Learn to Design Cloud Architecture

Let’s build a Real Time chat system

- A real-time chat application allows users to send and receive messages instantly over the internet.

WhatsApp

Facebook Messenger

WeChat

Line Messenger

Telegram

Signal

slack

Microsoft Teams

Discord

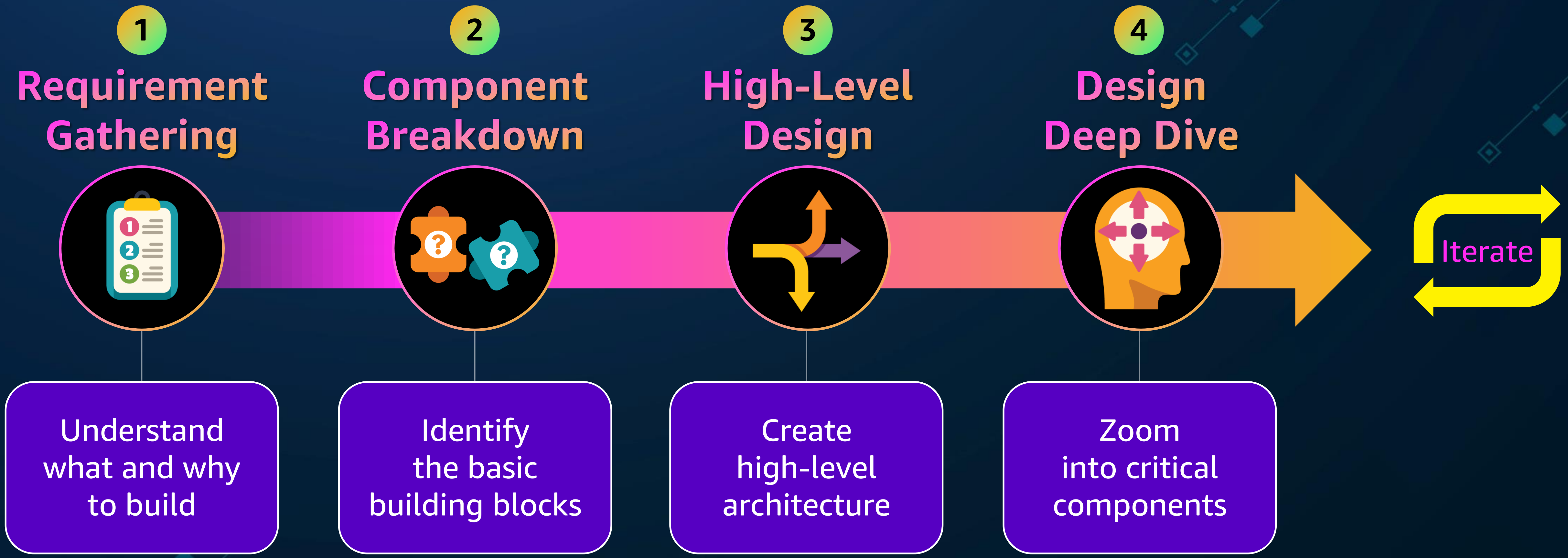
WhatsApp

- 2009:
  - WhatsApp Inc. was founded by former Yahoo employees
- 2013:
  - 200 million+ users
- 2024:
  - 2 billion+ active users

Top features:

- End-to-end encrypted chats
- Multi-device access support
- Free voice/video calling

Designing a System – A simple framework



1 **Requirement Gathering**

Understand what and why to build

2 **Component Breakdown**

Identify the basic building blocks

3 **High-Level Design**

Create high-level architecture

4 **Design Deep Dive**

Zoom into critical components

Iterate

Requirement Gathering

Functional Requirement

1 to 1 Chat

Group Chat

Media Attachment

Online/ Offline status

Delivery Confirmation

Push Notifications

Online
Offline

Non-functional Requirement

Performance

Send and receive messages in real-time

Scalability

Billions of users and billions of messages

Security

End-to-end encryption of messages

Reliability

Store messages until they are delivered

Availability

Minimal downtime or interruptions

Out of scope

- Authentication
- Encryption
- User registration
- User profiles
- Mobile app
- Audio/Video call
- Block lists
- Multiple device support

Component Breakdown

Core components

Mobile App

Front-end interface for the user

Status Service

Tracks user presence (online/offline)

Connection Service

Manages real-time connectivity

Storage Service

Storage for chat history and media

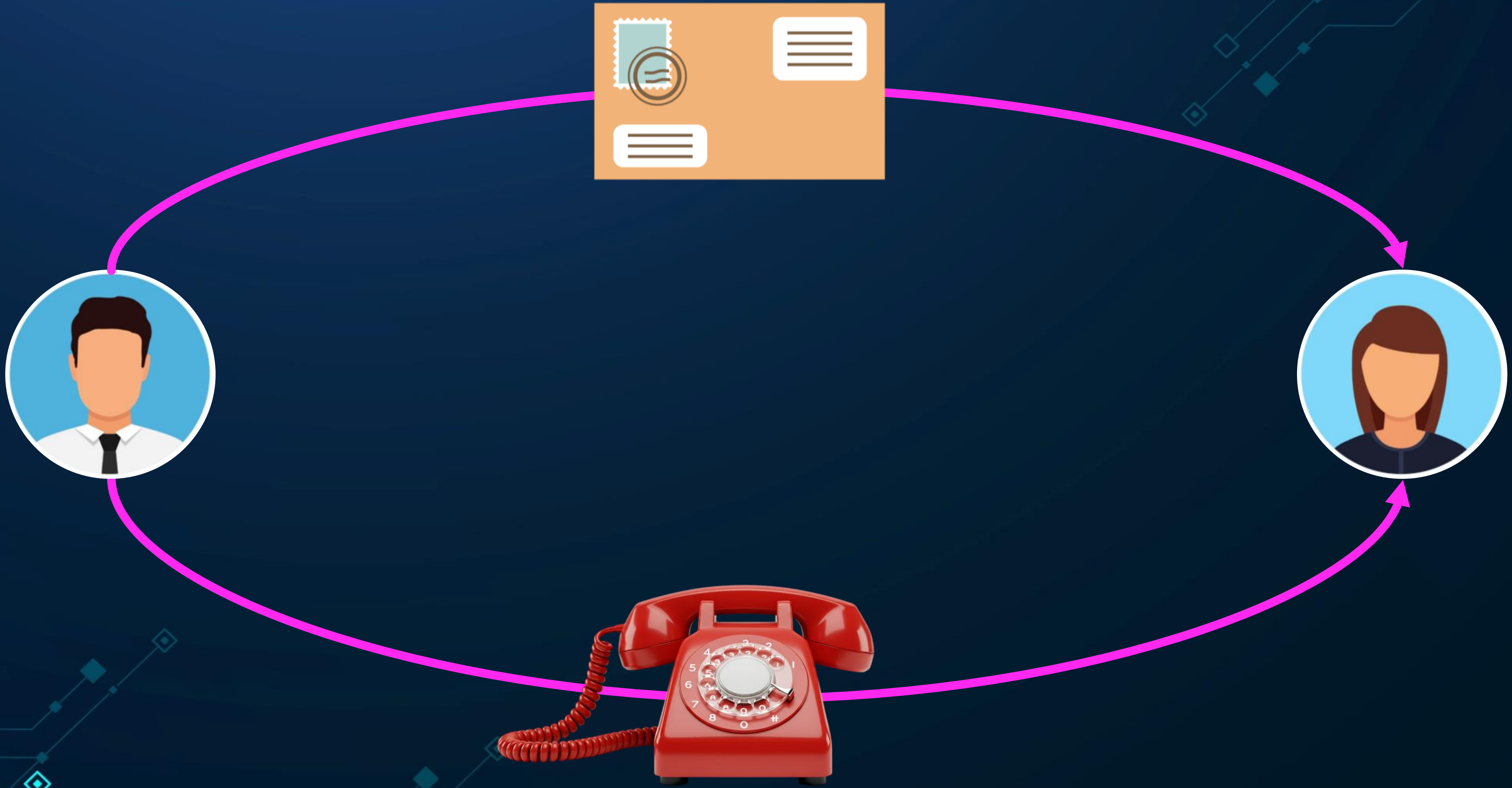
Message Service

Sending and receiving of messages.

Group Service

Manages group chat functionality

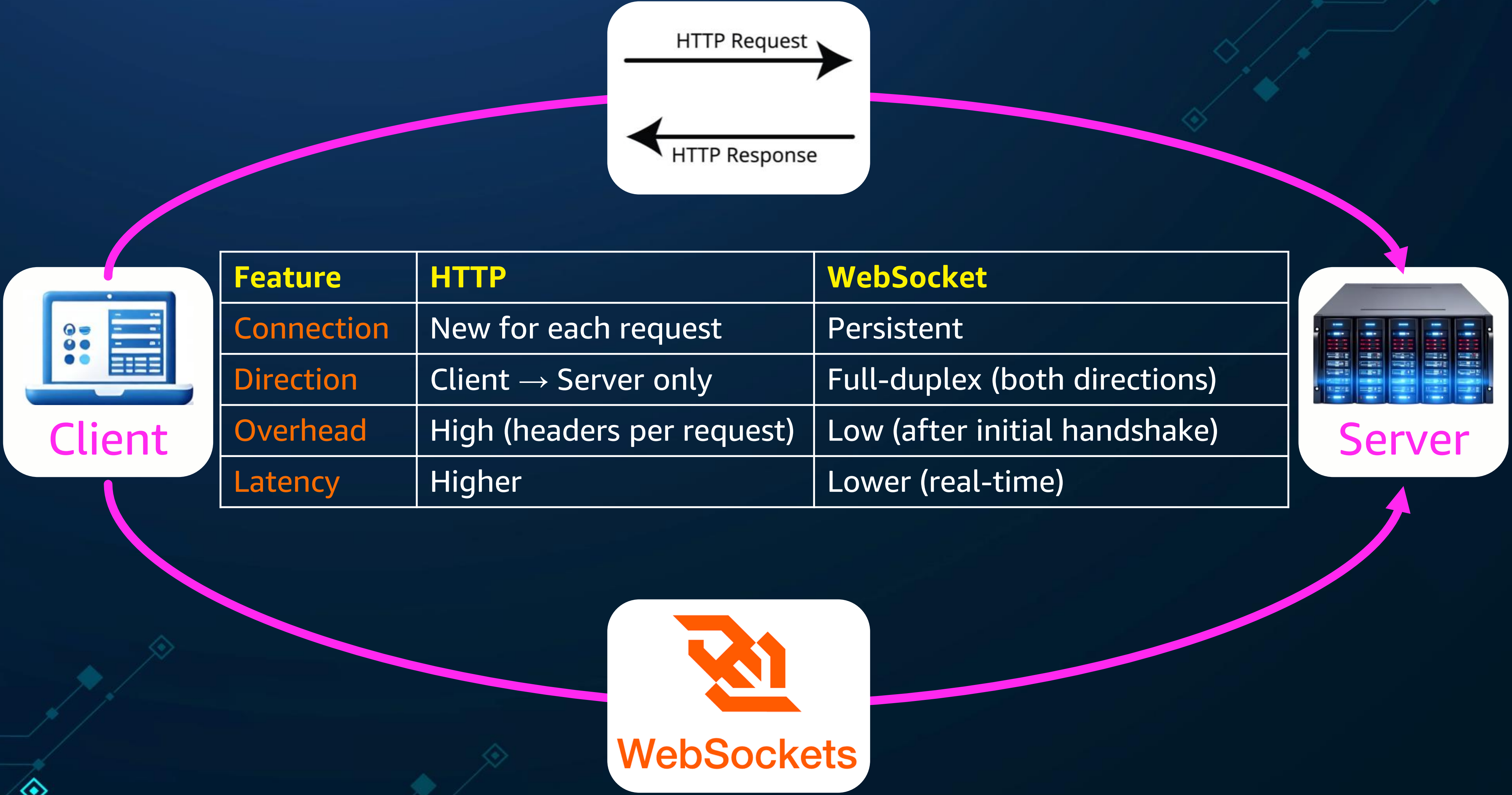
Analogy: Sending Letters vs. A Phone Call



What is WebSocket?

WebSocket is a communication protocol that provides full-duplex communication channels over a single, long-lived TCP connection.

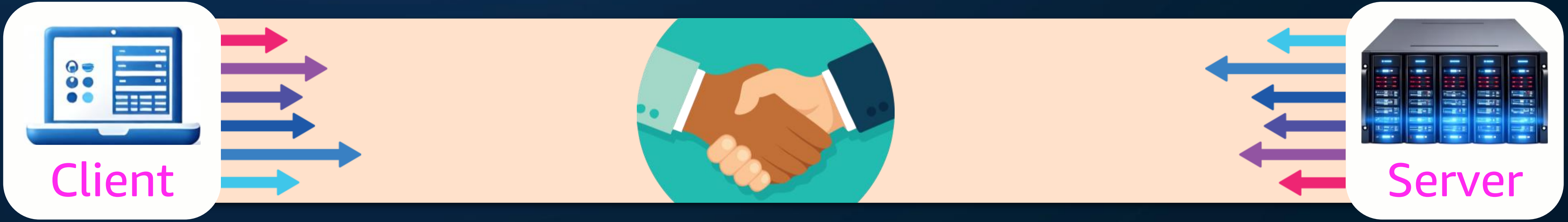
Analogy: Sending Letters vs. A Phone Call



Feature	HTTP	WebSocket
Connection	New for each request	Persistent
Direction	Client → Server only	Full-duplex (both directions)
Overhead	High (headers per request)	Low (after initial handshake)
Latency	Higher	Lower (real-time)

How WebSocket Works?

Step 1	Handshake (via HTTP)
Step 2	Persistent Connection
Step 3	Transfer of Data



Common Use Cases	
Live Chats	Online Games
Stock Tickers	Collaborative Editing

Alternative to WebSockets

- MQTT
  - MQTT stands for Message Queuing Telemetry Transport. It is a lightweight, open-source messaging protocol that is widely used in the Internet of Things (IoT) for communication between devices.
- XMPP
  - The full form of XMPP is Extensible Messaging and Presence Protocol. It is a communication protocol used for instant messaging, presence information, and contact list maintenance. XMPP is based on XML and enables near-real-time exchange of structured data between network entities.

