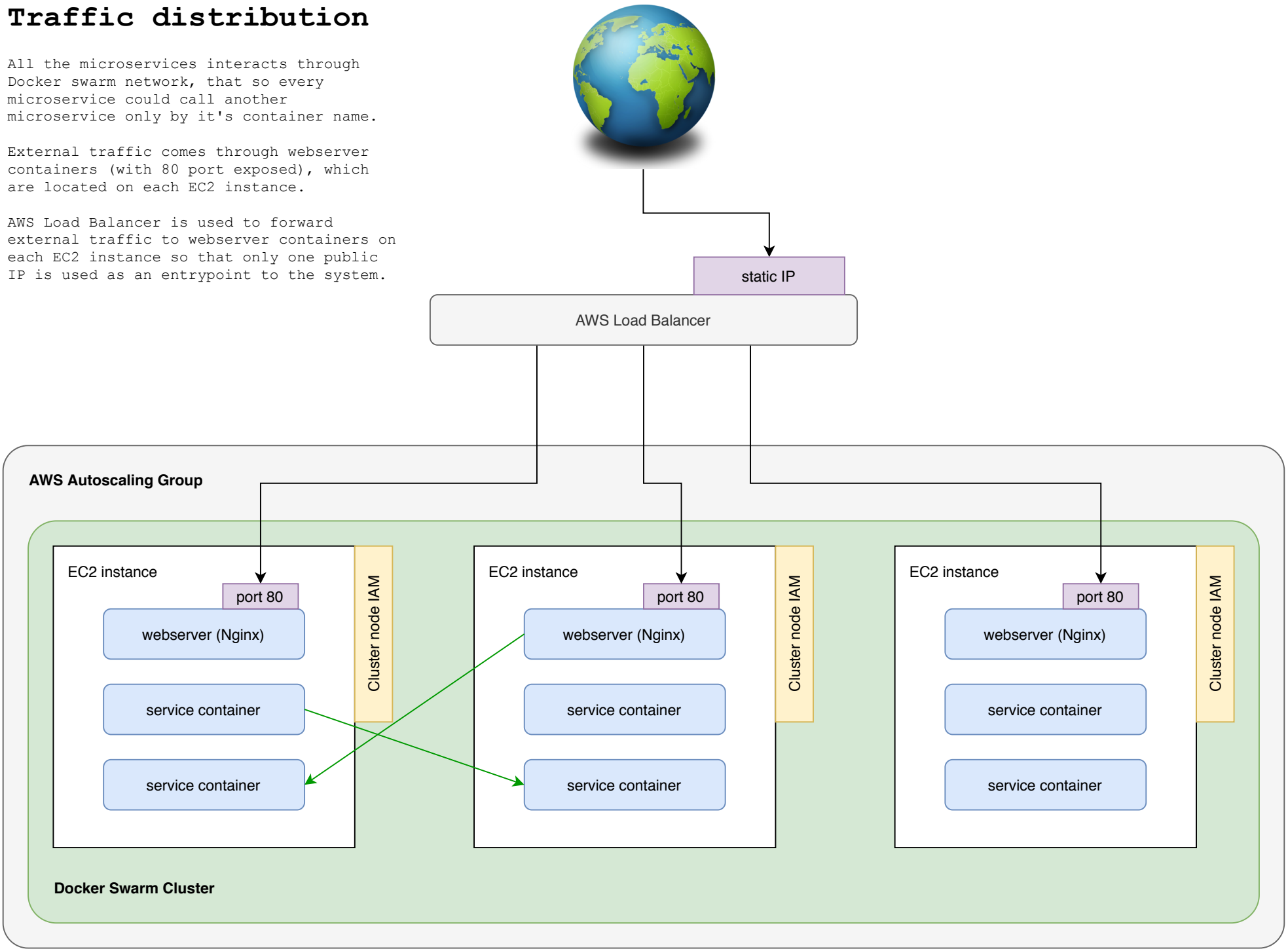
Traffic distribution

All the microservices interacts through Docker swarm network, that so every microservice could call another microservice only by it's container name.

External traffic comes through webserver containers (with 80 port exposed), which are located on each EC2 instance.

