Telemedicine Platform – System Design

By:- Abhinendra Singh Chauhan

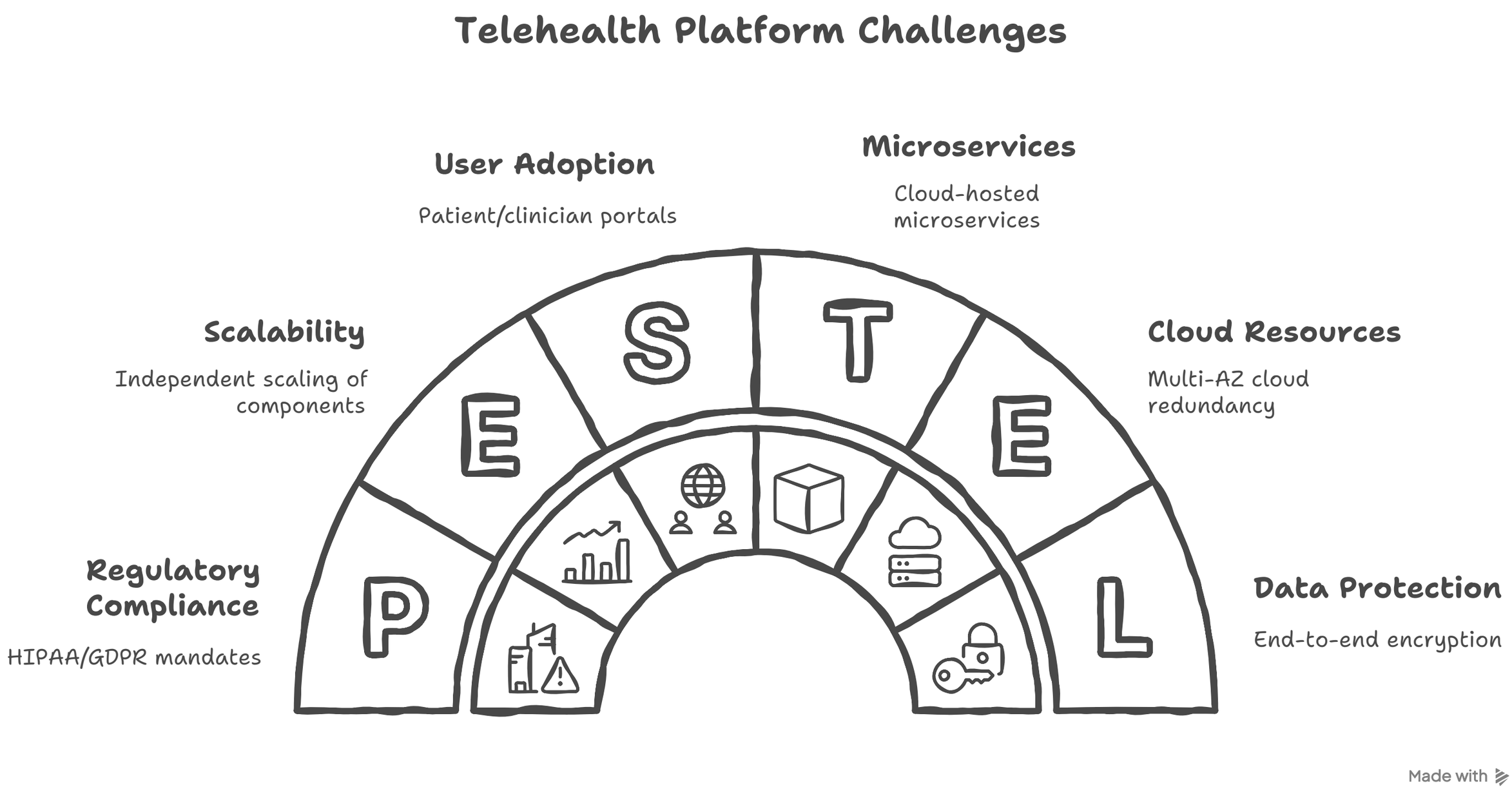
Project: Telemedicine Platform (Video Consults, Messaging, E-Prescriptions) This design connects patients, clinicians, and pharmacies via secure video visits, chat, triage, and prescriptions. It must support mobile and web clients, ensure HIPAA/GDPR compliance, and scale to thousands of users. Our high-level style is cloud-native microservices with layered frontends. Key goals include secure end-to-end encrypted communications, reliable video conferencing, instant messaging, eprescriptions, and full auditability. Targets might be ~99.95% availability, P95 video-join <2s, sub-2s chat latency under load.

1. Executive Summary

We propose a microservices-based architecture to meet the telehealth requirements. All services (scheduling, messaging, e-Rx, audit, etc.) run as independent services behind an API gateway with OAuth/ OIDC auth. A dedicated SFU media cluster (e.g. Janus or mediasoup) handles group video calls for scalability . Data is split across stores: a relational DB (Postgres) for PHI (encrypted at rest), a document store (e.g. MongoDB or Elastic) for notes, an appendonly event/audit store (Kafka or write-once DB), and S3-compatible object storage for images/recordings. We cache sessions and presence in Redis. A CDN serves static assets and attachments. 1

- Architecture Style: Cloud-hosted microservices (with micro-frontends for patient/ clinician portals) are chosen over a monolith or pure serverless. Microservices allow independent scaling of high-load components (like video and messaging) and independent deployments . (Monoliths are easier to start but become bottlenecks as scale grows .) · 2 2
- Non-functional Targets: e.g., 99.95% uptime (multi-AZ/cloud redundancy), TLS1.3/SRTP encryption, AES-256 at rest, audit logs immutable. All PHI is encrypted in transit (TLS 1.3) and at rest (AES-256). Compliance with HIPAA/GDPR mandates end-to-end encryption and audit trails. ·

Key Features: Patients can book and join video visits; clinicians see schedules and histories; asynchronous secure chat for follow-up; e-prescriptions delivered to patient's pharmacy; admins have audit dashboards.



