System Structure
Microservices: (exemples**)**

- Document Management Service (DMS): Handles document creation, updates, versioning, and metadata.
- Real-Time Collaboration Service (RTCS): Manages real-time editing and synchronization among multiple users.
- User Management Service (UMS): Handles authentication, authorization, and user profiles.
- Notification Service (NS): Sends notifications for changes, sharing requests, or activity summaries.
- File Storage Service (FSS): Manages file uploads, storage, and retrieval.
- Stream Processing Service (SPS): Processes real-time data streams for immediate insights (e.g., ad clicks, impressions).
- Batch Processing Service (BPS): Aggregates and processes historical data for deeper analysis.
- Analytics Query Service (AQS): Exposes APIs to query processed data for dashboards or reports.
- Data Storage Service (DSS): Manages the storage of raw, processed, and aggregated data.
- Product Catalog Service (PCS): Manages product information, categories, and search functionality.
- Cart Service (CS): Handles shopping cart operations like adding, updating, or removing items.
- Payment Gateway Service (PGS): Integrates with third-party payment processors (e.g., Stripe, PayPal).
- Analytics Service (AS): Tracks user behavior, purchases, and other metrics for reporting and insights.
- Ride Management Service (RMS): Handles ride requests, allocations, tracking, and status updates.
- Driver Management Service (DMS): Manages driver profiles, availability, and ratings.
- Pricing and Fare Calculation Service (PFCS): Calculates fares dynamically based on demand, distance, and traffic.
- Autocomplete Service (AS): Provides suggestions for partially typed queries.
- Indexing Service (IS): Crawls and indexes data for fast retrieval.
- Search Ranking Service (SRS): Processes indexed data to return ranked results.
- Data Ingestion Service (DIS): Handles web crawlers and external data ingestion pipelines.
- Booking Service (BS): Manages flight booking operations and reservation status updates.
- Inventory Management Service (IMS): Synchronizes seats and inventory with airlines or Global Distribution Systems.
- Fraud Detection Service (FDS): Analyzes transactions in real-time to detect potential fraud.

Communication Protocols:

- gRPC for internal microservice communication due to its low latency and efficient data serialization.
- RESTful APIs for inter-service communication/ For synchronous service-to-service communication..
- WebSocket for real-time bi-directional communication between clients and the RTCS.
- Message Queues: (e.g., Apache Kafka) for asynchronous communication and decoupling between services.

Databases

1. SQL Database(Use a leaderless approach (e.g., **DynamoDB** or **Cassandra**) for high availability and eventual consistency.):
    - Use for structured data such as user profiles, permissions, and document metadata.
    - Example: **PostgreSQL** for its strong ACID compliance and advanced JSON support.
    - **Partition** by user_id or object_id to ensure even data distribution
2. NoSQL Database
    - *Use a leaderless approach with consistent hashing to partition and replicate data across nodes.*
    - Use for real-time collaborative data (e.g., operational transformation/CRDT states) and file storage metadata.
    - Example: MongoDB for flexible schema design and high write throughput.
3. Data Warehouse:
    - Use for aggregated data and historical analytics. Example: Google BigQuery or Snowflake for OLAP (Online Analytical Processing).
    - Partition by time (e.g., day or hour) for efficient historical queries.
4. Time-Series Database:
    - Use for tracking time-stamped metrics like impressions and click-through rates. Example: InfluxDB or TimescaleDB.
5. Blob Storage:
    - Use for storing large files such as images or exported document formats.
    - Example: AWS S3 or Azure Blob Storage.
6. Search Engine:
    - Use for fast product searches and recommendations.
    - Example: **Elasticsearch** for full-text search and filtering capabilities.
7. Cache:
    - Use for frequently accessed data like product details and user sessions.
    - Example: **Redis** for in-memory caching.

Load Balancing and Service Discovery

- Load Balancing: Use to distribute traffic among service instances:
  - **NGINX** *Software-based , Highly configurable, Requires manual setup* (A high-performance web server and reverse proxy that handles large traffic with low resource usage. Benefits: Supports caching, load balancing, and SSL termination, making it versatile and efficient)for scaling web applications.
  - **AWS Elastic Load Balancer** *Cloud-based, Limited to AWS-specific, Automatically scales* (Automatically distributes incoming traffic across multiple targets to ensure availability and fault tolerance. Benefits: Scalable, integrated with AWS services, and supports health checks for dynamic traffic routing.)
- Service Discovery: Use a tool like
  - **Consul** *Supports multi-datacenter, Written in Go, microservices in Spring Cloud ecosystems.* (A multi-purpose tool for service discovery, health checks, secure communication, and configuration management. Ideal for complex, multi-datacenter setups and hybrid environments.)
  - **Eureka** *single-datacenter, Written in Java, Ideal for hybrid cloud* (A lightweight, Java-based service discovery tool, tightly integrated with the Spring ecosystem. Best for Spring Cloud microservices)
  - Both for dynamic service discovery and health monitoring.(Tools for dynamic service discovery and health checks in microservice architectures. Benefits: Automatically updates service locations, enabling seamless scaling, fault tolerance, and dynamic routing.)