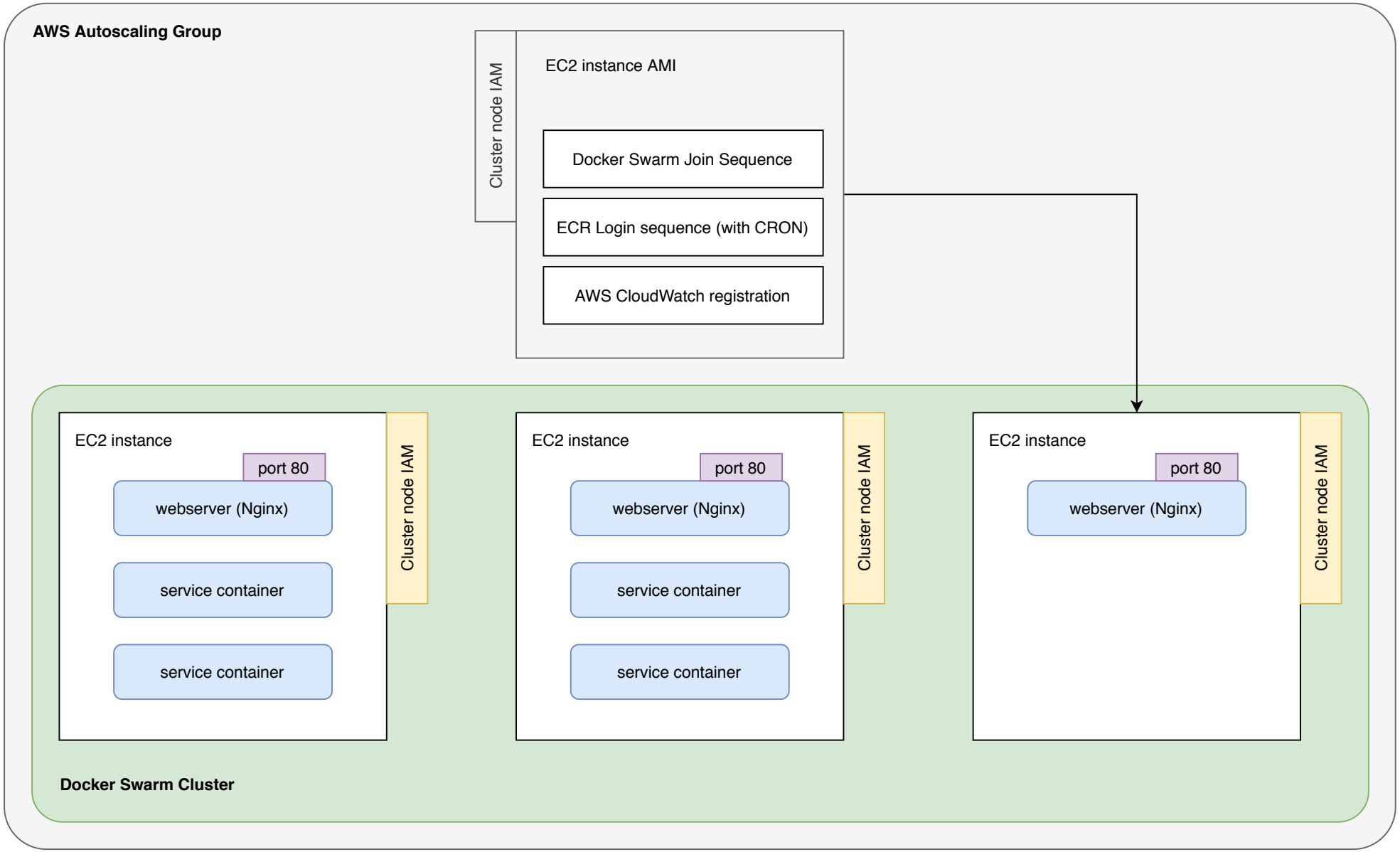
AWS Load Balancer is used to forward external traffic to webserver containers on each EC2 instance so that only one public IP is used as an entrypoint to the system.



Cluster Scaling

AWS AMI is used as a template for new cluster node set up. This AMI have to be prepared for each cluster. ECR credentials and Docker swarm initialization data should be included in it.

