Article type: project

**Devops assessment - 2nd phase**

**To-do**

Create the server architechture diagram to deploy microservices architecture **(ecommerce application)** in the aws cloud.

**Following services can be use**

Ec2

Cloud watch

Ecs

Eks

Route53

S3

Amazon eks

Mongodb atlas

**Architecture should support  following objectives**

-      10 micro services deployment on aws cloud

-      optimize service and resource cost

-      security and scalability

-      backups and log monitor

-      analytics & different reports

**Database**

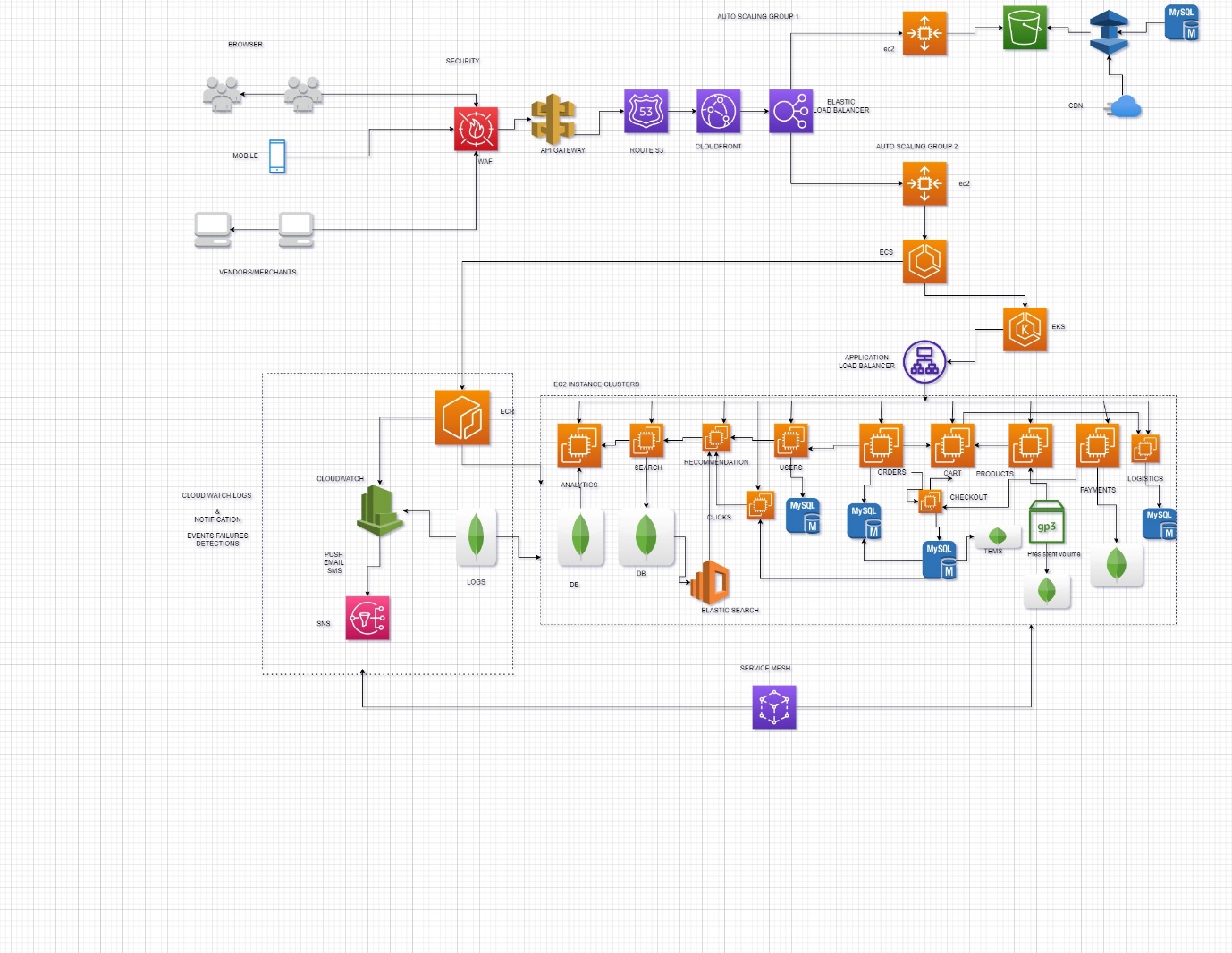
-      mongodb atlas

-      mysql

**Solution:**

E-commerce application:

* Identity
* User data
* Cart
* Search
* Checkout
* Product
* Payments
* Shipping
* Inventory



Benefits:

* Scalability
* Loosely coupled
* Fault isolation vs bring all down
* Allows try out new technologies
* Rewritten can be to 1 service
* Faster development and deployment
* De-centralised data.
* Messages are broken

**User profile:**

* Create user
* Get user
* Delete user
* Cart function:
* Create
* Add
* Delete

**Order:**

* Generate
* Approve
* Delete

**Ship order:**

* Shipping
* Get invoice
* Return order

|  |
| --- |
| Search + product |
| Inventory |
| Shipping (backend) |
| Rating& reviews |
| Recommendation engine |
| Merchants (other option to show) |
| Finance + insurance (emi, warranty) |

Api :

* Api gateway http(external):(http/https,web socket,rpc)(internal)
* Api composition n microservices one call to client
* Parallel call
* For serial call configure other on api

