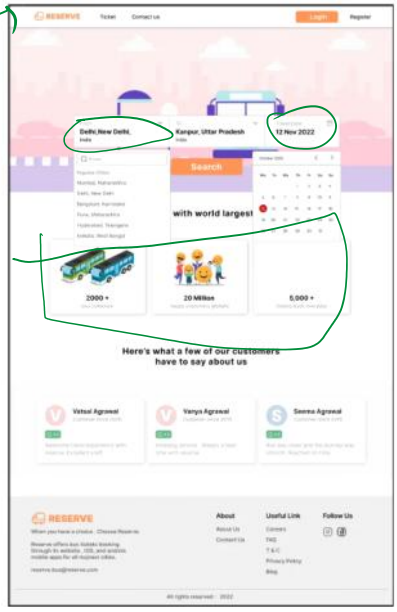
10 March 2025 19:42

Full Stack Project Overview

AlmaBetter



Full Stack Project Overview & Setup

1. Introduction to Full Stack Projects

- **Definition:**
  A full stack project involves building an entire web application—both the frontend (client-side interface) and the backend (server-side logic and database interaction). It delivers a complete user experience.
- **Why It Matters:**
  Full stack development allows you to build applications that are not only visually appealing and interactive but also capable of handling complex data operations, user requests, and scalable deployments.
- **Real-World Context:**
  Companies like Google, Facebook, and Amazon rely on full stack projects to power their products. For example, Google Search presents a friendly UI while processing complex backend queries for rapid results.

2. Components of a Full Stack Project

A. Frontend (Client-Side)

- **Technologies:**
  React is typically used to build interactive, reusable UI components. It leverages a virtual DOM to update only the necessary parts of the webpage.
- **Role:**
  Presents the user interface, handles user inputs (like clicks and form submissions), and communicates with backend APIs to fetch or send data.
- **Example in a Project:**
  In the Reserve Project (a bus ticket booking system), the homepage where users enter travel details is built using React.

B. Backend (Server-Side)

- **Technologies:**
  Node.js combined with *Express.js* is used to create the server. Node.js enables JavaScript to run on the server, while Express provides a framework to handle routing and middleware.
- **Role:**
  Manages server logic, processes API requests from the frontend, and handles tasks such as authentication, data processing, and business logic.
- **API Endpoints:**
  - **POST API:** To add new trip details.
  - **GET API:** To retrieve a list of trips (e.g., past trips limited to 50 entries).
  - **Other APIs:** For booking tickets, retrieving trip details by date or other parameters.

C. Database

- **Technologies:**
  MongoDB Atlas is used as a NoSQL database which stores data in JSON-like documents (BSON).
- **Role:**
  Persists data such as trip details, user information, and booking records. Its flexible schema design allows for storing varied data types.
- **Connection:**
  The Node.js backend connects to MongoDB using MongoDB drivers or libraries to fetch and insert data as required by API endpoints.

3. Case Study: The Reserve Project

- **Connection:**
  The Node.js backend connects to MongoDB using MongoDB drivers or libraries to fetch and insert data as required by API endpoints.

3. Case Study: The Reserve Project

- **Overview:**
  The Reserve Project is a bus ticket booking platform that integrates:
  - **Frontend:** User interface for searching trips, selecting buses, choosing seats, and entering user information.
  - **Backend:** API endpoints (using Express.js) to add trips, retrieve trip details, and manage ticket bookings.
  - **Database:** MongoDB Atlas to store trip and booking data.
- **User Flow:**
  1. **Homepage:** User inputs travel parameters (source, destination, date, etc.).
  2. **Bus Selection:** Users view available buses and select one based on details like amenities and pricing.
  3. **Seat Selection:** After choosing a bus, users pick their preferred seats.
  4. **User Information & Payment:** Users enter personal details and complete payment via Stripe integration.
  5. **Receipt Generation:** A confirmation or ticket receipt is provided upon successful payment.

4. Real-Life Example: Amazon's Full Stack System

- **Frontend (User Experience):**
  Amazon's website and mobile apps deliver a user-friendly shopping experience. Customers browse products, read reviews, and make purchases—all through intuitive interfaces.
- **Backend (Server-Side Logic):**
  Amazon's backend is built on a microservices architecture. Each service handles specific functions (e.g., order processing, inventory management, payment processing). This design allows for scalability and efficient handling of millions of transactions.
- **Database & Storage:**
  Amazon employs a mix of relational databases (for transactional data) and NoSQL databases (for flexible product catalogs and high scalability). Data is often distributed across multiple servers and data centers for resilience and speed.
- **System Interaction:**
  When you place an order on Amazon, the frontend collects your data and sends it to the backend. The backend:
  - Validates your order.
  - Interacts with various services (inventory, shipping, payment).
  - Retrieves and stores data in the appropriate databases.
  - Returns order confirmation and tracking information to your frontend.
- **Takeaway:**
  Amazon's system illustrates how a well-integrated full stack solution can support complex operations and scale to handle vast numbers of users and transactions.

5. System Design: High-Level and Low-Level Design

A. High-Level Design (HLD)

- **Definition:**
  HLD provides a bird's-eye view of the overall system architecture. It outlines the main components, their interactions, and the overall data flow.
- **Key Elements:**
  - **Components:** Frontend, backend, databases, external APIs, and integrations.
  - **Diagrams:** Often represented as block diagrams showing how components (like user interface, server, database, and third-party services) communicate.
  - **Focus:** Overall system structure and relationships, not the detailed inner workings of each component.
- **Example in Full Stack Projects:**
  In the Reserve Project, the HLD would show the user journey from the React-based frontend through the Node.js/Express backend to the MongoDB database, along with integrations (like the Stripe payment gateway).

B. Low-Level Design (LLD)

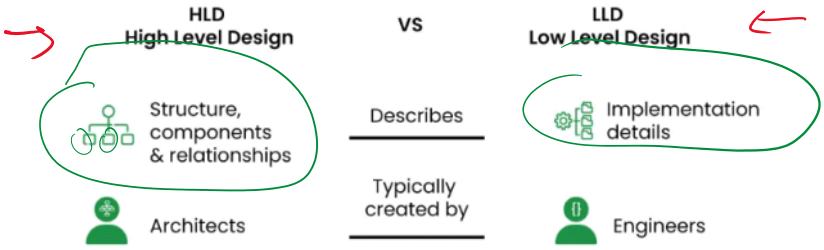
- **Definition:**
  LLD dives into the specifics of each component. It includes detailed designs such as class diagrams, API specifications, and database schemas.
- **Key Elements:**
  - **Detailed Modules:** Specific functions, classes, methods, and data structures used within each component.
  - **Database Schema:** The structure of collections/tables, fields, data types, and relationships.
  - **API Details:** Endpoints, request/response formats, error handling, and security measures.
- **Example in Full Stack Projects:**
  For the Reserve Project, the LLD would detail:
  - Specific API endpoints (e.g., /api/trips, /api/bookings) including request parameters and response structures.
  - The schema for the "trips" collection in MongoDB.
  - The logic for processing user inputs and managing session states.

C. Process: From HLD to LLD

1. **Start with HLD:**
   Define the overall architecture. Identify the major components (frontend, backend, database) and how they interact.
2. **Drill Down to LLD:**
   Break each component into detailed parts. Define exact API endpoints, database collections, and specific code modules.
3. **Integration:**
   Ensure that the detailed designs align with the overall architecture. Confirm that all interfaces between components are well-defined.

Break each component into detailed parts. Define exact API endpoints, database collections, and specific code modules.

3. **Integration:**
Ensure that the detailed designs align with the overall architecture. Confirm that all interfaces between components are well-defined.



1. High-Level Design (HLD)

A. Definition & Purpose

- **HLD Overview:**
  High-Level Design provides an overall architecture view of the system. It defines the main components, their interactions, and the data flow without getting into the specific details of implementation.
- **Purpose:**
  - **Big-Picture Perspective:** Helps stakeholders understand the system structure and major functionalities.
  - **Component Identification:** Identifies modules like the frontend, backend services, databases, external integrations, and communication protocols.
  - **Scalability & Resilience:** Outlines how the system can scale and handle load by showing distributed components and redundancy plans.

B. Example: Amazon's System – HLD Perspective

Imagine you are designing Amazon's e-commerce platform at a high level:

1. **Components & Architecture Diagram:**
   - **User Interface (Frontend):**
     - Web and mobile applications for browsing products, placing orders, and managing accounts.
   - **API Gateway:**
     - Acts as a single entry point for client requests. It routes requests to appropriate backend services.
   - **Microservices Architecture (Backend):**
     - **Catalog Service:** Manages product listings and details.
     - **Order Service:** Processes customer orders.
     - **Payment Service:** Handles payment transactions securely.
     - **Inventory Service:** Keeps track of stock levels.
     - **User Service:** Manages user profiles and authentication.
   - **Databases & Storage:**
     - **Relational Databases:** For transactional data (orders, payments).
     - **NoSQL Databases:** For scalable storage of product catalogs and customer reviews.
     - **Caching Systems:** (e.g., Redis) for quick data retrieval.
   - **External Integrations:**
     - Third-party services for shipping, fraud detection, and analytics.
2. **Data Flow & Interaction:**
   - **User Journey:**
     - A customer logs into the Amazon website (frontend). The request goes through the API gateway which directs it to the User Service for authentication.
     - Upon browsing products, the Catalog Service retrieves product data from a NoSQL database.
     - When an order is placed, the Order Service coordinates with the Inventory Service, Payment Service, and then stores the order in a relational database.
   - **High-Level Diagram (Conceptual):**
     - A block diagram would show boxes for each service, arrows indicating data flow between them, and external systems such as payment gateways.

