FRANCIUM TECH

High Volume background processing with Kubernetes and GKE

Premise

For one of our premium clients, we had to develop a highly scalable background processing engine based on Sidekiq. This is an intelligent system that parses documents and has an embedded machine learning component in it. The number and the size of the documents can vary drastically and the system should be efficient enough to scale for this change. In such a case, parallelism is the only solution for consistent delivery. So, we had decided to spawn multiple sidekiq jobs. But, the real challenge here lies in the confluence of these many modules and in managing them.

Needs and Tools

For the aforementioned platform, we needed the following items.

- **Machine Learning** - This is the core. We decided to use **Django** for this.
- **UI** - For visualizing the output we wanted a new UI component here. As **Rails** is one of our favourite frameworks, we opted it.
- **Relational DB** - We needed a centralized relational DB that also should be exposed to other related apps in the ecosystem, and we chose **Postgres** for this.
- **Background Jobs** - For parallelization, we need a few nodes for background processing. For managing multi-processes, we chose **sidekiq**.
- **Caching** - For centralized caching we needed an efficient caching server. We chose **Memcache** for that.
- **Containerization** - We had been using **Docker** for a long time both on our local and some production environments. So, we continued to used that as well.

Docker Compose?

We initially explored using docker-compose. We had been using docker-compose for on boarding new joinees to the team and for most of our development needs. We even had tried our hands in deploying docker-compose onto production. But, It generally requires several manual steps for setting it up in the production server including a shared directory for containers and secrets management. But we actually needed an orchestrator in this case. For instance, if one of the containers is stopped, an orchestrator has to start the container in no time in addition to providing scaling opportunities.

Why Kubernetes?

Few interesting things that influenced us to move to Kubernetes were

1. Rolling updates
2. Very High Availability
3. Extreme scaling of Sidekiq processes (upto 1000 Jobs/sec)

This type of features are difficult if not possible to achieve in other deployment tools like Capistrano without manual intervention.

Google Kubernetes Engine (GKE)

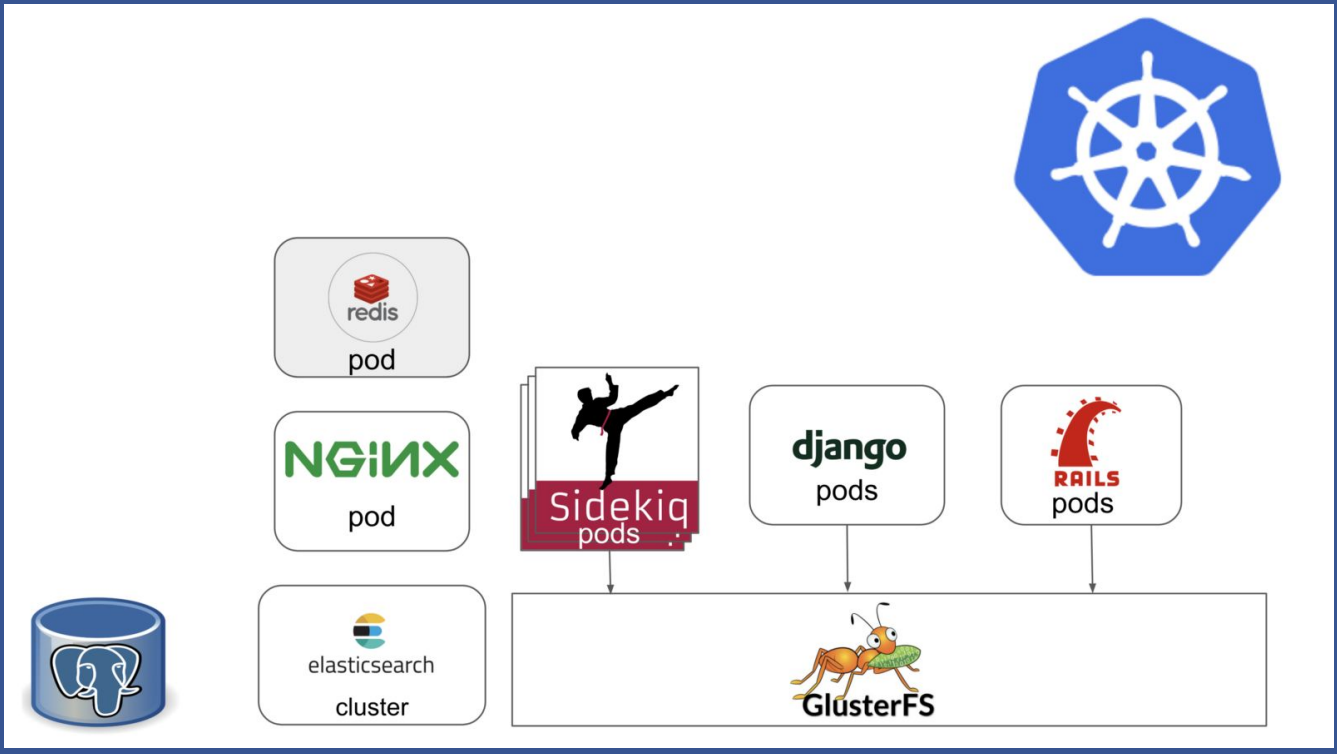
Since most of applications are deployed on Google Compute Engines, Google Kubernetes Engine was an obvious choice for us.

