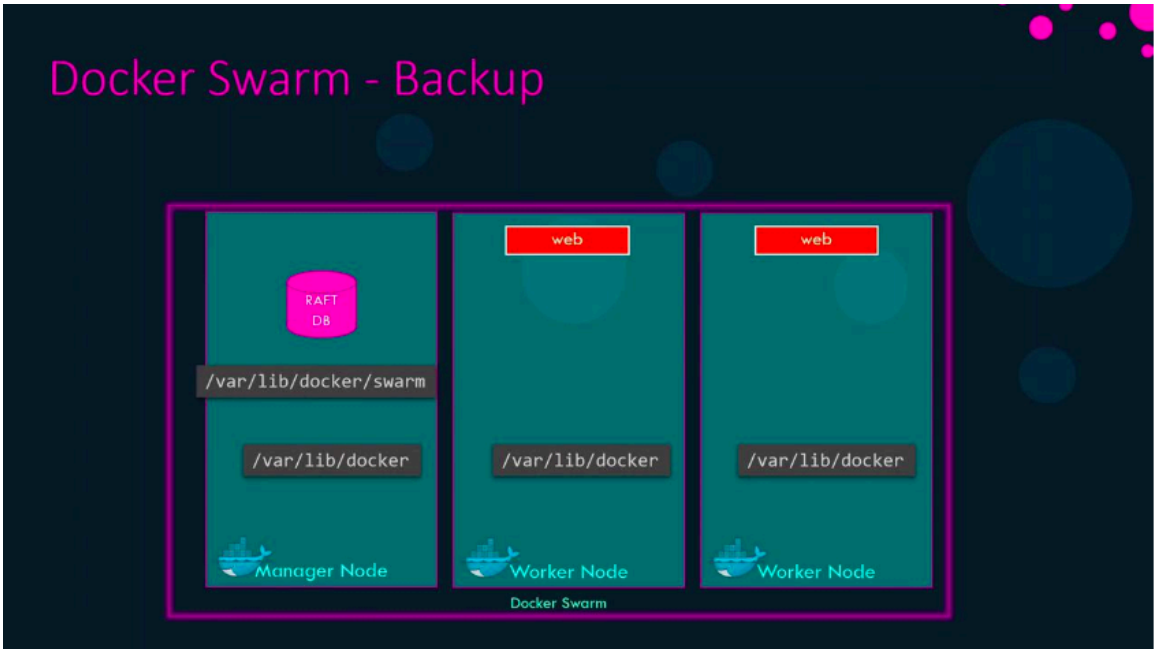
Docker Swarm DR setup in AWS

1. Overview of a typical Swarm setup on prem:

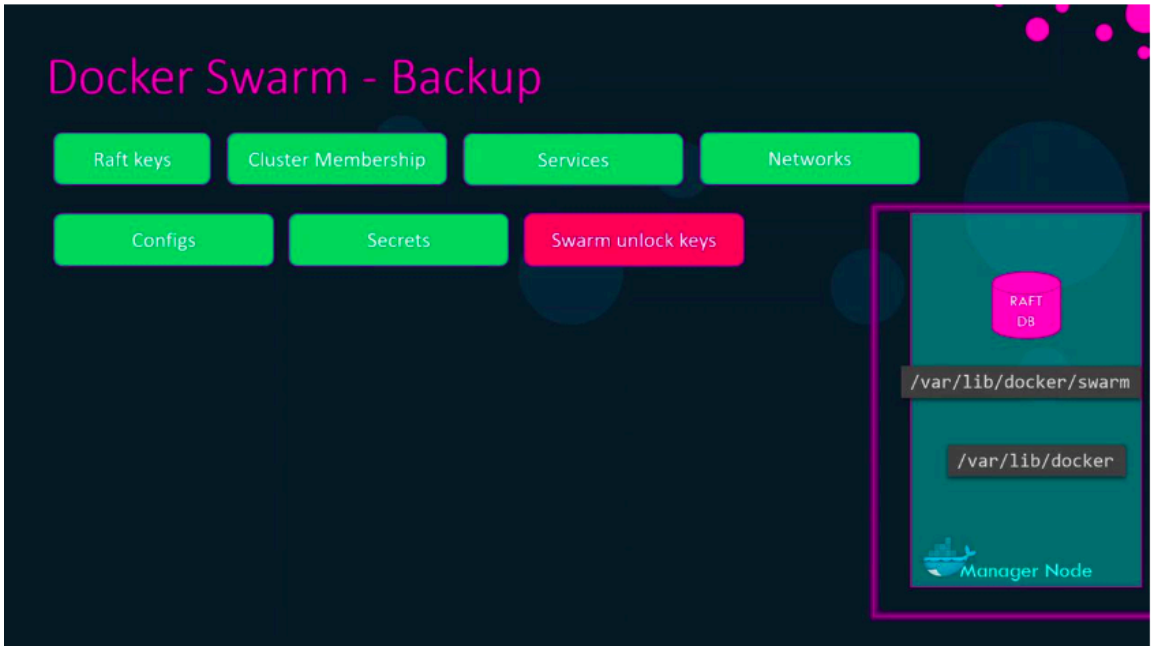
- 1 Master node(manager) and 2 slave nodes(worker)
- Major components:
  Master node: Saves the Swarm state in location: /var/lib/docker/swarm/
  Slave node: Executes the container tasks by the master node
  An External Database: Maintained Separately, not inside containers.

