Design Document
Project Name: All is Whale

Author: Mark S. Maglana <mark@maglana.com>

Introduction

This document briefly discusses the design of the All is Whale project, particularly the application components, the environment architecture, and the security considerations.

Application Components

The application is composed of the following components:

User Interface

The user interface serves as the frontend of the application, interacting with the end users. In the UI, users are able to send and receive messages belonging to a conversation. The UI interacts with the next component, the API, to send and receive these messages. The UI is available at https://ui.whale.kubekit.io.

NOTE: Due to time constraints, the UI is not able to send messages to the API. It is, however, able to retrieve conversations based on a conversation ID.

API

The API provides the business logic of the application and is responsible for serving requests from the UI. The API is available at https://api.whale.kubekit.io however, the root path does not provide anything. If you want to test it directly, use the swagger UI at https://api.whale.kubekit.io/docs instead.

It's important to note that because both the UI and the API are in different domains, CORS needs to be configured for them to be able to communicate. Thus, the API has been configured to allow requests originating from https://ui.whale.kubekit.io. This is done in the file api/quipper/main.py.

Database

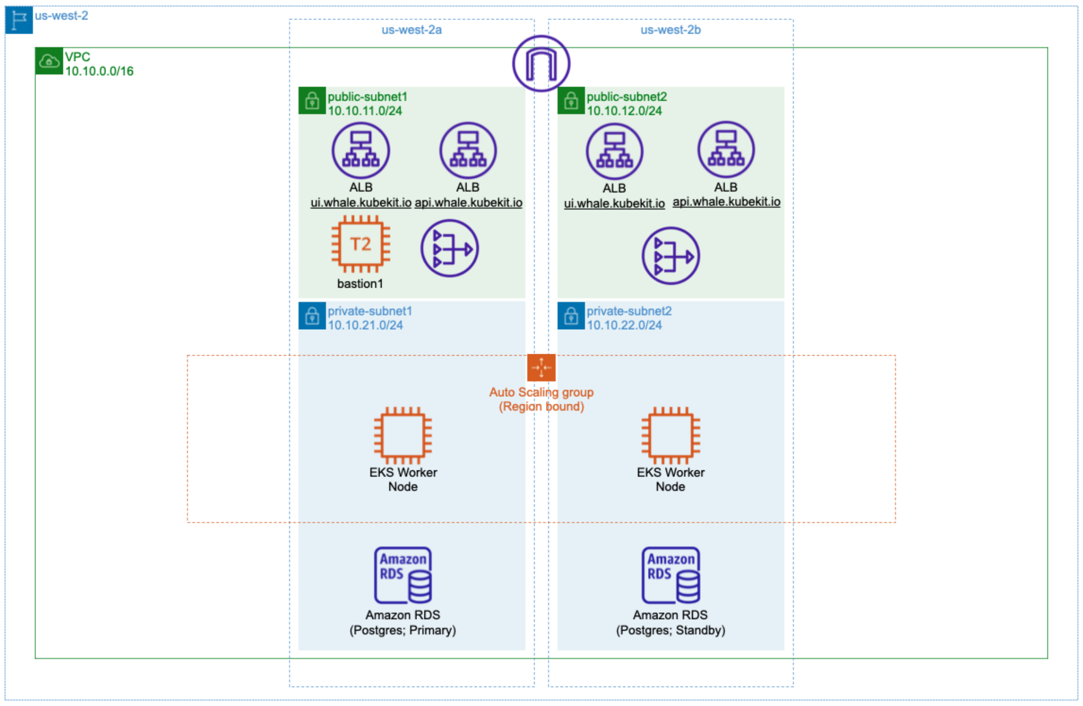
In order for the API server to persist the data that it receives, it needs to use a database. We have elected to use RDS Postgres here so as to minimize the administrative load while ensuring the durability of our data. With RDS, we are easily able to put up a Primary and Standy database in different AZs

Prometheus

To monitor the health of our services, we will use prometheus within the cluster. At the onset, we have elected to use the helm charts at https://prometheus-community.github.io/helm-charts since it is easy enough to deploy. However, we should re-examine the possibility of making this more robust in the future since not much testing was done during this project on this deployment of Prometheus.

Environment Architecture

For this section, we will use the following diagram as reference:



We will be deploying our cloud resources across two availability zones to reach a certain level of reliability. However, if we want to ensure zero to minimal downtime in case the region goes down, we should start thinking about deploying a standby cluster in another region.

Most of our services with the exception of Postgres will reside in EKS, specifically the worker nodes that are depicted above. They have been placed in private subnets for security purposes and managed by an autoscaling group to ensure that we always have enough workers to handle workloads.

Naturally our nodes and workloads may need to reach outside of the private subnets to be able to do their work. Thus NAT gateways have been placed in the public subnets and the routing table has been configured such that all traffic going to addresses other than those belonging to the VPC are allowed to go out the Internet Gateway.

Security Groups have also been put in place such that the worker nodes are able to access the database in the RDS instances which are also placed in the private subnets.

To allow for debugging, a bastion server was put up in the public subnet with the author's public key as the only entry in the node's authorized_keys file.

Finally, the UI and the API each get their own highly available (AWS default) L7 load balancer via the AWS ALB service. Both LBs have been configured to route http to https and have valid TLS certificates provided by Let's Encrypt. We will discuss how the latter was implemented in the next section.

ALB and TLS

To make the management of the load balancers easier, we used the AWS Load Balancer Controller. This allowed us to easily instantiate them via annotated Ingress resources in Kubernetes which are automatically linked to the Service fronting the Pods.

To automatically generate the certificates for both components (UI and API), we use cert-manager which is also triggered by the properly crafted Ingress resource. Specifically, one that has the tls stanza.

A downside to this combination is that while the cert-manager is able to produce the TLS key and cert and store it as a secret in the application namespace, ALB can only read secrets from AWS Certificate Manager. To get around this, a script was written in scripts/configure-tls-resources such that it copied the TLS key and cert from the Secret resource generated by cert-manager over to ACM and then update the associated Ingress with an annotation pointing to the ACM secret's ARN.

Source Code

This submission is accompanied by source code which is responsible for provisioning the infrastructure (via Terraform), kubernetes manifest files for defining the kubernetes resources, and actual code for the UI and API services. For more information please see the source's

README.md file. If you would like a quick video demo, here's an unlisted YouTube video for your viewing pleasure.