Telemedicine Platform Architecture

2. Requirements Pack

2.1 Stakeholders

- Patients: End users booking/attending virtual visits, viewing prescriptions, sending messages. ·

- Clinicians (Doctors/Nurses): Handle appointments, conduct video consults, document notes, send e-prescriptions. ·
- Pharmacists / E-Rx Networks: Receive and fulfill electronic prescriptions; verify doctor signatures. ·
- Admin/Ops/Compliance: Manage user accounts, audit logs, system configuration, ensure regulatory compliance. ·
- Third-party Integrators: EHR systems, lab services, insurers – need APIs or HL7/FHIR interfaces. ·
- SRE/Monitoring: DevOps/SRE team monitoring system health, performance, and reliability. ·

2.2 High-Level User Stories

- As a patient, I want to schedule and join a video visit so I can consult remotely. ·
- As a clinician, I want to view my upcoming appointments and patient history so I can prepare. ·
- As a patient, I want to send secure messages to my clinician for follow-up questions. ·
- As a clinician, I want to issue an e-prescription during a visit that the patient can send to a pharmacy. ·
- As an admin, I want complete audit logs of all actions so I can satisfy compliance audits. ·

These stories drive features like calendar/scheduling, WebRTC video calls, chat UI, e-prescription digital signing, and logging.

2.3 Acceptance Criteria (Examples)

- Video call latency: A WebRTC video session should connect (participants join) within 5 seconds 95% of the time on a 4G network. ·
- Messaging latency: Secure chats deliver within 2 seconds under normal load. ·
- E-Prescription delivery: Once issued by clinician, the e-Rx (with digital signature) is received by the patient's pharmacy system reliably. ·
- Data protection: All PHI (personal data, notes, transcripts) must be encrypted at rest and in transit; tamper-proof signatures on prescriptions; full audit trail. ·

2.4 Context / Use-Case Diagrams

(See attached context_diagram.png and usecase_diagram.png .) Actors include Patient, Clinician, Pharmacy, Auth provider, Audit service, Media SFU. The context diagram shows patient/clinician clients communicating via API gateway to core services and media SFU, while pharmacies receive e-Rx from prescriptions service.

2.5 Constraints & Assumptions

- Compliance: Must meet HIPAA (US) and relevant local health data laws (e.g. GDPR in EU). All data and communication must be secure and logged. ·
- Clients: Both web and mobile (iOS/Android) clients required. Likely a web portal for clinicians/admin, and cross-platform or native app for patients. ·
- Scale: Initial load might be modest (e.g. 10,000 daily active users, hundreds of daily appointments) but should be designed for growth (multi-tenant). Peak concurrent calls might reach hundreds or thousands. ·
- Third-Party Services: 3rd-party SMS/email for OTP or notifications; e-prescription networks; insurance/EHR APIs. ·
- Storage: Use cloud object storage (S3 or S3-compatible) for media (attachments, call recordings) with assumed high durability. ·
- Teams: Small-to-medium dev team initially, growing over time.

3. Architecture Overview

3.1 Chosen Architecture

We adopt a microservices architecture with micro-frontends. This allows independent development and scaling of features (video, chat, scheduling, etc.) . A single-page web/mobile frontend (built in React or similar) handles UI for each role. An API Gateway (e.g. AWS API Gateway, Kong, or NGINX) provides a unified endpoint, handles TLS termination, OIDCbased auth (OAuth2/JWT), rate limiting, and routing to services. 2
Behind the gateway are core microservices: Auth, User/Profile, Scheduling, Messaging, Media Signaling, Prescriptions, Audit/Logging, and Notifications. Each service exposes a example, the Scheduling service manages appointments; the Messaging service handles chat (persisting messages encrypted); the Prescriptions service creates and stores signed eRx docs; etc.
orchestration (Kubernetes or ECS). Service discovery and dynamic scaling ensure high availability. A CDN (e.g. CloudFront) caches static front-end assets and any large media (turn servers), and IAM for resource access. Monitoring (Prometheus/ Grafana), centralized logging (ELK/EFK), and distributed tracing (OpenTelemetry) are integrated for observability.
REST/JSON or gRPC API. Services communicate via lightweight JSON over HTTP or gRPC. For Media Service (SFU Cluster): Video calls use WebRTC. We deploy a fleet of SFU (Selective Forwarding Unit) servers (e.g. mediasoup, Janus, or Jitsi) to handle multi-party calls. SFUs forward encoded streams rather than mixing them (MCU), which greatly improves scalability and reduces client CPU load . In practice, SFUs subscribe to each participant's stream and relay it selectively (possibly adapting resolution based on client bandwidth) . We might front these with a media gateway that load-balances new session requests to SFU nodes. 1 1 Data Layer: A PostgreSQL cluster stores core relational data (users, patients, providers, appointments) – these contain PHI and are fully encrypted at rest. A document store (e.g. MongoDB or ElasticSearch) holds unstructured data like clinical notes or transcripts for fast text search. An append-only event store (e.g. Kafka or a write-once DB) logs all system events for auditing. A Redis in-memory cache handles session tokens, presence (who is online), and rate-limiting counters. S3-compatible object storage holds attachments (lab results, images) and optionally encrypted recordings of sessions. Infrastructure: We deploy in a cloud environment (AWS/Azure/GCP) using container downloadable content. Standard components include L4/L7 load balancers, NAT/firewall for

