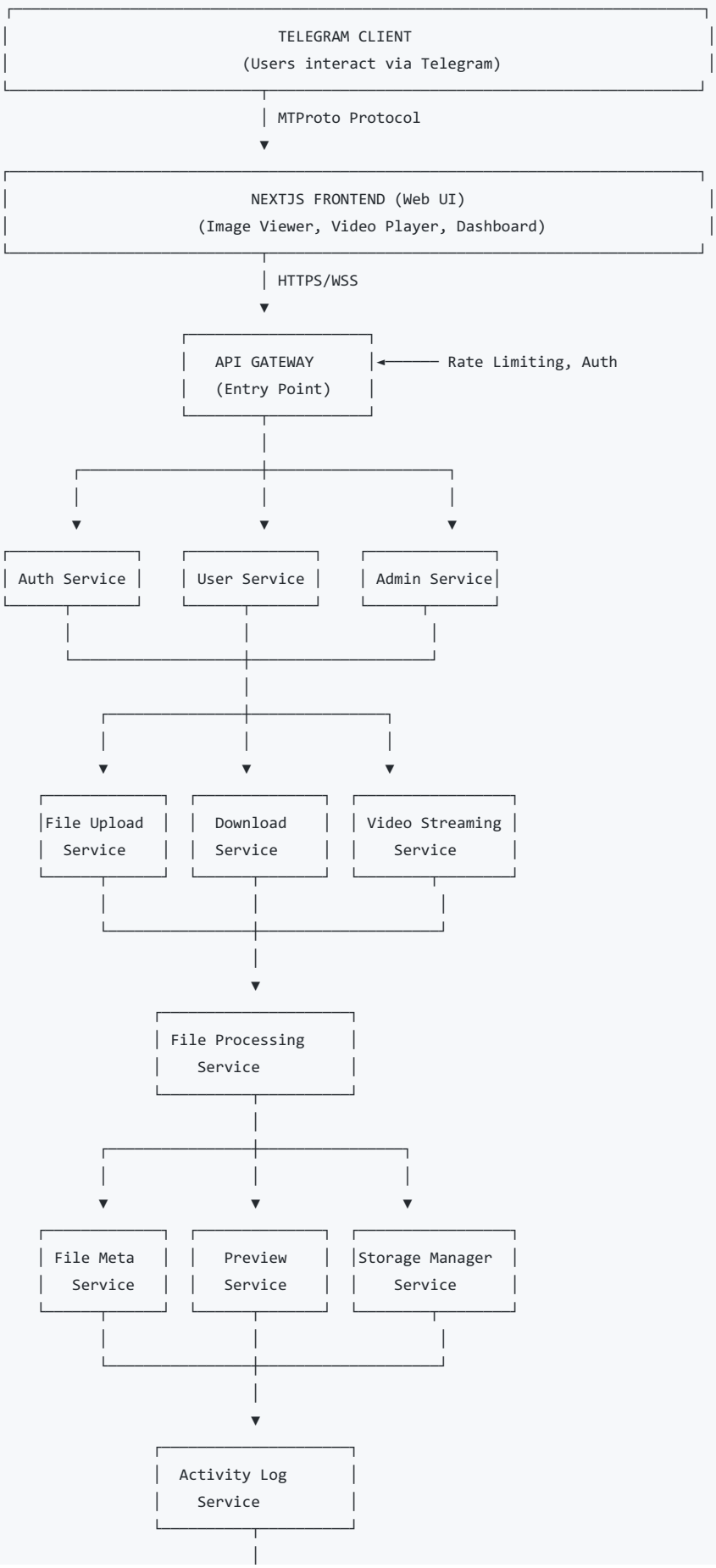
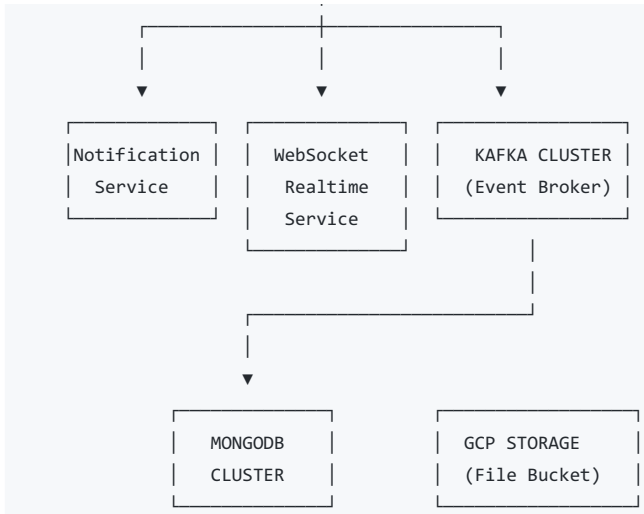
TeleDrive: Complete PRD + Technical Architecture Document