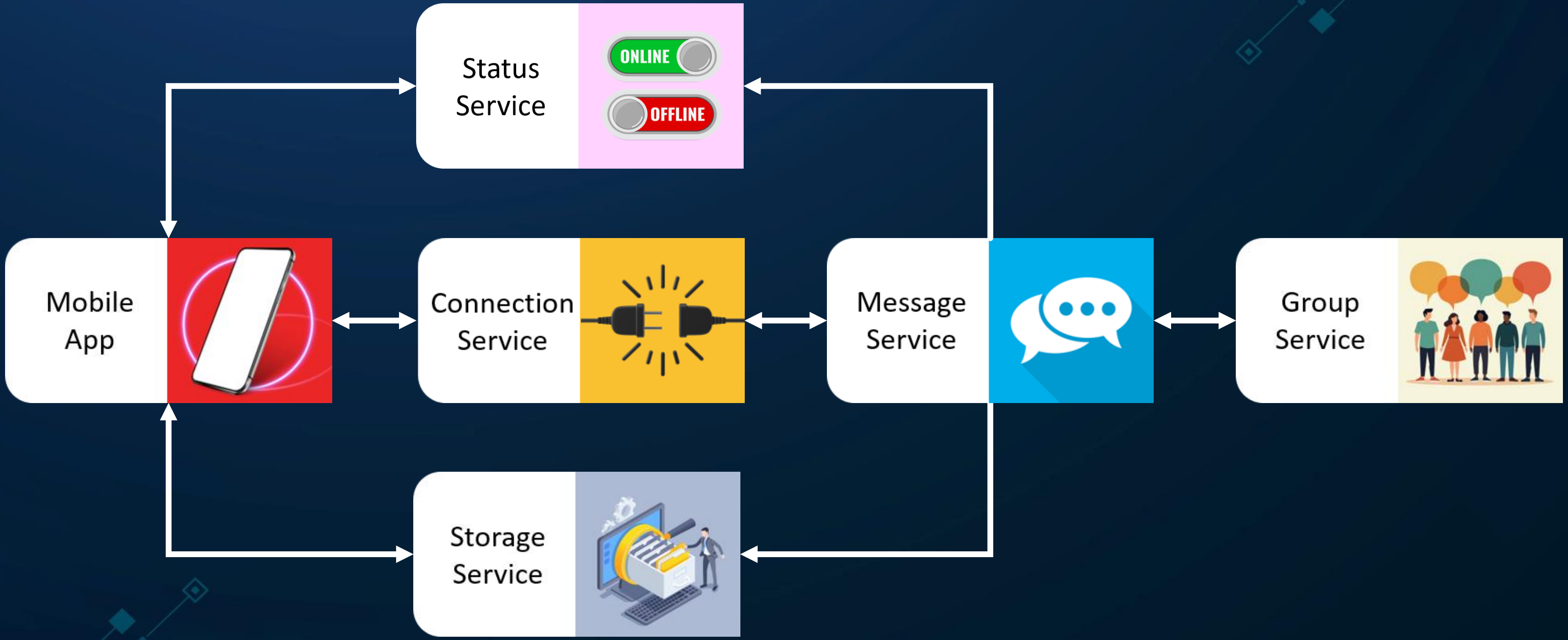
Feature	WebSocket	MQTT	XMPP
Communication Pattern	Full-duplex, bidirectional streams	Publish/subscribe via broker	Client-server messaging with presence
Data Format	Binary or text frames	Lightweight binary (topics + payload)	XML stanzas (text-based, verbose)
Built-in Presence	No	No	Yes, presence, contact lists, subscriptions
Protocol Overhead	Low (once handshake completes)	Very low, optimized for IoT/mobile	High (XML verbosity, overhead)
Applications (likely usage)	• Facebook Messenger (Custom WebSocket) • WeChat • Discord • Microsoft Teams • Line Messenger • Signal	• Facebook Messenger (Android/iOS clients) • Instagram DM	• WhatsApp - Custom XMPP "FunXMPP”

How to implement Web Sockets?

Platform	**Lightweight Option** These give you direct access to the WebSocket protocol (without additional logic or structure. You'll need to manually handle reconnects, rooms, user state, etc.	**High-Level Option** These libraries abstract away the raw WebSocket protocol and give you extra features out of the box, such as: Automatic reconnection Presence detection etc.
Java	Jetty, Tyrus	Spring WebSocket (STOMP)
.NET	System.Net.WebSockets	SignalR
Python	websockets, FastAPI	Django Channels, Socket.IO
Node.js	ws	Socket.IO

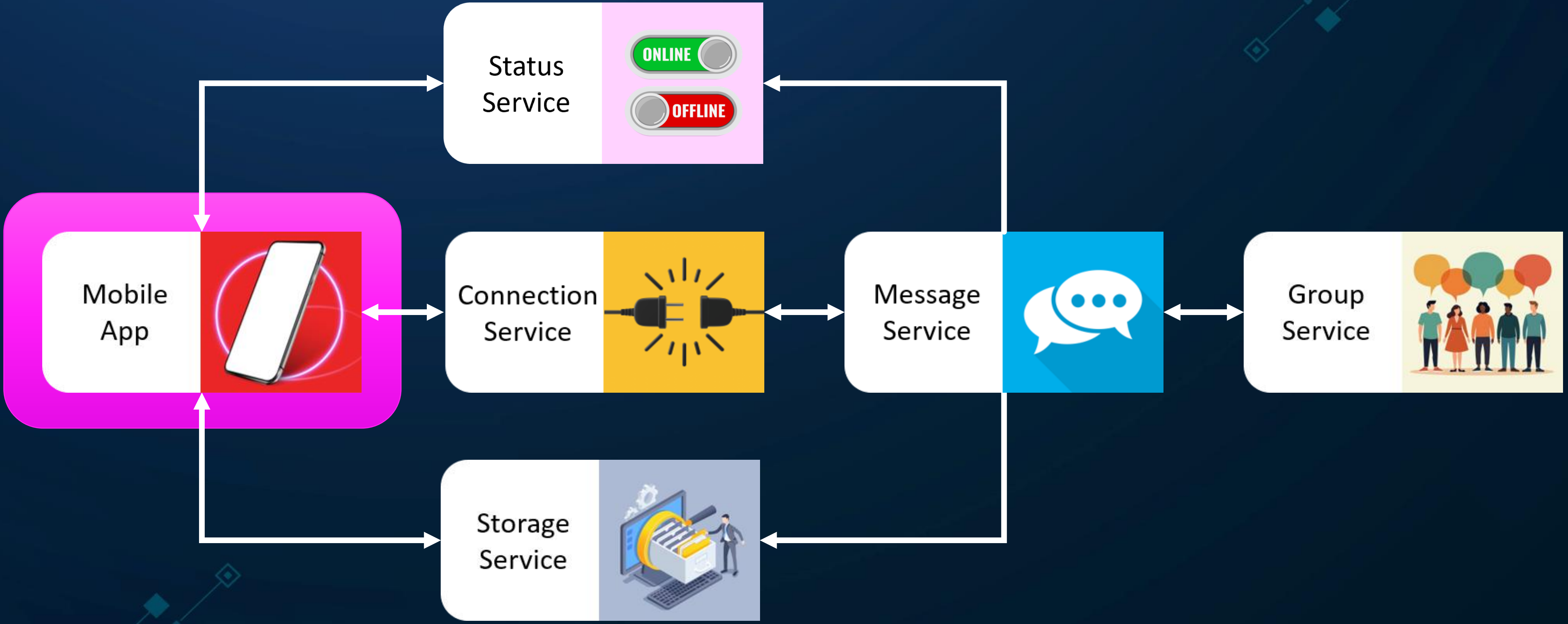
High-Level Design

High-level design

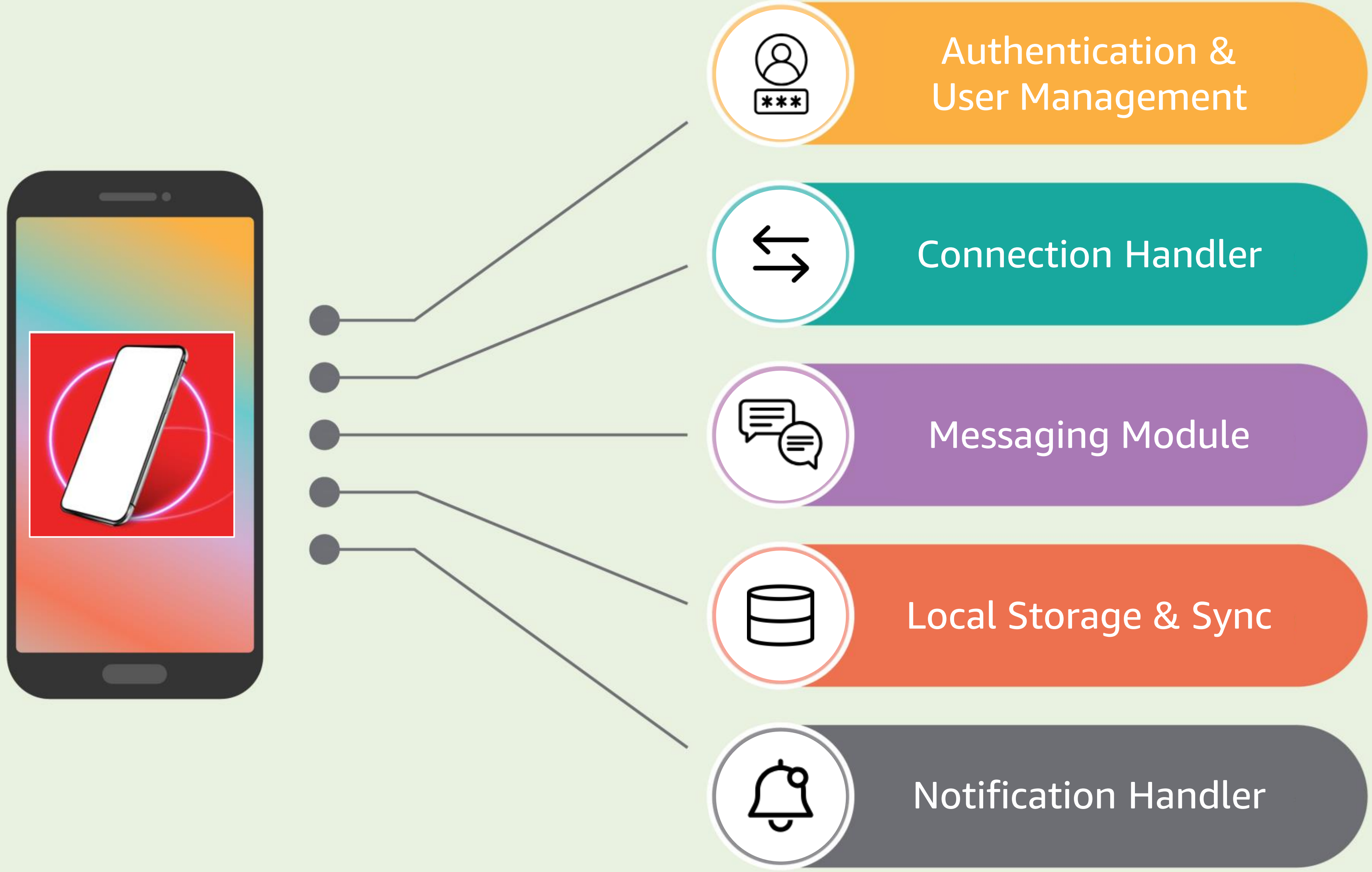


Design Deep Dive

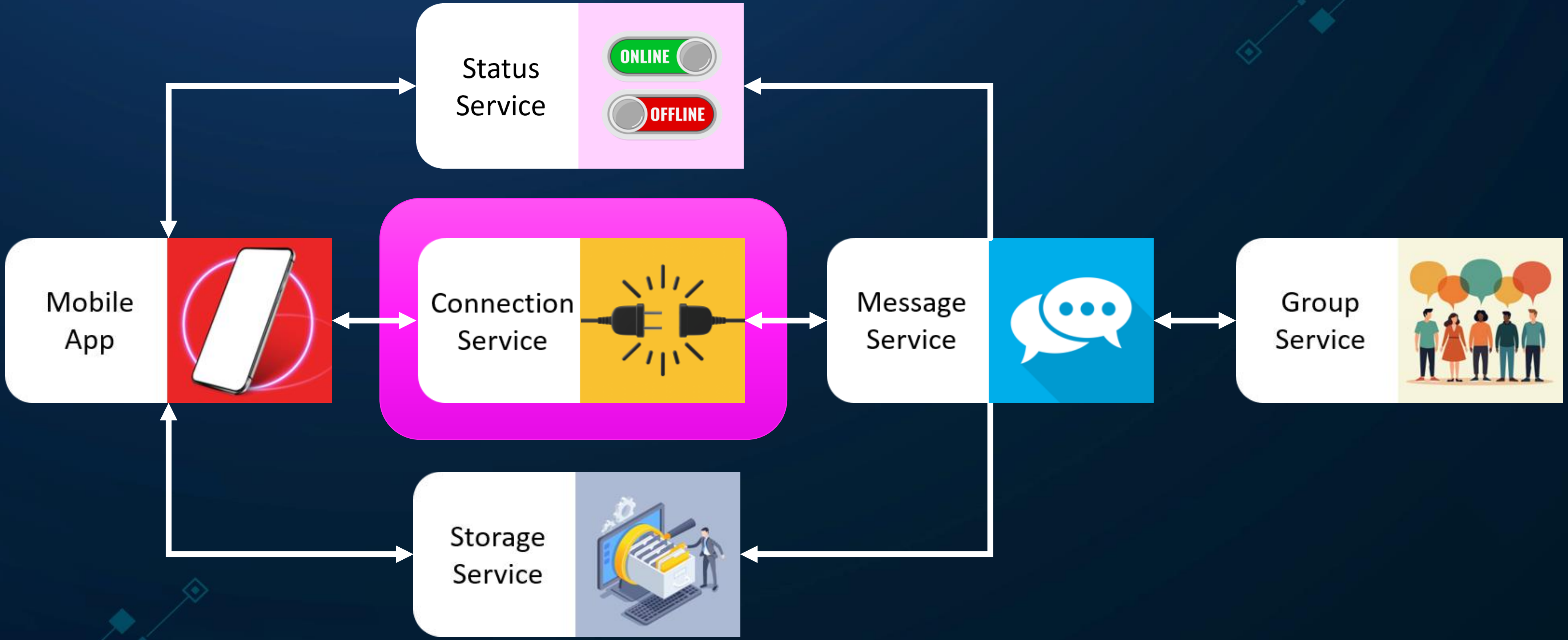
Mobile App


Status Service
ONLINE
OFFLINE
Mobile App
Connection Service
Message Service
Group Service
Storage Service