C. Thinking Process in HLD

- **Requirements Analysis:**
  Understand the business needs: scalability (millions of users), reliability (order processing without failure), and performance (fast search and recommendations).
- **Defining Boundaries:**
  Determine what should be independent modules (e.g., separate microservices for user management vs. orders).
- **Technology Choices:**
  Decide on technologies (e.g., React for frontend, microservices in Node.js/Java, relational vs. NoSQL databases) based on scalability and maintenance.
- **Stakeholder Communication:**
  HLD is shared with architects, product managers, and clients to ensure everyone agrees on the overall system.

2. Low-Level Design (LLD)

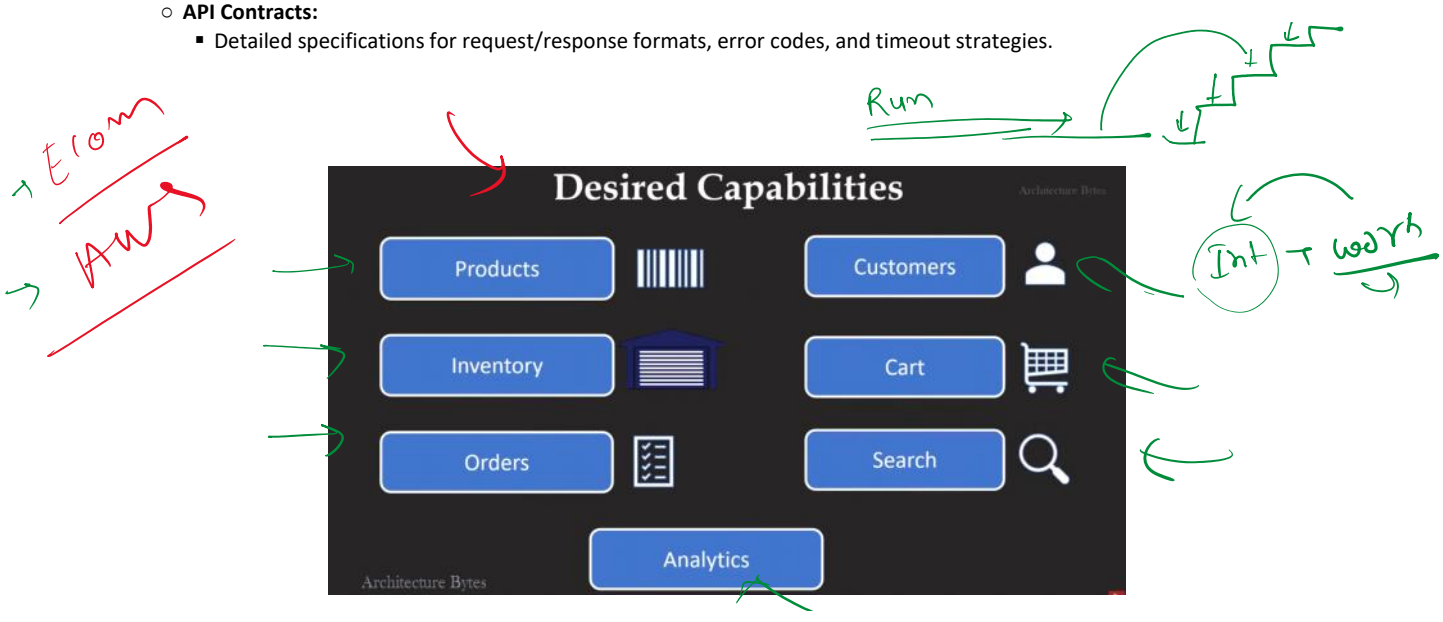
A. Definition & Purpose

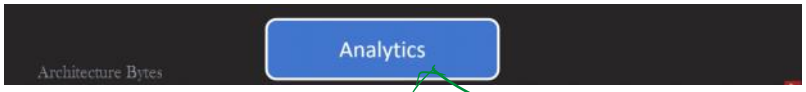
- **LLD Overview:**
  Low-Level Design translates the high-level blueprint into detailed specifications. It describes exactly how each component will be built.
- **Purpose:**
  - **Detailed Specifications:** Provides in-depth details like algorithms, class diagrams, API endpoints, data models, and interactions.
  - **Implementation Guide:** Acts as a guide for developers during coding.
  - **Error Handling & Security:** Specifies details about how errors are managed and how security is enforced.

B. Example: Amazon's System – LLD Perspective

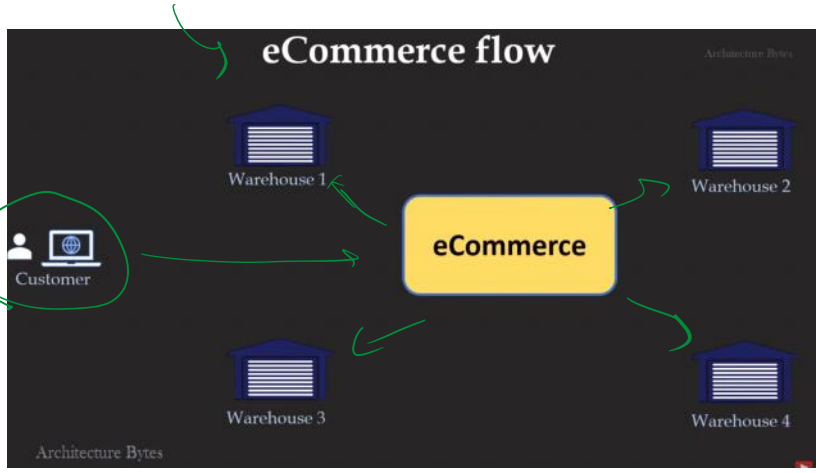
Taking our Amazon example further, here's how the LLD would look:

1. **Detailed Component Specifications:**
   - **Catalog Service:**
     - **API Endpoints:**
       - GET /api/products: Retrieves product listings.
       - GET /api/products/{id}: Retrieves detailed information for a specific product.
     - **Data Model:**
       - Product Schema: Fields such as productID, name, description, price, category, ratings, reviews.
     - **Caching Strategy:**
       - Use Redis to cache frequently accessed product data.
   - **Order Service:**
     - **API Endpoints:**
       - POST /api/orders: Creates a new order.
       - GET /api/orders/{orderId}: Retrieves order details.
     - **Business Logic:**
       - Validate stock levels from the Inventory Service.
       - Process payment by calling the Payment Service.
       - Transaction management to ensure order consistency.
   - **Payment Service:**
     - **Security Measures:**
       - Use secure payment gateways.
       - Encrypt sensitive data in transit and at rest.
     - **API Endpoint:**
       - POST /api/payments: Processes payment information.
   - **User Service:**
     - **Authentication:**
       - Use JWT (JSON Web Tokens) for session management.
       - Endpoints for user registration, login, and profile management.
2. **Database Design Details:**
   - **Relational Database (e.g., for Orders):**
     - **Schema Definition:** Tables for Orders, OrderItems, and Transactions.
     - **Indexes & Relations:**
       - Primary keys, foreign keys, and indexes to optimize query performance.
   - **NoSQL Database (e.g., for Product Catalog):**
     - **Document Structure:** Each product stored as a document in a collection, allowing flexible schema.
     - **Query Patterns:** Designed to support efficient product searches and filters.
3. **Inter-Service Communication:**
   - **Synchronous vs. Asynchronous:**
     - Define which services communicate in real-time (synchronous REST calls) and which use messaging queues (asynchronous order processing).
   - **API Contracts:**
     - Detailed specifications for request/response formats, error codes, and timeout strategies.

