QUEENSLAND UNIVERSITY OF TECHNOLOGY

ASSIGNMENT 2: CLOUD APPLICATION INDIVIDUAL REPORT

CAB432 – Cloud Computing

Jeremiah Dufourq, n9960651
25/10/2020

1. Statelessness, Persistence and Scaling

Please refer to the architecture diagram which is overleaf.

SentiStock+ has two levels of persistence, would be considered to be stateless, and scales with the load applied to analyzing multiple stock tickers. As a quick overview, the user hits the application load balancer which then points to the scaling pool. In this scaling pool is a (vue+quasar) client and (express+node) server. This then uses a redis cache and postrges database for persistence.

Regarding persistence on a deeper level, the application uses postgres through AWS's managed RDS as a first level of persistence. Every request that the user makes on the client which contains some sort of state is stored within this database. For this application this is particularly in regards to storing the current stock tickers, historical analysis and also the tweets. If one of the instances in the scaling pool fails, then the data for which the user currently was working with would be saved, as it is persited on the psotgres database. For a second level of persistence, the application uses elasticache (which uses redis). The user uses this data whenever they reload the application. Because the cache is updated whenever the user sends a request to analyse more data, and hence persist more data in the database, in this sense the cache will never become stale.

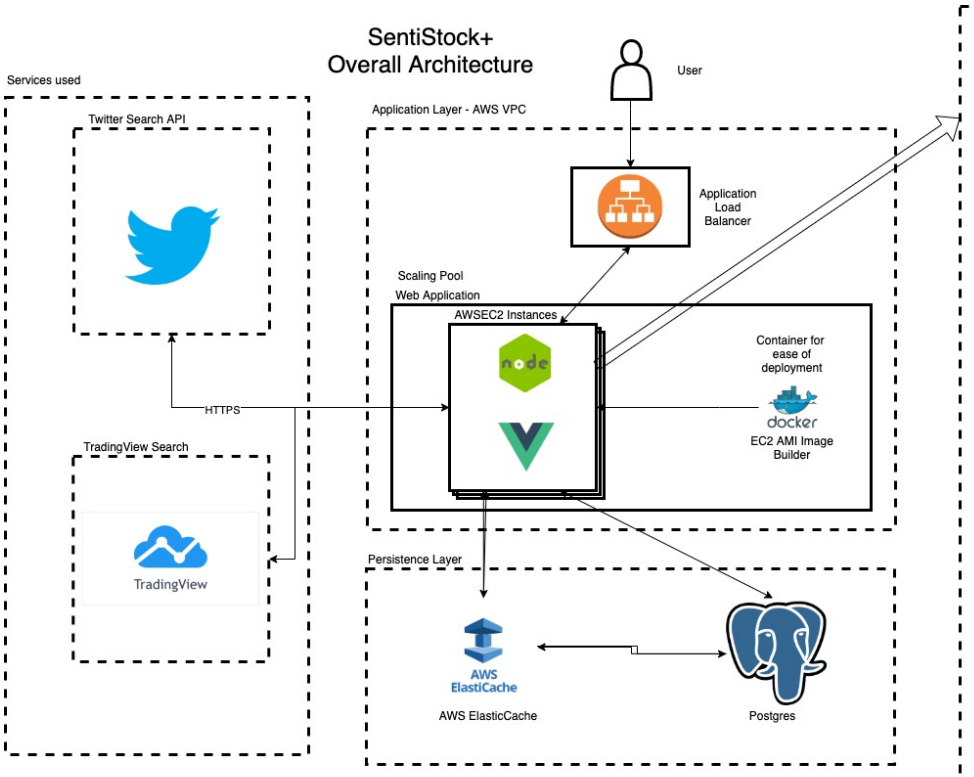
On the client side, there could be considered an aspect of statefulness with respect to passing data between components. This is particular apparent in storing the current tickers, and the data to use for the charts within vue variables. There was no way in getting around this as the charts need the data on the client side to update. However, one way the application did mange to mitigate some of this is that whenever the chart is update, it uses the data within the database.