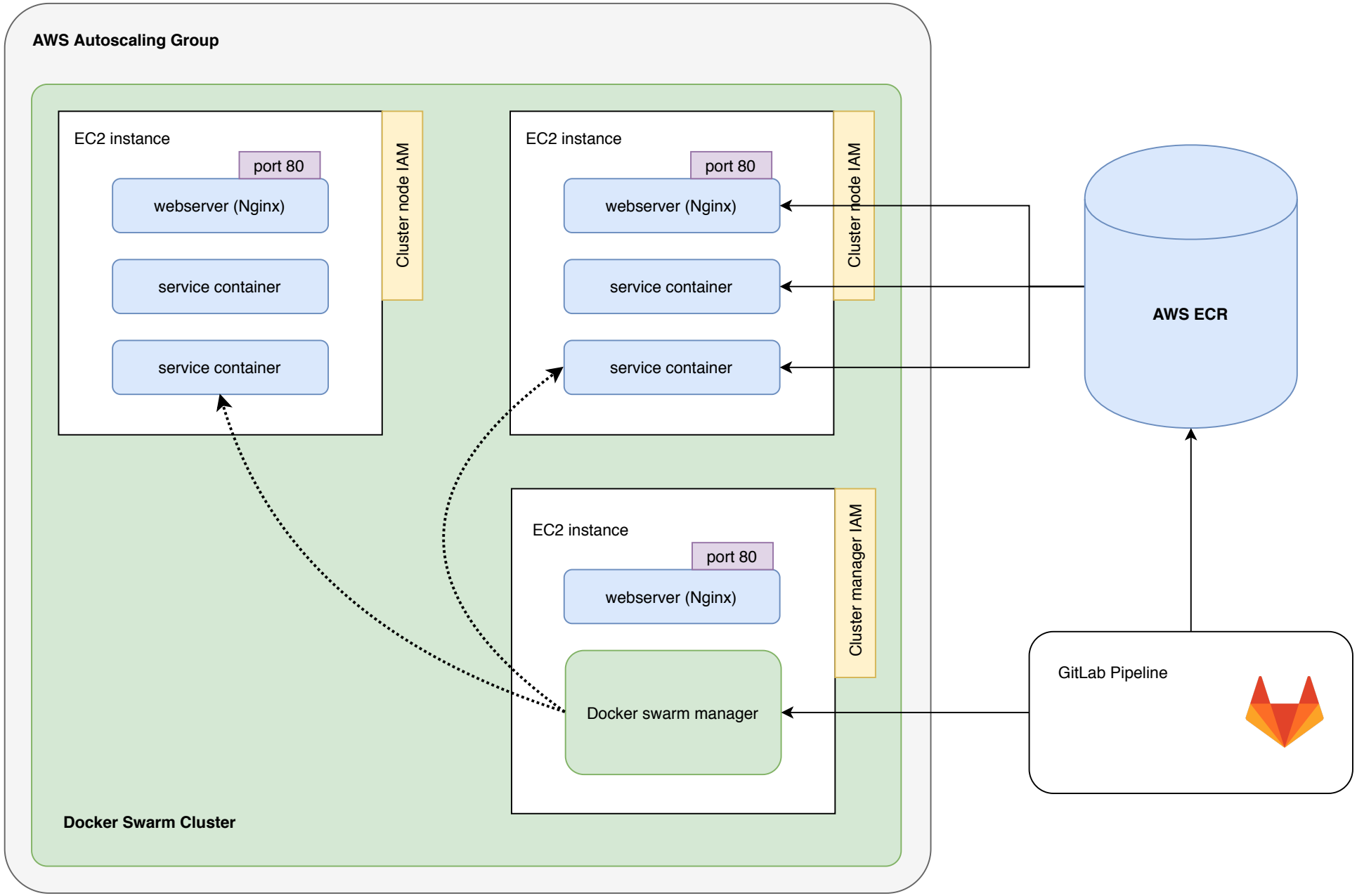
The set of needed commands should be placed at EC2 UserData script to initialize node on EC2 start up.



Service Initialization / Update

Predefined Docker Swarm manager will be used to manage services initialization and update processes.

Ci/Cd pipeline is leading this processes by triggering swarm manger on manager instance as well as storing and updating new images to ECR.