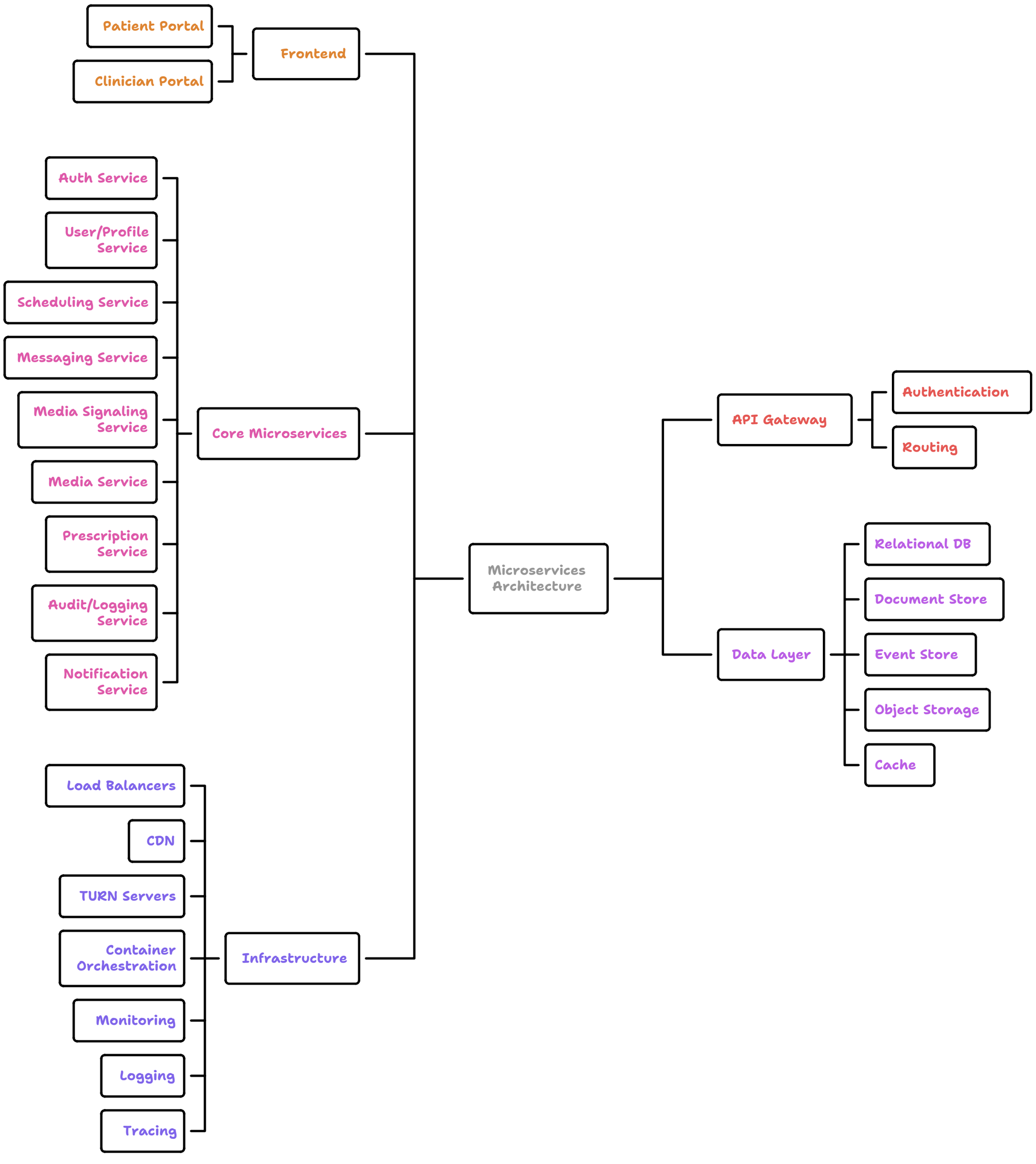
3.2 Components

- Frontend (Patient/Clinician Portal): Built as micro-frontends or separate single-page apps (e.g. React) for patient and clinician roles. They interact with backend via REST and WebSocket for live updates. The clinician UI includes calendar, patient record, video widget, and chat. The patient UI includes appointment booking, chat, and viewing prescriptions. ·
- API Gateway: Handles authentication (JWT/OIDC tokens, integration with Auth0/ Keycloak) and routes requests to microservices. It also enforces HTTPS, rate limits, and logs requests (without PHI). ·
- Auth Service: OIDC provider issuing access/refresh tokens. Supports multi-factor (SMS/ OTP or TOTP) for clinician logins. Integrates with external SSO if needed. ·
- User/Profile Service: Manages user identities (patients, providers), storing minimal demographic PHI (encrypted in DB) and link to patient/provider records. ·
- Scheduling Service: Manages appointments. Books time slots (with conflict checks), generates meeting links (room IDs), and triggers notifications. ·
- Messaging Service: Stores conversations and messages (each message body encrypted). Exposes APIs for sending/receiving chat (could use WebSockets for realtime). Maintains conversation indices for searching history. ·
- Media Signaling Service: Coordinates WebRTC signaling (ICE candidates, SDP exchange) but does not handle media streams. Clients connect to SFU for media after obtaining a room token from this service. ·
- Media Service (SFU cluster): As above, handles the actual media streams. It just routes SRTP packets between clients, performing any forwarding/adaptation. SFUs reduce per-client load and scale well . · 1

- Prescription Service: Issues e-prescriptions. Generates a signed document (using a certificate/ private key to ensure non-repudiation) and sends it to the patient's pharmacy. Persists a copy in DB. ·
- Audit/Logging Service: An immutable event store that records every action (user logins, data reads/ writes on PHI, configuration changes). This ensures full audit trails for compliance. ·
- Notification Service: Handles async notifications (email/SMS) for reminders (appointments, new messages, eRx ready). Can be serverless for cost-effectiveness on infrequent events. ·
- Data Stores: ·
- Relational DB (Postgres): Stores users, patients, providers, appointments, prescriptions metadata (with encrypted sensitive fields) . · 3
- Document Store (MongoDB/Elasticsearch): Holds clinician notes, transcripts, chat conversation metadata for search. ·
- Event Store (Kafka or write-only DB): Append-only log of all system events and audit entries. ·
- Object Storage (S3): Stores binary attachments (lab images, referral letters) and optionally recorded session media files (if recording is supported, must be encrypted). ·
- Cache (Redis): Caches session tokens, rate limits, and quick lookups (e.g. online presence, temporary OTPS). ·

Other Infrastructure: Managed load balancers (L4 for TCP connections, L7 for HTTP), CDN for static assets, TURN servers for WebRTC fallback if direct peer connection fails through NAT, container orchestration (K8s) for services, Prometheus/Grafana for metrics, ELK for logs, and OpenTelemetry for tracing.

