Angel Yang, cy2389@columbia.edu
James Xue, jx2218@columbia.edu
Kevin Ye, ky2294@columbia.edu
Bryan Yu, by2181@columbia.edu
Ishan Guru, ig2333@columbia.edu

Final Project Report: EverTweet

Topics in Software Engineering Spring 2017

Project Deliverables

Documentation on how to set up and run each application are included in each application's README.md.

EverTweet Application
Monolithic:
https://github.com/kevinye1/EverTweet
Service-Oriented with Servers:
https://github.com/nyletara/EverTweet-SOA
Service-Oriented without Servers:
https://github.com/nyletara/EverTweet-Lambda

Framework Application
All Architectures: https://github.com/nyletara/AppDevFramework

Technologies Used

We are using a variety of tools to bring our project to life, while incorporating all the aspects of each of our midterm papers. This section illustrates the tools used for each phase of our project:

- Development
  - Twitter API - to access the timeline content of Tweets for the input username
  - IBM Watson API - to conduct sentiment analysis on text using the Natural Language Understanding service
  - Python Flask - for each individual server/service that we build including the orchestrator
  - AWS Lambda + API Gateway - for each individual server/service that we build in the serverless application model

- Architecture
  - Monolithic: all services will be in the same server
  - Service-oriented with servers: each service will be deployed on its own EC2 instance (AWS)
  - Service-oriented without servers: serverless with AWS Lambda functions (AWS)
- Testing
  - Employ test driven development practices
  - Write tests using Python's unittest framework
  - Utilize a continuous integration tool Travis CI
- Experiment Tools
  - Git: version control, can use git diff to track change in code versions)
  - DateTime and Time modules: to track performance of the different application architectures

Experiment Results

Development Experience

Overall, the development of the three architectures for this experiment took considerable work over the course of four weeks, and the development of the framework, as well as documentation, took additional time.

The monolithic architecture was the simplest to develop, especially since most of us had experience developing Python/Flask applications before. The Flask structure was set up in about one day, working correctly with the Twitter and IBM Watson APIs. The front-end making an AJAX call to the Twitter/IBM functions, and rendering the data using Highcharts took a few additional hours to develop and refine. Overall, the experience was painless; documentation for the APIs was decent and the documentation for Highcharts was good.

The SOA architecture was slightly more complicated, and during the development process we ran into some issues with organizing the structure into an orchestrator, sentiments, and tweets. Modularizing the code this way actually helped keep each file shorter and made it easier for us to understand where things went, so in the long run the maintenance and debug process should be shorter. Development took a few days to complete.

Finally, the lambda architecture had even more moving parts, and we ran into some issues with cross-origin requests, data not being in the correct format, and deploying correct versions. Ultimately this took several days to set up and refine.

Response Time

	# Tweets				
	10	20	50	100	200
Monolithic	4.768	9.262	24.452	44.027	88.463
	5.066	8.968	22.388	45.608	88.972
	4.837	9.245	22.92	44.589	89.904
Average:	4.890333333	9.158333333	23.25333333	44.74133333	89.113
SOA	5.193	8.829	21.613	42.809	85.751
	4.758	8.798	21.42	42.657	86.046
	4.938	8.831	21.103	43.503	85.139
Average:	4.963	8.819333333	21.37866667	42.98966667	85.64533333
Lambda	6.838	10.899	28.238	53.293	100.89
	6.472	11.342	27.949	54.494	101.383
	6.514	11.984	28.102	55.389	104.321
Average:	6.608	11.40833333	28.09633333	54.392	102.198