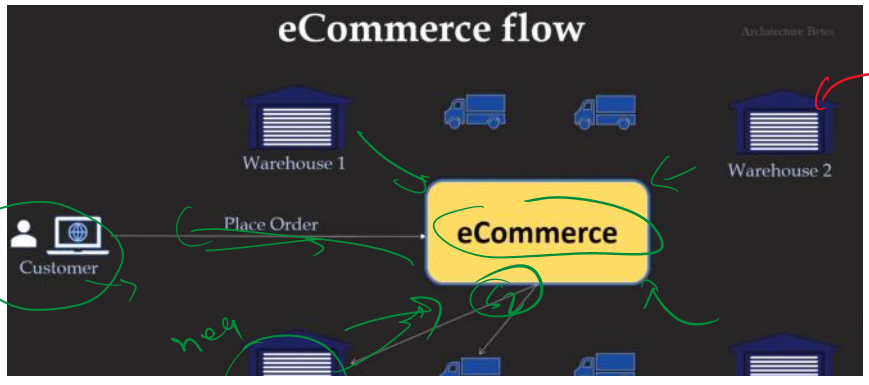


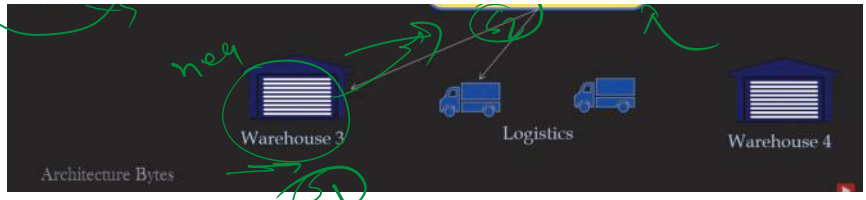

Analytics
Architecture Bytes

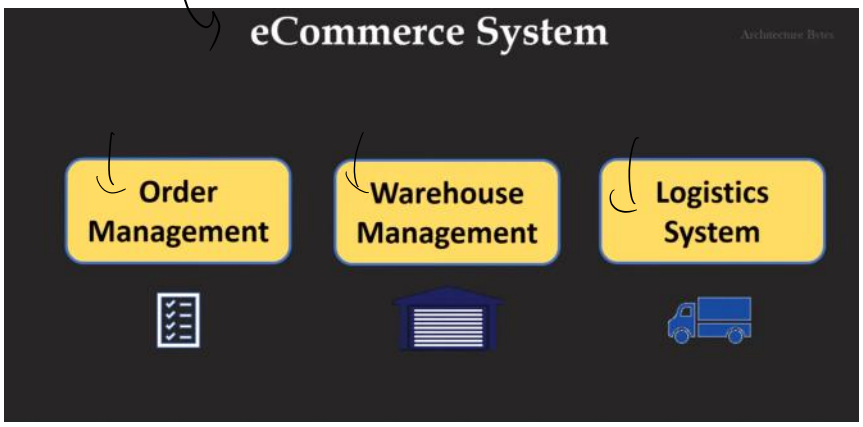

Design Approach
Microservices Architecture
Orchestration via Workflows

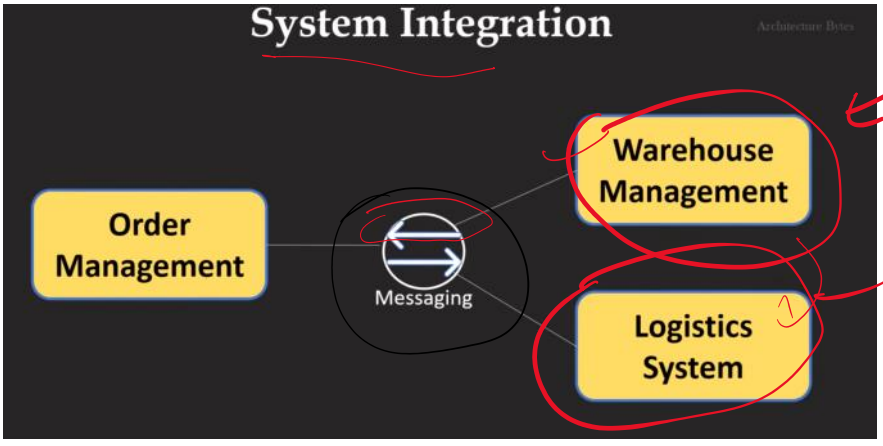

eCommerce flow
Warehouse 1
Warehouse 2
Customer
eCommerce
Warehouse 3
Warehouse 4
Architecture Bytes

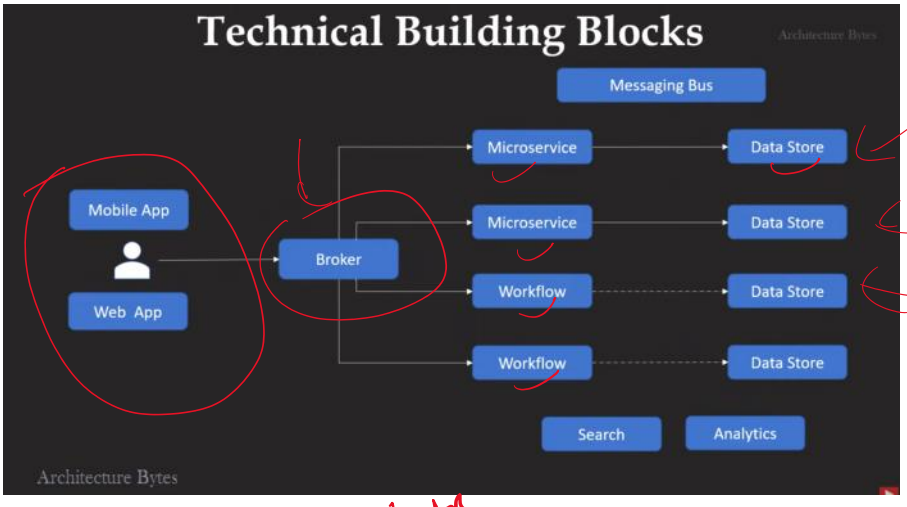

eCommerce flow
Warehouse 1
Warehouse 2
Customer
Place Order
eCommerce
Warehouse 3
Warehouse 4
Architecture Bytes

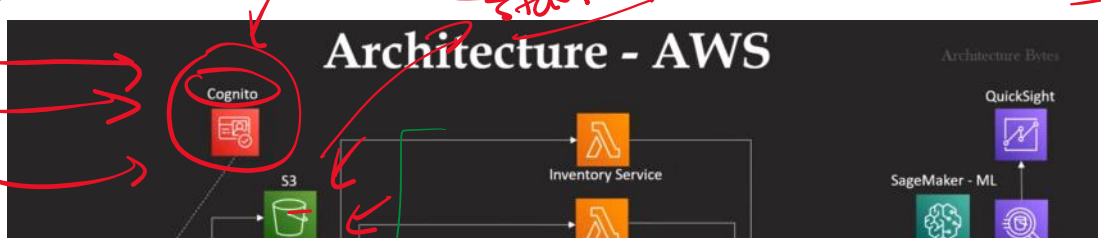

eCommerce flow
Warehouse 1
Warehouse 2
Customer
Place Order
eCommerce

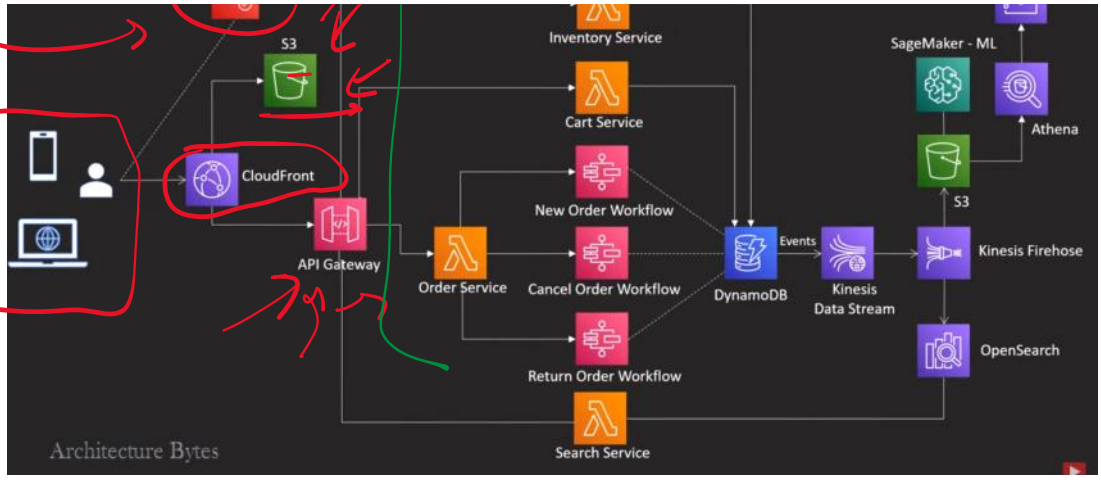

Warehouse 3
Logistics
Warehouse 4
Architecture Bytes

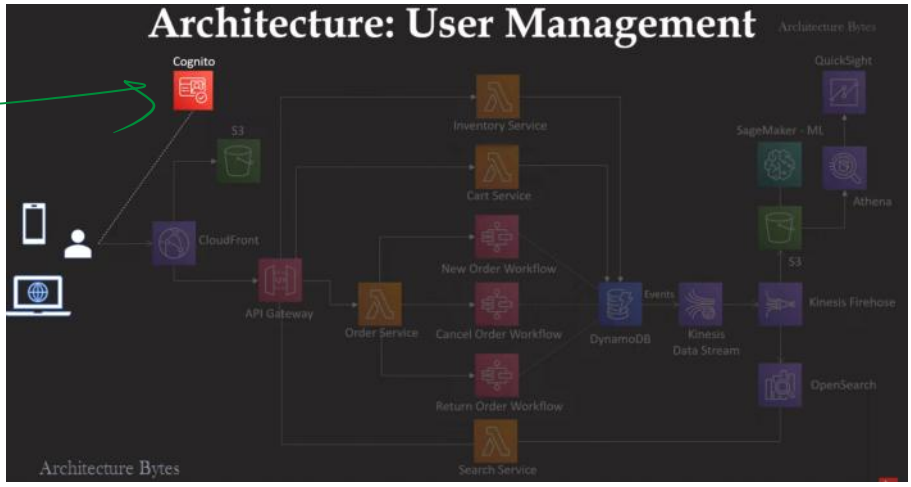

eCommerce System
Order Management
Warehouse Management
Logistics System

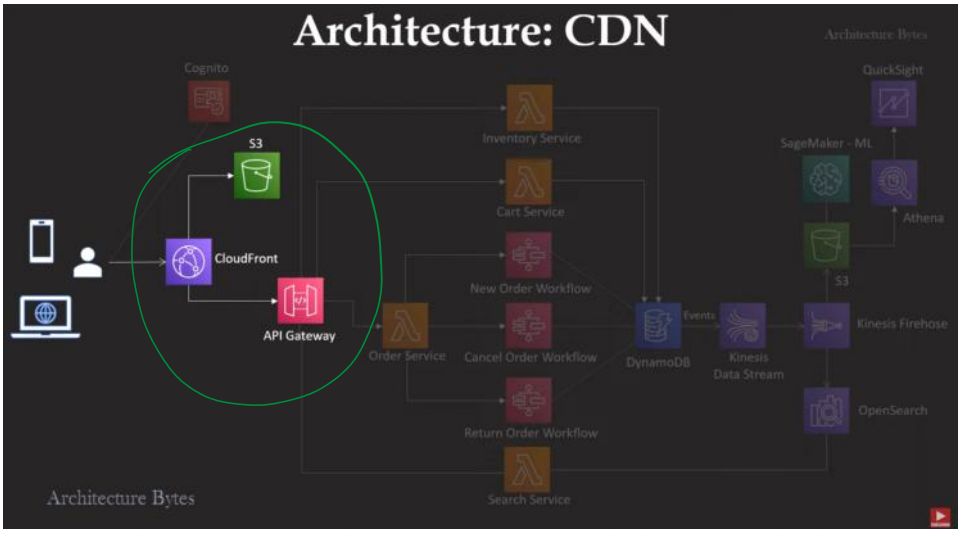

System Integration
Order Management
Messaging
Warehouse Management
Logistics System