1️⃣ HIGH LEVEL ARCHITECTURE DIAGRAM






```
KAFKA TOPICS (Event Flow):
├─ file.uploaded
├─ file.processing.started
├─ file.processing.completed
├─ file.deleted
├─ user.registered
├─ download.requested
├─ video.streaming.started
├─ notification.created
└─ activity.logged

GCP DEPLOYMENT LAYOUT:
├─ Cloud Run (Each Microservice as Container)
├─ Cloud Storage (File Storage)
├─ MongoDB Atlas (Database)
├─ Cloud Pub/Sub or Kafka on GKE (Event Streaming)
├─ Cloud Load Balancer (API Gateway)
└─ Cloud CDN (Static Assets, Video Streaming)
```

□ PRODUCT REQUIREMENTS DOCUMENT (PRD)

1□ EXECUTIVE SUMMARY

TeleDrive is a cloud storage platform that leverages Telegram as the primary interface for file management. Users can upload, download, view images, and stream videos directly through Telegram, while also accessing a web dashboard built with NextJS for enhanced viewing experiences.

The platform demonstrates modern microservices architecture principles using TypeScript, MongoDB, Kafka for event-driven communication, and WebSockets for real-time updates. It's designed as a learning project that maintains production-level architectural patterns while keeping complexity manageable.

Key Value Propositions:

- Seamless Telegram integration using MTProto protocol
- Real-time file processing notifications
- Scalable microservices architecture
- Cross-platform access (Telegram + Web)
- Secure cloud storage with GCP infrastructure

2□ OBJECTIVES

Primary Objectives:

1. Create a functional cloud drive accessible via Telegram
2. Demonstrate microservices architecture best practices
3. Implement event-driven communication patterns
4. Provide real-time user notifications
5. Build a scalable, production-ready system architecture

Learning Objectives:

1. Master microservices design and implementation
2. Understand event-driven architecture with Kafka
3. Implement WebSocket real-time communication
4. Work with Telegram MTProto protocol
5. Deploy containerized applications on GCP
6. Design MongoDB schemas for distributed systems
7. Handle file processing and streaming

Success Metrics:

- Successful file upload/download operations
- Video streaming with minimal buffering
- Real-time notifications delivery within 2 seconds
- System handles concurrent users efficiently
- 99.9% uptime for core services

3️⃣ TARGET USERS

Primary User Personas:

1. Individual Cloud Storage Users

- Age: 18-45
- Tech-savvy individuals who use Telegram daily
- Need: Personal cloud storage accessible from Telegram
- Pain Point: Want simple file storage without complex apps

2. Small Team Collaborators

- Age: 25-50
- Small businesses or remote teams
- Need: Quick file sharing and access via familiar interface
- Pain Point: Existing cloud solutions are expensive or complex

3. Media Consumers

- Age: 18-35
- Users who store and stream personal media
- Need: Easy video/image streaming and viewing
- Pain Point: Want unified storage and streaming platform

4. Microservices Learners

- Age: 20-40
- Developers learning distributed systems
- Need: Realistic microservices project
- Pain Point: Most tutorials lack real-world complexity

4️⃣ CORE FEATURES

4.1 File Upload

Users can upload files of any type through Telegram. The system accepts files up to 2GB, processes them asynchronously, and provides real-time upload progress notifications.