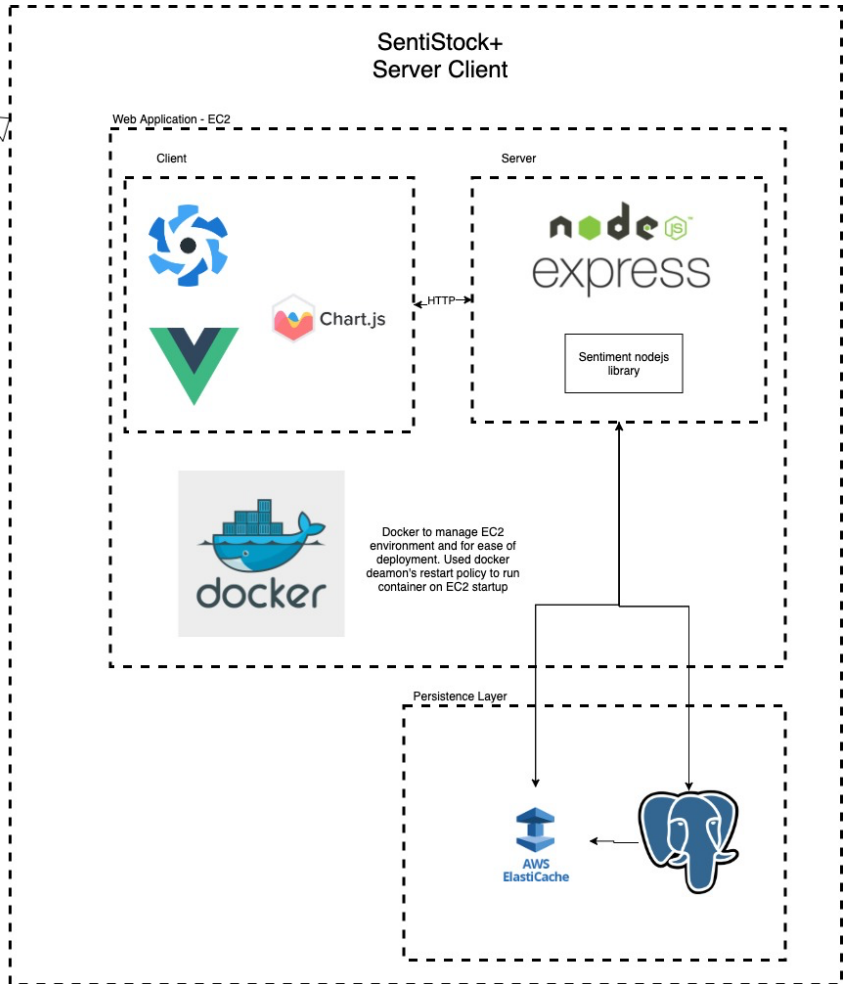
For this application, an SQL database was a bit of an overkill. This is because we have no relations which we needed to mange in the application. A better alternative would have been a noSQL option, such as MongoDB which would have been feasible by using DynamoDB provided by AWS. Because the application did not require consistency due to a low volume of transactions, and did not have any relations, a noSQL database would have been the preferred option. If the application was to require a user login, and a live twitter feed, then potentially the current architecture would be valid due to consitency and relational requirements.

Regarding scaling, the approach taken by the application was to use the CPU usage as a metric to increase the number of instances in the scaling pool. The application had this limit set to 20% . This metric was used becase the application requires more resources to calculate the sentiment for a larger number of tickers, and hence if you increase the number of tickers for which the application has to process, then it will increase the resources on that instance, and will require more instances. This made sense for this application because there was a high number of computational tasks required to perform the sentiment analysis.

An alternative solution would be to use the number of requests as well as the CPU resources. I could see this being feasible if this application was used at scale. For

example, if there are users associated with the application (say 100k + users), the application will need to scale with the amount of requests, rather then the CPU usuage.