GKE is a managed, production-ready environment for deploying any containerized applications. It improves developer productivity, resource efficiency, automated operations, open source flexibility and hence accelerates the project delivery. It enables rapid application development and iteration by making it easy to deploy, update, and manage the applications and service

The Architecture

We sat with the DevOps team for the deployment architecture. We decided that we need to run the following containers

- One Redis container
- One UI container
- Two(at-Least) Sidekiq containers
- One Django container
- A cluster of GlusterFs containers



The machine learning stuff was supposed to be run in Django based application that exposes the calculated data in the api. Also we decided to keep PostgreSQL, the database server in Google Compute Engine as a separate server outside the cluster as this is used by other projects as well. GlusterFs containers are required as data is shared between most other containers through the file system.

Considering GlusterFs

Since the directory based data is shared between Rails app, Sidekiq and Django app, we had to initially resort to running all these containers in a single pod. This means that we can't scale it as file uploaded in a pod will not be available in a pod running another instance of sidekiq. This forced us into considering GlusterFs as a first class citizen and create GlusterFS cluster and getting each of the containers running into its own pod. This is a recommended and best practice too. The configuration for these containers are of Kind (in Kubernetes terms) Deployment In General, Deployments are scalable to greater limits.

Setting up GlusterFs was still a large pain in Kubernetes cluster. So keeping that running outside of the Kubernetes cluster was initially the solution. After we configured Glusterfs with the Kubernetes and running each containers into separate pods worked out great for us.

Issue with StorageClass

After running all these, the application worked well for us. However the application was very slow. On review of the configuration files and GKE dashboard, we noticed that the cluster was still making use of Google's default pd-standard storage in PersistentStorage kind, like shown below.

```
1  apiVersion: storage.k8s.io/v1
2  kind: StorageClass
3  metadata:
4    name: gluster-storage
5  provisioner: kubernetes.io/glusterfs
```

After we defined this StorageClass and referencing gluster-storage in all the other deployment configurations, it worked like a charm and performance was really good. This is one of the early mistakes we did while setting up the QA environment.

Few more nuances we need to be aware of while deploying to Kubernetes is that Kubernetes Engine doesn't yet support SSD root disk in Node Pool. This is especially strange in 2018. In our benchmarks,

the application deployed in Kubernetes is still slow by a minor margin. This could be due to the use of standard storage in GKE NodePool.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sidekiq
spec:
  selector:
    matchLabels:
      app: sidekiq
  strategy:
    type: RollingUpdate
  replicas: 2
  template:
    metadata:
      labels:
        app: sidekiq
    spec:
      containers:
      - image: gcr.io/mers-200418/bitbucket-tracker:28fedaf455041990ee0aa1bda68fc406908ec9cc
        name: sidekiq
        imagePullPolicy: Always
        command: ['/bin/bash']
        args: ["-c", "bundle exec sidekiq"]
        resources:
          requests:
            cpu: '.6'
            memory: '1.2Gi'
        volumeMounts:
          - name: gluster-claim
            mountPath: efs
      volumes:
      - name: gluster-claim
        persistentVolumeClaim:
          claimName: gluster-claim
```

Following is the one of the deployment kind file for Sidekiq. Names have been changed of course. We could ask for Kubenetes controller to allocate CPU and RAM as mentioned in the resource below. For Sidekiq, I needed 1.2 GB of minimum RAM. If you don't specify the replicas size, its value is considered 1 by default.

Google container registry

We are also using Google container registry for building the container. So every time we pushed the code, Google container registry automatically pulled the code and builds the container. This makes it deployment speed slower than Capistrano based deployment. Google doesn't seem to maintain cache for bundles or caching the layer for us automatically until you use cloud builder. We also did a thorough study on speeding up the building of the containers.

Also every time we push the rails code, we also build Nginx container that serve as a reverse proxy for the rest of the cluster.

It is also recommended to only expose ClusterIP Service for resources in the cluster except for the Nginx and That's exactly what we have done for the security reasons.

The amount of machines needed in NodePool is greater than deploying all the apps in a single environment. This is what we used to do for staging. However if you reduce the request in the configuration, it would just work alright.

We haven't mentioned about Liveness and Readiness probe and setting up SSL and allotting a static ip for the application for the pods and graceful shutdown of sidekiq and other applications and commands to deploy, redeploy, rollback and monitoring in this article which necessitates another set of articles.

Conclusion

Now, It has been months since we pushed our application into Kubernetes and we enjoy the returns as it is painless to manage and deploy them. There are about 10 queues processing different kind of jobs in Sidekiq. At a point we achieved about 35000 jobs a minute post Kubernetes implementation. In addition to this, we also deployed the Ruby on Rails and Django applications, Redis store, PostgreSQL Database and Nginx running on Kubernetes. This also helped us in reducing the overall working cost to its half of what of we spent earlier and along with greater benefits of Kubernetes.