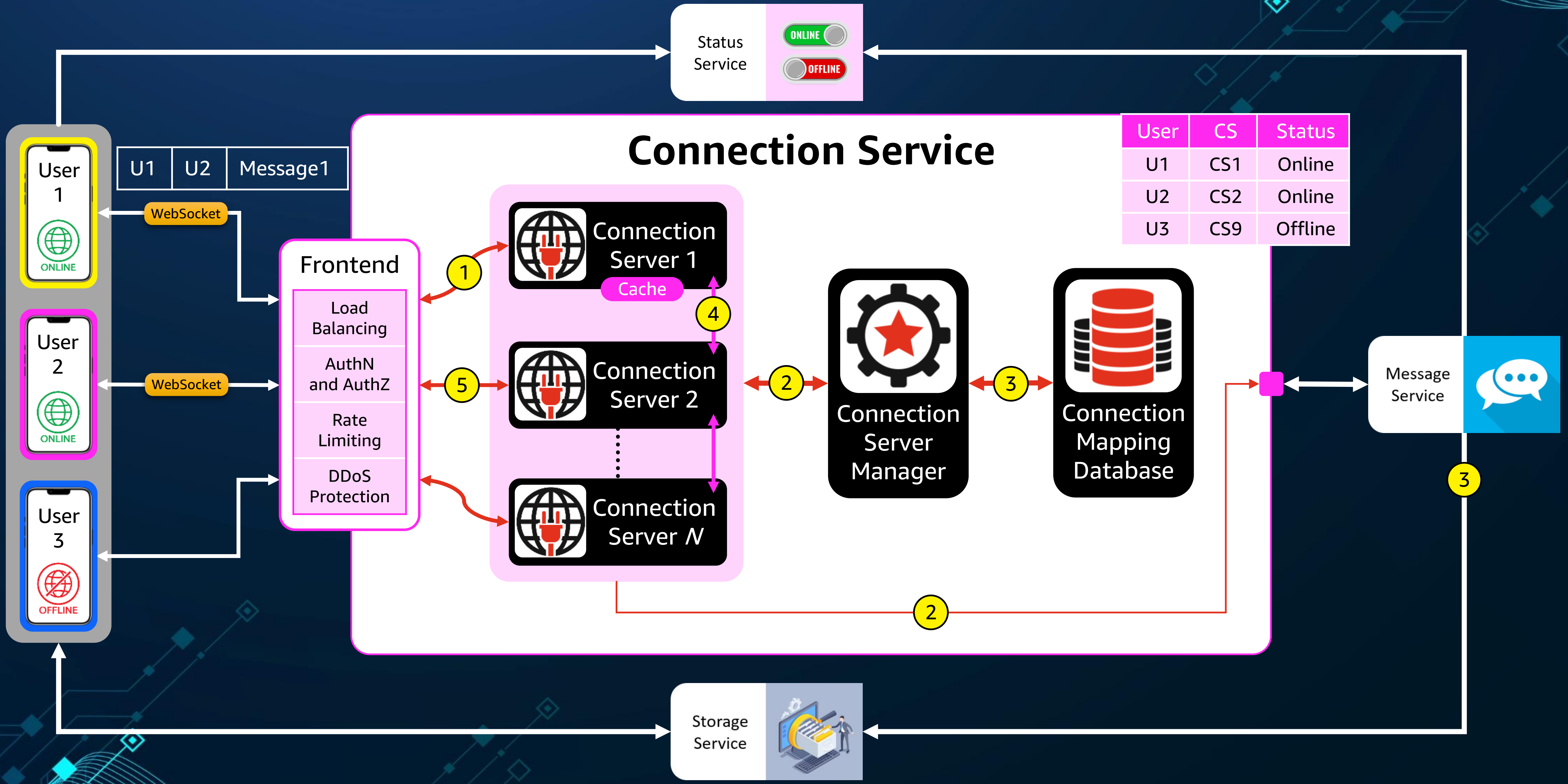
Mobile App


Authentication & User Management
Connection Handler
Messaging Module
Local Storage & Sync
Notification Handler