SentiStock+
Overall Architecture
User
Services used
Twitter Search API
Application Layer - AWS VPC
Application
Load
Balancer
Scaling Pool
Web Application
AWSEC2 Instances
Container for
ease of
deployment
EC2 AMI Image
Builder
HTTPS
TradingView Search
TradingView
Persistence Layer
AWS
ElastiCache
AWS ElasticCache
Postgres


SentiStock+
Server Client
Web Application - EC2
Client
Server
Chart.js
HTTP
express
Sentiment nodejs
library
docker
Docker to manage EC2
environment and for ease of
deployment. Used docker
deamon's restart policy to run
container on EC2 startup
Persistence Layer
AWS
ElastiCache

2. The Global Application

When considering this application on the global level, there would be some modifications that would need to be required for it to scale. However, these modifications would also come with the changing of the requirements for the application.

Persistance:

Currently, the application uses a SQL database with cache to manage persistence. This application would be considered stateless. As described before, for this application a noSQL database would probably have been better because we don't have entity relations, and we do not have the need for consistency and could rely on eventually consistency. However, as our application gets larger, this is where having an SQL database might come in handy (even though they do not scale horizontally but rather scale vertically). These are the reasons why; 1) They support consistency → this would be beneficial if the application was to include some live tweet feed or some other high transactional task, 2) They support relational entities → this would be beneficial if the application was to have users on the platform and profiles for their logins 3) They are 'cheaper' compared to noSQL → This is trivial, whilst SQL Databases are cheaper than noSQL databases, you do have the problem that requires resources to be added as the application scales. So in this sense, it is really application specific. However on the whole, they are cheaper then noSQL databases.

Caching + CDN:

With regards to caching, the application currently has caching in the form of AWS elasticache. However, if we think about this application on a global level, the region our application is hosted in, and how it will be delivered will make a big difference if our application requires a high volume of users. This is where something like a CDN service as well as edge caching could help. To extend on our application, we could provide a CDN service such as cloud front to serve out web application close to where the requests are being sent from. This service will also provide the edge caching needed to make sure that our end customers are getting the application served to them with the lowest latency possible.

Scaling:

The current application scales with regards to the CPU usage. On a global scale, this is not a stable policy to scale on and we should think about something a bit more robust. If you think about the application with 100s of thousands of users, each performing a certain amount of requests, then a CPU scaling policy is not going to cut it. What is required is a certain combination of policies, or just to consider the application to scale based off the number of requests which are sent to the server, as this is a better indication of the load. The changes that would be made is to make the scaling policy dependant on

the number of requests. If a server also exceeds a certain CPU threshold, sure we can also scale based off this. However currently, 20% is a waste of resources and hence will cost money in the long run. Therefore, it should be proposed that the application scales on requests, and if the servers reach more then 80% of their load.

