to get into a node.
4. In S3, all the minio bucket are buckets used by the team for different purposes. **Minio** is a microservice hosted in the k8s cluster that acts as the middleman between our mission apps and the actual AWS instance. So minio creates that connection and it also provisions those buckets using **minio provisioner**. If you Look at the bucket name, you will see that the have the following format “**minio-<app-name>-<cluster-name-prefix>-<random-10-hexadecimal-character>.** For example, “minio-genesis-sipr-mission-test-123kjn123kjn123kjn” or “minio-fuelai-sipr-mission-prod-0b448e6afd”. The random-10-hexadecimal-character is created using **openssl.**
5. A better job should be done naming these buckets using the appropriate fabrics. Like if it is jwics, then “sipr” need to be replaced by “jwics”. But that might get done in the future. Just keep in mind that, although sipr is used in the name of those buckets, it does not mean that they are all part of the sipr fabric.
6. These buckets are the operational buckets for the applications. If you click on one of them, you will see that there are folders inside. And those folders are used by the developer to develop the application. Their code will have permission to access these buckets and call for stuff out of those folders. Those folders can also be used for database seeding,
7. Most of the time the namespace name is going to be the app name, but not always.
8. In the sipr-mission-bootstrap bucket is used as transient folder location to store bundles, synker packages prior to taking it to the high side.
9. In RDS you will see all the database supporting the ODIN project, all the database for big-bang, and mission apps, etc…
10. The db identifier (name) usually has this format on nipr “odin-mission-prod-<app-name>” dor example, “odin-mission-prod-force”. On the high side the format is “<app-name>-prod or staging or test”. On nipr the “prod” in the name of the RDS does not necessarily mean that they are production data, except for the trinity apps (fuelai,wingmanai,trinityai).
11. We manually create the RDS instances as empty databases for the apps. In k8s we have provisioners that run as jobs for RDSs, and they will build out the databases that the customer expects to be in place when the pod spins up, so a sql connection can be created to use or feed data or seed the database.
12. Whenever we create and RDS, we create it with a root admin, a root password and at least one database. This database never gets used because the provisioner makes a new database that gets used by the application.
13. **MGMT-PROD** is the cluster that runs the **keycloak** instance that manages all the cluster’s access to all the apps unless we bypass it with an admin gateway.
14. How does keycloak works? When you type the url of the app in your browser, DNS receives the request and redirects it toward the app load balancer, the load balancer sends it to the node in the node group it manages. Once the request hit the node port, which is protected by SDO, k8s kicks in and redirect the requests to keycloak, which then checks the client ID, identify the appropriate group and see if the person is a member, then attaches a Javel and cookie with a web token and a keycloak certificate, then redirected toward the load balancer. Then k8s redirects it to the virtual services to compare against one of the matching virtual service url.
15. In UC2S-ODIN-MGMT-PROD, you will find creds for keycloak and more.
16. In UC2S-ODIN-MISSION-PROD, we store all the creds for all the apps RDS connections (root admin and password), ArgoCD, Grafana, elastic-kibana, Prometheus, etc….
17. Navigate to VPC, then security groups. There is one security group per VPC. One security group cannot be used for two different VPC. Security groups are important when building the RDS. Postgress (5432) and mysql (3307 or 3306) talk on very specific ports identifies as default ports that AWS creates. Traffic from k8s goes through those ports to get into the RDS.
18. We admin and public load balancers.
19. The difference between the ALB and the NLB is that the NLB responds on the transport layer (TCP:443) whereas the ALB operates on the application layer (it is not limited to port 443)