Microservices Architecture for Healthcare Platform



4. Architecture Trade-off Analysis

- Monolithic: Pros: Simple initial development; single deployment; easier local testing. Cons: Hard to scale individual parts (video and chat load), updates require full redeploy, risk of a "big ball of mud" as feature set grows. When to pick: Small MVP with limited scope and one team . Example: a quick pilot app might start monolithic, but will likely outgrow this model rapidly. ·

- Microservices: Pros: Fine-grained scaling of components (e.g., scale up SFU cluster or messaging independently), independent deployments, technology heterogeneity (use best tool per service), fault isolation (one service failing doesn't take down all). Cons: Higher operational complexity (networking, deployments, distributed data), need for robust devOps. When to pick: Medium/large scale, many users/teams. For telemedicine, this fits because different features (video, messaging, ·

records) have distinct load patterns . (RadiologyKey notes that "monolith design... is a bottleneck when your app is larger. Instead, work with microservices...each performing a single activity such as scheduling appointments, video consults, billing..." .)

- Serverless: Pros: Minimal ops, pay-per-use cost model, easy horizontal scaling for short tasks. Cons: Cold starts, not suited for long-lived tasks (cannot run real-time media sessions directly), vendor lock-in. When to pick: Event-driven parts like notifications, batch jobs (e.g. nightly report generation), or when usage is very spiky/low volume. For the core telehealth (real-time media, interactive UI), serverless alone would be inadequate. ·

Recommendation: Use microservices for core platform services plus a scalable SFU media cluster for video. Use serverless functions selectively for auxiliary tasks (e.g., sending email/SMS, file processing) to reduce costs on low-frequency jobs. This aligns with advice to favor microservices for scalable telehealth systems .

5. Data Design

5.1 Data Stores & Purpose

- Relational DB (Postgres): Stores structured PHI data – user accounts, patient profiles, provider credentials, appointments, prescription records. Sensitive fields (patient demographics, medical history pointers) are encrypted (AES-256) at rest. ·
- Document Store (MongoDB/Elastic): Stores semi-structured data such as clinical notes, conversation transcripts, or chat metadata. Useful for full-text search and flexible schemas · 3
- Audit/Event Store (Kafka or Write-only DB): Captures all events and changes in an append-only fashion. Once logged, entries cannot be altered, enabling compliance auditing and traceability. This can also back event sourcing/CQRS patterns (see below). ·
- Object Storage (S3): Holds uploaded files (scanned documents, test results) and encrypted video recordings of sessions. Durable and scalable storage for large blobs. ·
- Cache (Redis): In-memory store for ephemeral data like session tokens, Redis can also hold chat presence, rate-limit counters, and serve as a pub/sub bus for live updates (e.g. new message notifications). ·

5.2 ERD / Schema (Summary)

Major tables (with example columns):

- -users: (user_id PK, role, name, email, password_hash, metadata_encrypted...] – basic identity (patients and staff).
- -patients: (patient_id PK, user_id FK, demographics_encrypted, insurance_info, emergency_contact...].
- -providers: (provider_id PK, user_id FK, license_number, specialty, credentials_hash...].
- -appointments: (appt_id PK, patient_id FK, provider_id FK, start_ts, end_ts, status, video_room_id) – schedules.
- -messages: (message_id PK, conversation_id FK, sender_id FK, body_encrypted, timestamp).
- -conversations: (conversation_id PK, participants, last_msg_ts).
- -prescriptions: (rx_id PK, appointment_id FK, provider_id, patient_id, content_signed, issued_at).
- -audit_logs: (log_id PK, actor_id, action, resource_type, resource_id, timestamp, details_hash) – immutable audit trail (hash chaining optional).

(See attached ERD.png for a diagram.)

We will encrypt sensitive columns (demographics, message body, prescription content) at the application level. Audit logs store hashes of details to detect tampering.

5.3 Normalization / Denormalization Rationale

We normalize core tables (users, patients, appointments) to enforce consistency and reduce duplication (e.g., patients linked to user accounts). However, for performance-critical read views (such as clinician dashboards), we may denormalize or use materialized views/caches. For example, a clinician's schedule view could use a precomputed join of providers, patients, and appointment details to avoid heavy joins on each query. NoSQL stores inherently hold denormalized data (e.g., a patient note document may embed the author's ID rather than join to a user table).

5.4 Indexing & Partitioning

- Indexes: Primary keys on all tables. Indexes on foreign keys (e.g. patient_id on appointments, conversation_id on messages). Composite indexes for common queries (e.g. (provider_id, start_ts) on appointments to quickly find a provider's upcoming slots). Full-text or inverted indexes on note text or message bodies for search. ·
- Partitioning: We can partition large tables by date or tenant: e.g., monthly partitions for messages and audit_logs to archive older data efficiently. For multitenant support, partial or sub-partitioning per organization. ·
- Sharding: If a single Postgres cluster can't handle peak writes, consider horizontal sharding by patient or provider ID (hash-based) for distributed load. NoSQL collections (e.g. Mongo) can shard collections by shard key (like patient_id) as well. ·

5.5 Backups & Retention

- Backups: Daily snapshots of databases and continuous write-ahead log (WAL) shipping for point-intime restore. Use managed DB (e.g. AWS RDS) for built-in backups and replication. ·
- Retention: Follow regulatory retention: for example, HIPAA requires retaining audit logs and related documentation for 6 years . (Medical records may have state-specific periods, often 5–10 years.) · 4
- Archival: Move aged data (e.g. >2 years) to cold storage or cheaper object-storebacked databases. ·

E.g., older appointment history could go to an archival schema. Ensure encryption keys for archived data remain available as long as retention period.

6. Design Patterns & Anti-Patterns

6.1 Selected Patterns

- Proxy/Gateway: API Gateway and media proxy (TURN servers) for NAT traversal and load balancing. ·
- Builder/Factory: For constructing complex e-prescription documents (adding doctor signature, formatting) before sending. ·
- CQRS: Separate write-model (handling commands like "create appointment", "send message") from read-model (querying schedules, chat history). This allows optimizing reads (dashboards) and audit logging of commands. ·
- Event Sourcing: (Partial) Write-events (e.g. "AppointmentBooked") to the audit log. These events can rebuild state or serve as an immutable change history. ·
- Circuit Breaker & Retry: For all external calls (SMS API, pharmacy API). Use retries with exponential backoff and circuit breakers to isolate failures. Include idempotency tokens for safe retries (e.g. while sending a prescription). ·
- Observer / Pub-Sub: Internal pub/sub (e.g. via Kafka or Redis streams) to broadcast events like "new message" or "appointment updated" to interested services (notifications, cache invalidation). ·
- Repository: Each microservice uses a repository layer to abstract DB access (e.g. using an ORM or data mapper). This facilitates testing and potential DB swaps. ·