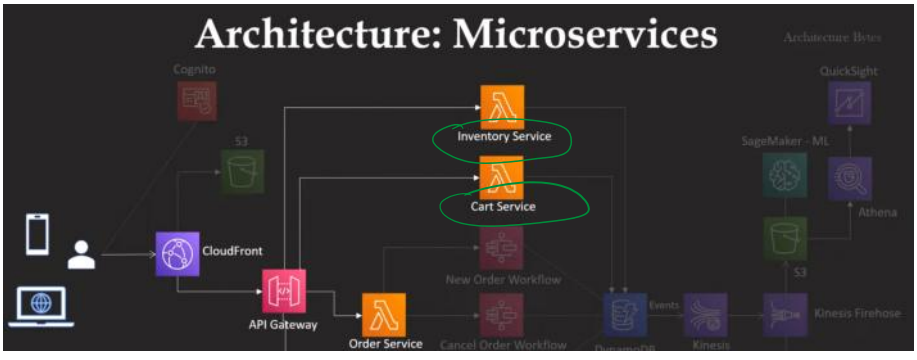

Technical Building Blocks
Messaging Bus
Mobile App
Web App
Broker
Microservice
Microservice
Workflow
Workflow
Data Store
Data Store
Data Store
Data Store
Search
Analytics
Architecture Bytes

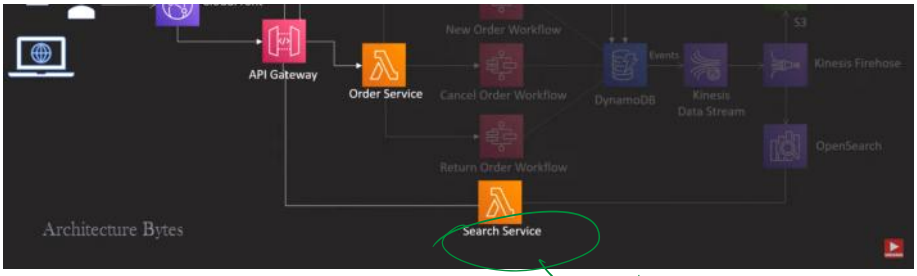

Architecture - AWS
Cognito
S3
Inventory Service
QuickSight
SageMaker - ML

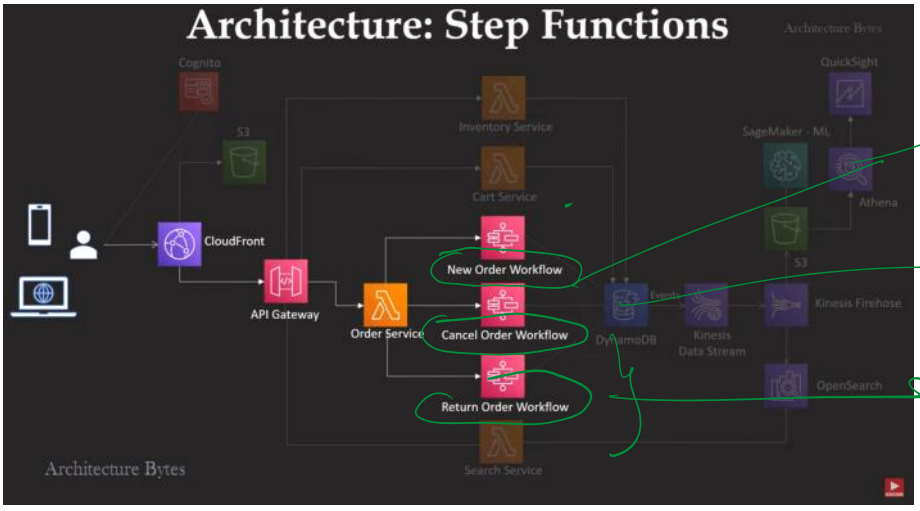

S3
Inventory Service
SageMaker - ML
Cart Service
Athena
CloudFront
New Order Workflow
S3
API Gateway
Order Service
Cancel Order Workflow
DynamoDB
Events
Kinesis Firehose
Kinesis Data Stream
OpenSearch
Return Order Workflow
Search Service
Architecture Bytes

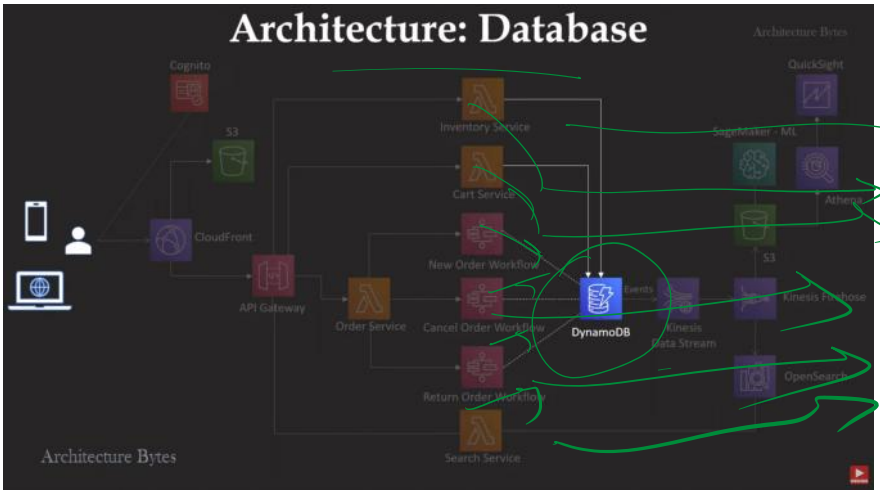

Architecture: User Management
Architecture Bytes
Cognito
QuickSight
S3
Inventory Service
SageMaker - ML
Cart Service
Athena
CloudFront
New Order Workflow
S3
API Gateway
Order Service
Cancel Order Workflow
DynamoDB
Events
Kinesis Firehose
Kinesis Data Stream
OpenSearch
Return Order Workflow
Search Service
Architecture Bytes


Architecture: CDN
Architecture Bytes
Cognito
QuickSight
S3
Inventory Service
SageMaker - ML
Cart Service
Athena
CloudFront
New Order Workflow
S3
API Gateway
Order Service
Cancel Order Workflow
DynamoDB
Events
Kinesis Firehose
Kinesis Data Stream
OpenSearch
Return Order Workflow
Search Service
Architecture Bytes

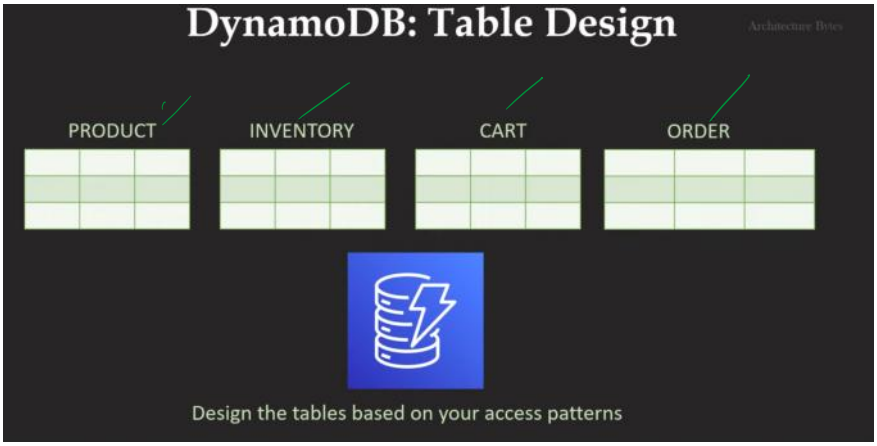

Architecture: Microservices
Architecture Bytes
Cognito
QuickSight
S3
Inventory Service
SageMaker - ML
Cart Service
Athena
CloudFront
New Order Workflow
S3
API Gateway
Order Service
Cancel Order Workflow
Events
Kinesis Firehose