Some configuration components are:

a. **Raft database:** Cluster membership, service definitions, overlay networks



b. **Configurations and Secrets:** Application configuration and sensitive data
c. **Node Cryptographic Identity:** Keys for secure inter-node communication
d. **Service Definitions:** Container deployment specifications



More info [here]

2. Risks with the present setup:

Docker Swarm’s high dependencies on the Manager node, Raft Consensus Protocol

The user's current configuration, a Docker Swarm cluster with a single master node and two worker nodes, presents a significant and immediate risk. While a single-manager swarm is fully functional, it represents a critical single point of failure. This is due to the inner workings of Docker Swarm's cluster management.

Docker Swarm manager nodes implement the Raft Consensus Algorithm to maintain a consistent global cluster state. This state includes all service definitions, configurations, network settings, and secrets. For the cluster to remain healthy and operational, a majority of its manager nodes, known as the quorum, **must be available to agree on changes.** The quorum is defined as

(N/2)+1, where N is the total number of manager nodes.

In the case of a single-master cluster, **losing that one manager node means the quorum is immediately lost, bringing the entire control plane of the swarm to a halt.** While existing tasks on worker nodes will continue to run, no new tasks can be scheduled, and the cluster cannot respond to failures or rebalance workloads.

3. Our Available Solutions for DR strategy:

But First,

a. Deconstruction of the Docker Swarm for DR:

Some important elements like **Swarm state, container image and the application's persistent data (Container Level)** must be carefully considered for a successful DR, not just only a container image or application code is enough. Read here.

1. Swarm Cluster State and Configuration:

The state of a Docker Swarm is critical and is stored on the manager nodes in the /**var/lib/docker/swarm** directory. This data includes service definitions, secrets, configurations, and network settings. The loss of this state would require a complete rebuild of the cluster from scratch.
A robust DR plan necessitates a reliable procedure for backing up and restoring this data. The recommended procedure involves taking a "cold" backup by stopping the Docker daemon on a manager node to ensure no data is being changed during the backup process. The entire **/var/lib/docker/swarm** directory is then archived and stored securely. **A "hot" backup, performed while the manager is running, is possible but not recommended due to less predictable results.**

Standard procedure for this is a six-step procedure:

1. Shut down the Docker daemon on the target host.
2. Remove the existing contents of the `/var/lib/docker/swarm` directory on the new host.
3. Restore the archived backup of the `/var/lib/docker/swarm` directory into the new location. *It is important to note that the new node must use the same IP address as the node from which the backup was made, as the swarm data does not reset the IP address.*
4. Restart the Docker daemon on the new node.
5. Run the `docker swarm init --force-new-cluster` command to re-initialize the swarm, ensuring the new node does not attempt to connect to the old, non-existent nodes.
6. Add additional manager and worker nodes to the new swarm to bring it to full operating capacity and reinstate the backup regimen.

2. Container Image Registry Replication:

A crucial component of any containerized application is its image registry. A DR plan requires that all application images are available in the AWS cloud for deployment during a recovery event.

The simplest approach is a manual `docker push` workflow, but this is only suitable for low-volume or infrequently updated images. For a production-grade solution, Amazon Elastic Container Registry (ECR) offers powerful replication features. There are two primary methods for synchronizing an on-premises registry with ECR.