Connection Service



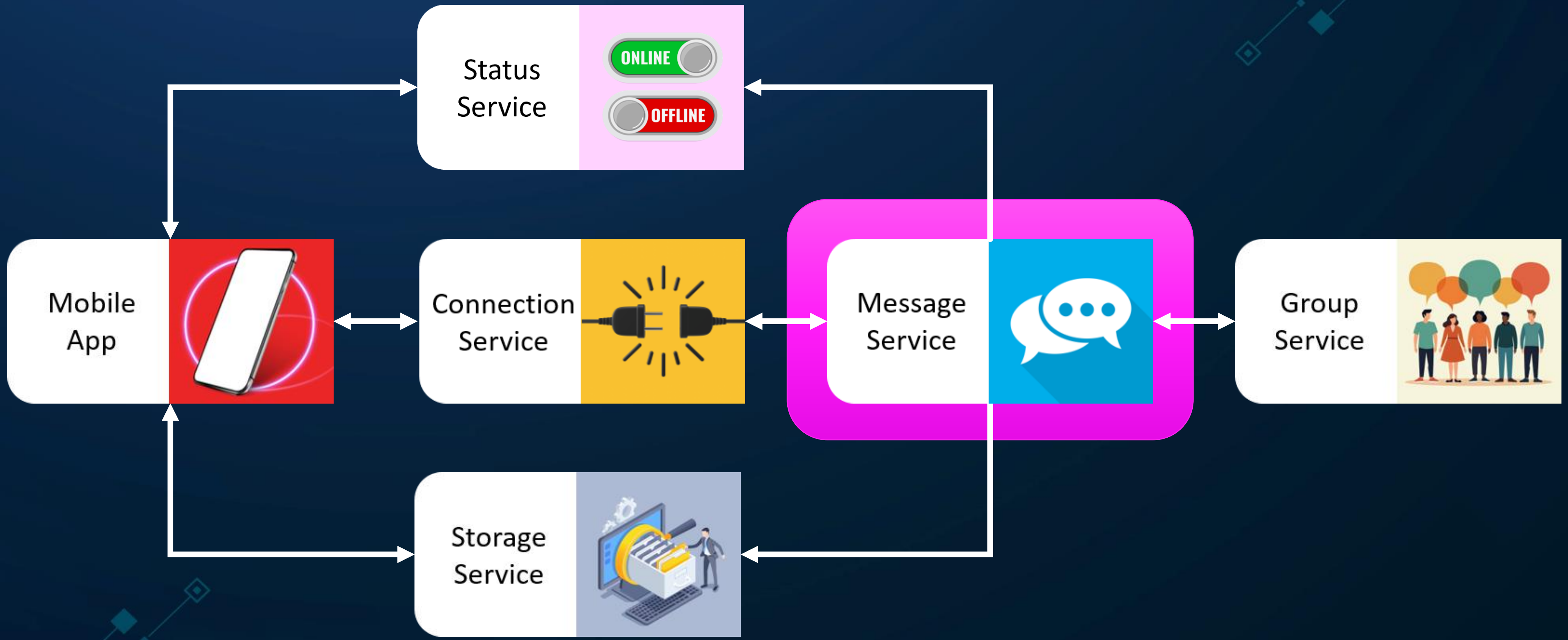
Connection Service



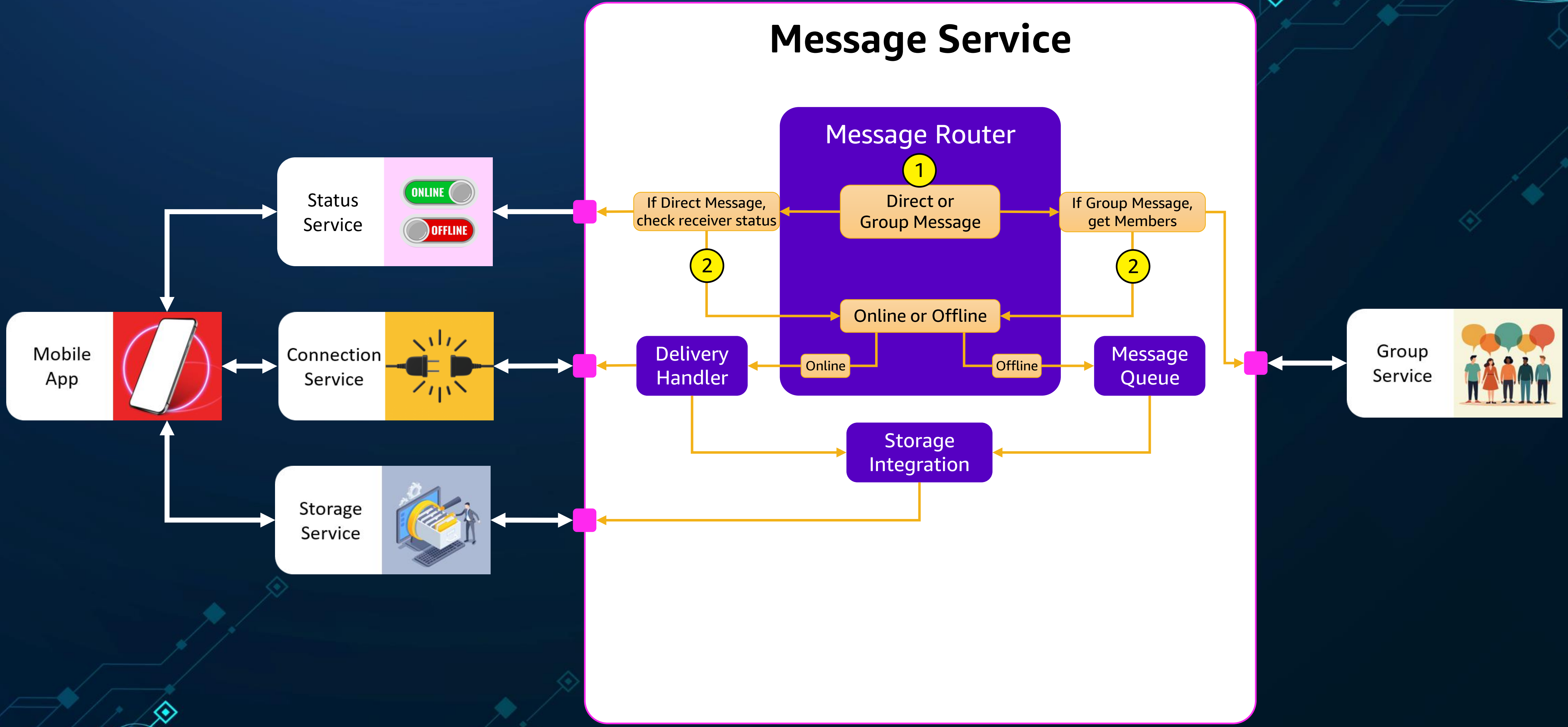
U1	U2	Message1

User	CS	Status
U1	CS1	Online
U2	CS2	Online
U3	CS9	Offline

Message Service

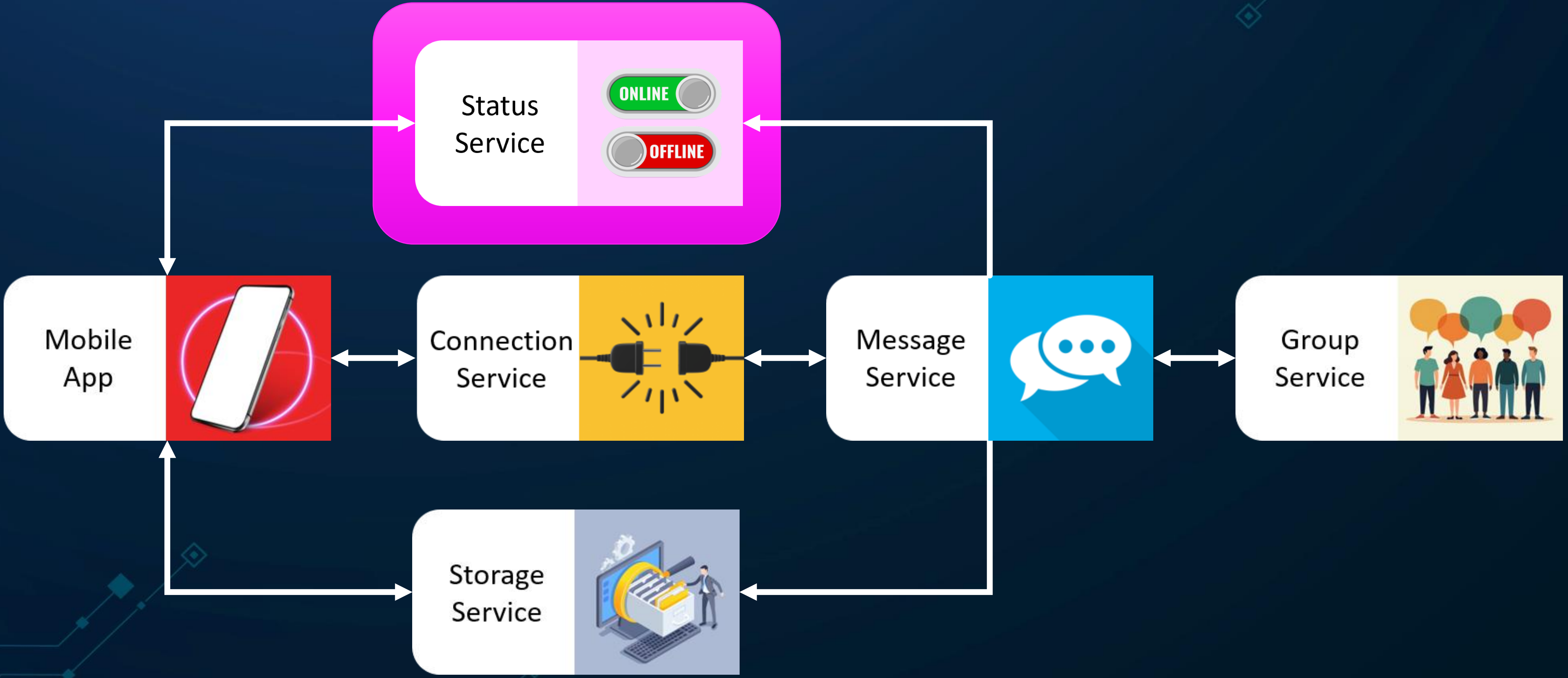


Message Service



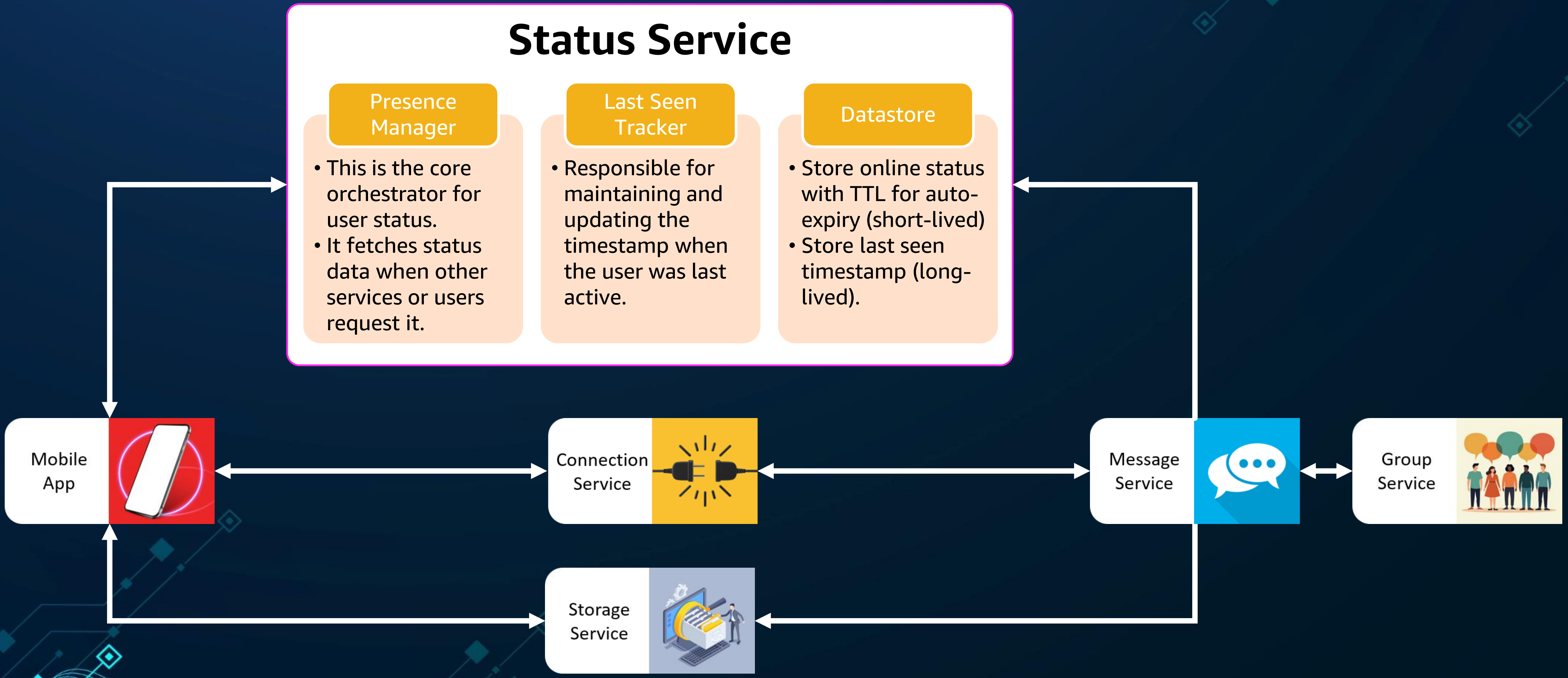
Status Service

- Track user presence (online/offline) and last seen time.



Status Service

- Track user presence (online/offline) and last seen time.



Status Service

Presence Manager

- This is the core orchestrator for user status.
- It fetches status data when other services or users request it.

Last Seen Tracker

- Responsible for maintaining and updating the timestamp when the user was last active.

Datastore

- Store online status with TTL for auto-expiry (short-lived)
- Store last seen timestamp (long-lived).

Mobile App

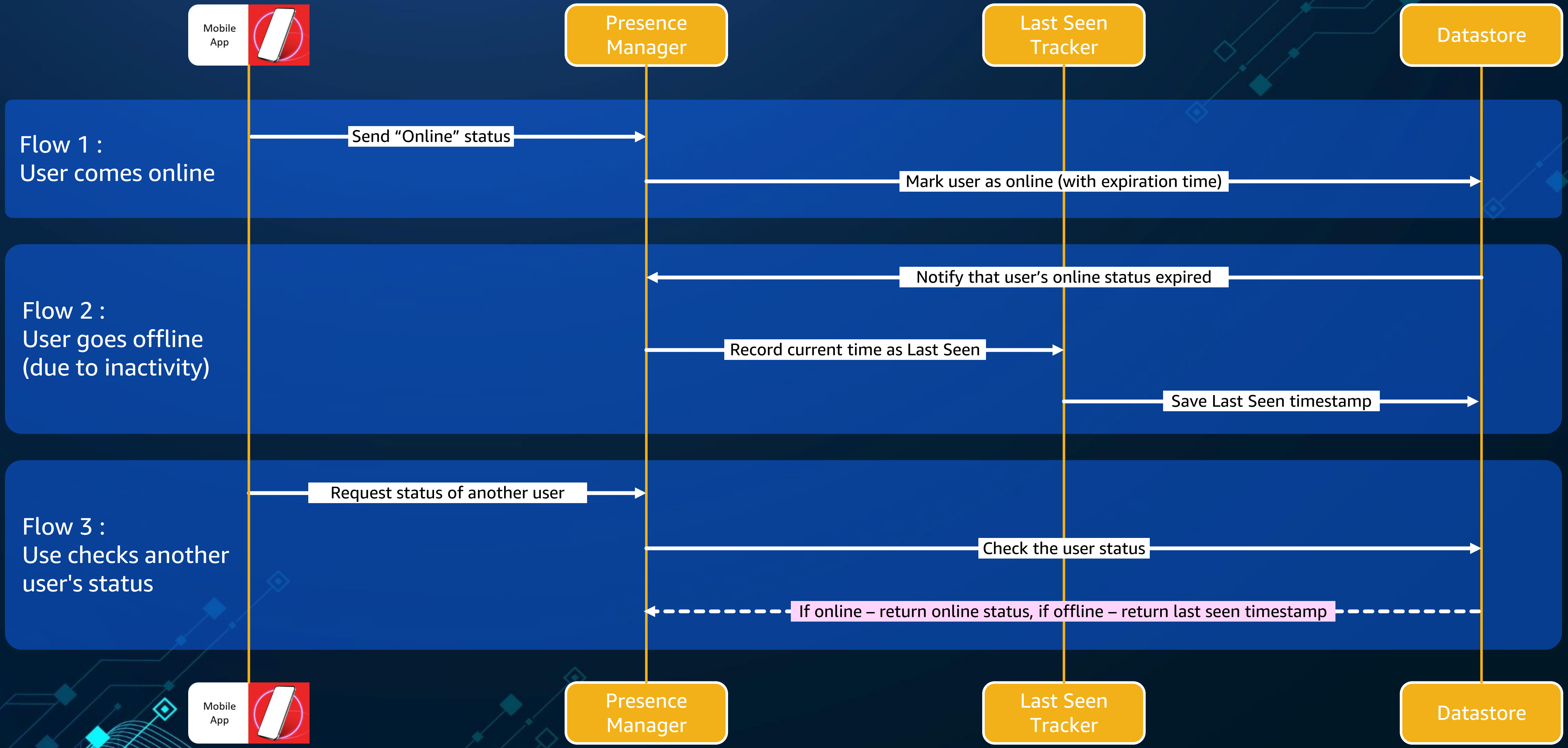
Connection Service

Message Service

Group Service

Storage Service

Status Service



Mobile App

Presence Manager

Last Seen Tracker

Datastore

Flow 1 :
User comes online

Send “Online” status

Mark user as online (with expiration time)

Flow 2 :
User goes offline
(due to inactivity)

