Mike Dito
Joe Mogannam
Edson Hernandez
CS 385
Final Project Report

Our application is a cloud-based, scalable, RESTful payroll API hosted on the Google Cloud Platform. It includes several custom components, as well as a few Google Cloud components. Users of our service send their data to us as JSON objects, and we send data back to them as JSON objects. Users are first met by a Load Balancer, which routes them to an available Application server. The Application server uses Google Cloud's PubSub messaging service to publish data to various topics that our custom components subscribe to. We created three custom components: Translator, Accruals, and Delivery. In order for our data to persist, we needed a database. Since our data is structured, and we need the ability to scale horizontally, we opted for Google's Cloud Spanner database.

For our service to be scalable, the Application server needs to have the ability to scale horizontally. Because of this, we needed to have a Load Balancer distribute our users to various Applications that are in service. We decided to use the Load Balancing service that is built in to the Google Cloud Platform. To enable Load Balancing, we needed to first create a managed instance group. Managed instance groups are just a group of identical instances that are built from an instance template. So, we created an instance template for our Application server. This template is built from an image of our Application. A great feature that is a part of managed instance groups is that If any changes are made to the instance template it uses, these changes can be rolled out to all instances in the managed instance group.

We also enabled Google's Auto-Scaling feature. This feature will monitor the CPU level of our Application instance, and when it gets too high it will spin up as many Application instances as it needs to reduce the load. When the traffic dies down, this feature will also scale our application down.

The Application server runs Flask, a web micro-framework for Python. We opted for Flask because we were all familiar with Python, and Flask allows our application to be flexible. By flexible, we mean that it supports many outside libraries, and we aren't restricted to doing things in a particular way. Since the Google Cloud Platform has an extensive Python API Client, we could easily use their libraries. For this architectural layer, we needed to use PubSub, a messaging service, to publish incoming data for our other components to receive. The Flask server has 5 routes that users can hit, these are: "insertCompany", "addEmployee", "submit", "accruals", "deliveryRequest", and "paystub". The first two allow our users to join our service by submitting data about their company, and employees. When we receive their data, we publish it to the topics, "insert-new-company" and "insert-new-employee". The "submit" route allows our companies to submit a batch of timesheets for their employees. These timesheets get published to the "insert-timesheets" topic. The "accruals" route triggers a process that will calculate a company's pay stubs. The data for this route gets published to the "calculate-accruals" topic, which our Accruals instance subscribes to. The "deliveryRequest" route triggers a process that will fetch an employee's pay stub and bring it to our front-end application. When this route is hit, the data it receives is published to the "delivery-request" topic, which our Delivery instance subscribes to. Now, since their pay stub is at the front-end, they can hit the "paystub" route to retrieve their pay stub.

So, we're publishing all these messages using PubSub, but where are they going? Who subscribes to them? And what happens to the message? Our Translator component subscribes to "insertCompany", "addEmployee", and "submit". When it receives messages on these topics, it parses the message (JSON), generates a Cloud Spanner SQL-like insert statement, and then inserts the data into our database. We decided to make this component so that our Application tier only had to deal with incoming requests. By using PubSub, we were able to offload all of our incoming data to Translator, which handles most of the writing to the database.

The “accruals” topic is subscribed to by our Accruals instance. Accruals has one job, calculate accruals, and insert them into our database. This frees up Application to continue just handling incoming requests.