- Strangler Pattern (for modularization): If migrating an old system, new features (video chat, etc.) are built as microservices that slowly replace old modules. ·

6.2 Anti-Patterns to Avoid

- Chatty Services: Avoid overly chatty synchronous calls between microservices. Whenever possible, use asynchronous messaging/event-driven integration to reduce tight coupling and improve resilience. ·
- God Object: Don't create a single "UberService" that contains all business logic. Keep services focused (e.g. Scheduling service does only scheduling). ·
- All-in-one Media Control: Don't force all media handling through one monolithic control plane. Keep media servers (SFUs) separate from core API logic; the API only handles signaling and room management. ·
- Bypassing API Gateway: All client and inter-service calls should go through defined channels. Do not allow ad-hoc database or service access (enforces security and audit). ·

7. Security & Compliance

7.1 Threat Model (Summary)

- External Attacks: MITM (counter with TLS), brute-force/credential stuffing (rate-limit logins, MFA), XSS/CSRF (secure cookies, tokens), injection (use parameterized queries). Ensure tokens are signed. ·
- Insider Threat: Limit access via RBAC, least privilege, audit all accesses. Encrypt sensitive data so even DB or log admins can't read raw PHI. ·
- Availability Attacks: DDoS protections (WAF, rate limiting), design for graceful degradation (e.g. if video SFUs are overloaded, fallback to audio-only or smaller resolutions). ·
- Data Leakage: Prevent PHI leaks via overly broad APIs or logs. Scan attachments for PII before sending (Data Loss Prevention). ·

7.2 AuthN / AuthZ

- Authentication: Use a standards-based OIDC provider (Auth0/Keycloak/Cognito) for user sign-on. Require MFA (SMS or TOTP) for clinician accounts. Secure password storage (bcrypt/Argon2) and account lockout on repeated failures. ·
- Authorization: Implement Role-Based Access Control (RBAC): roles (patient, clinician, admin) with scopes/permissions. For example, patients can only access their own data; clinicians can only access patients they are treating. Also include attribute-based rules (e.g. a nurse role vs specialist role). ·
- Token Model: Issue short-lived JWT access tokens (e.g. 15 min) and long-lived refresh tokens. Support token revocation (e.g. if user is disabled or reported compromised). ·

7.3 Encryption & Data Loss Prevention

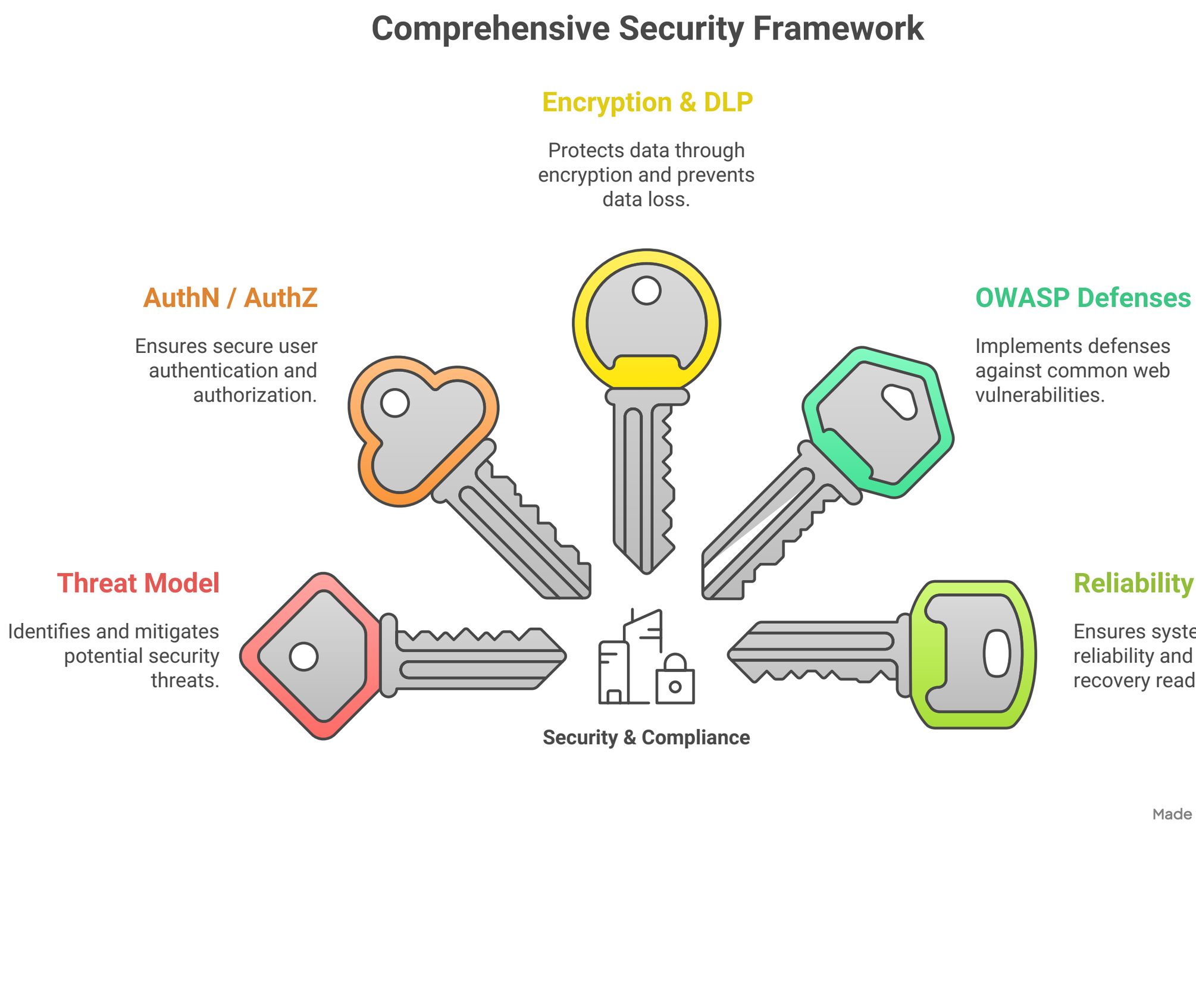
- Transport Encryption: All communication (API calls, WebRTC signaling, web UI) uses TLS 1.3. Per HIPAA best practices, use at least TLS 1.2/1.3 and AES-256 ciphers. WebRTC media is encrypted with SRTP (Datagram TLS) end-to-end. ·
- At-Rest Encryption: AES-256 encryption for databases (Postgres, Mongo) and object storage. Keys managed in a secure KMS (Key Management Service) with rotation. Field-level encryption for highly sensitive data (e.g. patient SSN) is recommended. ·
- Prescription Signatures: Use asymmetric signing (e.g. RSA/ECC) for e-prescription documents so pharmacies can verify origin and integrity. Keep the private key in a Hardware Security Module (HSM) or cloud KMS. ·
- DLP (Data Loss Prevention): Inspect file uploads and messages for malicious content or unintended PHI leaks (e.g. patient inadvertently sharing someone else's data). Use antivirus and phishing detection on attachments. ·
- Backup Encryption: Backups (WAL archives, snapshots) must also be encrypted. ·