Notify that user’s online status expired

Record current time as Last Seen

Save Last Seen timestamp

Flow 3 :
Use checks another
user's status

Request status of another user

Check the user status

If online – return online status, if offline – return last seen timestamp

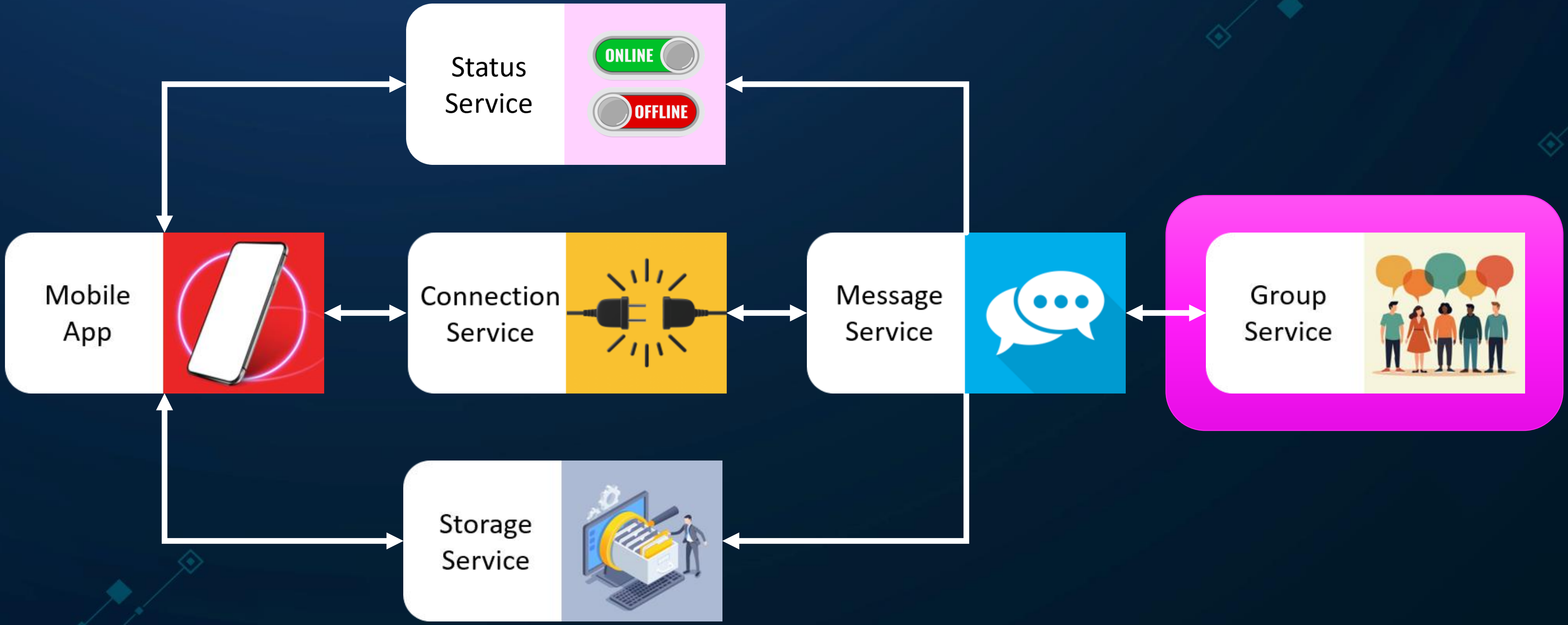
Mobile App

Presence Manager

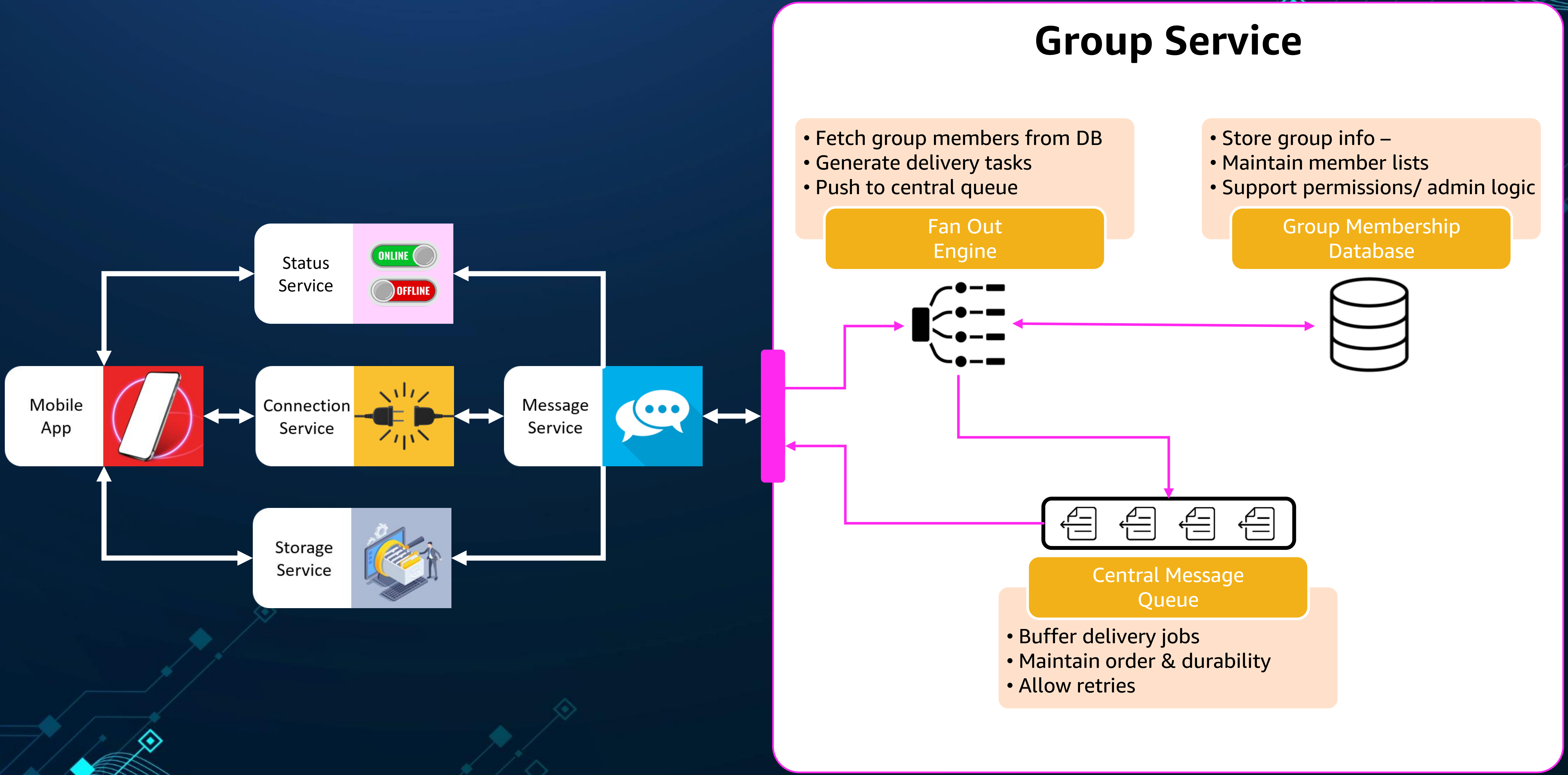
Last Seen Tracker

Datastore

Group Service



Group Service



Group Service

- Fetch group members from DB
- Generate delivery tasks
- Push to central queue

Fan Out Engine

- Store group info –
- Maintain member lists
- Support permissions/ admin logic

Group Membership Database

Central Message Queue

- Buffer delivery jobs
- Maintain order & durability
- Allow retries

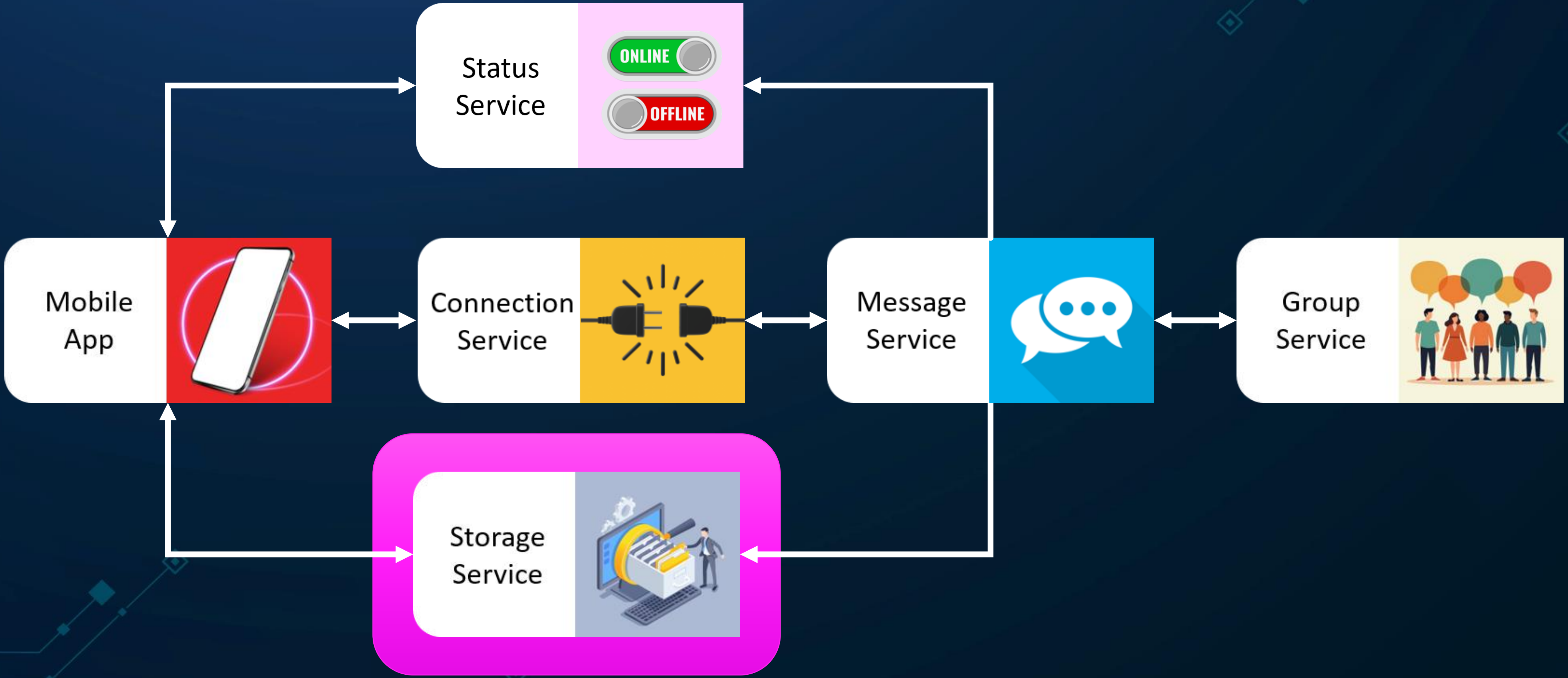
Fan out design – Analogy

- Want to share a party invitation to your colleagues

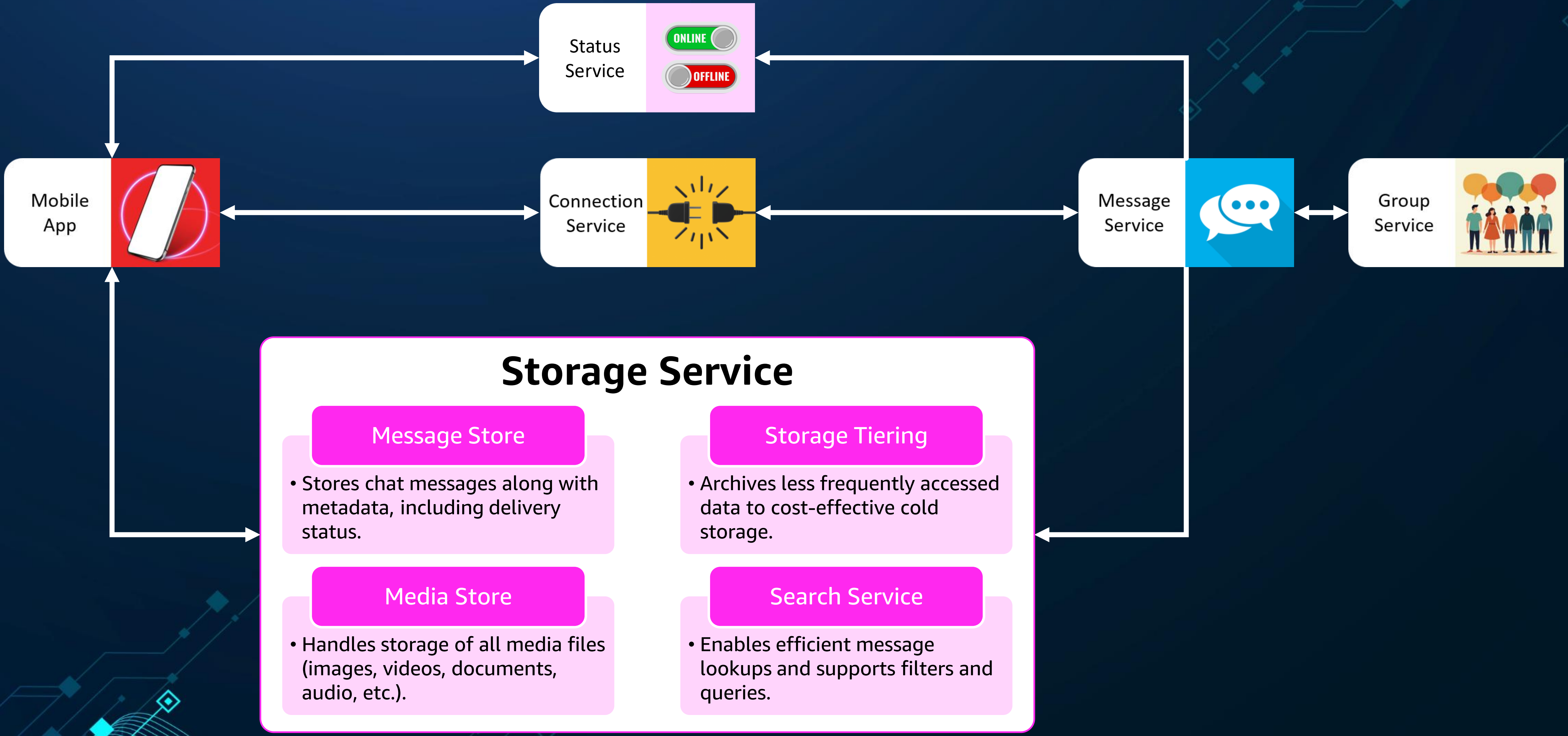
Send individual letters to each person	Post the invitation once on a notice board	Cards for close friends, board for everyone else
Everyone gets it quickly, but it’s resource-intensive	Efficient and scalable, but delivery timing depends on the reader	Mix of speed (for VIPs) and scale (for general public)
Fan-out on Write	Fan-out on Read	Hybrid Strategy

Approach	How It Works?	Pros	Cons	Best Suited For
Fan-out on Write	Message is written to each group member’s queue or inbox at send time	• Low latency delivery • Simple read path • Easy offline support	• High write amplification (1 → N) • Costly for large groups- • Duplicate storage • Complex retries on failure	• Small to medium groups (<1,000 members) • Private group chats
Fan-out on Read	Message is stored once; delivered/fetched by users when they come online or request it	• Write once, read many • Storage-efficient • Ideal for large groups • Scales to millions of members	• Higher read latency • Must track user read pointers • Harder for real-time delivery • More complex read logic	• Large/public groups (>10K members) • Broadcast channels
Hybrid (Tiered)	Uses fan-out on write for small groups, fan-out on read for large groups	• Flexible and scalable • Optimizes both read and write • Supports varied group types and use cases	• Architecturally complex • Must manage two delivery models • Higher dev and testing effort	• Platforms like WhatsApp • Tiered user experiences (e.g., premium vs. free)

Storage Service



Storage Service



Storage Service

Message Store

- Stores chat messages along with metadata, including delivery status.

Storage Tiering

- Archives less frequently accessed data to cost-effective cold storage.

Media Store

- Handles storage of all media files (images, videos, documents, audio, etc.).

Search Service

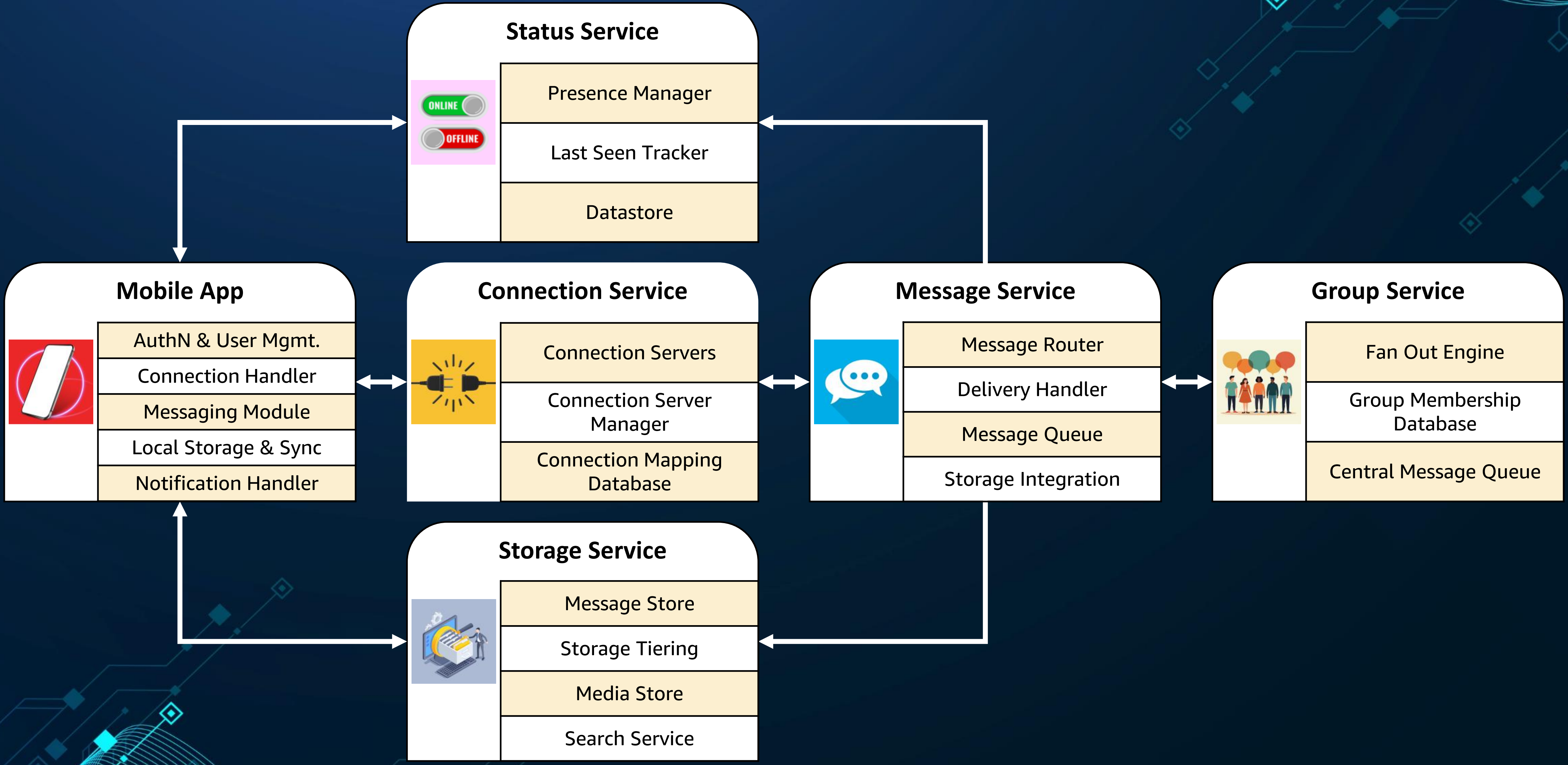
- Enables efficient message lookups and supports filters and queries.

Storage Tiering

Tier	Purpose	Storage Type (Example)
Hot Storage	For active, recently accessed media	High-speed object store (e.g., S3 Standard)
Warm Storage	For moderately accessed media	Cheaper but slower object storage (e.g., S3 Infrequent Access)
Cold Storage	For archival or rarely accessed data	Archival tier (e.g., S3 Glacier)

Criteria	Why It Matters?
Last Access Time	Most common — if media hasn’t been accessed in X days, move it.
Message Age	Move media older than N days/weeks/months regardless of access.
Message Read Status	Media in messages that are read by all group members may be cold.
User Activity	If sender and receiver are inactive, media can be tiered down.
Content Type	Larger video files may move to cold faster than small images.
Group Size & Engagement	Media from inactive or low-engagement groups can be cold-tiered.