Tech Stack

- Frontend:
  - **React.js** with **WebSocket** integration (or Angular for big projects and dashboards).
  - **React Native** for cross-platform mobile applications
- Backend:
  - **Node.js** for RTCS
  - **Python** (FastAPI) for RESTful APIs and for DMS and NS.
  - **Java/Scala** for stream processing with Kafka Streams or Apache Flink.
- Databases: **PostgreSQL**, **MongoDB**, **AWS S3**, **BigQuery**(specialized database -time-stamped data, perfect for tracking metrics like system performance or IoT sensor reading), **InfluxDB**(time-series BD built on PostgreSQL, -> scalability and SQL analisys).
- Infrastructure: **Kubernetes** for container orchestration, AWS for hosting.
- Messaging: **Apache Kafka** for event streaming (System for handling real-time data, letting apps share and process information quickly and reliably. Is good because it handles large amounts of data in real-time,ideal for tracking events - user activity or system logs)
- Monitoring:
  - **Prometheus**(monitoring tool for collecting + querying metrics) and **Grafana**(Visualization and dashboard tool)
  - **OpenTelemetry** (standard toolkit for collecting telemetry data (traces, metrics, logs)) or **Jaeger** (Distributed tracing tool for visualizing) for distributed tracing (End-to-end visibility of requests across microservices, easier to pinpoint performance bottlenecks)

Availability, Performance, Consistency, Scalability

1. Availability:
   - Use replication in MongoDB and PostgreSQL.
   - Ensure high availability using multi-region deployments.
   - Replicate services and databases across multiple regions.
   - Use leader election (e.g., ZooKeeper) to ensure failover.
2. Performance:
   - Use caching (Redis) to minimize database hits.
   - Optimize WebSocket connections using load balancers with sticky sessions.
   - Implement sharding and indexing in databases.
3. Consistency:
   - Use eventual consistency in RTCS for collaborative updates.
   - Ensure strong consistency in DMS for metadata updates.
   - For critical operations (e.g., ad billing), prioritize strong consistency.
   - For analytics, allow eventual consistency to prioritize performance.
4. Scalability:
   - Horizontal scaling for stateless services.
   - Partition data in both SQL and NoSQL databases, (e.g., by region for ride data) and data pipeline scaling to handle increased data volume.
5. Centralized Logging: **ELK Stack** (Elasticsearch, Logstash, Kibana) or **Splunk** for aggregating logs from all microservices. *Benefit*: Quick troubleshooting when issues arise across distributed components.

Trade-offs

1. Databases:
    - SQL provides strong consistency but can be less scalable.
    - NoSQL offers flexibility and scalability but may lead to eventual consistency.
2. Consistency vs. Availability:
    - Real-time collaboration prioritizes availability over strong consistency.
    - Metadata updates prioritize consistency to avoid conflicting permissions.
3. Performance vs. Cost:
    - High-performance caching (Redis) increases cost.
    - Multi-region deployments ensure availability but add latency and cost.
4. Consistency vs. Availability: Prioritize availability for user-facing operations but ensure consistency for transactions.

Transit Security

Objective: Protect data exchanged between clients, servers, and databases to prevent interception or tampering.

- Encryption:
    - Use TLS (Transport Layer Security) to encrypt all communication channels (e.g., HTTPS for APIs, WebSocket Secure (WSS) for real-time editing).
    - Encrypt communication between microservices using mutual TLS or gRPC with encryption enabled.
- Data Integrity:
    - Use hashing algorithms (e.g., HMAC) to verify data integrity during transmission.
- Certificate Management:
    - Use trusted Certificate Authorities (CAs) for server certificates.
    - Automate certificate rotation with tools like Let's Encrypt or AWS Certificate Manager.

Authentication Security

Objective: Ensure only authorized users can access the system.

- Authentication Methods:
    - Implement OAuth 2.0 for secure user authentication and third-party integrations.
    - Use OpenID Connect (OIDC) for federated login (e.g., Google or Microsoft account login).
- Token-Based Authentication:
    - Use JSON Web Tokens (JWT) for stateless authentication. Ensure tokens are signed and include expiration times to prevent reuse.
    - Rotate and revoke tokens regularly using a blacklist/whitelist mechanism.
- Multi-Factor Authentication (MFA):
    - Require MFA for high-privilege actions, such as document sharing or administrative access.
- Password Security:
    - Hash and salt passwords using algorithms like Argon2, bcrypt, or PBKDF2.
    - Enforce strong password policies and offer options for password recovery via secure email/SMS verification.