We have two methods for Container registry replication:

Pull-Through Caching: An ECR private registry can be configured to act as a pull-through cache for the on-premises registry. When an image is requested from the AWS ECR registry that it does not have, ECR pulls the image from the on-premises registry and caches it. The documentation outlines the procedure for setting up this rule and storing authentication credentials in AWS Secrets Manager. ECR automatically checks for updates to the upstream image at least once every 24 hours to ensure the cached version remains current. Link here

Third-Party Registry Replication: For complex, enterprise-level environments, a third-party registry like **Harbor** can be deployed on-premises. Harbor supports replication rules to synchronize with an AWS ECR endpoint. This method provides granular control over replication policies and can be an ideal solution for organizations that already use or require a sophisticated on-premises registry. Link here (EKS too, heavy on Kubernetes)

3. Data and Persistent Volume Management:

(This is from gemini, we can have multiple approaches for DB migration and sync) Need more emphasis on using NFS.

For applications with low RPO requirements, continuous data replication is essential. AWS Database Migration Service (DMS) is an ideal tool for this task, enabling continuous replication from an on-premises database to a managed AWS RDS instance. The workflow for setting up DMS involves creating a replication instance, defining source and target endpoints, and then creating a migration task. For live, ongoing changes, the migration type can be set to "Migrate existing data and replicate ongoing changes".

For a Multi-Site Active/Active architecture, continuous replication must be bi-directional, meaning that changes made in the AWS cluster are replicated back to the on-premises site and vice-versa. This introduces significant

complexity related to data consistency and requires a different architectural approach. This model shifts from a traditional failover plan to a high-availability strategy, where the application must be designed to handle eventual consistency and conflicts. Services such as Amazon DynamoDB global tables and Amazon Aurora global databases are designed for these distributed, multi-write scenarios and serve as a reference architecture for how a DR plan necessitates a fundamental change in the application's data model.

B. A more Detailed Implementation for Each DR Strategy:

Now with Each of the DR Strategy Blueprint provided with AWS, we purpose different strategy, namely Backup and Restore, Pilot light, Warm Standby and Active-Active.

1. Backup and restore:

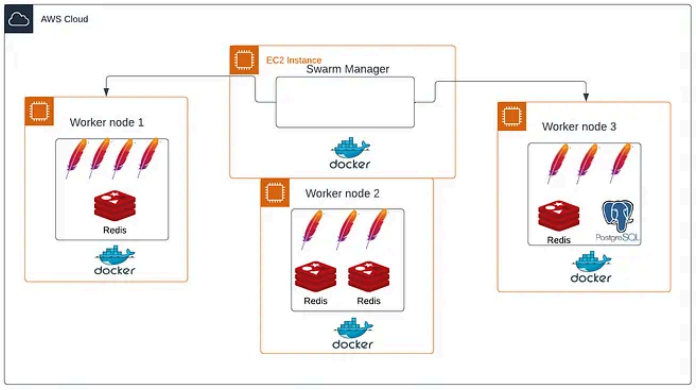
Automated Backup of Swarm State: A daily cron job is configured on one of the on-premises manager nodes. The script will first perform a *systemctl* stop docker to take a cold backup of /var/lib/docker/swarm. It will then use the AWS CLI to securely archive and upload the backup to a versioned S3 bucket with Object Lock enabled.

Database Backup to S3: The on-premises database is configured to perform a nightly full backup to a local file system. A separate script then uses the AWS CLI to upload the backup dump to a different S3 bucket. For SQL Server 2022, a native BACKUP DATABASE TO URL command can be used to send the backup directly to S3.

Image Replication: A workflow is established to automatically push any new or updated container images to the designated Amazon ECR repository.

This official doc is useful for migrating Swarm Applications to AWS.

Alternatively, We can go for a more bare metal approach by deploying EC2 instances and self - managing nodes.



Here's a more descriptive doc

So with this strategy, our Recoverery (Disaster Recovery) will include the following steps:

1. **Launch and Restore the Swarm Manager:** The ASG launches the initial EC2 instance. A user data script on this instance downloads the most recent Swarm state backup from S3, restores the /var/lib/docker/swarm directory, and then executes the docker swarm init --force-new-cluster command to establish the new Swarm cluster.

