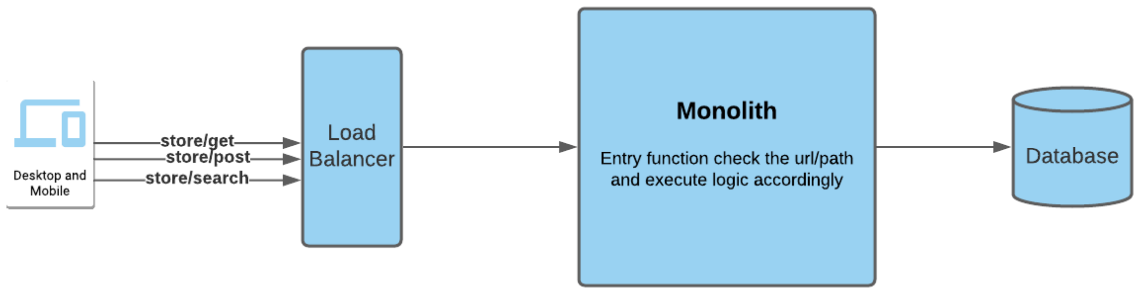
Mercado System

Current design:

Mercado e-commerce product is currently a monolithic application that is hosted on-premises.

Below is a high-level architecture for the current design



Monolith architecture is good in small-scale applications but it has the following cons for large-scale applications:

- Modularity is hard to enforce as app grows
- Scaling is a challenge
- All or nothing deployment because everything is contained in a single code base
- Leads to long release cycles because over the time the single code base gets bigger and all functionalities should be tested even not touched.
- So it’s slow to react to customer demand

Based on the above, Mercado system has the following technical issues:

- Scalability
- Low availability
- High latency

Solution

To have a scalable, high available and low latency application we will **refactor** and modernize it. The **application modernization** process will include three transformations at once:

1. Architecture: from Monolith to Microservices
2. Infrastructure: from On-Premises to Cloud
3. Delivery: from Traditional/Waterfall to DevOps/CICD pipelines

Splitting the application into loosely coupled collections of modular services that can be updated, tested and scaled independently. Each service will have its own code repository and delivery pipeline. Each service is containerized and deployed in auto-scale infrastructure.

So, to design a system architecture, let’s assume a scenario that has some use cases and will be modeled. These uses cases will be represented below in functional and non-functional requirements.

Functional Requirements

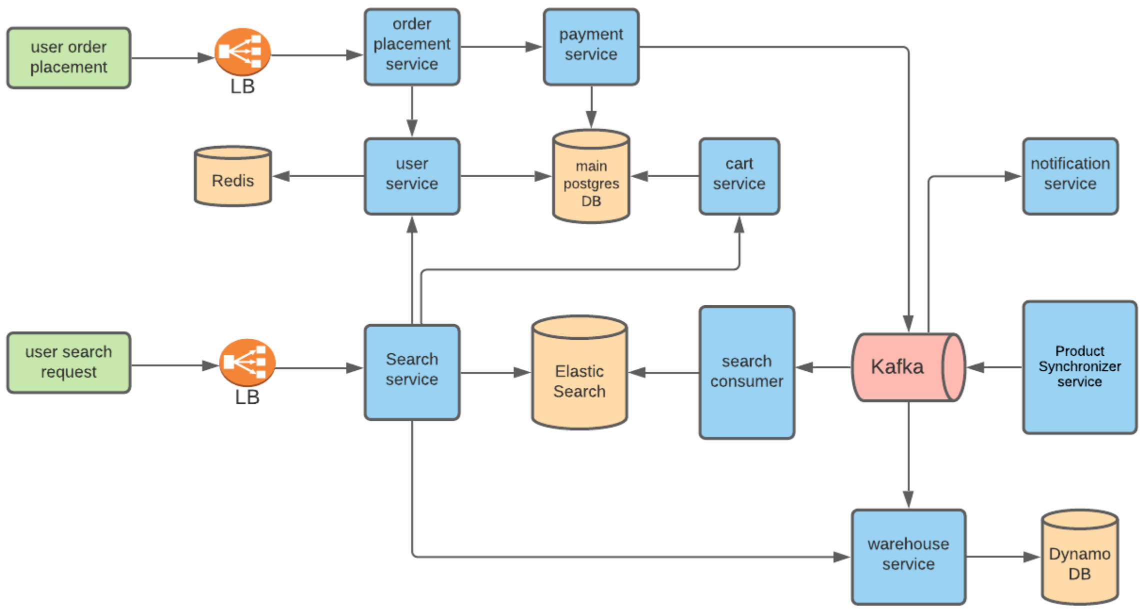
Mercado application has many functions, will focus here in a high-level architecture on the below basic features:

- The application has a store of products(Product Information Management system – PIM)
- ProductSynchronizer is a component that feed PIM up from an ERP
- Users can search for products
- Users can add/delete/update item in a cart
- Users can purchase items in the cart by making payments
- The application will send notifications to users after payments

Non-Functional Requirements:

- High availability
- Low latency
- Scalability

System Architecture Design:



- **Product Synchronizer service**: this service will fetch all the products from an ERP and publish it into a Kafka queue
- The **search consumer service** that subscribes to that Kafka queue will read the new updates from the queue and will add them into the elastic search database.
- When the user searches for a product, the **Search service** will be invoked and search results will be returned based on user preferences returned by user service
- **User service**: it provides API's to manage users info (profile, location, contacts...) and maintains Redis cache. If the requested data is not available in Redis, it will be fetched from

the **main Postgres DB** and update Redis DB. This cache DB will save user search preferences and user orders history. Cache DB is used to reduce the latency.