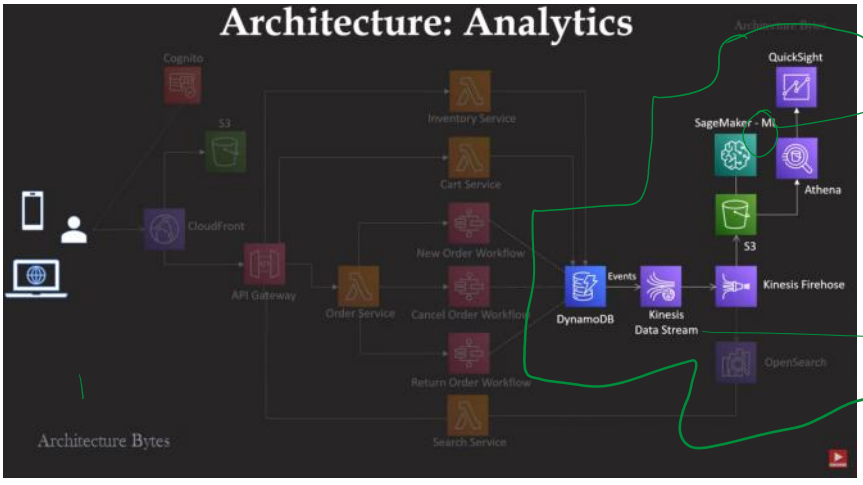

API Gateway
Order Service
New Order Workflow
Cancel Order Workflow
Return Order Workflow
DynamoDB
Events
Kinesis Data Stream
Kinesis Firehose
S3
OpenSearch
Search Service
Architecture Bytes


Architecture: Step Functions
Architecture Bytes
Cognito
S3
CloudFront
API Gateway
Order Service
Inventory Service
Cart Service
New Order Workflow
Cancel Order Workflow
Return Order Workflow
DynamoDB
Events
Kinesis Data Stream
Kinesis Firehose
QuickSight
SageMaker - ML
Athena
OpenSearch
Search Service
Architecture Bytes


Architecture: Database
Architecture Bytes
Cognito
S3
CloudFront
API Gateway
Order Service
Inventory Service
Cart Service
New Order Workflow
Cancel Order Workflow
Return Order Workflow
DynamoDB
Events
Kinesis Data Stream
Kinesis Firehose
QuickSight
SageMaker - ML
Athena
OpenSearch
Search Service
Architecture Bytes


Data Entities
Architecture Bytes
Customers
Buyers
Products
What is being sold
Inventory
Product quantity available for sale
Orders
Customers request for Products





Requirements:
1. Search for Product
2. Recommendations on Homepage
3. Place Order
4. Check Order status
5. Write/View product reviews

Assumptions:
1. User Profile Creation is provided
2. Product Onboarding is provided
3. Payment Service is provided

Non-Functional Requirements:
1. Low Latency (Recommendations & Search)
2. High Consistency (Placing order, order status and payments)

Capacity Estimations:
Active Users: 300M Monthly Active users
each user search for 10 product a month
= 300M * 10 Search/month
=3B Search /(30 days * 24 hours* 60 mins * 60 seconds)
=3*10^9 / (30 *10^5)
= 1K Searches/second

Total Products: 10M
Assume: 1 Product requires 10MB storage(Images+ Description)
Total Product Storage: 10M * 10MB
= 10 * 10^6 * 10 * 10^6
=. 100 * 10^12
= 100TB storage required to store all product information

Assume: 1 Product requires 10MB storage(Images+ Description)
Total Product Storage: 10M * 10MB
= 10 * 10^6 * 10 * 10^6
=. 100 * 10^12
= 100TB storage required to store all product information

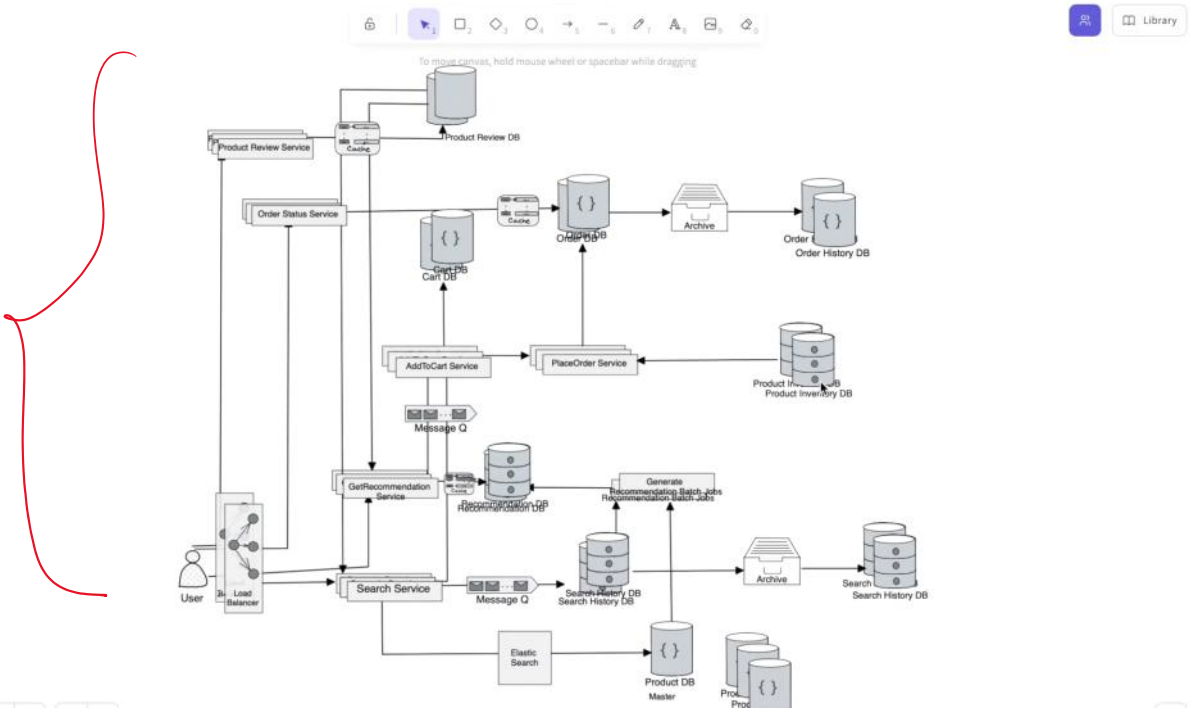
Database Design:
1. User DB- (SQL)

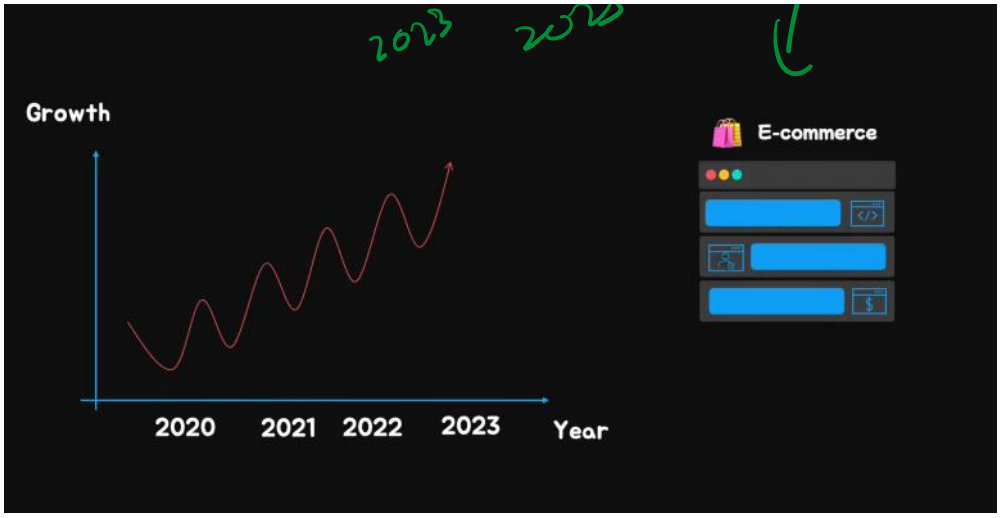
User ID (PK)
User Name(String) Password: (String) Encrypted First Name: (String) Last Name: (String) Email: (String) Last Login: DTTM Created: DTTM

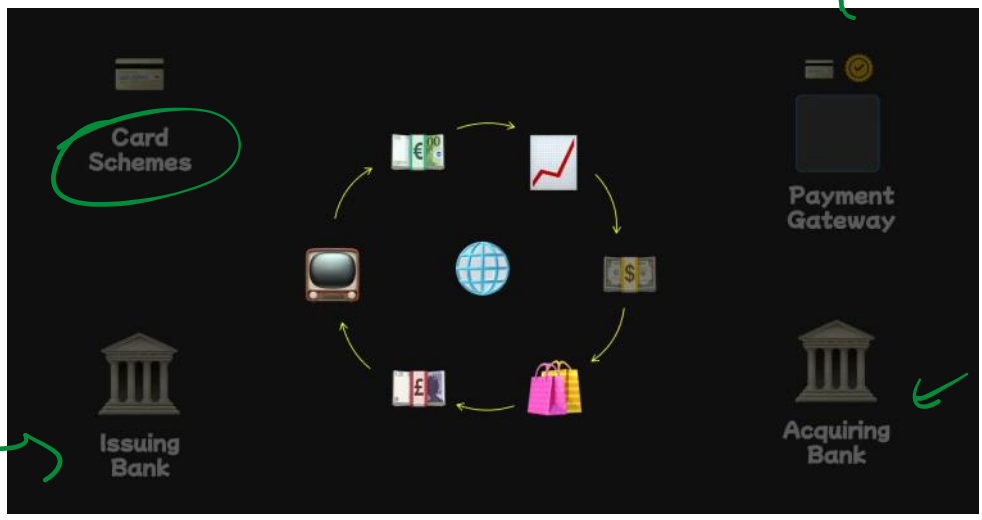
2. Address DB- (SQL)