Behavior:

- User sends file to Telegram bot
- System validates file size and type
- File is uploaded to GCP Storage
- Metadata is extracted and stored
- User receives confirmation with file details
- Real-time progress updates via WebSocket

4.2 File Download

Users can request file downloads through Telegram commands or web interface. System generates secure temporary download links.

Behavior:

- User requests file by name or ID
- System validates user permissions
- Temporary signed URL is generated
- Link is delivered to user
- Download activity is logged
- Link expires after specified time

4.3 Image Viewing

Users can view uploaded images directly within the web platform with thumbnail previews.

Behavior:

- System generates multiple image sizes on upload
- Thumbnails displayed in gallery view
- Full-size images load on demand
- Lazy loading for performance
- Image metadata displayed (size, date, dimensions)

4.4 Video Streaming

Users can stream videos directly in the web player with adaptive quality.

Behavior:

- Videos are processed for streaming on upload
- System generates multiple quality variants
- Adaptive bitrate streaming based on connection
- Progress saved across sessions
- Playback controls (play, pause, seek, volume)
- Real-time buffering status

4.5 Authentication & Authorization

Secure user authentication through Telegram OAuth and JWT tokens for API access.

Behavior:

- Telegram OAuth flow for user login
- JWT tokens issued for API access
- Token refresh mechanism
- Role-based access control (user, admin)
- Session management
- Secure token storage

4.6 Storage Management

Users can view storage usage, organize files, and manage their data.

Behavior:

- Display total storage used
- List all uploaded files
- File organization (folders/tags)
- Search and filter capabilities
- Bulk operations (delete, move)
- Storage quota enforcement

4.7 Notifications

Real-time notifications for file operations, processing status, and system events.

Behavior:

- Upload completion notifications
- Processing status updates
- Storage quota warnings
- Shared file notifications
- System maintenance alerts
- Delivered via Telegram and WebSocket

4.8 Real-time Updates

WebSocket-based real-time updates for all file operations and system events.

Behavior:

- Live upload progress
- Processing status changes
- New file notifications
- Storage updates
- Multi-device synchronization
- Connection resilience

4.9 Activity Logging

Comprehensive logging of all user activities and system operations.

Behavior:

- All file operations logged
- User action tracking
- System event recording
- Audit trail for admin
- Analytics data collection
- Searchable log history

5️⃣ SYSTEM ARCHITECTURE EXPLANATION

5.1 Why Microservices?

Modularity and Independence: Each service handles a specific business capability (authentication, file upload, streaming, etc.). Services can be developed, deployed, and scaled independently, making the system more maintainable and easier to understand.

Technology Flexibility: Different services can use different technologies or databases if needed. For example, the video streaming service might have different requirements than the authentication service.

Fault Isolation: If one service fails, others continue operating. A failure in the notification service doesn't prevent file uploads from working.

Team Scalability: Multiple teams can work on different services simultaneously without conflicts or coordination overhead.

Learning Value: Demonstrates real-world distributed systems architecture, preparing developers for production environments.

5.2 Communication Methods

Synchronous Communication (HTTP/REST): Used for direct request-response patterns through the API Gateway. Example: User requests file download, API Gateway calls Download Service, receives response, returns to user.

Asynchronous Communication (Kafka): Used for event-driven patterns where services react to events without blocking. Example: File Upload Service publishes "file.uploaded" event, multiple services (Processing, Metadata, Notification) react independently.

Real-time Communication (WebSocket): Used for pushing updates to connected clients without polling. Example: Upload progress, processing status, new file notifications pushed directly to user's browser.