7.4 OWASP Top-10 Defenses (Example)

- Injection (SQLi/XSS): Use ORM or parameterized queries. Sanitize any rich text (WYSIWYG) in notes/ messages. ·
- Broken Auth: MFA, secure password policies, no sensitive info in tokens. Revoke refresh tokens on sign-out. ·
- Sensitive Data Exposure: Strictly avoid exposing PHI in error messages, URLs, or logs. Use HTTPS and HSTS. ·
- Access Control: Enforce per-API and per-field access checks. For example, only allow a clinician to prescribe if they have the "prescribe" role. ·
- CSRF: Use anti-CSRF tokens for state-changing requests from web clients. ·
- Security Headers: Content Security Policy (CSP), X-Frame-Options, secure cookies (HttpOnly, SameSite). ·
- Rate Limiting: Prevent credential stuffing and brute-force (especially on login/OTP endpoints) by limiting attempts and returning 429. ·
- Logging & Monitoring: Centralize logs, detect anomalies (e.g. unusual access patterns). ·

7.5 Reliability & Disaster Recovery

- High Availability: Deploy services in multi-AZ/region clusters. For example, Postgres in a primaryreplica setup across zones. SFU instances auto-scale and have multiple workers per AZ. ·
- Failure Handling: Use circuit breakers (e.g., Netflix Hystrix pattern) when calling thirdparty APIs (pharmacy, SMS). Automatic retries with exponential backoff. ·
- Data Replication: Synchronous replication for critical data (appointments) and async for read replicas (for queries). Kafka topics replicated across brokers for durability. ·
- DR: Maintain a hot/warm standby in another region or cloud; regularly test failover of DB and services. Define RTO/RPO (e.g. RTO < 1 hour, RPO ~ 15min for DB). ·
- Penetration Testing: Schedule regular security audits and pen tests. Respond to vulnerability disclosures promptly (CVE tracking for all components). ·



8. Performance & Scalability

8.1 Caching

- CDN: Use a global CDN (CloudFront/Akamai) for all static assets (JS/CSS/images) and for serving large attachments or recordings. This reduces load on origin servers and speeds up content delivery. ·
- Application Cache: Redis caches sessions, user profiles, and recent data to reduce DB hits. For example, cache a clinician's patient list or auth token lookup. ·
- Data Caching: Use in-memory or Redis-backed materialized views for heavy dashboards (e.g. "today's schedule") to minimize complex joins. ·

8.2 Load Balancing & Scaling

- API Layer: L4/L7 load balancers distribute incoming HTTP(s) API calls across multiple instances of each microservice (auto-scaled by CPU/RPS). If using Kubernetes, an ingress controller or service mesh can do this. ·
- Media Layer: SFU servers sit behind their own load balancer or use DNS-based load balancing. Use sticky routing by room ID: once a participant joins a room on one SFU node, all their subsequent streams route to the same node. Auto-scale SFU instances based on CPU/network load. ·
- Connection Pooling: Services use pooled DB connections and tuned connection limits. Use read replicas for scaling query throughput (reports, analytics) . · 3
- Back Pressure: Leverage message queues (Kafka/RabbitMQ) for non-blocking operations. If the notification service is busy, it can queue emails instead of dropping them. Downstream: if a pharmacy API is slow, use a retry queue. ·

8.3 Replication & Partitioning

- Databases: Use a primary-replica Postgres setup. Critical writes (user signup, updates to appointments) go to the primary; heavy reads (reporting) hit replicas. If very high scale, shard by customer or region. ·
- Messaging/Events: Kafka for event streaming; it provides durability and decouples producers/ consumers for load leveling. We can partition topics by patient ID so consumers (e.g. analytics) can scale. ·
- Media Routing: For multi-region support, you might deploy SFUs in each region and use client IP/ geolocation routing to connect users to nearest media server for low latency. ·

8.4 Backpressure & Flow Control

- Rate Limiting: Per-user and per-API rate limits. e.g., limit message sends per second, login attempts, etc., to prevent floods. Return HTTP 429 when exceeded. ·
- Graceful Degradation: If certain services are under extreme load (e.g. video quality drops), degrade non-critical features (e.g. pause background transcription) and inform the user. ·
- Queue Sizing: Configure message broker and worker queues with limits. Monitor queue lengths and apply backpressure (slow producers) if queues backlog. ·

8.5 Performance Targets & Benchmarks

- API Latency: 95th percentile API response time <200ms under expected load. ·
- Messaging: 95% of chat messages delivered within 2s. ·
- Video Join: 95% of video sessions established within 5s (as in user story). ·
- Scale Tests: Load testing (e.g. with k6 or JMeter) must simulate peak usage (hundreds of concurrent calls, thousands of messages/minute) to validate scaling. (See attached load_test_report.pdf for example scenarios.) ·