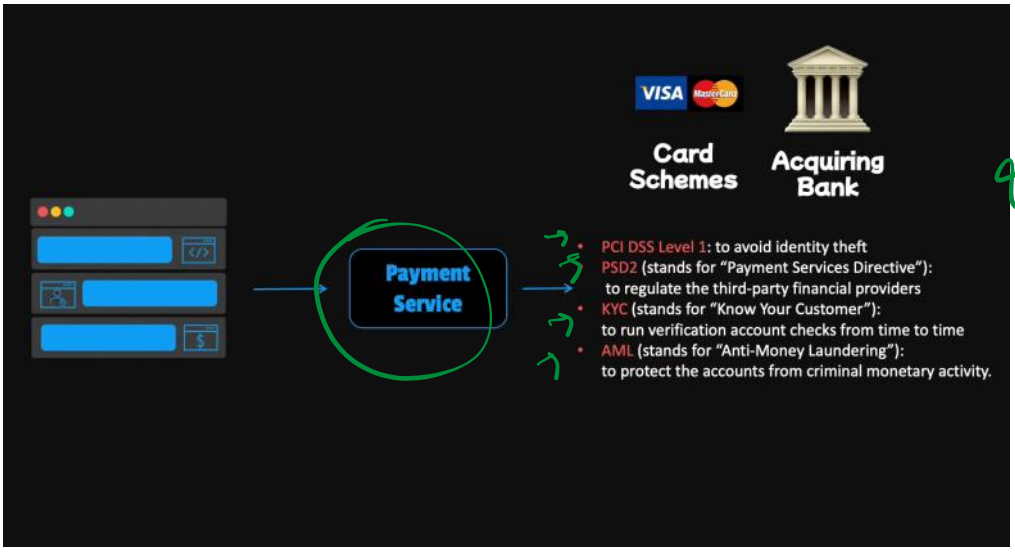
Address ID (PK)
User ID (FK) Effective Date: Date AddressLine 1: String Address Line 2: String City: String Country: String Zip: Alpha numeric

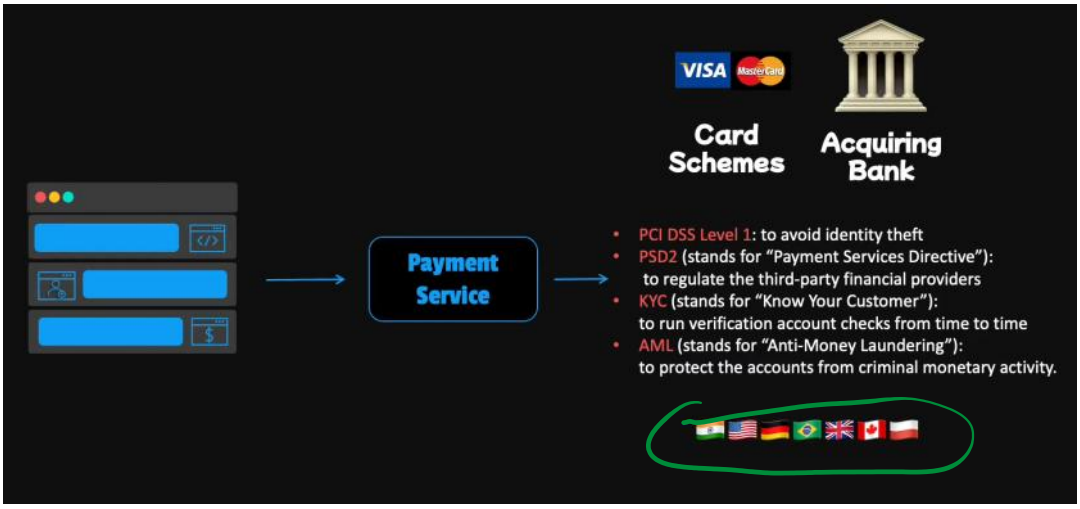
APIs:
1. GetRecommendations(UserID) -Return a list of 10 product recommendations
2. Search(Search String, UserID) - GET method& returns a list of products
3. AddToCart(UserID, ProductIDs,Qty, Amt)- Returns a boolean (true/false) status
4. PlaceOrder(UserID,OrderID,AddressID,PaymentStatus) - returns boolean
5. CheckOrderStatus(OrderId) - GET - Returns order status

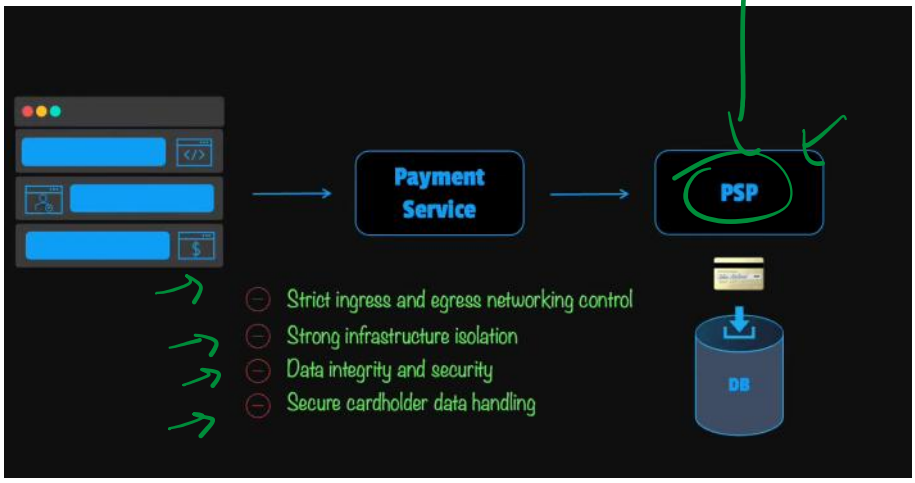


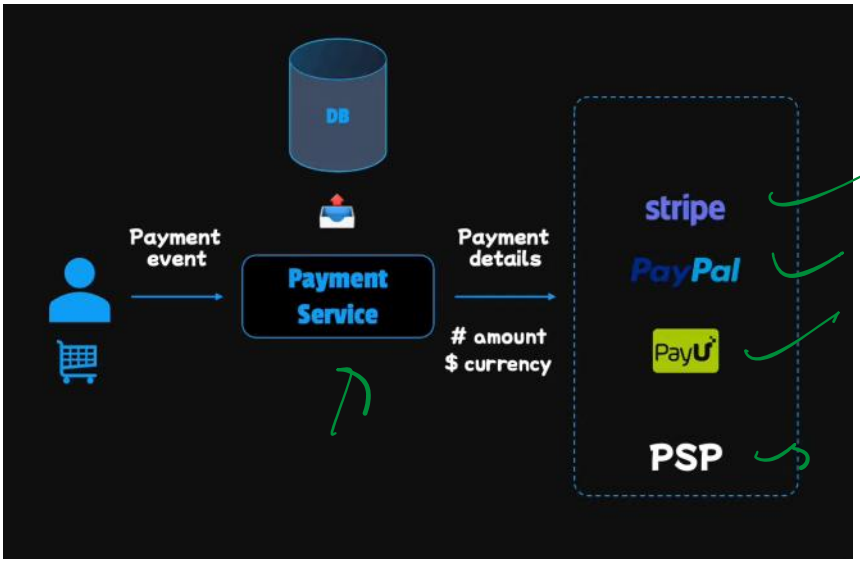

Growth
2020
2021
2022
2023
Year
E-commerce

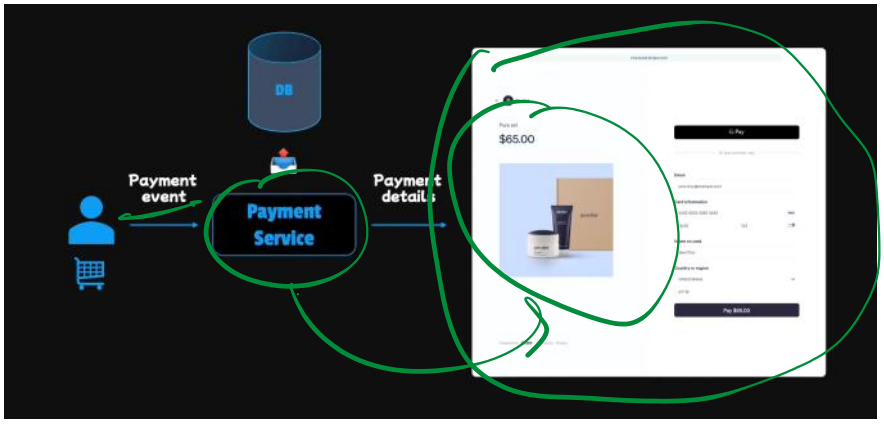

Card Schemes
Payment Gateway
Issuing Bank
Acquiring Bank

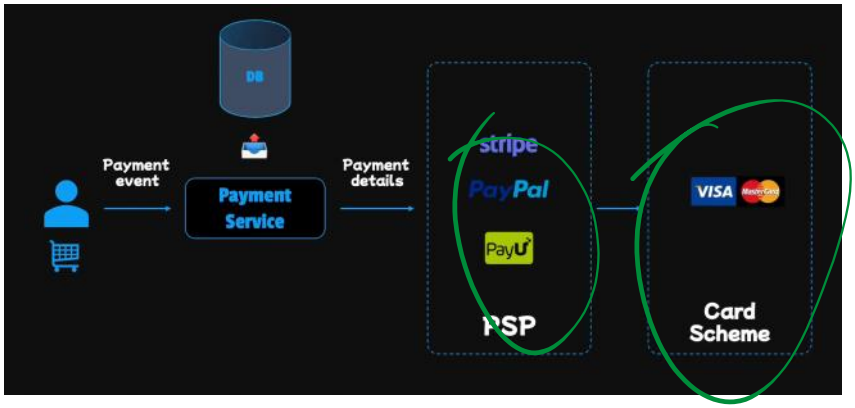

VISA
Card Schemes
Acquiring Bank
Payment Service
PCI DSS Level 1: to avoid identity theft
PSD2 (stands for "Payment Services Directive"):
to regulate the third-party financial providers
KYC (stands for "Know Your Customer"):
to run verification account checks from time to time
AML (stands for "Anti-Money Laundering"):
to protect the accounts from criminal monetary activity.