Complete architecture



Status Service

- Presence Manager
- Last Seen Tracker
- Datastore

Mobile App

- AuthN & User Mgmt.
- Connection Handler
- Messaging Module
- Local Storage & Sync
- Notification Handler

Connection Service

- Connection Servers
- Connection Server Manager
- Connection Mapping Database

Message Service

- Message Router
- Delivery Handler
- Message Queue
- Storage Integration

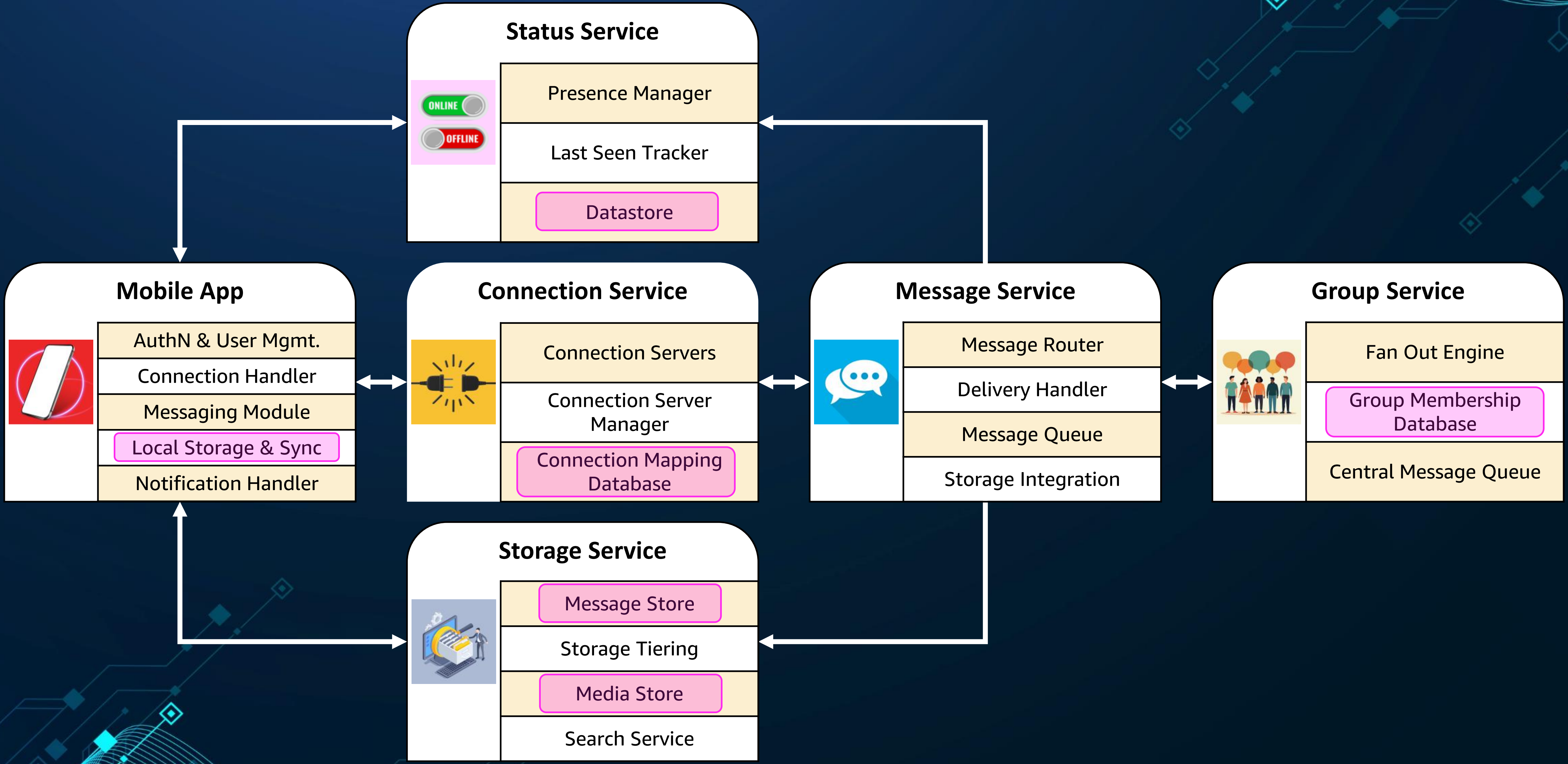
Group Service

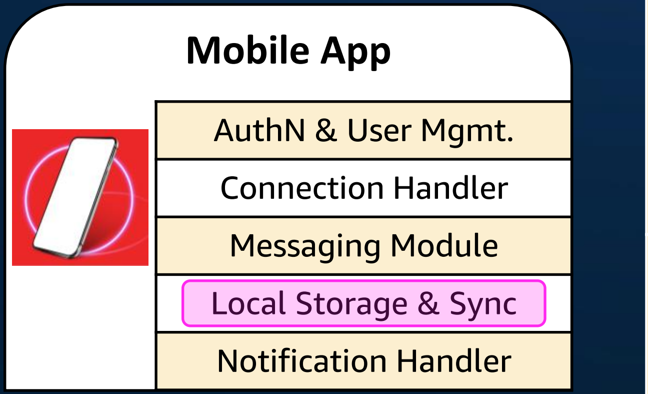
- Fan Out Engine
- Group Membership Database
- Central Message Queue

Storage Service

- Message Store
- Storage Tiering
- Media Store
- Search Service

Which storage?





Mobile App

AuthN & User Mgmt.

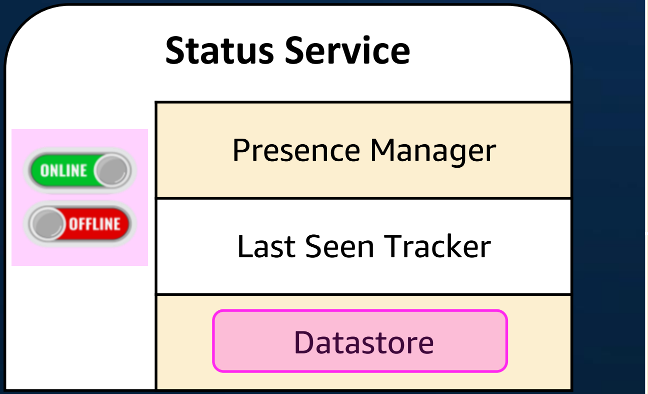
Connection Handler

Messaging Module

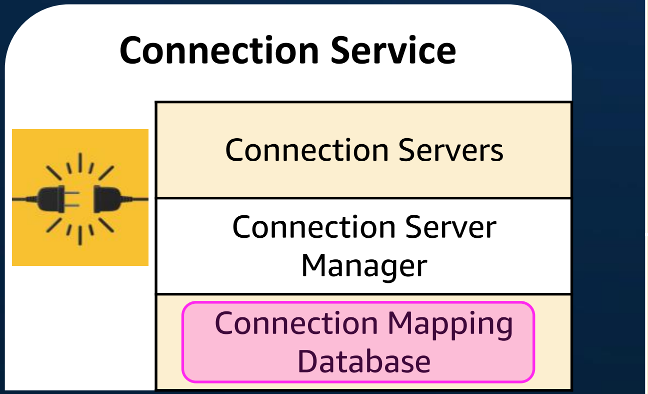
Local Storage & Sync

Notification Handler

Sub-component	Mobile App > Local Storage
Purpose	• Store chat history • Message metadata • User preferences locally
Consideration	• Structured, relational data • Needs offline access • Must support search • Pagination • Joins
Relational or Non-relational	**Relational** • Lightweight and embeddable on mobile devices • Supports ACID compliance for reliability • Efficient for structured, tabular data (messages, chats, users) • Optimized for complex local queries and indexes • E.g. - SQLite



Sub-component	Status Service > Datastore
Purpose	• Track online/offline status • Typing indicators • Last seen
Consideration	• Ephemeral and fast-changing • Needs very low latency • Time-based auto-expiry needed – Time-To-Live (TTL)
Relational or Non-relational	**Non-relational** • Ideal for high-frequency updates and quick lookups • Supports TTL for auto-expiry of status • In-memory option enables real-time speed • Horizontally scalable with predictable performance • No schema rigidity — easy to evolve status format • E.g., Redis, DynamoDB



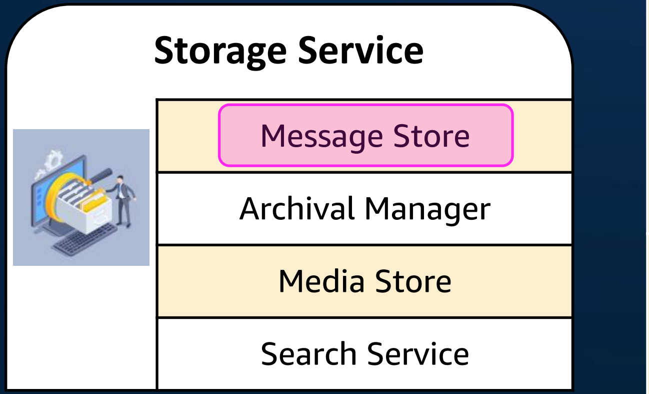
Connection Service

Connection Servers

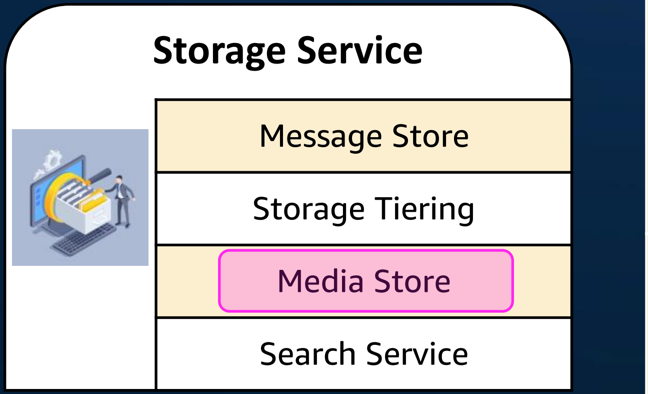
Connection Server Manager

Connection Mapping Database

Sub-component	Connection Service > Connection Mapping Database
Purpose	• Store mapping of user ID to active connection server
Consideration	• Real-time connection state • High concurrency • Simple key-value access pattern
Relational or Non-relational	**Non-relational** • Key-value store perfectly suits user-to-connection mapping • High-throughput and low-latency for billions of users • Built-in TTL support for auto-cleanup of stale connections • Scales horizontally across regions/data centers • No need for relational constraints or joins • E.g., Redis, DynamoDB



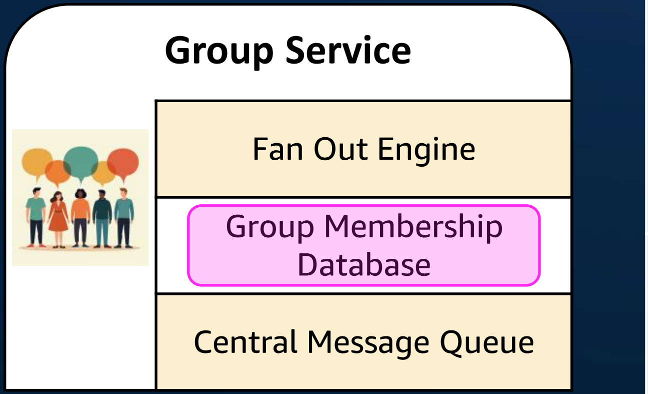
Sub-component	Storage Service > Message Store
Purpose	• Store and query text messages and metadata (excluding media)
Consideration	• Massive write throughput • Append-only model • Partitioning by user/chat needed • Flexible schema
Relational or Non-relational	**Non-relational** • Supports high-velocity writes and sequential reads • Tuned for append-only workloads (no updates) • Horizontal scalability with consistent performance • Schema flexibility as message format evolves • Denormalized design improves read performance for chat history • E.g., Cassandra, DynamoDB



Storage Service

- Message Store
- Storage Tiering
- Media Store
- Search Service

Sub-component	Storage Service > Media Store
Purpose	• Store media files • Images, Videos, Document, Audio
Consideration	• Large binary data • Static after upload • Size limits • Retention limit
Relational or Non-relational	**Object Storage** • Built to handle unstructured, large binary objects • Content is immutable and served via CDN links • Scalable to petabytes with high availability • Metadata can be indexed in a relational or NoSQL DB • Lifecycle management (e.g., archival, deletion) is built-in • E.g., Amazon S3, GCS, Azure Blob