9. API Specification (Sample)

We will expose RESTful JSON APIs for client interactions and (optionally) gRPC for internal high-throughput service calls. Version all endpoints (e.g. /api/v1/...) for future evolution. A brief table of sample endpoints:

patients/ {id}/ appointments	GET	List a patient's appts	{path & query params}	200 [{appointment}]	403, 404	Yes
/api/v1/ appointments	POST	Create a new appointment	{patient, provider, slot}	201 {appointment}	400,409	Yes {client token}
/api/v1/ video/ sessions	POST	Create video session {room token}	{appointment_id}	201 {room_id, token}	403 {no auth}, 410 {appt closed}	Yes key}
/api/v1/ messages	POST	Send a chat message	{conversation_id, body}	201 {message}	400 {body too large}, 413	Use to avoid duplicate

10. Observability & SRE

10.1 Logging

All services produce structured logs (JSON) with a common schema. Logs include correlation IDs (e.g. request_id) to trace a user request end-to-end. We never log raw PHI (log only metadata, user IDs, action names). Logs are sent to a central ELK/EFK stack for analysis. Admin actions and errors are logged at INFO/ ERROR levels accordingly.

10.2 Metrics

Key metrics collected (via Prometheus exporters): request rates, HTTP status code distribution, latency percentiles (p50/95/99) for each API. Business metrics: number of active calls, queued messages, appointments/day. Infrastructure metrics: CPU, memory, network IO, SFU channel count. Dashboards in Grafana provide live views of health (API latency, error spikes, concurrency). Alerts configured on error rate thresholds, saturation (e.g. SFU CPU >80%), or queue buildups.

10.3 Tracing

Distributed tracing is enabled with OpenTelemetry. Each API request carries a trace ID that flows across services and even into the SFU signaling. This allows latency breakdown (e.g. time spent in scheduling DB query vs network). Traces help diagnose slow requests or identify problematic services. Traces are visualized in a tracing system (Jaeger or Tempo).

10.4 SLOs / SLIs / Alerts

- SLIs: e.g. API availability (HTTP 2xx rate), p95 latency for API calls, message delivery time, video session success rate. ·
- SLOs: e.g. 99.95% API uptime monthly, 95% messages delivered <2s, 99.9% successful call connections. ·
- Alerts: Trigger if SLO violations look imminent. Example: >1% 5xx on core APIs in 5min, or video call failure rate >2%. Alerts go to ops via email/slack/pager. Each alert has an associated runbook. ·

10.5 Runbook Example (Video Degradation)

- Alert: Media worker error rate >5% for 5min. ·
- Steps: Check SFU instance health (logs for ICE failures). Verify TURN relay status. Scale up SFU pool or redistribute rooms if some nodes are unhealthy. If recent deploy caused issue, roll back. Confirm new sessions work, then help affected users reconnect. Notify stakeholders if degraded service persists. ·

11. Security/Reliability Runbook (Summary)

- Incident Triage: Identify incident scope (e.g. compromised key, DDoS, service outage). Contain (block IP, disable user) and eradicate (revoke tokens, restart service) as needed. ·
- Escalation: Predefined contacts (SRE, legal, compliance) notified for high-severity incidents. ·
- Breach Response: If PHI breach suspected, notify legal/compliance. Follow HIPAA breach notification rules (notification within 60 days, etc.). Preserve forensic logs. ·
- Recovery: Failover to backup systems, restore from last good backups if needed. After resolution, conduct post-mortem.

12. Tech Stack Options & Justification

(Example comparison table in attached report.) We evaluate key tiers:

Frontend: ·

- React/Redux (Recommended): Mature ecosystem, many UI libraries, excellent support for WebRTC (e.g. use react-webrtc), strong community. ·
- Angular: Enterprise-grade, but heavier; overkill for small teams. ·
- Vue/Svelte: Lighter, but smaller ecosystem around WebRTC. ·

Backend: ·

- Node.js/TypeScript: Fast development, many libraries (WebRTC signaling, JSON). Good for I/O heavy tasks (chat, signaling). ·
- Go: High concurrency, efficient CPU (good for SFU control plane or media servers). Simpler deployment. ·
- Python/Django: Rapid prototyping for scheduling/logic, but less ideal for sustained high concurrency. ·
- Java/Spring Boot: Strong typing, mature (especially for healthcare clients), but heavier.

.

Media (SFU): ·

- mediasoup (Node C++): Highly customizable SFU. Recommended if we need custom features. ·
- Janus (C): Proven, low-latency. Good performance out of box. ·
- Jitsi (Scala/Java): Full stack (includes SFU and signaling) – faster to deploy but less flexible. Could be used for MVP. ·