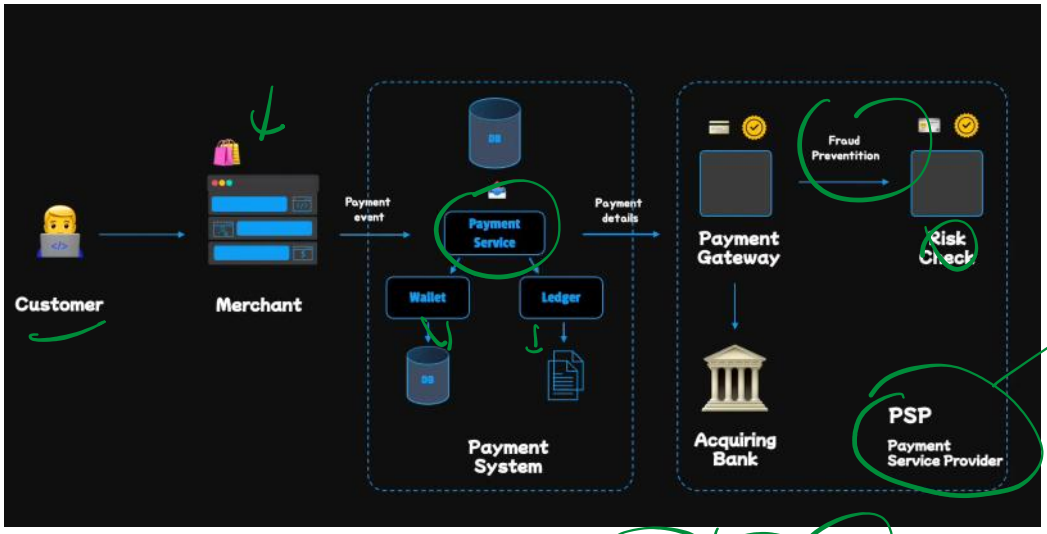

VISA
Card Schemes
Acquiring Bank
Payment Service
PCI DSS Level 1: to avoid identity theft
PSD2 (stands for "Payment Services Directive"): to regulate the third-party financial providers
KYC (stands for "Know Your Customer"): to run verification account checks from time to time
AML (stands for "Anti-Money Laundering"): to protect the accounts from criminal monetary activity.

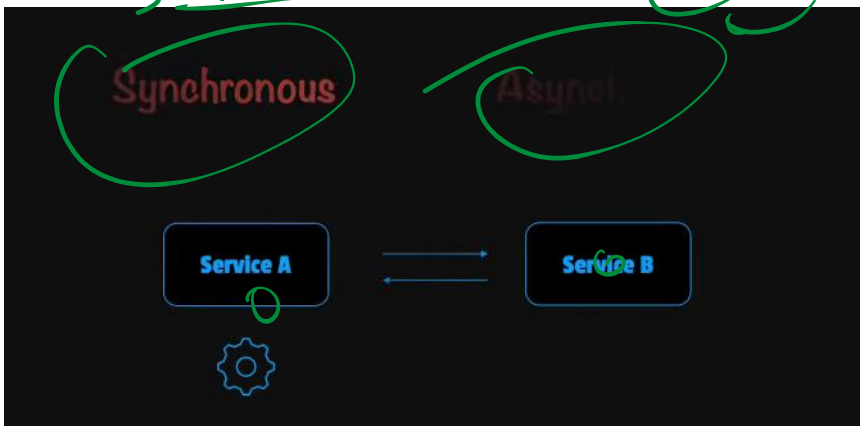

Payment Service
PSP
DB
Strict ingress and egress networking control
Strong infrastructure isolation
Data integrity and security
Secure cardholder data handling

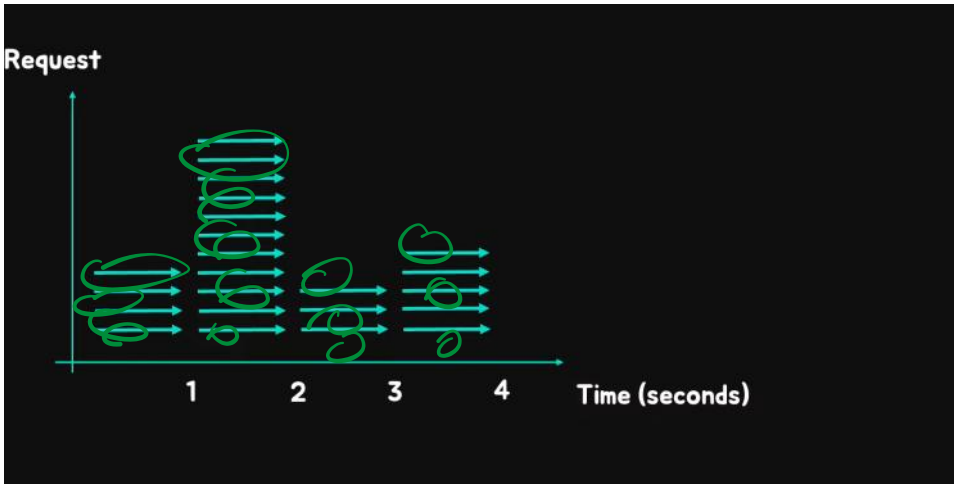

DB
Payment event
Payment Service
Payment details
# amount
$ currency
stripe
PayPal
PayU
PSP

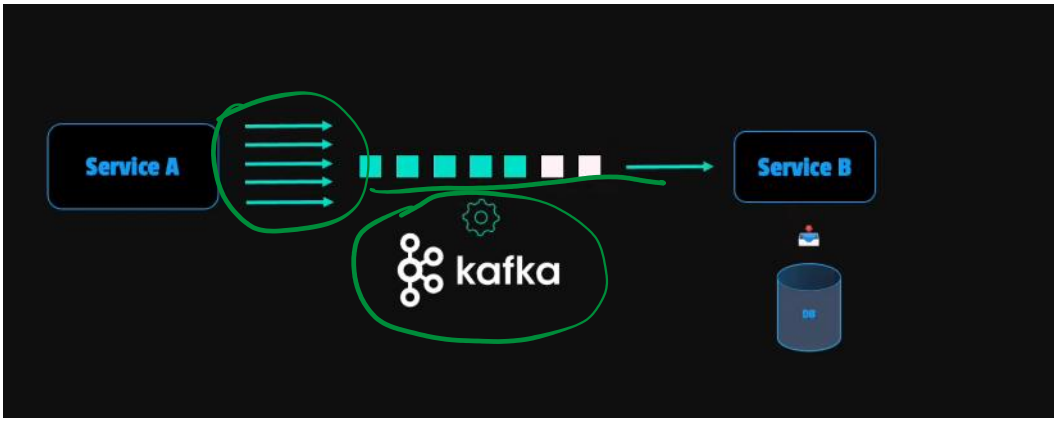

DB
Payment event
Payment Service
Payment detail
$65.00

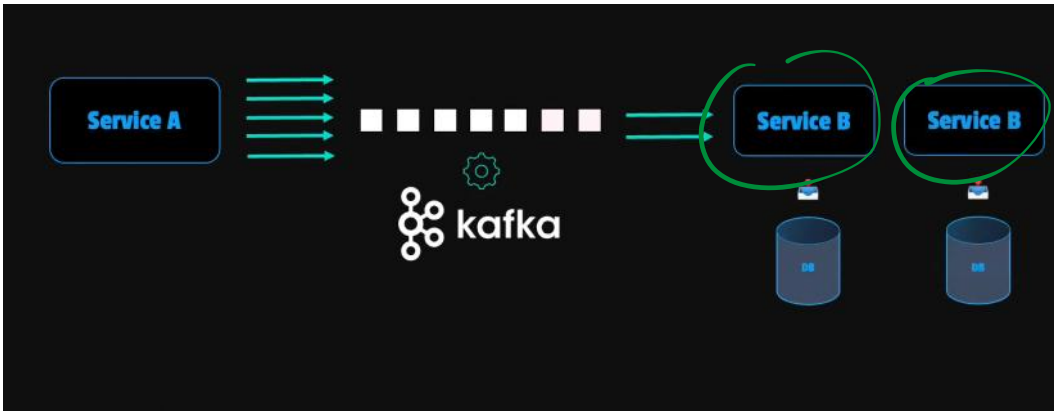

DB
Payment event
Payment Service
Payment details
stripe
PayPal
PayU
PSP
VISA
Card Scheme