2. **Restore Persistent Data:** The database backup from S3 is downloaded and restored to a newly provisioned database on an EC2 instance or to a new Amazon RDS instance.

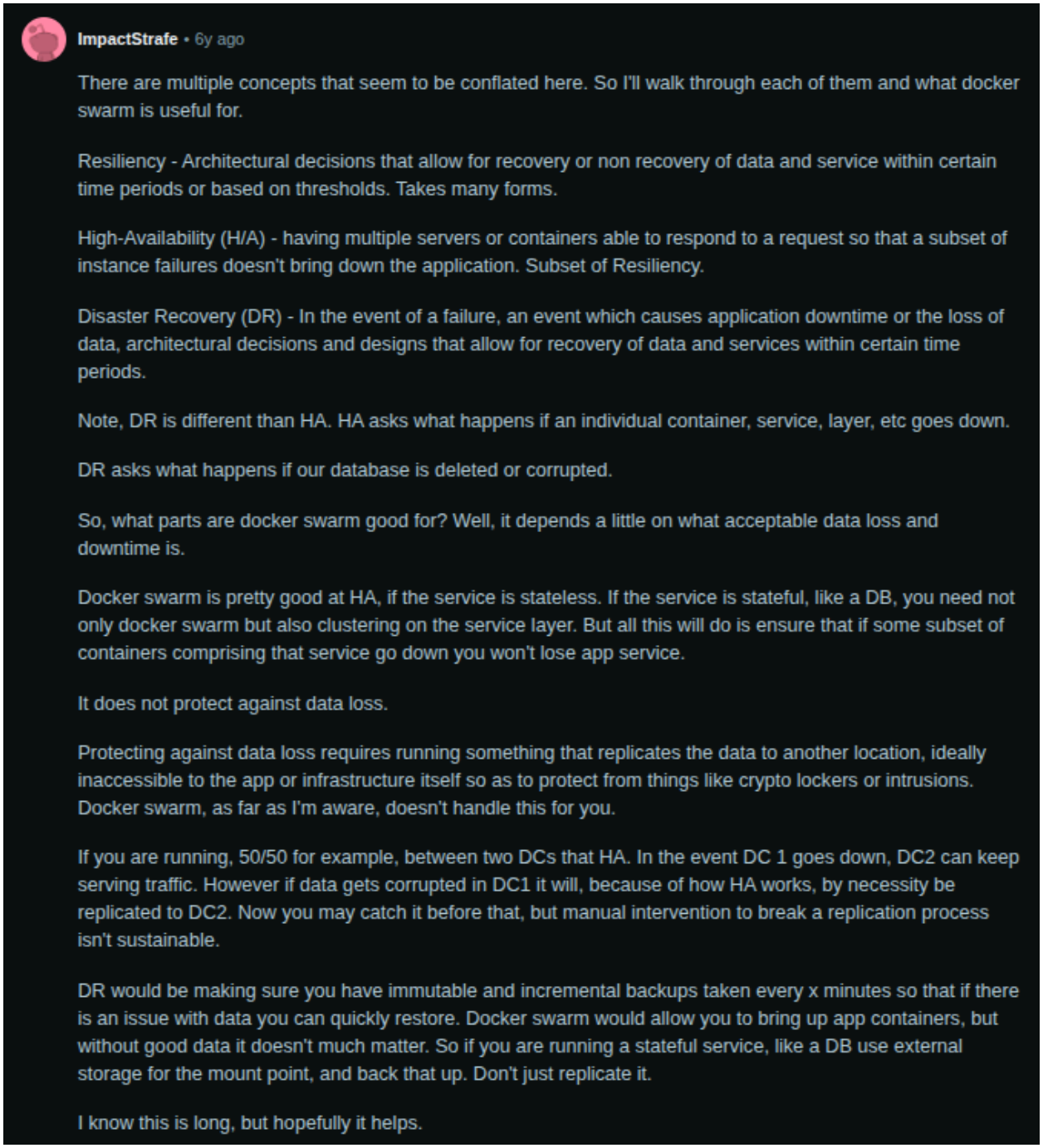
3. **Re-hydrate the Cluster:** The Auto scaling group desired capacity is increased to add the necessary number of manager and worker nodes. The user data scripts on these nodes will automatically execute docker swarm join commands to join the new cluster. (This has to be tested for a good grasp)

4. **Deploy Application:** A script or manual process uses the docker stack deploy command to launch the application services from the images in ECR.