- User can select a product from the search results and **Cart service** is invoked to add the product to the Cart and it's saved into Postgres DB.
- A relational DB (**the main Postgres DB**) is selected to maintain the relationships between DB tables and enforce ACID.
- Once the user decide to purchase the product, the **Order Placement service** will be invoked and store related info in main DB then call **the payment service**
- After payment is done, an event be published to a Kafka queue to which **the Notification service** is subscribing. The notification service will send the proper notifications to users.
- Back to the **Product Synchronizer service**, below is the code to ready data from the source ERP and publish the read data into a Kafka queue

```
public class FetchFiles {
    public void readFiles() throws IOException {
        //Creating a File object for datasrource
        File directoryPath = new File("D:\\Demo");
        //List of all files and directories
        File filesList[] = directoryPath.listFiles();
        Scanner sc = null;
        for (File file : filesList) {
            sc = new Scanner(file);
            String input;
            StringBuffer sb = new StringBuffer();
            while (sc.hasNextLine()) {
                input = sc.nextLine();
                sb.append(input + " ");
            }
            System.out.println("File contents: " + sb.toString());
            //publish to Kafka queue
        }
    }
}
```

To handle a CSV product file including hundreds of thousands of rows I used java.util.Scanner to run through the contents of the file and retrieve lines serially, one by one. This solution will iterate through all the lines in the file – allowing for processing of each line without keeping them in memory.

- **Warehouse service** listens to Kafka to onboard new products and will expose APIs to add, update, and fetch products. It stores data into a DynamoDB because products related data will be unstructured where various types of products will have different attributes. This information is fetched and used by the search service to display whether a product is currently available or not. Below is a code for warehouse service :

```
@RestController
@RequestMapping("/api/v1/products")
public class ProductController {
    @Autowired
    private final ProductRepository productRepository;

    public ProductController(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }
    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Product create(@RequestBody Product product){
        return this.productRepository.save(product);
    }
    @PutMapping("{sku}")
    @ResponseStatus(HttpStatus.OK)
    public Product update(@RequestBody Product product,
@PathVariable("sku") String sku){
        Product dto =
productRepository.findOneBySku(sku).orElseThrow();
        dto.setDescription(product.getDescription());
        dto.setPrice(product.getPrice());
        dto.setQuantity(product.getQuantity());
        dto.setTitle(product.getTitle());
        return this.productRepository.save(dto);
    }
    @PutMapping("{sku}")
    @ResponseStatus(HttpStatus.OK)
    public Product get(@PathVariable("sku") String sku){
        return productRepository.findOneBySku(sku).orElseThrow();
    }
}
```

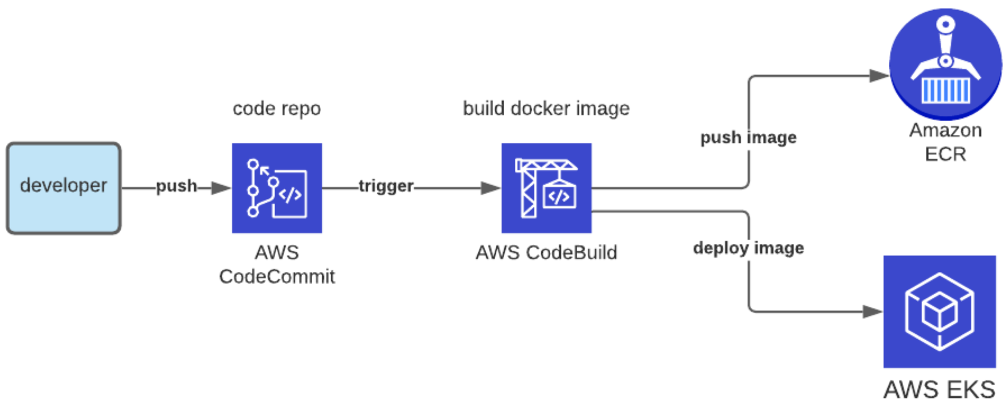
Product Model:

```
@Entity(name = "product")
public class Product {
    @Id
    private String sku;
    private String title;
    private String description;
    private int price;
    private int quantity;

//    setter and getter for each attribute here
}
```

Deployment:

As mentioned above, each service will have its own code base containerized and will follow the below CICD pipeline:



Conclusion:

Using microservices to speed up the delivery of the application updates where each service has its own CICD pipeline and can be updated independently. Each service is running on a container on a Pod on AWS EKS cluster which can scale independently and failover exists which enhance availability and scalability.