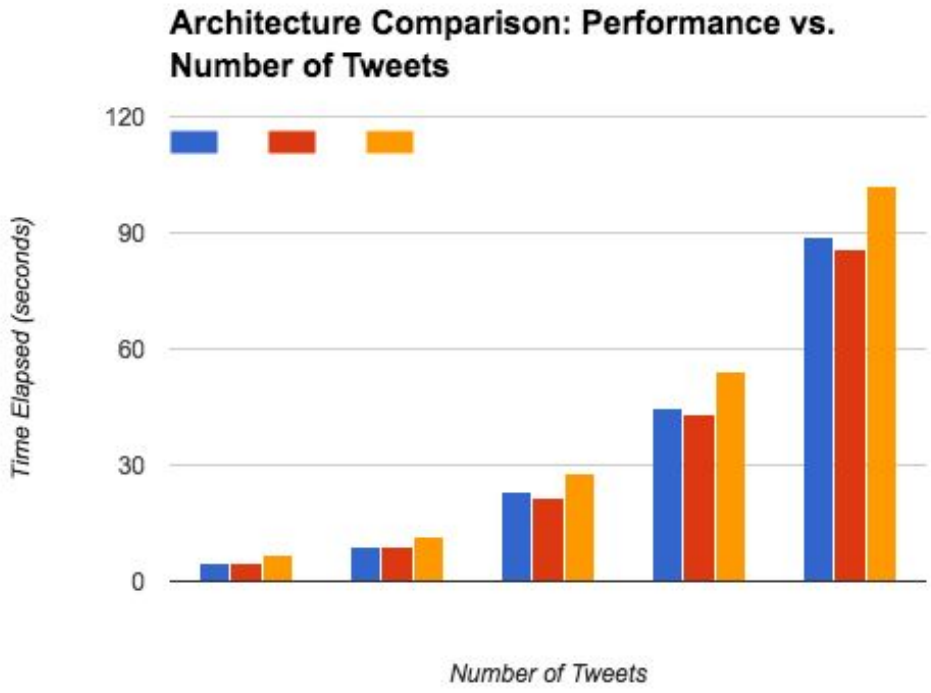

Figure 1: Average Time Elapsed by Number of Tweets (10, 20, 50, 100, 200) for each of the three architectures (blue = monolithic, red = SOA, yellow = lambda)

The response times for each of the architectures is displayed in table format and in chart format above. From these results, we saw that on average, SOA performed slightly better than monolothic architecture, but also, the lambda architecture performed the worst out of the three.

Documentation

As expected, documentation for Service-oriented and Lambda architectures were quite vague. Firstly, because Service-oriented architecture is a style of code design rather than a guided framework or library, online documentation usually either only described the code design principle or used specific development platforms that did not substantially help us. Because of this, our team took the time to develop the Service-oriented architecture with Python and Flask and created our own documentation for it. In the provided Github link (https://github.com/nyletara/AppDevFramework), we hope that our skeleton framework will assist future COMS4156 students in seamlessly transitioning from monolithic to SOA to Lambda architecture or begin develop immediately within an architecture.

AWS Lambda documentation was also quite difficult to parse through. Our team had a general concept of serverless service-oriented architecture, however had never used AWS Lambda before. Reading through the documentation and picking through the needed aspects of AWS Lambda took a considerable amount of time, but it was well worth it. We hope that our framework will save COMS4156 students time and difficulty by giving them a concrete code framework to build from.

Cost Predictions

Ultimately, we predicted that using AWS Lambda is much cheaper than hosting your own servers or even using an EC2 instance.

In our application, our lambda function request took 300 microseconds to run. According to the AWS Lambda pricing, each tier is priced per 100 milliseconds at $0.20 per one **million** requests. This is a very cheap option for developers, and also includes bonus free tier seconds per month.

Overall, we predict that using AWS Lambda is much cheaper and more scalable. Even though EC2 instances are popular, having a live server that's running, even if it's not being used, means that the developers have to pay for it. This could be impractical for certain teams.

Code Reuse

Overall, there was a surprisingly large amount of code reuse. From a Monolithic architecture to Service-Oriented architecture, the functions within each service were simply partitioned into separate Flask servers exactly as it was. The only code changes were that each service had additional code to initialize it as a Flask server and that the functions had to be refashioned into Flask routes. Through our provided SOA framework, there should be very minimal code changes required to switch from monolithic to service-oriented architectures.

From SOA with Servers to Serverless Service-Oriented architecture with AWS Lambda, there was again a substantial proportion of code reuse. Because they are both Service-Oriented, code for each service was almost identical, with the only addition being event handlers which are required by AWS Lambda. One interesting note about using AWS Lambda is that packages must downloaded into the local codebase so that AWS Lambda has the proper packages to run functions on the cloud. While our own application seems to have an immense amount of additional files, these are simply downloaded packages, such as watson_developer_cloud, that each service requires. Within our Lambda framework, users will be advised to download their own packages into each service as well, so while there is a lot of additional code, the actual code reuse from SOA is high, while strenuous effort is kept minimal.