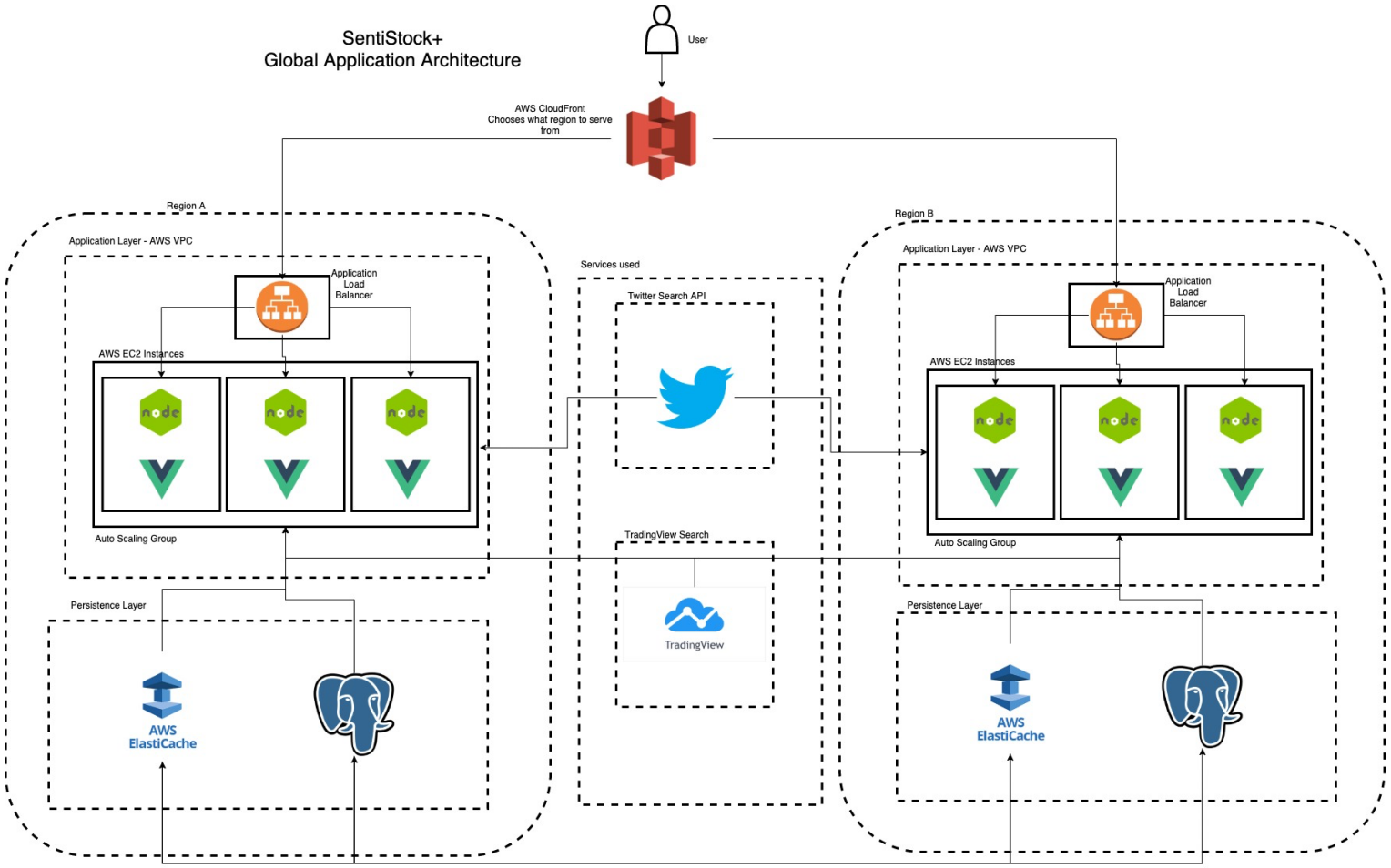
Proposed architecture:

The proposed revised architecture consists of the following. We will have the browser which the user will access the application from. They will then send a request to CloudFront (a managed CDN by AWS), which will go out and find the closests location to the user to serve the application from. This will still have the same application load balancer, however the CDN adds the benfit of serving the content closer to the user, and decreasing the latency. In terms of persistence, for our current application, it is most likely better to use a noSQL database (as our current application does not require consistency and relational entites). However, I am proposing a scalable global web app, and with that in mind, the scalable global web app will have a SQL database with redis cache. This is because we would like to run a high amount of relational entity transactions with consistency. With the scaling policy, we will scale based on the number of requests and CPU usage above 80% to not waste resources.

Security Comments:

Somew comments about security. There are two main things for security which we need to consider. The first is authentication for our API endpoints. Currently, the application has not way to authenticate a user which is connecting to our API endpoint. This means that anyone with the access to the current URL for the servers will be able to modify the data in the database if they know the API endpoints. This is a major security flaw. One way to address this in the global application would be to build authentication into the server. This can be done using Oauth 2 and JWTs (for stateless) or some other authentication service.

The second vulnerability is SQL injection attacks. The current application mitigats this by using nodejs libraries to insert the query statements into the pool. This should not pose a large threat on the global application level.


SentiStock+
Global Application Architecture
User
AWS CloudFront
Chooses what region to serve from
Region A
Application Layer - AWS VPC
Application Load Balancer
AWS EC2 Instances
Auto Scaling Group
Persistence Layer
AWS ElastiCache
Services used
Twitter Search API
TradingView Search
TradingView
Region B
Application Layer - AWS VPC
Application Load Balancer
AWS EC2 Instances
Auto Scaling Group
Persistence Layer
AWS ElastiCache