1. Definição da arquitetura

Um sistema com necessidade de alta disponibilidade, acarretam vários problemas, qual linguagem, Framework, cloud e qual o custo que a empresa terá que desembolsar para manter toda esta arquitetura operando com toda da disponibilidade possível.

Tendo os fatores acima, foram testados os Frameworks Java Spring Boot, Nodejs com Express e Sequelize e Serverless retornando um “Hello word”

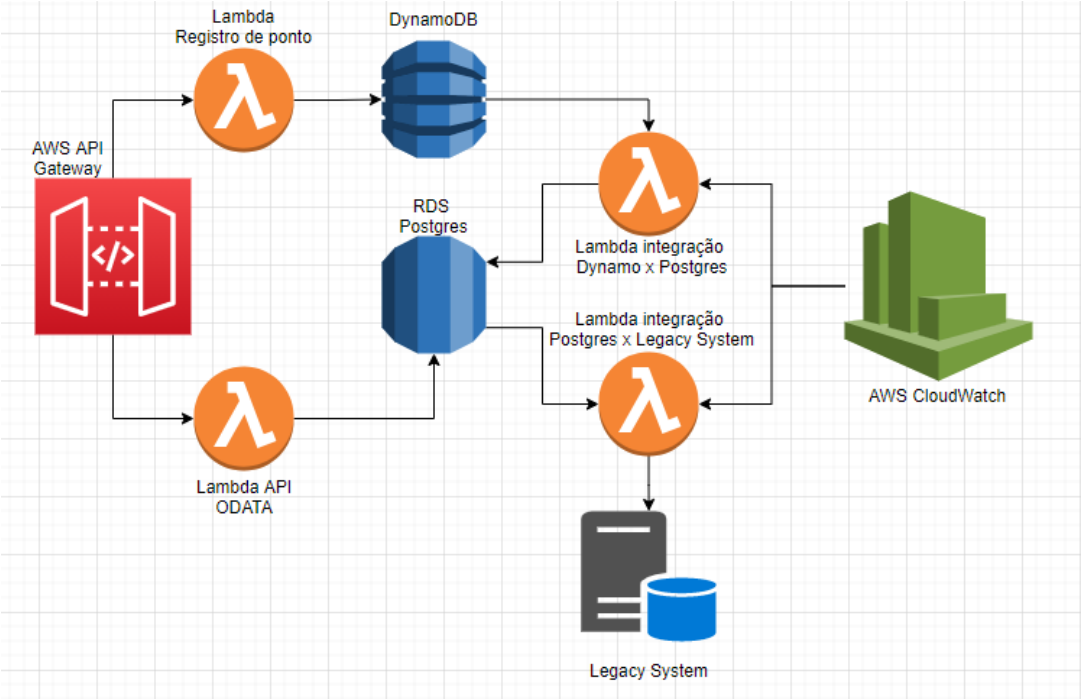
- Com Spring Boot foi possível processar um milhão de requisições em pouco mais de 5 minutos, através de uma aplicação monolítica em uma máquina no EC2 da AWS em modo grátis.
- Com Nodejs foi possível processar um milhão de requisição com um pouco mais de 7 minutos, através de uma aplicação monolítica em uma máquina no EC2 da AWS em modo grátis.
- Com Serverless publicado na AWS e no Google Cloud foi possível processar uma milhão de requisição em menos de 20 segundos.

Com estas informações obtidas acimas o Serverless quando publicado em um cloud é muito mais eficiente em relação a custo-benefício de que qualquer Framework, com o ponto positivo que não há necessidade de controlar o escalamento horizontal quando vertical, o que é feito automaticamente pela própria arquitetura do cloud.

Agora com a tecnologia de desenvolvimento definida é necessário definir o modelo de persistências dos dados, foram elencados o MySql com a engine Myisam e o DynamoDB. O Mysql com Myisam é um banco relacional que não possui ACID, permitindo várias inserções sem Locks em sessões, atendo somente ao número de conexão. Quanto o DynamoDB é um banco de dados NoSql auto escalável.