**Bff (backend)**

* More api gateway(mobile, web browser,3rd party)
* Mobile: ios/android
* Also we can track and rate limit the 3rd party api usages
* Authentication(jwt(json web token))
* SSL termination

**Load Balancer**

* Insulaton(security-)
* Api gateway translation call combine or split.
* Helps to imlplement
* Caching
* Managing access quotas
* Api health monitoring
* Api versioning
* Chaos monkey testing
* A/b testing
* Static is not possible
* So our api act as lb
* Id add instances or not working
* One micro to other microservice – service discovery(find all n/w address)

**Service register- db(list of all service)**

1.self registers(themself)

2.third party

Discovery(update with new cases)

1.client-><-sr(ports and ip)

2.server

Order processing microservice

Http/rpc call

Generate bill

Update-order-table

Notify merchants-> merchant microservice call

**Circuit breaker:** -> cache response to recover or create new instances to full fill the service

* Cached repo
* Fallback(3rd part )-> fails of microservice due to overload
* Recover.(timeout)-> try to contact mc service healed and back to connection to response

**Service** **mesh:**

* Challenges use cases between microservices
* Load balance
* Service discovery
* Retries
* Circuit breaking
* Timeout(std time out to configure)
* Side car☹proxy)agent or mediator blocking some sites to access
* Service mesh will call for other service for their address
* Send to service registry req.get()

**Control plane:**

* Centralized hub all of proxy sideload
* Data plane: this proxy doesn’t know other proxies

**Deployment patterns:**

* Scalable & throughput
* Reliable & available
* Isolation
* Resource limit
* Monitor
* Cost effective

1.)Multiple service per host

* Vm server-> instance
* Pros: Efficient resource utilization
* & fast deployment

Cons:

* Poor isolation
* No resource limit
* Dependency conflict

2.) Service per vm/container

* Image built for microservices separate prebuild image s1 s2 s3
* Auto scaling on aws/vm kubernetes and docker
* Pros:
* Isolation & secure
* Manageable
* Fast (container only)
* Autoscaling (kube/docker)
* Cons:
* Slow(vm only)
* Not efficient(vm only)
* Not secure(continers)

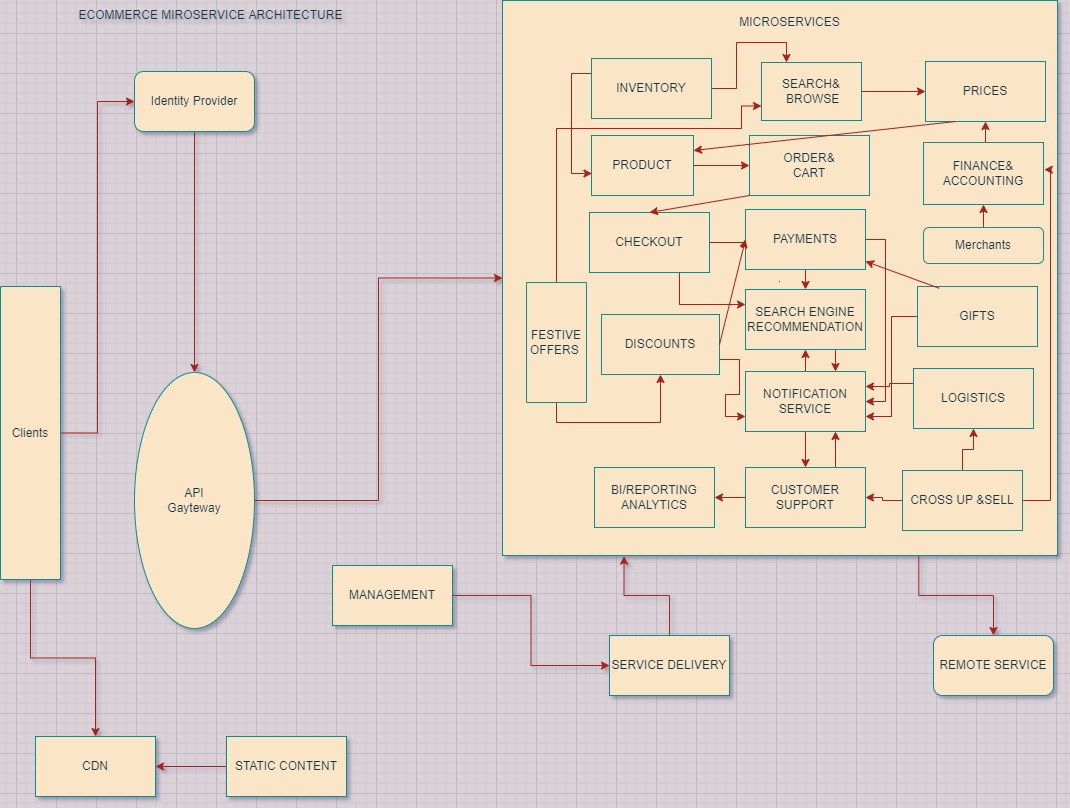
3.) Serverless:

* AWS fargate
* Pros :
* Focus on code
* Auto scaling
* Pay you go

Cons:

* Runtime support
* Expensive
* Vendor lock
* Debugging pain
* Stageless & short running process only

**Overview:**



**Cart Services:**