5.3 Kafka Role

Event Streaming Platform: Kafka acts as the central nervous system for asynchronous communication between services. It provides reliable, ordered, and persistent message delivery.

Key Responsibilities:

- Decouple producers from consumers
- Buffer messages during service downtime
- Enable event replay for debugging
- Support multiple consumers per event
- Guarantee message delivery order
- Provide fault tolerance

Usage Pattern: Services publish domain events to Kafka topics. Other services subscribe to relevant topics and react to events independently. This creates a loosely coupled system where services don't need to know about each other.

5.4 WebSocket Role

Real-time Bidirectional Communication: WebSocket provides persistent connections between client and server, enabling instant push notifications without polling overhead.

Key Responsibilities:

- Push real-time updates to connected clients
- Maintain persistent connections
- Broadcast system-wide events
- Send user-specific notifications
- Handle connection lifecycle
- Provide connection resilience

Usage Pattern: Clients establish WebSocket connections to Realtime Service. When events occur (file uploaded, processing complete), services publish to Kafka, Realtime Service consumes events and pushes to appropriate WebSocket connections.

5.5 Telegram MTProto Role

Telegram Integration Protocol: MTProto is Telegram's custom protocol for secure, fast communication with Telegram servers. It enables TeleDrive to act as a Telegram bot and receive/send messages.

Key Responsibilities:

- Authenticate with Telegram servers
- Receive incoming messages and files
- Send messages and files to users
- Handle Telegram bot commands
- Process file uploads from Telegram
- Deliver notifications via Telegram

Usage Pattern: Services interact with Telegram through a client library that implements MTProto. When users send files via Telegram, MTProto receives them, and services process them. When system sends notifications, MTProto delivers them to Telegram users.

5.6 Simple Scaling Explanation

Horizontal Scaling: Each microservice runs in a container and can be replicated. When load increases on File Upload Service, deploy more instances. Load balancer distributes requests across instances.

Database Scaling: MongoDB supports replica sets for read scaling and sharding for write scaling. Read-heavy operations (file listing) use read replicas. Write operations go to primary.

Kafka Scaling: Kafka topics can be partitioned. Multiple instances of a service can consume from different partitions in parallel, increasing throughput.

Storage Scaling: GCP Storage automatically scales. No manual intervention needed as storage needs grow.

Stateless Services: All services are stateless (no in-memory session data). State stored in MongoDB or cache. This allows unlimited horizontal scaling without sticky sessions.

Auto-scaling Strategy:

- CPU-based: Scale when CPU > 70%
- Request-based: Scale when request queue > threshold
- Time-based: Pre-scale during predicted peak hours

5.7 GCP Deployment Idea

Cloud Run Deployment (Recommended for Learning): Each microservice deployed as a Cloud Run service. Cloud Run provides automatic scaling, built-in load balancing, and simple deployment. Pay only for actual usage, making it cost-effective for learning projects.

Architecture:

- API Gateway: Cloud Load Balancer + Cloud Run
- Microservices: Individual Cloud Run services
- Database: MongoDB Atlas (managed)
- Message Queue: Cloud Pub/Sub or self-hosted Kafka on GKE
- Storage: Cloud Storage buckets
- CDN: Cloud CDN for video streaming

Alternative: GKE Deployment (Production-like): Deploy all services to Google Kubernetes Engine for more control. Use Kubernetes deployments, services, and ingress. Provides full container orchestration but more complex to manage.

Deployment Flow:

1. Build Docker images for each service
2. Push images to Google Container Registry
3. Deploy to Cloud Run or GKE
4. Configure environment variables
5. Set up load balancing and routing
6. Configure auto-scaling policies
7. Set up monitoring and logging

MICROSERVICES DETAILED SPECIFICATION

---

1. Auth Service

Purpose: Handles all authentication and authorization operations including Telegram OAuth, JWT token generation, validation, and refresh operations.