Data Storing

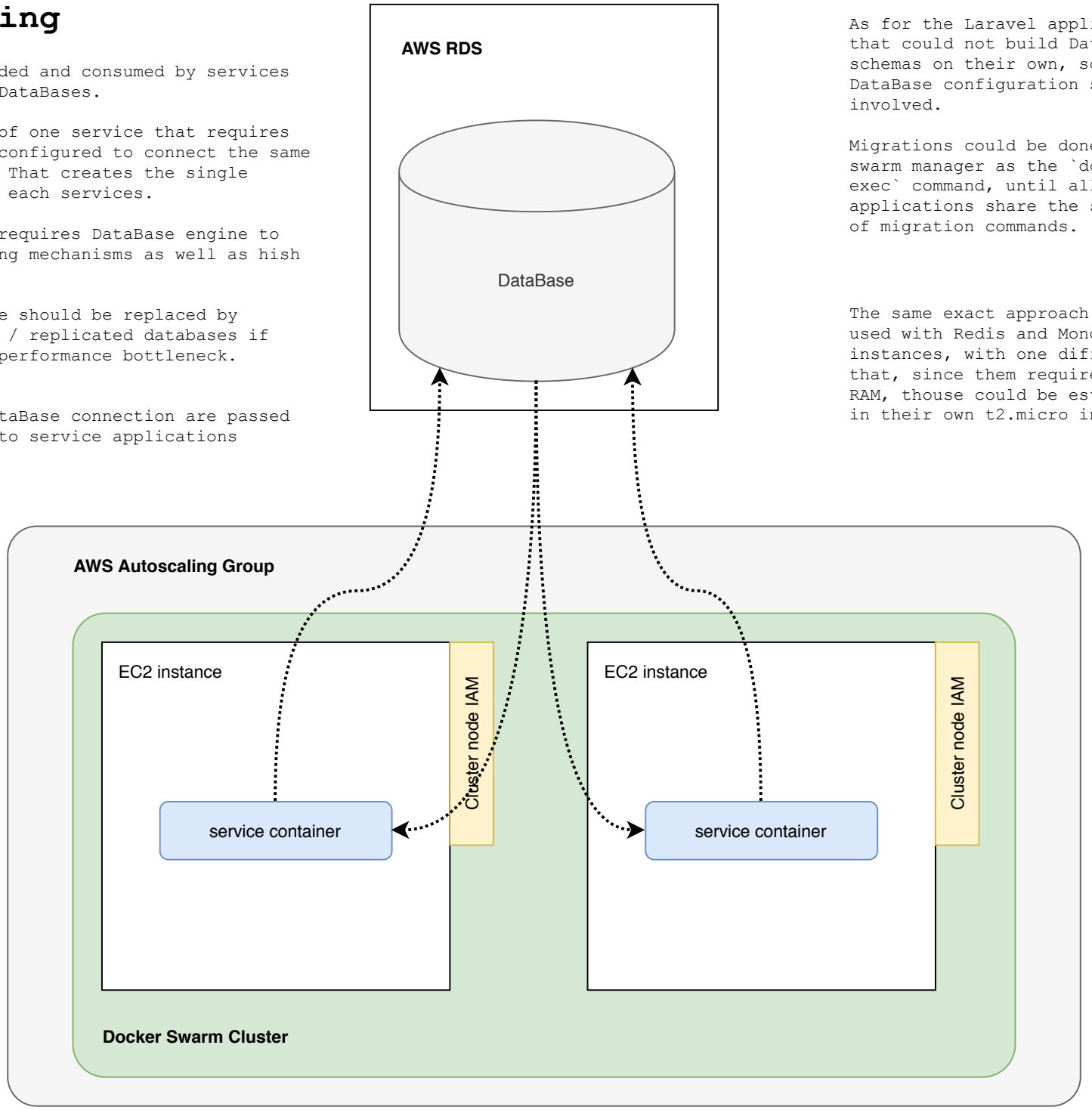
All the data provided and consumed by services are stored in RDS DataBases.

Multiple replicas of one service that requires the same data are configured to connect the same DataBase instance. That creates the single source of true for each services.

But this approach requires DataBase engine to use kind of blocking mechanisms as well as hish I/O efficiency.

Later this DataBase should be replaced by cluster of sharded / replicated databases if data usage become performance bottleneck.

Credentials for DataBase connection are passed as env parameters to service applications containers.



As for the Laravel applications, that could not build DataBase schemas on their own, some manual DataBase configuration should be involved.

Migrations could be done through swarm manager as the `docker exec` command, until all applications share the same set of migration commands.

