Paul Kenaga

Professor Alam

20 April 2023

CS 470 Final Reflection

YouTube Presentation: https://youtu.be/LhWk68C1qnc

The skills I have learned in this course to become a more marketable candidate in Software Engineering are moving a MEAN stack web application from an on-premise setup to a cloud environment, containerizing applications using Docker and Docker Compose, utilizing various AWS cloud computing services such as Lambda and API Gateway, and implementing security protocols in the cloud to safeguard data and resources. My strengths as a software developer are perseverance and curiosity. While at this moment I have limited professional experience with software development, I am confident in my drive to succeed, learn, and put forth my best work. Professional experience and expertise will come in due time. The types of roles that I am prepared to assume in a new job are junior software developer and junior web developer; a position where I can apply my studies and that offers a learning atmosphere. This course will help me reach my professional goals by providing me with the opportunity to work on the AWS platform and gain exposure with the microservices it offers. I feel that the future for many businesses is cloud development and it is valuable for me to have these skills in my toolbox because it will appeal to employers and, hopefully, make me a more effective employee.

To produce management efficiencies and scale my web application in the future I would utilize some of the features of the microservices I am already employing as well as other microservices available in the AWS ecosystem. To handle scale and error handling I would create separate protocols for Amazon API Gateway, AWS Lambda, and Amazon DynamoDB. For API Gateway, I would establish throttling limits for my methods to limit the rate of requests and regulate processing. Even though this microservice has the benefit of autoscaling to handle incoming requests, it is important to optimize my API's performance and protect my resources by limiting the number of requests that are handled within a certain time period. I would also set a burst rate to limit the number of concurrent requests my API can handle and cache results to requests so that they are stored client-side and can be access quickly without contacting the back end again. For error handling, I would create HTTP responses that errors from my Lambda functions can route to so that my API can respond with an appropriate HTTP status code and response body.

AWS Lambda and Amazon DynamoDB will also automatically scale to meet my usage needs. However, there are concurrency limits in AWS Lambda that limit the number of concurrent execution environments where my Lambda functions can be invoked. To prevent any of my Lambda functions from using all the available concurrency I can configure concurrency settings so that reserved concurrency is allocated for each Lambda function. To handle errors, I would set retry limits and a maximum event age as to control flow and configure a dead letter queue (DLQ) that prevents message loss in the case invocation of my Lambda functions fail. The DLQ would make it easier for me to inspect and correct errors. Try-catch statements can also be added to my functions so that they respond to errors smoothly.

To handle scale in DynamoDB I would use AWS Application Auto Scaling which is a service that automatically adds or removes capacity to meet real-time demand. This would allow my web application

to better handle sudden spikes in traffic by increasing provisioned read and write capacity for my database tables. It would also ensure that I am not using or paying for more capacity than I need. Similar to AWS Lambda, I would configure error retries and an exponential backoff algorithm to handle errors. This would allow requests to be attempted again but also increase the wait time in between them. The logic in try-catch statements within my Lambda functions would serve to handle errors as well for failed database requests.

I would use the AWS Cost Explorer tool to predict the cost of using AWS, but I know that this tool would take some time to be effective because the forecast is based on my past usage. The more time that goes by, the more accurate the forecast will be. Another way to facilitate predicting the cost would be to set limits on the requests for my microservices so that I can essentially set a maximum spending amount. Since containers are constantly running, I believe their cost is more predictable. However, they are not nearly as cost-effective as serverless architecture. The reason why it is difficult to predict the cost of AWS is because of its elasticity and pay-as -you-go model; you only pay for the resources you use and resources are dynamically allocated to match your needs. AWS microservices only run when they need to so the cost is going to align with usage while the cost of containers aligns with the amount of space being occupied on a vendor's server.

The deciding factors in plans for expansions would involve traffic volume, application performance, and future application functionality. Perhaps users will be required to sign into the web application in the future to make it more secure. This would require another table for usernames and passwords. The pro of expansion is that it is very easy to provide resources when they are needed. Another pro of expansion is that AWS CloudWatch makes it easy to monitor resources in a single view and create custom views and alarms for resources. A third pro is still only being charged for the resources you use, which means that expansion is possible without racking up high upfront costs. Cons that would be deciding factors in plans for expansion are increased potential for vendor lock-in, more dependence on internet connectivity, and increased security concerns. Even when there is a way to prevent vendor lock-in, the difficulty to migrate off the cloud or to another platform would increase as an application expands. When you have more resources and data in the cloud, you become more reliant on your internet connection to access them. This can create unpleasant situations such as when your internet service provider is experiencing issues. Additionally, expansion in the cloud requires more trust that your resources and data are protected and will be available.

Elasticity and the pay-as-you-go model play a significant role in the decision making for planned future growth. It removes some of the risks associated with expansion in an on-premise environment because accrued charges are based on usage and resources are allocated dynamically. You do not have to purchase a whole new server or pay for its maintenance. Nor do you need to employ or pay for more resources than you need at any given moment. Future growth can be focused on development, not where or how the application will run. Elasticity and the pay-as-you-go model make the AWS platform, and serverless architecture in general, cost efficient and facilitates decision making for planned future growth.