Responsibilities:

- Process Telegram OAuth flow
- Generate and sign JWT access tokens
- Generate and manage refresh tokens
- Validate incoming JWT tokens
- Handle token refresh requests
- Revoke tokens on logout
- Manage user sessions
- Enforce rate limiting on auth endpoints

Input/Output Description:

- **Input:** Telegram auth code, username, password (for admin), refresh tokens
- **Output:** JWT access tokens, refresh tokens, user session data, authentication status

MongoDB Usage: Stores authentication-related data in "auth_sessions" collection. Each document contains user ID, refresh token hash, issued timestamp, expiry timestamp, device information, and revocation status. Indexes on user_id and refresh_token for fast lookups.

Kafka Topics Involvement:

- **Publishes to:** user.logged_in, user.logged_out, token.revoked
- **Consumes from:** user.registered (to set up initial auth credentials)

WebSocket Usage: Does not directly use WebSocket. Auth events are consumed by WebSocket Realtime Service to notify other devices about login/logout events for session management.

Failure Handling:

- Invalid tokens return 401 Unauthorized
- Failed Telegram auth returns descriptive error
- Token generation failures logged and retried
- Rate limiting prevents brute force attempts
- Failed operations logged to Activity Log Service via Kafka

---

2. User Service

Purpose: Manages user profiles, preferences, storage quotas, and user-related metadata.

Responsibilities:

- Create new user profiles
- Update user information and preferences
- Manage storage quota allocation
- Track storage usage per user
- Handle user settings (theme, notification preferences)
- Provide user search and listing (admin)
- Soft delete user accounts
- Calculate and enforce storage limits

Input/Output Description:

- **Input:** User registration data, profile updates, settings changes, user ID queries
- **Output:** User profile data, storage usage statistics, user lists, operation status

MongoDB Usage: Stores user data in "users" collection. Fields include telegram_id, username, email, profile_picture, storage_quota, storage_used, preferences object, created_at, updated_at, is_active. Indexes on telegram_id and email for authentication lookups. Compound index on storage_used and storage_quota for quota enforcement queries.

Kafka Topics Involvement:

- **Publishes to:** user.registered, user.updated, user.deleted, user.quota_exceeded
- **Consumes from:** file.uploaded, file.deleted (to update storage_used)

WebSocket Usage: Storage usage updates pushed via WebSocket when files are uploaded or deleted, allowing real-time quota display in UI.

Failure Handling:

- Duplicate user registration returns conflict error
- Storage quota violations prevent file uploads
- Invalid user updates return validation errors
- Database failures trigger retry with exponential backoff
- Critical failures logged to monitoring system

---

3. File Upload Service

Purpose: Handles file upload operations from Telegram, validates files, manages upload to cloud storage, and coordinates the upload workflow.

Responsibilities:

- Receive files from Telegram MTProto
- Validate file size and type
- Generate unique file identifiers
- Upload files to GCP Storage
- Create initial file metadata
- Calculate file checksums
- Handle multipart uploads for large files
- Provide upload progress tracking
- Trigger file processing workflows

Input/Output Description:

- **Input:** File data from Telegram, user ID, file metadata (name, size, type)
- **Output:** File ID, upload status, storage URL, checksum, progress updates

MongoDB Usage: Creates initial document in "files" collection with fields: file_id, user_id, filename, size, mime_type, storage_path, checksum, upload_status, upload_started_at, upload_completed_at. Uses "upload_sessions" collection to track multipart uploads with session_id, file_id, parts_uploaded array, total_parts.

Kafka Topics Involvement:

- **Publishes to:** file.uploaded, file.upload.failed, file.upload.progress
- **Consumes from:** None (entry point for file uploads)

WebSocket Usage: Publishes upload progress events to WebSocket Realtime Service via Kafka, which pushes updates to user's connected clients showing real-time upload percentage.