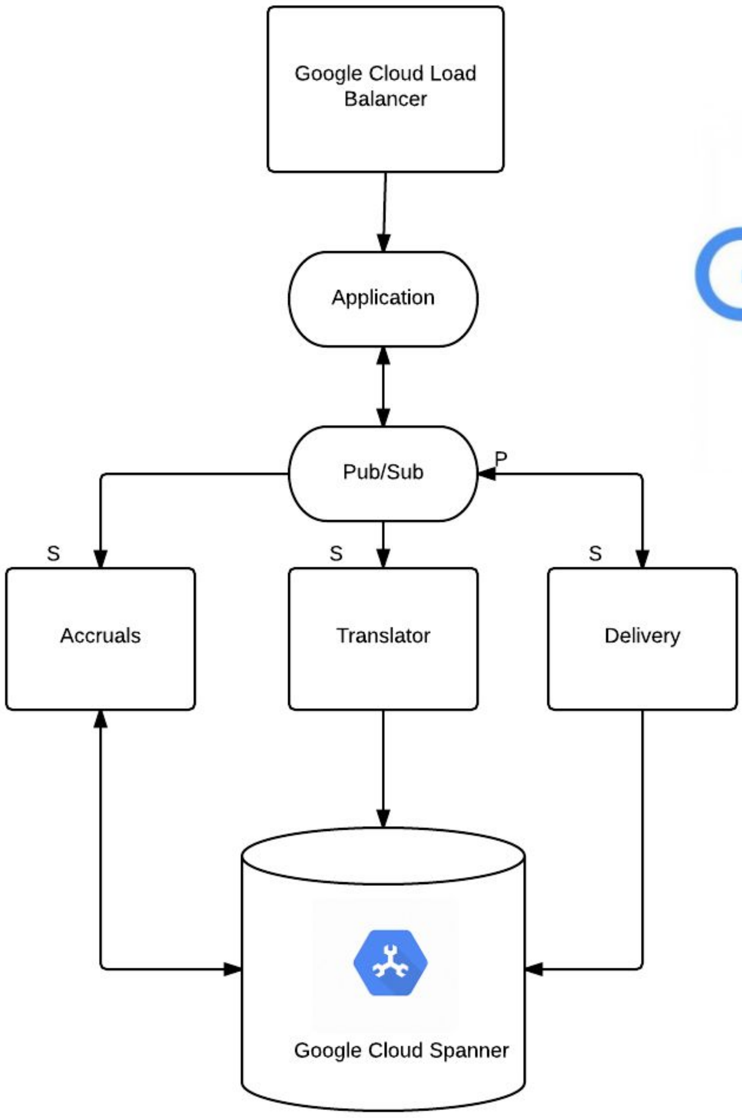
The “deliveryRequest” topic is subscribed to by our Delivery instance. It’s sole job is to query the database for pay stubs and deliver them to the front-end Application. The decision was made to create this Delivery instance to keep with the theme of having each component take care of one aspect of the service.

The database that we needed for this service had to be both relational and horizontally scalable. This led us to discover Cloud Spanner, Google’s high-performance, database built for the cloud specifically to provide the benefits of a relational database with the ability to scale horizontally. Spanner has strong consistency, high availability, automatic replication, and is horizontally scalable.

We faced quite a few design challenges throughout this project. The first of which was designing our components. We had to think about what they would do, and how they would interact with the rest of our components. We had to design the JSON objects that our users would be required to send, and design the database to work with that JSON. Since our data is structured, we knew that we needed a relational database, but we needed it to be scalable too. We first used Google SQL, but ran into trouble when we tried to scale. That’s when we found Cloud Spanner, and solved our database scale problem.

Originally, we tried many Python Kafka libraries to publish and subscribe messages, and we were successful in testing it on one instance. But we ran into trouble when we began tried to have two instances communicate. If instance A published to topic T and instance B subscribes to topic T, B would not receive any messages. To solve this problem, we discovered Google's PubSub messaging service. Their service makes communication between instances in a project very easy.

Another scale problem we encountered was how should Translator, Delivery, and Accruals be scaled? Because Translator subscribes to messages from Application, if we had more Translators, that each subscribed to the same topic they would each be inserting identical data into the database. To solve this, custom topics could have been made for each Translator. Each delivery instance would need to deliver to its own Application, and Accruals would calculate pay stubs requested by a specific Application. One last problem to address is how we store the pay stubs once they are delivered to the front-end. They're currently sitting in a Python dictionary, which will not persist if the server happens to fail. To solve this, we could try using a cache.


Google Cloud Load Balancer
Application
Pub/Sub
P
S
S
S
Accruals
Translator
Delivery
Google Cloud Spanner