5. **Redirect Traffic:** The DNS record in Amazon Route 53 is updated to point to the new load balancer or public IP of the AWS-hosted Swarm cluster.

This is the tentative workflow for our Backup and Restore strategy.

Consider these paragraphs from a docker pro:



ImpactStrafe • 6y ago

There are multiple concepts that seem to be conflated here. So I'll walk through each of them and what docker swarm is useful for.

Resiliency - Architectural decisions that allow for recovery or non recovery of data and service within certain time periods or based on thresholds. Takes many forms.

High-Availability (H/A) - having multiple servers or containers able to respond to a request so that a subset of instance failures doesn't bring down the application. Subset of Resiliency.

Disaster Recovery (DR) - In the event of a failure, an event which causes application downtime or the loss of data, architectural decisions and designs that allow for recovery of data and services within certain time periods.

Note, DR is different than HA. HA asks what happens if an individual container, service, layer, etc goes down.

DR asks what happens if our database is deleted or corrupted.

So, what parts are docker swarm good for? Well, it depends a little on what acceptable data loss and downtime is.

Docker swarm is pretty good at HA, if the service is stateless. If the service is stateful, like a DB, you need not only docker swarm but also clustering on the service layer. But all this will do is ensure that if some subset of containers comprising that service go down you won't lose app service.

It does not protect against data loss.

Protecting against data loss requires running something that replicates the data to another location, ideally inaccessible to the app or infrastructure itself so as to protect from things like crypto lockers or intrusions. Docker swarm, as far as I'm aware, doesn't handle this for you.

If you are running, 50/50 for example, between two DCs that HA. In the event DC 1 goes down, DC2 can keep serving traffic. However if data gets corrupted in DC1 it will, because of how HA works, by necessity be replicated to DC2. Now you may catch it before that, but manual intervention to break a replication process isn't sustainable.

DR would be making sure you have immutable and incremental backups taken every x minutes so that if there is an issue with data you can quickly restore. Docker swarm would allow you to bring up app containers, but without good data it doesn't much matter. So if you are running a stateful service, like a DB use external storage for the mount point, and back that up. Don't just replicate it.

I know this is long, but hopefully it helps.

Here's a productive discussion

2. Backup and Restore:

A single, minimal Docker Swarm cluster (e.g., one manager and one worker) is maintained in AWS. Critical databases are continuously replicated from on-premises to AWS using AWS DMS. Application container images are continuously replicated to Amazon ECR. A pre-configured AWS CloudFormation template for the full-capacity Docker Swarm ASG is ready to be launched at a moment's notice.

Hence we use the following failover steps:

1. **Monitoring and Triggering:** Amazon CloudWatch agents on the on-premises nodes send system and application metrics to AWS. A CloudWatch alarm is configured to trigger when a key metric (e.g., on-premises host CPU utilization) indicates a failure.

2. **Resource Provisioning:** The CloudWatch alarm triggers a pre-configured AWS Lambda function. This function's purpose is twofold: it executes the CloudFormation template to provision the full-capacity EC2 ASG, and it also initiates the restoration of data from the DRS staging area.

3. **Application Deployment:** Once the ASG has scaled the cluster to full capacity, the application services are deployed from ECR.

4. **DNS Failover:** The Lambda function or a separate process updates the Route 53 failover record to redirect traffic to the AWS-hosted environment.

3. Warm StandBy:

Overview: A Docker Swarm cluster is continuously running in AWS with a reduced number of nodes and service replicas. Bi-directional data replication or a one-way continuous replication solution is in place to keep the on-premises and AWS databases synchronized with a low RPO**.**

Failover Steps:

1. Monitoring: CloudWatch monitors the health of both the on-premises and AWS environments.
2. Trigger: A CloudWatch alarm detects a failure in the on-premises environment.
3. Scaling: The CloudWatch alarm triggers a scaling policy on the EC2 ASG to immediately scale the cluster to its production capacity. The Warm Standby environment, which is already running, only needs to scale its node count. Simultaneously, a separate automation script can issue a
4. *docker service scale* command to increase the number of running service replicas within the swarm.
5. DNS Switch: The Route 53 health checks for the on-premises site detect the failure and automatically redirect traffic to the AWS environment.