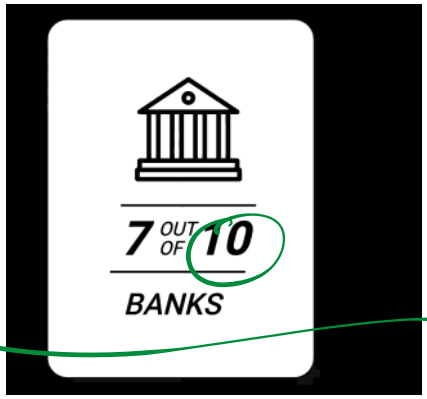

Customer
Merchant
Payment event
DB
Payment Service
Wallet
Ledger
Payment System
Payment details
Fraud Preventition
Payment Gateway
Risk Check
Acquiring Bank
PSP
Payment Service Provider

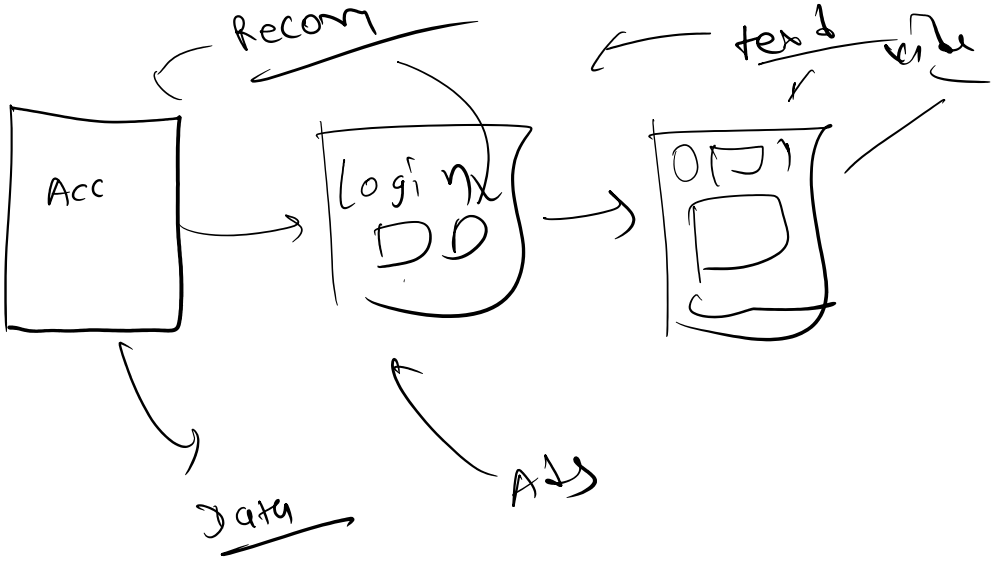

Synchronous
Service A
Service B


Request
1
2
3
4
Time (seconds)


Service A
Service B
kafka


Service A
Service B
Service B
kafka

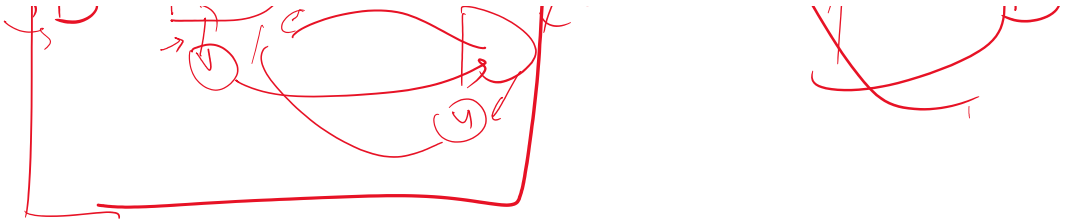




→ MVC (model-view controller)

VS

→ UML (Unified modeling (ang



MVC

- Model : manages data & Business logic
- view : Handles UI →
- Controllers : interact between model view & input

UML :(

class Digram : struct → model, contr, view

Sequence Digm : interact - user, com, model

Use case dia

MVS

↓

Architect pat

UML

↓

System dis → visual

→ GitHub Actions

CI/CD → developers → Automate workflow

↓
building
testing
deploying

- Automation

- CI/CD → interaction -

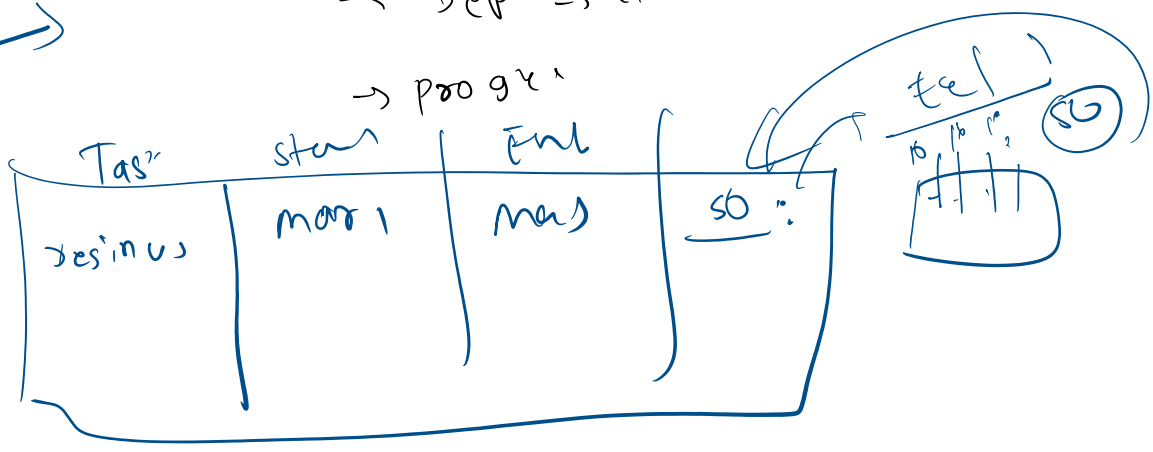
→ Jenkins → Pull, Pull - request

→ Built - in GitHub integration :

→ Gantt - Chart → time tool

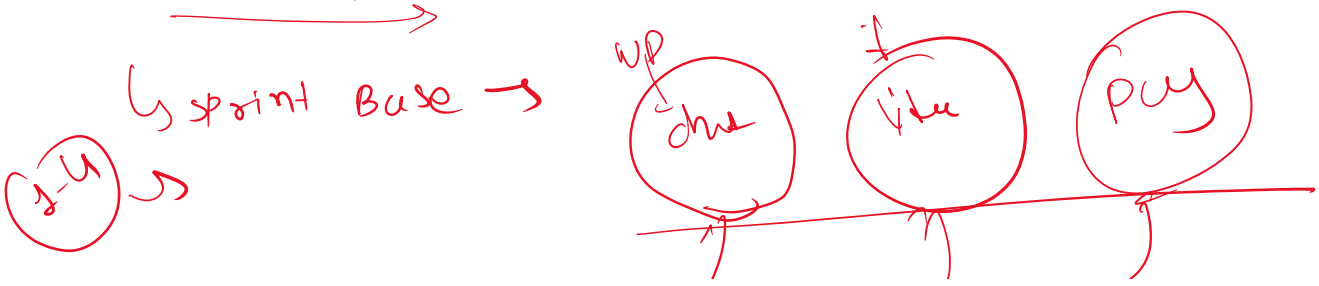
↳ Task - schedulin
→ dep → link func
→ progr

3 tasks



2. Scrum Board → task tool → Agile -

↳ sprint Base →

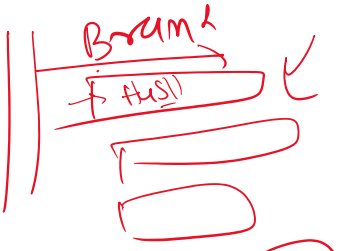


→ what is git cherry pick → command

↳ Branch → merge

→ specific bug fix or feature from onth Branch

→ wrong → move



↳ switch

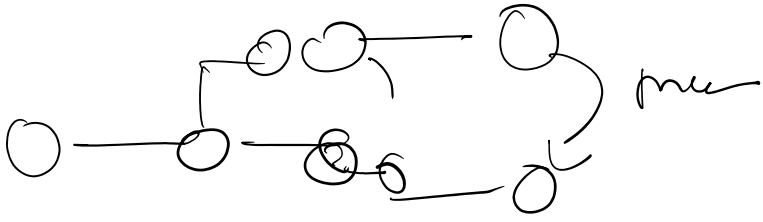
git checkout main

git cherry pick (Hash)

1. → mkdir git → P1
   → CD P1
   → git init

2. → Hello word
   → git add .
   → git commit

Create Branch



git checkout -b feature-Branch

make Changes

→ Feature A

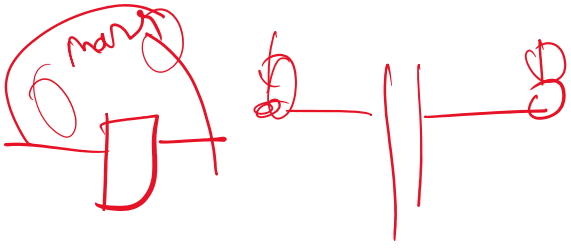
git add
git commit

Feature B

git add .
git com

→ git Check main

→ git -log - online



b2c3d4e (HEAD -> feature-branch) Added Feature B
a1b2c3d Added Feature A
e5f6g7h (main) Initial commit

Add feat

→ new → Branch

git cherry-pick a1b2c3d

Reverting Commit

→ git revert a1b2c3d → undo

Undo

→ git reset --hard HEAD~1 (2)

→ git reset --hard HEAD~1