Failure Handling:

- File size validation fails immediately with error message
- Partial uploads stored temporarily with resume capability
- Network failures trigger automatic retry with exponential backoff
- Storage failures logged and user notified via Telegram
- Failed uploads cleaned up after 24 hours
- All failures logged to Activity Log Service

---

4. File Processing Service

Purpose: Processes uploaded files asynchronously, generates thumbnails for images, converts videos for streaming, extracts metadata, and performs virus scanning.

Responsibilities:

- Queue files for processing from Kafka events
- Generate image thumbnails (small, medium, large)
- Extract image EXIF metadata
- Transcode videos to multiple resolutions
- Generate video thumbnails and previews
- Extract video metadata (duration, codec, resolution)
- Perform virus/malware scanning
- Update processing status in database
- Retry failed processing operations

Input/Output Description:

- **Input:** File ID, storage path, file type from Kafka events
- **Output:** Processed file variants, thumbnails, extracted metadata, processing status

MongoDB Usage: Updates "files" collection with processing status and results. Adds processed_variants array containing URLs of different sizes/resolutions. Updates metadata object with extracted information (dimensions, duration, codec). Uses "processing_jobs" collection to track job status with fields: job_id, file_id, processing_type, status, started_at, completed_at, error_message.

Kafka Topics Involvement:

- **Publishes to:** file.processing.started, file.processing.completed, file.processing.failed
- **Consumes from:** file.uploaded

WebSocket Usage: Processing status updates pushed to users in real-time via Kafka to WebSocket Realtime Service, allowing users to see when their files are ready.

Failure Handling:

- Processing failures logged with detailed error messages
- Failed jobs automatically retried up to 3 times
- Permanent failures mark file as "processing_failed" in database
- User notified via notification service
- Corrupt or invalid files quarantined
- Resource exhaustion handled with job queuing and rate limiting

---

5. File Metadata Service

Purpose: Manages comprehensive file metadata, search indexing, tagging, and provides fast metadata queries for listing and searching files.

Responsibilities:

- Store and index file metadata
- Provide file search functionality
- Manage file tags and categories
- Handle file organization (folders/collections)
- Generate file listings with pagination
- Filter files by type, date, size, tags
- Update metadata from processing results
- Maintain search indexes

Input/Output Description:

- **Input:** Metadata updates, search queries, filter parameters, tag operations
- **Output:** File metadata objects, search results, filtered lists, aggregated statistics

MongoDB Usage: Reads from "files" collection with extensive indexes. Creates text index on filename for search. Compound indexes on user_id + created_at for listing, user_id + mime_type for filtering. Stores tags in "tags" collection with many-to-many relationship to files. Uses aggregation pipelines for complex queries and statistics.

Kafka Topics Involvement:

- **Publishes to:** metadata.updated, metadata.search.completed
- **Consumes from:** file.uploaded, file.processing.completed (to update metadata)

WebSocket Usage: New files and metadata updates broadcast to connected clients so file lists update automatically without refresh.

Failure Handling:

- Search queries timeout after 5 seconds with partial results
- Invalid queries return validation errors
- Database read failures return cached results if available
- Metadata inconsistencies logged for manual review
- Index rebuild scheduled automatically if corruption detected

---

6. Download Service

Purpose: Handles file download requests, generates secure temporary signed URLs, manages download permissions, and tracks download activity.

Responsibilities:

- Validate download permissions
- Generate secure temporary signed URLs for GCP Storage
- Set appropriate expiration times
- Handle direct download requests
- Support range requests for resumable downloads

- Track download statistics
- Enforce rate limits on downloads
- Manage bandwidth throttling if needed

Input/Output Description:

- **Input:** File ID, user ID, download options (range, expiration)
- **Output:** Signed download URL, download token, expiration time, file metadata

MongoDB Usage: Queries "files" collection to verify file existence and permissions. Uses "downloads" collection to track download history with fields: download_id, file_id, user_id, downloaded_at, ip_address, user_agent, download_size. Indexes on file_id and user_id for analytics.