API Security

Objective: Prevent unauthorized access and protect APIs from attacks like injection, DDoS, or cross-site scripting.

- Authentication:
    - Use API keys or OAuth tokens for identifying and authenticating API consumers.
    - Require signed requests to verify the authenticity of API calls.
- Rate Limiting and Throttling:
    - Implement rate limiting at the API Gateway level to mitigate abuse (e.g., per user, per IP).
- Input Validation and Sanitization:
    - Validate all incoming data to prevent SQL injection, cross-site scripting (XSS), or other injection attacks.
- CORS (Cross-Origin Resource Sharing):
    - Restrict origins allowed to interact with the system using CORS policies.
- Monitoring and Logging:
    - Monitor API usage with tools like AWS CloudWatch, Elastic Stack, or DataDog.
    - Log all access attempts, including failed ones, to detect potential breaches.
- Security Headers:
    - Use headers like `X-Content-Type-Options`, `Content-Security-Policy`, and `X-Frame-Options` to protect against common web vulnerabilities.

Permissions and Authorization
Objective: Ensure users can only perform actions they are permitted to.

- Role-Based Access Control (RBAC):
  - Assign roles (e.g., Viewer, Editor, Admin) to users based on their access needs.
  - Define granular permissions for actions like viewing, editing, sharing, and deleting documents.
- Document-Level Permissions:
  - Implement access control at the document level (e.g., read-only, edit, comment).
  - Allow document owners to grant and revoke access.
- Context-Aware Authorization:
  - Consider context (e.g., location, device type, IP address) to enforce adaptive access controls.
- Audit Trails:
  - Maintain logs of permission changes and document access to track potential misuse.

Data Processing

1. **Stream Processing:**
   - Use for real-time analytics, like calculating CTR (Click Through Rate) or impressions per second. ()
   - Tools: Apache Flink(Advanced stream processing with low latency and stateful handling. Best for complex, large-scale pipelines) , Kafka Streams (Lightweight, Kafka-integrated for simpler real-time processing.Ideal for Kafka-centric setups with straightforward analytics)
2. **Batch Processing:**
   - Use for aggregating historical data, generating reports, and training machine learning models.
   - Tools: Apache Spark(Optimized for speed and versatility, supports in-memory and real-time processing), Hadoop(Suited for batch processing and long-term data storage.).

Data Ingestion

1. **Path:** User actions (e.g., browsing, adding to cart) and system events are sent to the **Analytics Service**. Events are ingested via Kafka for streaming to downstream services.
2. **Real-Time Data:** Ingest clickstream data for personalization and dynamic recommendations.
3. **Batch Data:** Periodically import bulk data (e.g., inventory updates from suppliers) via ETL pipelines.

Data Retrieval

1. **Path:** Dashboards and APIs query services like PCS, OMS, and Analytics for user-facing data. Product searches hit Elasticsearch for fast, filtered results.
2. **Caching:** Use Redis for caching product details and session data to reduce load on databases.

- **gRPC**: A system for apps to communicate by sending requests and getting responses over the internet. It works by using small, efficient messages with HTTP/2 for speed. | It's fast, supports many languages, and is great for real-time communication between services.
- **DynamoDB**: A fully managed database by AWS, designed for fast and reliable data storage. It automatically scales to handle large amounts of traffic.| Easy to use, highly scalable, and provides fast performance for applications needing quick reads and writes.
- **Cassandra**: An open-source database designed for managing large amounts of data across multiple servers with no single point of failure | Highly scalable, fault-tolerant, and ideal for applications needing consistent uptime and handling big data.
- **Redis:** A fast, in-memory database for storing and retrieving data quickly, often used for caching and real-time applications | Extremely fast, easy to use, and ideal for reducing database load or handling time-sensitive data like user sessions or leaderboards.
- **RESTful APIs**: A standard way for services to communicate over HTTP using clear and structured requests (e.g., GET, POST) | Simple, widely used, and easy to implement, making it ideal for connecting different apps or services.
- **NGINX:** A high-performance web server and load balancer that distributes traffic among servers efficiently. | Handles large traffic volumes, supports caching, and ensures availability by spreading the load.
- **Consul:** A tool for service discovery, health checking, and secure communication in distributed systems. | Helps microservices find and connect with each other automatically, ensuring scalability and fault tolerance.
- **Eureka:** A lightweight service registry for tracking running services and helping them communicate. | Simplifies microservice interactions and is tightly integrated with Java and Spring ecosystems.
- **AWS S3:** A cloud storage service for storing large files like documents, videos, and images. | Highly scalable, durable, and cost-effective for managing large volumes of data.
- **WebSocket:** A protocol for real-time, two-way communication between clients and servers. | Enables instant updates and is ideal for live features like chat or collaborative editing.