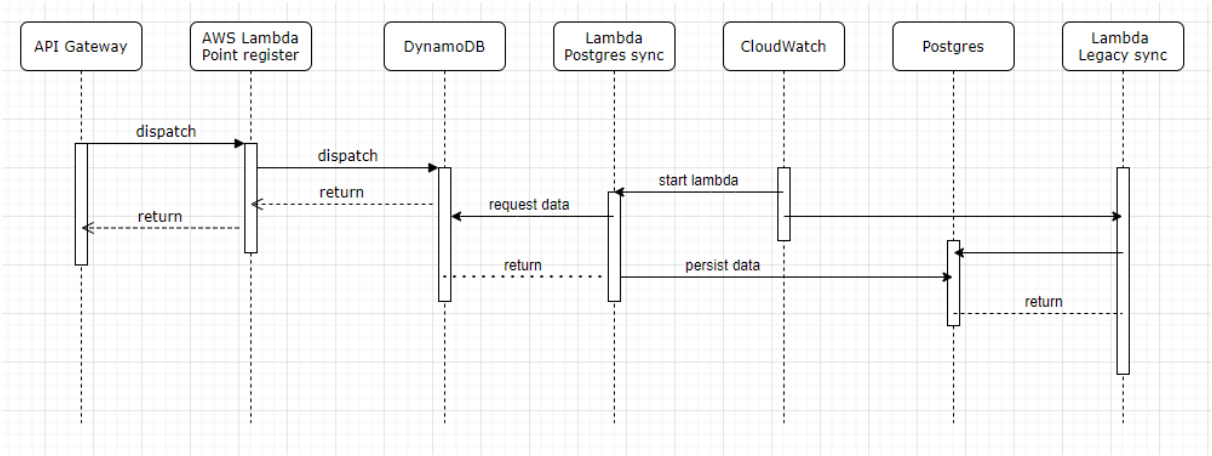
Com estas informações foi escolhido o DynamoDB devido a ter toda a sua arquitetura gerencia pela AWS e com poder de auto escaling, e assim vai ser possível armazenar todos os apontamos não importam os números de usuários simultâneos. Porém os bancos NoSql tem vários problemas e então os dados serão replicados a um Banco relacional Postgres, por um serviço assíncrono.

Agora com a tecnologia e armazenamento definidas, é necessário definir o cloud onde foi escolhido a AWS por se demonstrar mais robusta na questão de Lambdas, onde a arquitetura ficou como a representação gráfica abaixo.



Ou seja, foi criada uma Lambda para processar as requisições provenientes do API Gateway da AWS, onde esta lambda grava os dados em uma tabela do DynamoDB, outra Lambda com uma API ODATA recebe requisição do API Gateway para consultas dos apontamentos e status de integração com o sistema legado. E a partir do CloudWatch são chamadas de tempos em tempos Lambdas para sincronizar as bases DynamoDB x Postgres, quando levar os dados de apontamento para o sistema legado.

Com o diagrama de sequência abaixo podemos ver como toda a integração funciona.



2. Heurística de integração com o legado

Para a integração ser o mais rápido possível com o sistema legado foi criado um algoritmo inteligente que aprende o número máximo de processamento atual da API legado, ou seja, caso o sistema legado esteja sobrecarregado a integração irá funcionar de forma lenta ou em casos de o legado estar altamente disponível a integração irá acontecer mais rapidamente, com o menor número de erros.

A heurística foi montada através da seguinte logica, um contador de chamadas em aberto e um limite de chamadas, quando o limite é atingido e não houver erros, o limite é aumentado em 1.5% e em caso de existências de falhas na integração o limite é subtraído 10%. As requisições que falham são integradas novamente na próxima iteração.

Podemos ver na imagem abaixo a integração ocorrendo, onde foi iniciado com 6 requisição e conforme as chamadas ocorreram foi diminuída para 5 chamadas em paralelo.

```
Open requests: 1 Max open requests: 6 Errors: 0
Open requests: 2 Max open requests: 6 Errors: 0
Open requests: 3 Max open requests: 6 Errors: 0
Open requests: 4 Max open requests: 6 Errors: 0
Open requests: 5 Max open requests: 6 Errors: 0
Open requests: 6 Max open requests: 6.09 Errors: 0
Open requests: 5 Max open requests: 6.18135 Errors: 2
Open requests: 1 Max open requests: 5.563215 Errors: 0
Open requests: 2 Max open requests: 5.563215 Errors: 0
Open requests: 3 Max open requests: 5.563215 Errors: 0
Open requests: 4 Max open requests: 5.563215 Errors: 0
```

3. Conclusões

Com a arquitetura assíncrona implementada acima foi possível obter os seguintes resultados no número de chamadas à API de registro, integração com o Postgres e integração com o sistema legado.

Chamadas em 15 segundos	Processamento Lambda	Postgres sync	Legacy sync
5.000 chamadas	20 segundos	Cerca de 1.5 minutos	10 minutos
50.000 chamadas	22 segundos	Cerca de 12 minutos	3 horas e 37 minutos
100.000 chamadas	35 segundos	Cerca de 24 minutos	Não realizado

Com os resultados apresentados acima, foi verificado que a arquitetura desenvolvida consegue aguentar um fluxo gigantesco de dados, e em poucos minutos os registros de ponto já integrarão no banco relacional e disponível na API e em algumas horas disponível no sistema legado.

A recomendações seria deixar sempre habilitado o auto escalonamento da AWS para conseguir a melhor performance nos testes.

Quanto a próximos passos seria melhorar a lógica de integração com o sistema legado através de um algoritmo genético, para se utilizar da aleatoriedade para conseguir resultados melhores em um tempo mais curto do que uma simples estatística.

4. Fonte e ambiente

Todas as definições e requisitos para rodar a arquitetura local e em cloud estão no repositório https://github.com/jorgekg/point-record-serverless .