Kafka Topics Involvement:

- **Publishes to:** download.requested, download.completed, download.failed
- **Consumes from:** None (entry point for downloads)

WebSocket Usage: Does not directly use WebSocket. Download events logged and can trigger notifications for shared files (notify owner when someone downloads).

Failure Handling:

- Invalid file IDs return 404 Not Found
- Permission failures return 403 Forbidden
- Expired signed URLs generate new ones automatically
- Download interruptions support resume via range requests
- Rate limit violations return 429 with retry-after header
- All failures logged for analytics

---

7. Video Streaming Service

Purpose: Provides adaptive video streaming capabilities with multiple quality options, handles video playback requests, and manages streaming sessions.

Responsibilities:

- Serve video stream segments
- Implement adaptive bitrate streaming
- Generate m3u8 playlists for HLS
- Handle seeking and time-range requests
- Manage buffering and cache headers
- Track playback analytics
- Support multiple quality levels
- Implement video player API

Input/Output Description:

- **Input:** File ID, quality preference, time offset, playback session data
- **Output:** Video stream data, playlist files, stream metadata, playback state

MongoDB Usage: Queries "files" collection for video file information and processed variants. Uses "playback_sessions" collection to track viewing progress with fields: session_id, file_id, user_id, current_position, quality_level, started_at, last_updated_at. Allows resume functionality across devices.

Kafka Topics Involvement:

- **Publishes to:** video.streaming.started, video.streaming.stopped, video.playback.progress
- **Consumes from:** file.processing.completed (to know when video variants are ready)

WebSocket Usage: Can push buffering status and quality changes to clients in real-time for better user experience indication.

Failure Handling:

- Missing video variants fallback to original file
- Streaming errors attempt quality downgrade
- Network interruptions support automatic reconnection
- Corrupt segments skipped with gap handling
- Buffer underruns trigger quality reduction
- Failed sessions logged with error details

---

8. Preview Service

Purpose: Generates and serves preview content for files including image thumbnails, video previews, document previews, and quick-view content.

Responsibilities:

- Serve image thumbnails in various sizes
- Generate and cache document previews
- Provide video thumbnail frames
- Create preview cards for link sharing
- Handle lazy loading requests
- Implement image optimization
- Support responsive image serving
- Cache frequently accessed previews

Input/Output Description:

- **Input:** File ID, preview type, size parameters, format preferences
- **Output:** Preview images, thumbnail URLs, optimized content, cache headers

MongoDB Usage: Queries "files" collection for preview URLs and metadata. Uses "preview_cache" collection to track generated previews with fields: preview_id, file_id, preview_type, size, format, storage_path, generated_at, access_count. Implements cache eviction based on access patterns.

Kafka Topics Involvement:

- **Publishes to:** preview.generated, preview.cache.miss
- **Consumes from:** file.processing.completed (to cache new previews)

WebSocket Usage: Does not directly use WebSocket. Preview generation status can be pushed for large files requiring on-demand preview generation.

Failure Handling:

- Missing previews trigger on-demand generation
- Generation failures return placeholder images
- Invalid image data returns error image
- Cache misses logged and preview generated
- Excessive preview requests rate limited
- Failed generations retried with lower quality

---

9. Notification Service

Purpose: Manages all notification delivery across multiple channels including Telegram messages, in-app notifications, and email alerts.

Responsibilities:

- Send Telegram notifications via MTProto
- Deliver in-app notifications via WebSocket
- Send email notifications (if configured)
- Queue notifications for delivery
- Handle notification preferences
- Retry failed notification deliveries
- Track notification read status
- Support notification templates
- Batch notifications to avoid spam

Input/Output Description:

- **Input:** Notification events from Kafka, user IDs, notification content, priority levels
- **Output:** Delivery status, read receipts, notification history

MongoDB Usage: Stores notifications in "notifications" collection with fields: notification_id, user_id, type, title, message, priority, status, channels array, created_at, delivered_at, read_at. Indexes on user_id + read_at for unread notifications query. Stores user notification preferences in "notification_preferences" collection.