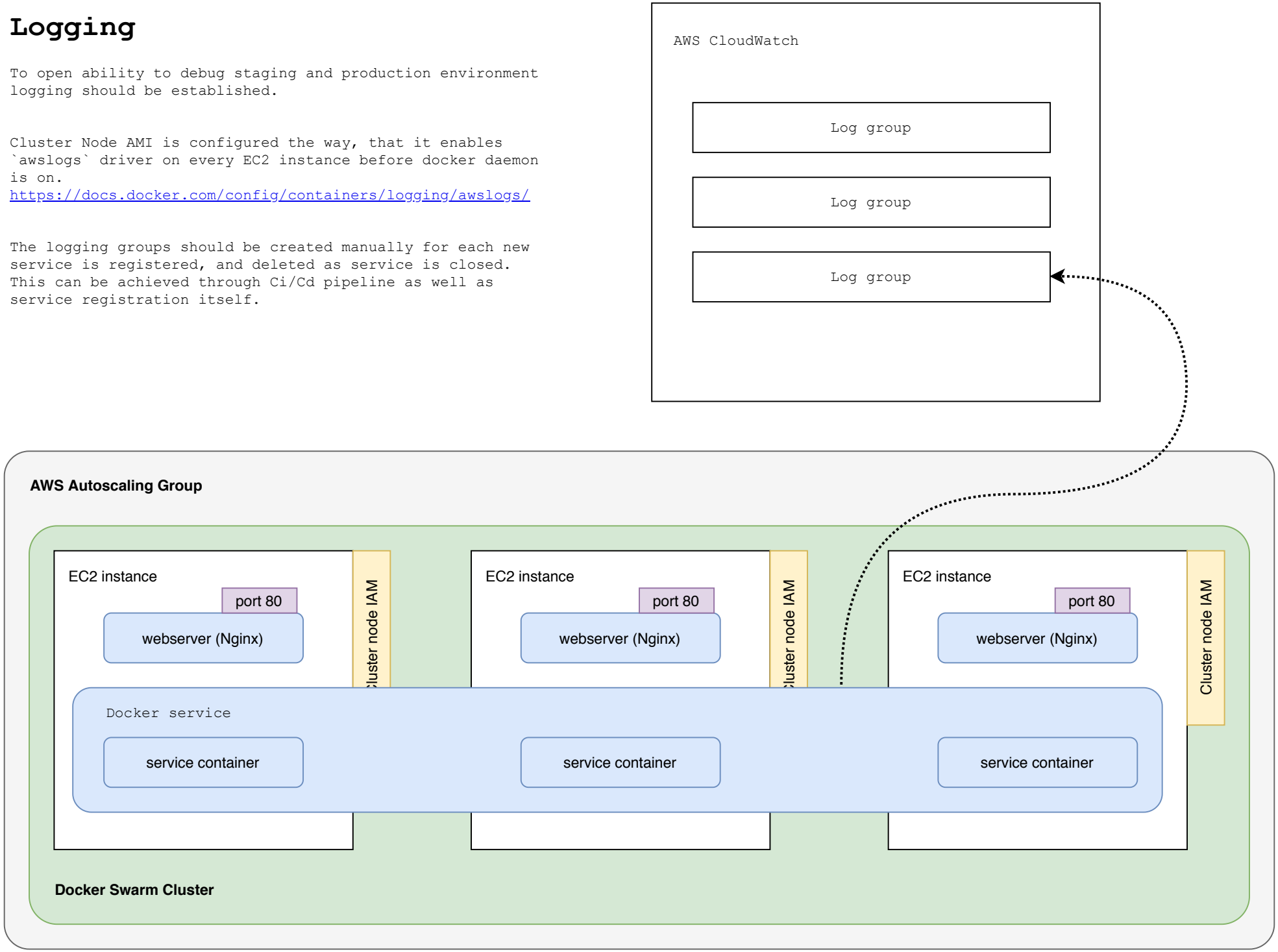
The same exact approach could be used with Redis and MongoDB instances, with one difference that, since them requires only RAM, thouse could be established in their own t2.micro instances

Logging

To open ability to debug staging and production environment logging should be established.

Cluster Node AMI is configured the way, that it enables `awslogs` driver on every EC2 instance before docker daemon is on.
https://docs.docker.com/config/containers/logging/awslogs/

The logging groups should be created manually for each new service is registered, and deleted as service is closed. This can be achieved through Ci/Cd pipeline as well as service registration itself.



Conclusion

This approach of the cluster structure implementation covers next set of questions:

- Routing;
- Scaling;
- Service init and update;
- Data storing;
- Logging.

This is the minimum set of requirements to the cluster structure on the initial stage of
production and development environment establishing.

However, this set is not enough to produce high-load zero-down time scalable environment.
So that this approach should be modified later with respect of new technical requirements.

Questions that are not covered in this approach:

- Performance metrics implementation and cluster visualization;
  (AWS Auto Scaling is responsible only for instances scaling, but how about replication of services containers?)
- Scaling of management nodes;
  (For now it's look like special AMI could be created to allow fast docker swarm manager initialization)
- Cluster nodes redistribution due to auto scales;
  (Fresh instances will run no containers except webserver until new containers will be started.
- AWS Cloud Formation Stack preparation;
  (It's possible to wrap all needed manipulation to Cloud Formation Stack, that process of cluster init can be done in no time)
- Data Backup;
- Configurations Backup;
- Usage of application load balancer upon multiple regions;
  (How to make request be process by the nearest docker container to requester?)
- DataBase clusterization;
- Frontend applications storage and serving;
  (This approach serves like a giant API, but what about frontend apps? Somehow they also have to be optimized).