Group Service

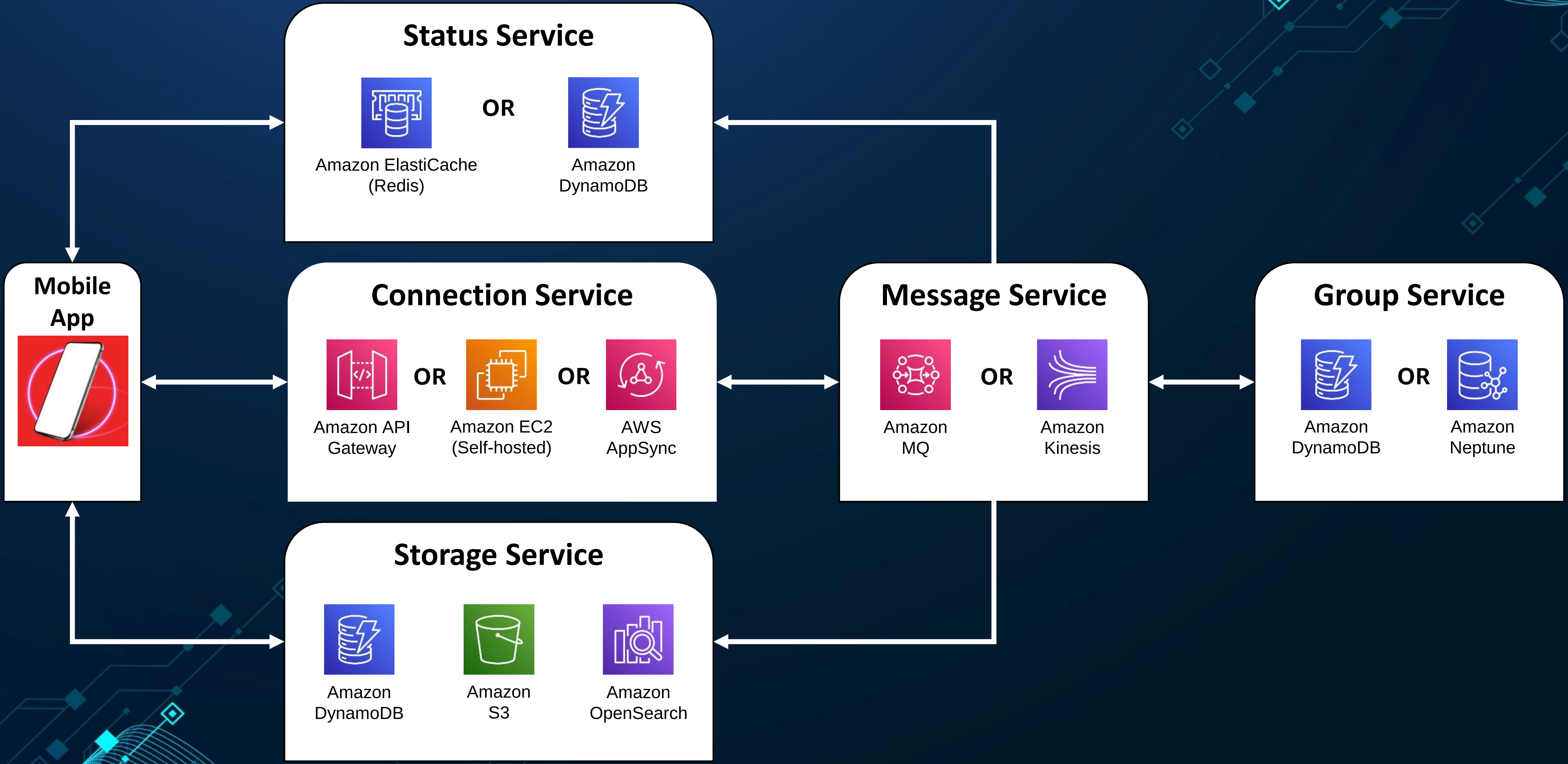
Fan Out Engine

Group Membership Database

Central Message Queue

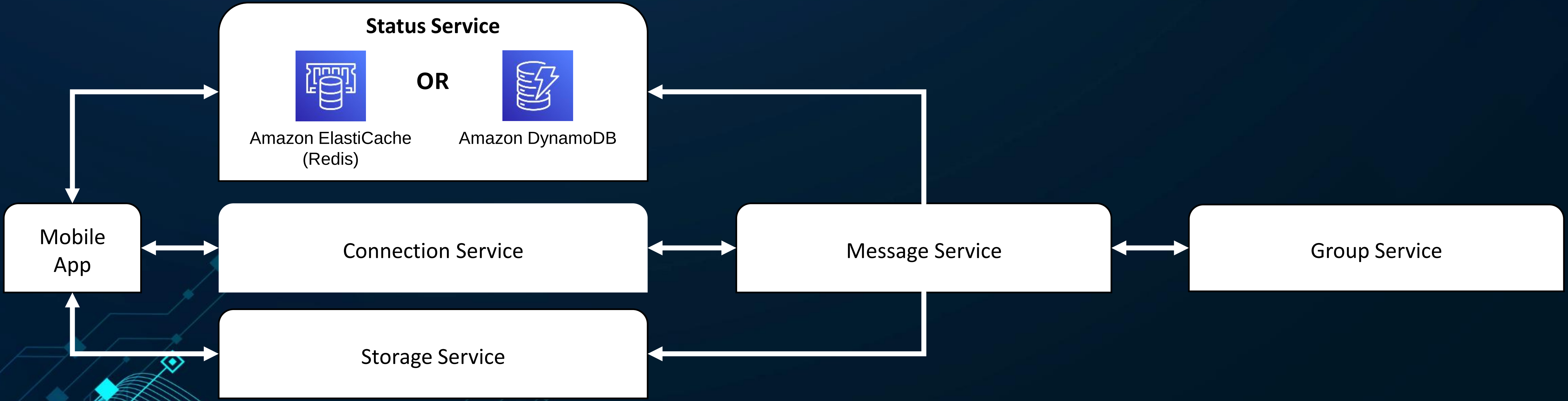
Sub-component	Group Service > Group Membership Database
Purpose	• Store user-to-group mapping • Roles and permissions
Consideration	• Many-to-many relationships • High fan-out read performance • Requires fast group membership resolution
Relational or Non-relational	**Non-relational** • Can handle denormalized lists of group members at scale • Optimized for fast fan-out during group messaging • Eliminates costly joins needed in relational models • Supports flexible role/permission metadata per member • E.g., Cassandra, HBase

Implementing on AWS



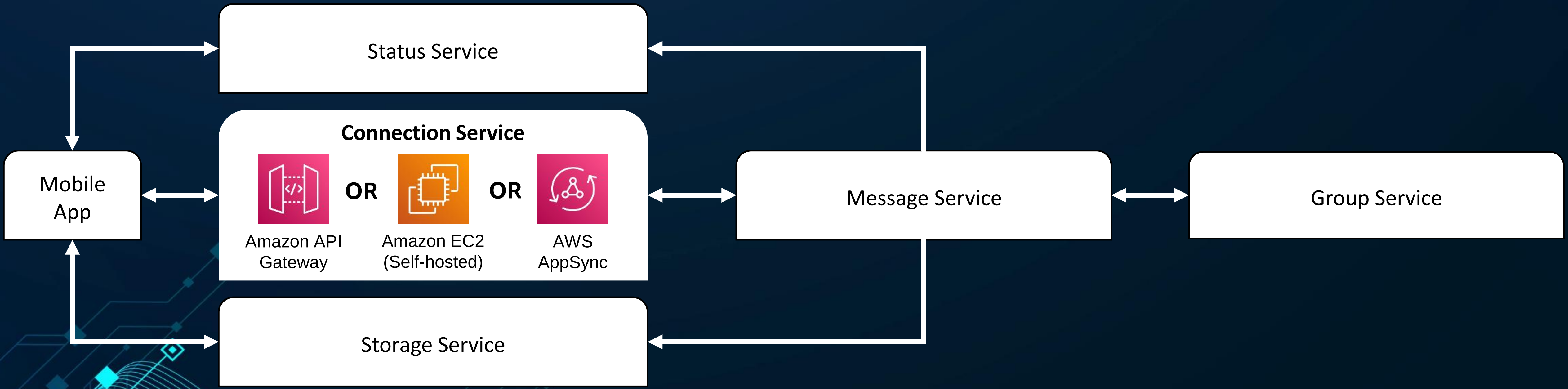
Pros and Cons – Status Service

Option	Pros	Cons
Amazon ElastiCache (Redis)	Low latency, Pub/Sub support, built-in TTL for presence.	Not persistent by default; costs increase with scale.
Amazon DynamoDB	Durable, scalable, supports TTL.	Not real-time; needs DynamoDB Streams + Lambda for triggers.



Pros and Cons – Connection Service

Option	Pros	Cons
Amazon API Gateway	Native WebSocket support, serverless, scalable.	Limited customization, message size limits.
Amazon EC2 (Self-hosted)	Full control over connections and protocols.	Requires scaling, monitoring, and maintenance.
AWS AppSync	Real-time communication, managed.	Might not fit well with highly-custom chat needs.



Status Service

Connection Service

Amazon API Gateway OR Amazon EC2 (Self-hosted) OR AWS AppSync

Mobile App

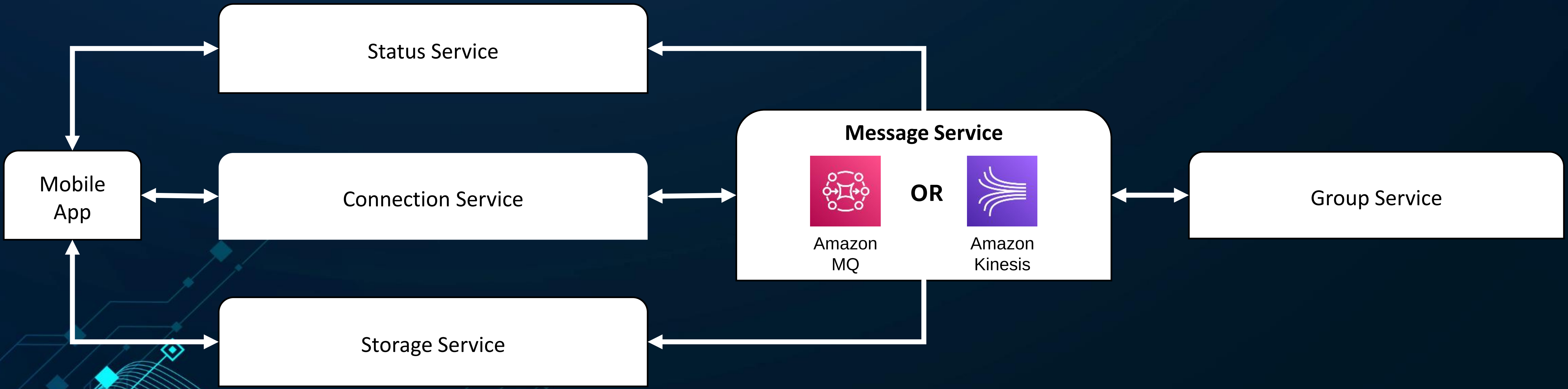
Message Service

Group Service

Storage Service

Pros and Cons – Message Service

Option	Pros	Cons
Amazon Kinesis	High-throughput, real-time stream processing.	Slightly complex to integrate with downstream consumers.
Amazon MQ	Full-featured message broker.	Heavier, more expensive, managed but not serverless.



Status Service

Mobile App

Connection Service

Storage Service

Message Service

Amazon MQ

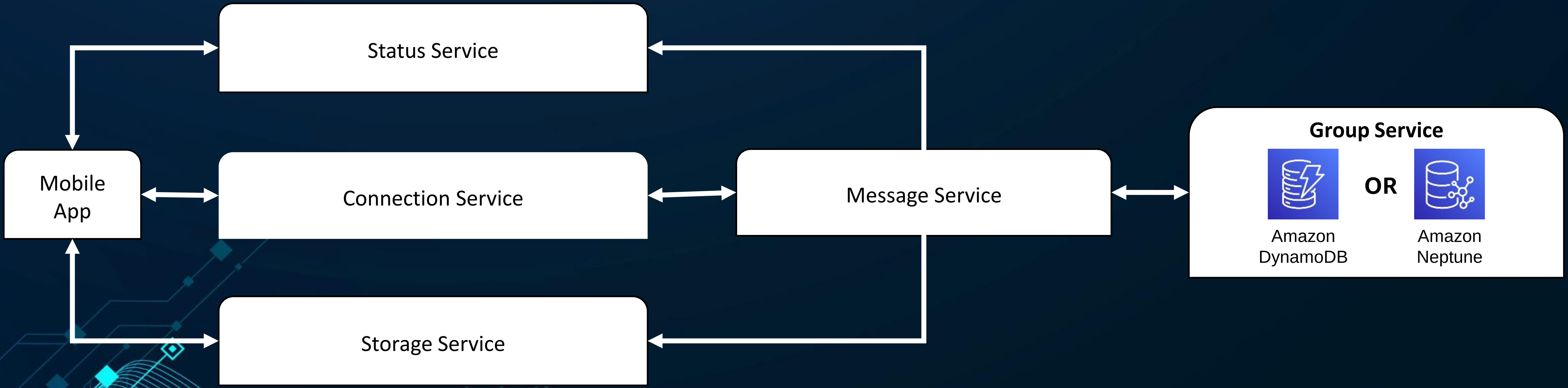
OR

Amazon Kinesis

Group Service

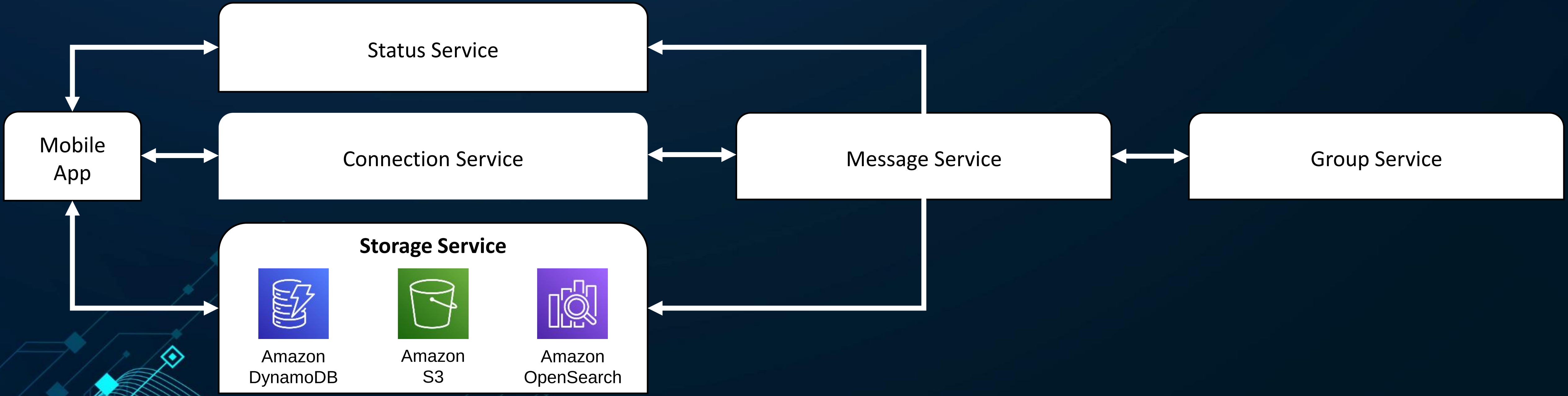
Pros and Cons – Group Service

Option	Pros	Cons
Amazon DynamoDB	Scalable, fast key-value access for group/user mapping.	Requires careful key design; no direct relational queries.
Amazon Neptune	Graph database for complex relationships.	Niche; requires learning curve and integration effort.

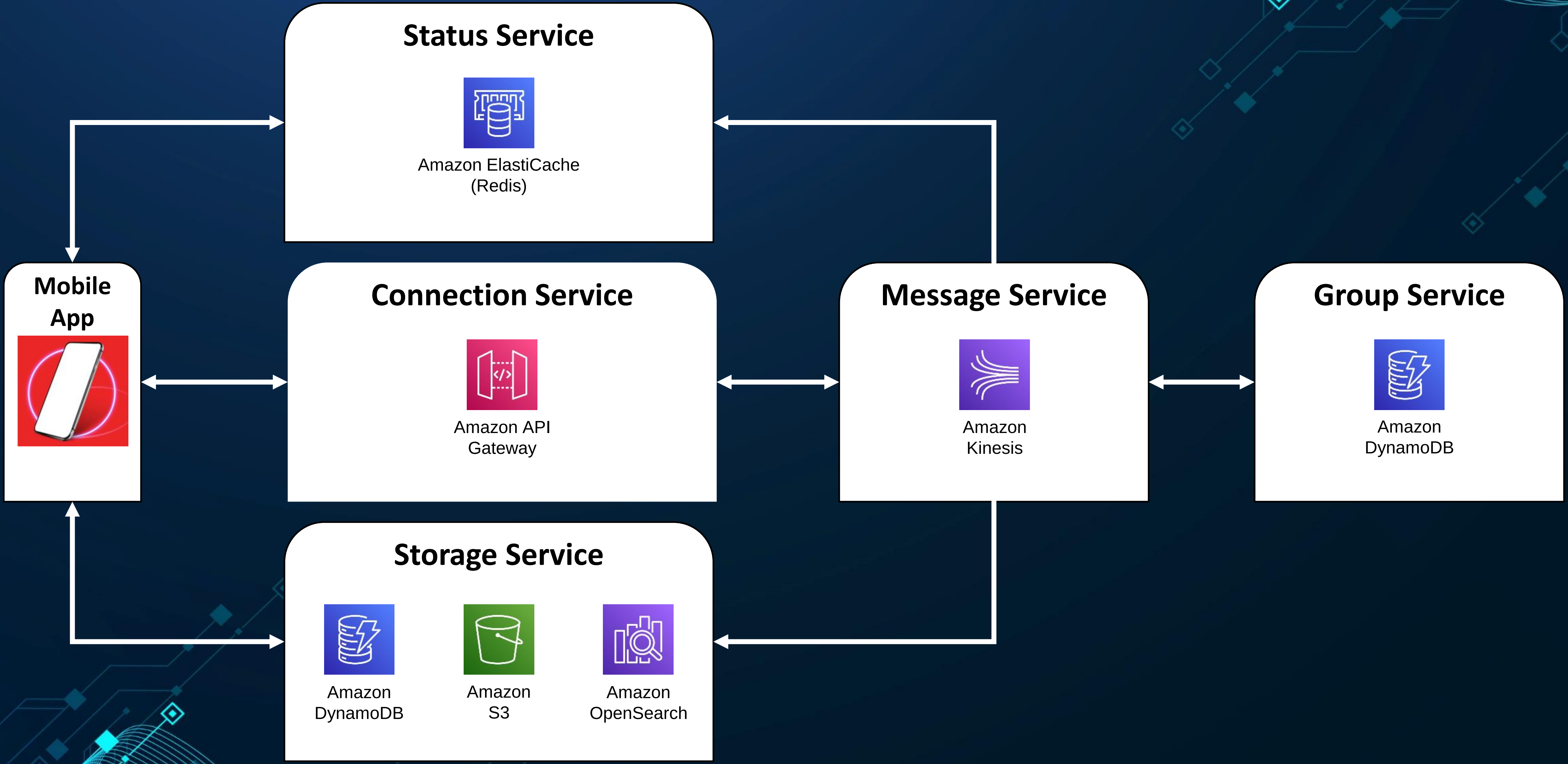


Pros and Cons – Storage Service

Subcomponent	Recommended Service(s)
Message Store	Amazon DynamoDB (with TTL and Streams)
Media Store	Amazon S3 (Intelligent Tiering, Signed URLs)
Cold Storage Manager	S3 Lifecycle Policies
Search Capabilities	Amazon OpenSearch