Databases: ·

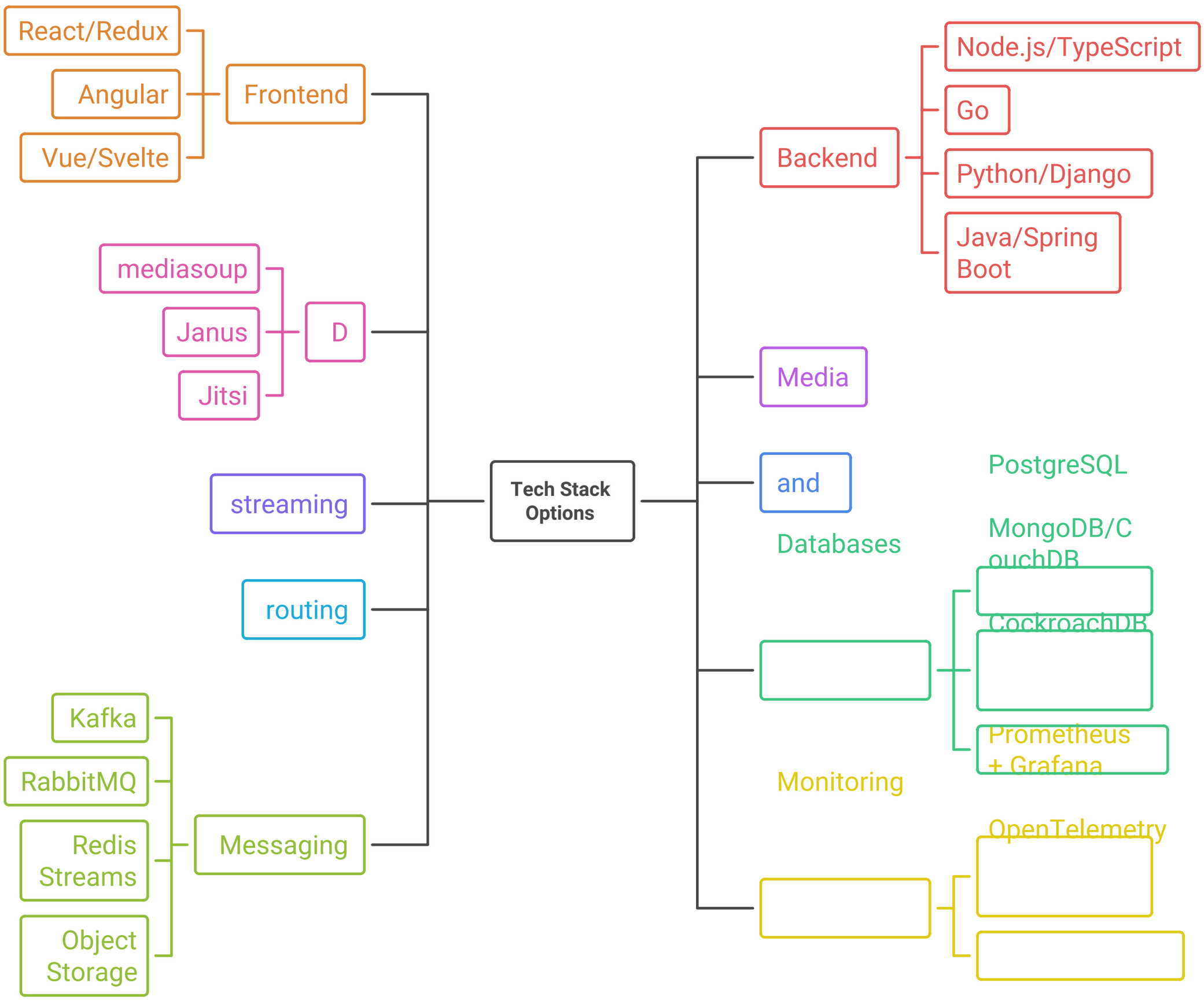
- PostgreSQL: Highly reliable ACID DB for PHI (recommended). Strong support for encryption, replication, JSONB. ·
- MongoDB/CouchDB: Flexible schema (notes), but handle in conjunction with Postgres. ·
- CockroachDB: Distributed SQL for multi-region, but higher complexity – use only if needed. ·

Messaging: ·

- Kafka (Recommended): High-throughput event streaming (audit logs, event sourcing). ·
- RabbitMQ: Easier to set up for low-to-mid throughput, supports complex routing. ·
- Redis Streams: Lightweight event log for simpler cases, but less durable. ·
- Object Storage: S3 or S3-compatible (minIO etc.) recommended for reliability and cost. ·

- Monitoring: Prometheus + Grafana (open-source) with OpenTelemetry for tracing. · Each choice balances cost, developer familiarity, scalability, and ecosystem maturity. E.g., we recommend React/Node.js for rapid development and strong community support in realtime apps, and Kafka for durable event logs in an audit-critical environment.

Tech Stack Options and Justifications



13. Testing, QA & Deployment

13.1 Testing Strategy

- Unit Tests: Each service should have high coverage on business logic (e.g. booking logic, auth checks). Mock external dependencies (DB, other services). ·
- Integration Tests: Stand up key services (database, cache) and test end-to-end flows within a service (e.g. DB+app). ·
- Contract/API Tests: For APIs, ensure backward compatibility; use OpenAPI/Swagger to define schemas and test clients. ·
- End-to-End (E2E) Tests: Automated tests simulating user flows (booking, video call, chat). Use headless browsers or device emulators, including network impairment (latency, packet loss) to ensure resiliency. ·

- Load Testing: Simulate expected load for video (e.g. 100s concurrent calls) and messaging (msgs/ sec) to verify SLAs. Adjust autoscaling. ·
- Security/Compliance Tests: Static code analysis (check for secrets), dynamic analysis (OWASP ZAP), and regular penetration tests. ·

13.2 CI/CD

- Pipeline: On commit, run linting, unit tests. Merge to main triggers integration and contract tests. Deploy to staging and run smoke tests. ·
- Canary/Blue-Green: Gradual rollout of new versions (especially for stateful services). For example, shift 10% of traffic to new deploy, monitor, then complete rollout. ·
- Rollbacks: Automated rollback on failure (high error rate or SLO breach). No human intervention needed for reversions. ·
- Tools: Jenkins/GitHub Actions/GitLab CI, Docker/K8s for artifacts and deployment, Helm/Terraform for infra as code. ·

13.3 Environment Layout

Separate environments with identical topology: dev (for rapid changes), qa, and staging (prod-like for final tests), and production. Each in separate VPCs with strict access controls. Use feature flags to deploy new features in dev/staging before enabling in prod. Manage secrets via a vault (e.g. HashiCorp Vault).

14. Privacy, Legal & Compliance

Data Flow Mapping: Identify all PHI flows (in transit, at rest). Ensure encryption and access controls at each point. ·

- Consent & Privacy: Build UI workflows for patient consent and data sharing preferences (e.g. sharing data with specialists or analytics). ·
- Data Residency: Ensure that PHI storage complies with local laws (data may need to stay in-region). Possibly use geo-replication constraints. ·
- Retention & Deletion: Implement data retention policies per above (e.g. delete chat records if user requests, after minimum retention). Provide "right to be forgotten" as applicable. ·
- Audit Trails: Keep full logs of who accessed/modified patient data, with timestamps (meeting HIPAA Security Rule). Use these in compliance reports. ·
- HIPAA/PCI/GDPR Checks: If handling payments (PCI) or EU patients (GDPR), ensure appropriate measures (e.g. PCI-DSS for payments, GDPR data export controls). ·
- Legal Documents: Prepare Business Associate Agreements (BAAs) with any vendors (e.g. cloud provider, API partners). ·

HIPAA requires covered entities to ensure their telehealth platforms maintain "secure communications and data storage". We will document our compliance, undergo regular audits, and train staff on privacy (HIPAA training).