What We Learned

Angel: One of the most interesting takeaways I had was how converting an application from monolithic to service-oriented affected code quality and readability. The idea of service-oriented and single function code has definitely been stressed in past classes, but actually going through the code refactoring process really helped me understand and solidify the

benefits of service-oriented architectures, especially when working on a team where everyone is developing different features. Additionally, setting up the framework for the summer ASE students to work with also helped with understanding how the technologies and architectures we were using actually worked and how to generalize what we were doing to be applicable for other contexts. A more technical thing I learned was how to implement AWS lambda functions, which was particularly interesting because it allows service-oriented applications to be deployed without servers.

James: This project was a great opportunity to learn more about both using the cloud and implementing service oriented-architecture. Previously to this project, I have developed on Google Cloud Services, so switching to the deploying on AWS was an interesting experience, especially concerning setup and deployment. I also found it very interesting to learn about service oriented architectures and switching between monolithic, service oriented and finally, serverless lambda architectures. Functionally, they are all the same, but in terms of scalability and code reuse, they can range considerably. Additionally, I really enjoyed using and learning about Twitter's Tweet API and IBM's Watson Natural Language Understanding API.

Kevin: Learning how to switch a service-oriented application from server-based to serverless was very challenging but interesting. The structure of a serverless service-oriented application was completely new to me, and it was interesting to see how the structure of the application can affect things like security (i.e. with the use of handlers). Additionally, I wrote about continuous integration in my paper, but setting this up with Travis showed me additional benefits that I had not considered previously, such as logging how long it takes to run the test suite. Lastly, I enjoyed learning about the innovative technical aspects needed to build our specific application, particularly the use of the IBM Watson Natural Language Understanding API Service to conduct sentiment analysis on the text.

Bryan: This project was extremely interesting and new for me, especially because I've never had the opportunity to work with the cloud or service oriented architecture before. Being able to work

with these different frameworks and architectures was a great learning experience. I was particularly pleased with learning how to setup and deploy different AWS EC2 instances, as well as the functionality of lambda functions. Learning about service-oriented architecture was also very interesting because I've never thought about separating the structure of my functions before. Separating an application by its services into different servers seems like a great new idea to me. Finally, learning how to effectively set up a framework for others to use was a fun challenge, because I had to think back on my COMS4156 days and try to imagine a framework that I could effectively use.

Ishan: The most rewarding part of the project was learning about lambda functions and how to switch from a service-based service-oriented architecture to a service-less service-oriented architecture. Even though this was something I had not covered in my paper, the efficiency in regards to resource usage makes lambda functions a very applicable and important thing to learn. Prior to this project, I had never employed continuous integration during the development process so I found CI particularly interesting. Especially when developing in such a large team setting, this was especially helpful in making sure that the changes we make to the code does not change the intended behavior of the program. Any code push that caused a build fail would automatically notify our team members so we could fix the bug before other team members develop with the bug.

Unexpected Problems / Deviation from Plans

In our original plan, we wanted to formulate experiments that implemented and compared the deployment of a monolithic and service-oriented application as EC2 instances on two major cloud service platforms: Google Cloud and Amazon Web Service. Following the feedback received from our project presentation, we realized that such a comparison would be trivial and decided to pivot the focus of our experiments. Instead, we decided to choose one cloud service provider, Amazon Web Services, and compare different deployment architectures on that (monolithic, services with servers as EC2 instances, and services without servers as AWS Lambda Functions). With this change, we were able to delve into

more complex comparisons than just the usability of cloud platforms.

Member Roles

We divided the team to be primarily in charge of the following roles: project manager, system architect, UI guru, external API guru, and testing guru. These roles deem the team member as the primary decision maker in relation to their role in case of disagreement within the team, not as the sole responsibility of the member. Everyone still are developing the application and testing suite together, and the roles are more so to pinpoint a go-to person regarding specific portions of the application. The scope of each role is outlined below:

Project Manager: Angel Yang

- Managing the timeline and schedule of the project
- Making sure that the team on track for the timeline
- Organizing periodic check-ins with team members

System Architect: Ishan Guru

- Decide on how features are split into services
- Decide and maintain project and framework structure

User Interface (UI) Guru: James Xue and Kevin Ye

- Decide on and develop the UI for application
- Create structure of front-end and back-end interaction

External API Guru: Bryan Yu and James Xue

- Find external APIs needed for application
- Create wrappers for external APIs
- Write service that calls + handles API requests

Testing Guru: Kevin Ye and Bryan Yu

- Build and maintain the testing suite
- Setup automated testing