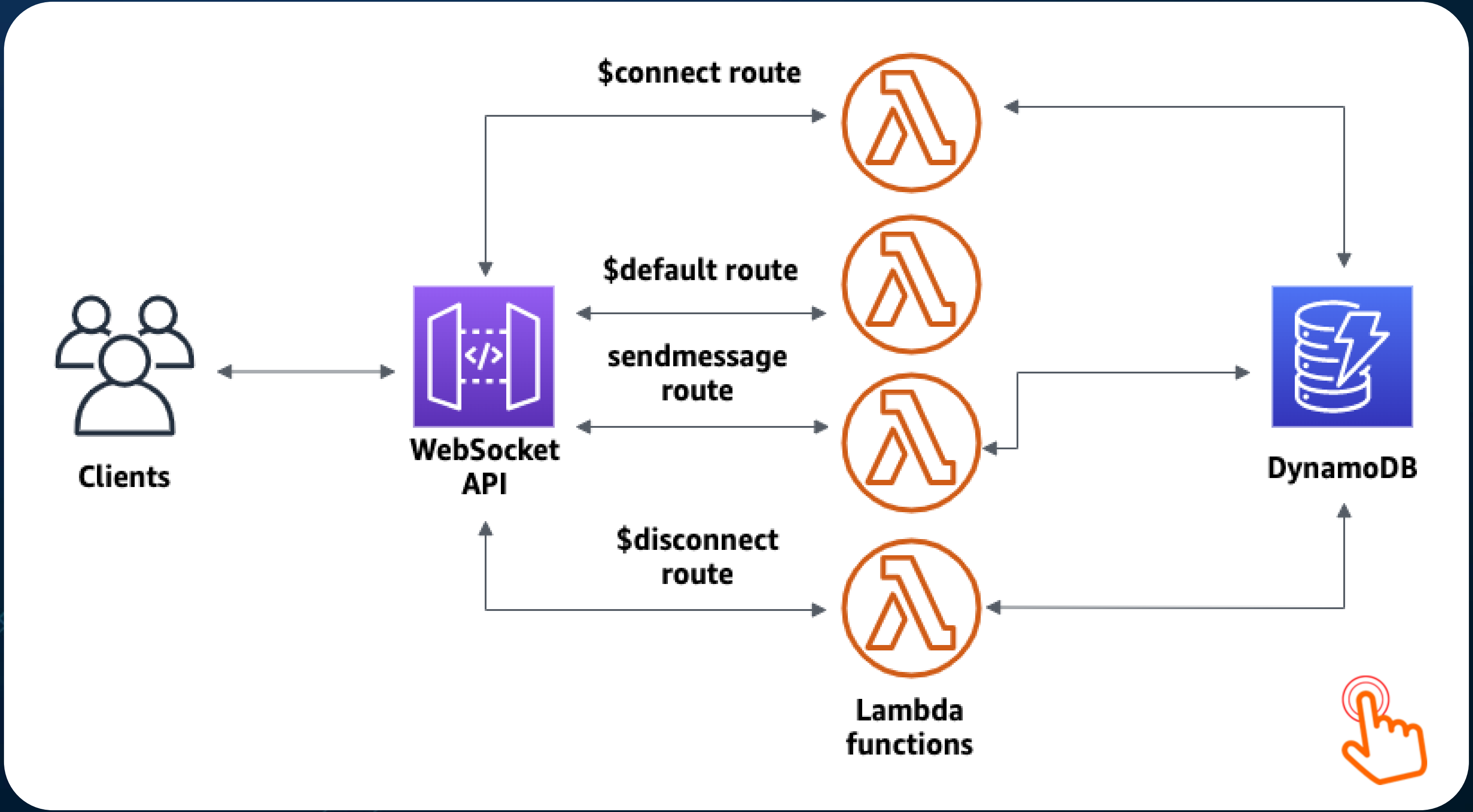


Implementing on AWS



Tutorial – Create a WebSocket chat app

- Create a WebSocket chat app with a WebSocket API, Lambda and DynamoDB



Non-functional Requirement

Performance

Send and receive messages in real-time

Scalability

Billions of users and billions of messages

Security

End-to-end encryption of messages

Reliability

Store messages until they are delivered

Availability

Minimal downtime or interruptions

Performance

- Goal: Send and receive messages in real-time
- Recommendations:
  - Use persistent connections like WebSockets for bi-directional, low-latency communication.
  - Minimize network hops and reduce the number of intermediary services in the message path.
  - Introduce in-memory caching layers (e.g., Redis-like systems) for user presence, session info, and message routing.
  - Design lightweight message formats (e.g., binary or compacted JSON) to reduce serialization/deserialization overhead.
  - Push vs Pull: Prefer push-based message delivery over polling to reduce latency and resource usage.
  - Load balance traffic across multiple message handlers and connection servers.

Scalability

- Goal: Billions of users and billions of messages
- Recommendations:
  - Use stateless microservices wherever possible so they can scale horizontally.
  - Introduce message queues or streams to decouple services (e.g., routing, storage, delivery).
  - Partition data by user, region, or chat to avoid bottlenecks and hotspots.
  - Use elastic storage and compute layers that can automatically expand with traffic.
  - Auto-scale connection and processing layers based on metrics like CPU, memory, queue depth, or user activity.

Security

- Goal: End-to-end encryption of messages
- Recommendations:
  - Implement end-to-end encryption (E2EE) so only sender and receiver can read messages—ensure encryption is handled at the client.
  - Use strong user authentication with token-based mechanisms (e.g., OAuth2 or JWT).
  - Encrypt all data in transit using HTTPS/TLS and encrypt at rest using managed keys.
  - Isolate sensitive components using private networks or access-controlled environments.
  - Apply least-privilege access controls for both services and users.
  - Log and monitor all access events for audit and anomaly detection.

Reliability

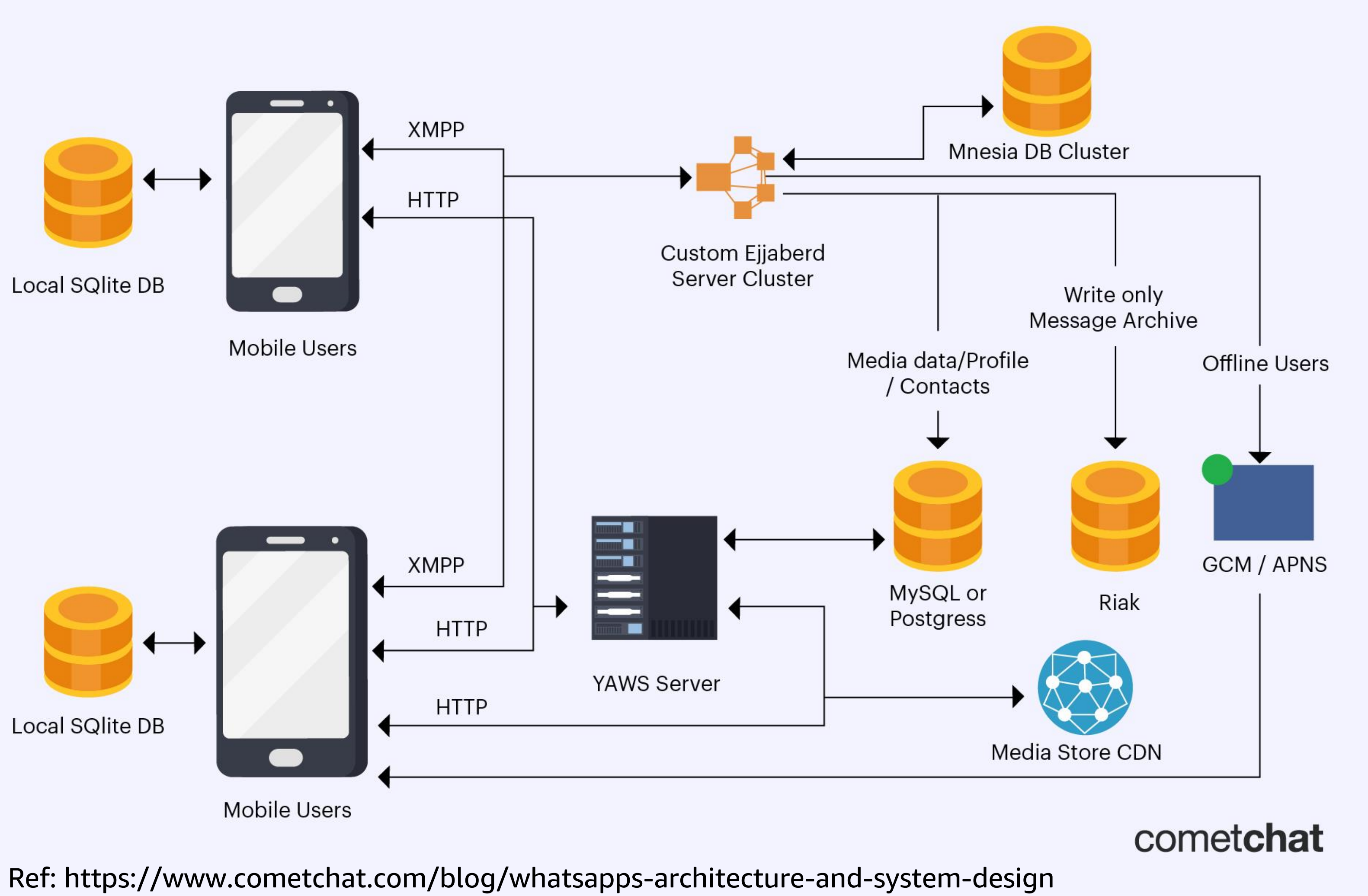
- Goal: Store messages until they are delivered
- Recommendations:
  - Persist messages before delivery to a durable store to avoid data loss during crashes.
  - Implement delivery acknowledgements and retries with exponential backoff.
  - Design for idempotency: duplicate messages or retries should not cause unintended side effects.
  - Use message queues with dead-letter support to handle failed deliveries gracefully.
  - Add health checks and service monitoring for automatic failure detection and failover.
  - Redundancy at all layers: Have multiple replicas of critical services and data stores.

Availability

- Goal: Minimal downtime or interruptions
- Recommendations:
  - Deploy services across multiple zones or regions to tolerate infrastructure failures.
  - Use load balancers and traffic routers to distribute traffic and handle node failures.
  - Gracefully degrade features: if group messaging fails, allow P2P to continue; if media upload is slow, queue for retry.
  - Implement self-healing infrastructure: automatically restart failed services or reroute traffic.
  - Regularly back up data and validate recovery processes for disaster recovery readiness.
  - Monitor availability with real-time alerts, dashboards, and SLO tracking.

WhatsApp Architecture – based on publicly shared information

- No officially published architecture



Ref: https://www.cometchat.com/blog/whatsapps-architecture-and-system-design

Programming Languages	• Erlang
Media-related components	• C++
Database	• Mnesia (Erlang's distributed DB) • Later scaled with MySQL (for long-term storage) • RocksDB (for high-speed access). • Riak (for media storage)
Messaging Protocol	• FunXMPP
Web Server	• YAWS (Yet Another Web Server)
End-to-end encryption	• Signal Protocol
Infrastructure	• Initially using FreeBSD, Now likely hosted within Meta's global data centers

Aspect	WhatsApp	Telegram	WeChat	Facebook Messenger
Focus Area	Privacy, simplicity, end-to-end communication	Speed, openness, cloud sync, developer-friendly APIs	Ecosystem of services, mini-apps, payments	Social interaction, integration with FB/Instagram/Threads
Unique Architecture Trait	Mobile-first, peer-to-peer encryption core	Cloud-first, centralized but secure transport layer	All-in-one “super app” model (chat, pay, services, games)	Deep integration with Facebook ecosystem
Message Protocol	Custom XMPP-based protocol + proprietary extensions	MTProto (Telegram's own protocol, optimized for speed & security)	Custom protocol (based on Tencent’s internal standards)	MQTT (lightweight pub/sub messaging protocol)
Data Storage	Mostly on-device, only metadata stored on servers	Messages stored in cloud (server-side history)	Server-side storage for chat, mini-programs, payments	Cloud-based storage, integrated with Facebook infra
Message Encryption	End-to-End by default (Signal Protocol)	Optional End-to-End (Secret Chats only)	Encrypted during transmission; not E2E by default	E2E only for Secret Conversations (not default)
Scalability Approach	Sharded Erlang clusters; client-heavy	Stateless backend, API-centric, CDN for media	Monolithic + Microservices for super app ecosystem	Microservices, large-scale sharding, FB global infra
Account Identity	Phone-number based	Phone optional; uses username model	Phone-number based + WeChat ID	Facebook account-based
Multi-Device Support	Recently added (client still primary source)	Built-in from start; true cloud sync	Yes; supports multiple devices natively	Fully supported

NEXT
STEPS

Incorporate following in design

- Authentication
- Encryption
- User registration
- User profiles
- Mobile app
- Audio/Video call
- Block lists
- Multiple device support