Kafka Topics Involvement:

- **Publishes to:** notification.sent, notification.delivered, notification.failed
- **Consumes from:** All service events that trigger notifications (file.uploaded, file.processing.completed, user.quota_exceeded, etc.)

WebSocket Usage: Primary channel for in-app notifications. Pushes notifications to connected clients instantly via WebSocket Realtime Service.

Failure Handling:

- Failed Telegram deliveries retried up to 5 times
- User unreachable notifications queued for later
- Invalid notification data logged and discarded
- Rate limiting prevents notification spam
- Delivery failures don't block application flow
- Critical notifications escalated to multiple channels

---

10. API Gateway

Purpose: Single entry point for all client requests, handles routing, authentication, rate limiting, request validation, and API versioning.

Responsibilities:

- Route requests to appropriate microservices
- Validate JWT tokens on protected routes
- Implement rate limiting per user/IP
- Handle request/response transformation
- Provide API versioning support
- Aggregate responses from multiple services
- Implement circuit breaker patterns
- Generate API documentation
- Log all API requests

Input/Output Description:

- **Input:** All HTTP requests from clients (web, mobile, Telegram bot)
- **Output:** Proxied responses from microservices, aggregated data, error messages

MongoDB Usage: Uses "api_keys" collection for API key validation (if public API provided). Stores rate limit counters in "rate_limits" collection with fields: key (user_id or IP), endpoint, request_count, window_start, window_end. Implements sliding window rate limiting.

Kafka Topics Involvement:

- **Publishes to:** api.request.logged, api.error.occurred

- **Consumes from:** None (entry point for HTTP requests)

WebSocket Usage: Handles WebSocket connection establishment and routes to WebSocket Realtime Service. Validates WebSocket auth before upgrade.

Failure Handling:

- Service unavailable returns 503 with retry-after
- Timeout errors return 504 Gateway Timeout
- Circuit breaker opens after repeated failures
- Failed services bypass with degraded functionality
- All errors logged with request context
- Health checks determine service availability

---

11. WebSocket Realtime Service

Purpose: Manages persistent WebSocket connections with clients, receives events from Kafka, and pushes real-time updates to connected users.

Responsibilities:

- Maintain WebSocket connections
- Authenticate WebSocket connections
- Subscribe to Kafka topics for events
- Route events to appropriate connections
- Handle connection lifecycle (connect, disconnect, reconnect)
- Implement heartbeat/ping-pong
- Support room-based broadcasting
- Handle connection scaling across instances
- Provide connection statistics

Input/Output Description:

- **Input:** WebSocket connection requests, Kafka events, user IDs for targeting
- **Output:** Real-time event pushes to clients, connection status, presence information

MongoDB Usage: Uses "websocket_connections" collection for connection registry with fields: connection_id, user_id, connected_at, last_ping, server_instance. Enables routing events to correct server instance in multi-instance deployment. Short TTL for automatic cleanup of stale connections.

Kafka Topics Involvement:

- **Publishes to:** None (pushes to WebSocket, doesn't publish to Kafka)
- **Consumes from:** ALL event topics that require real-time updates (file.uploaded, file.processing.completed, notification.sent, download.completed, etc.)

WebSocket Usage: Core service for WebSocket functionality. Manages all WebSocket connections and event broadcasting. Implements different event channels (personal, broadcast, room-based).

Failure Handling:

- Disconnected clients automatically reconnect with backoff
- Failed event deliveries logged but don't block other deliveries
- Connection overload triggers graceful rejection with retry
- Server crashes don't lose events (Kafka retention)
- Reconnection delivers missed events from Kafka offset
- Load balancing distributes connections across instances

---

12. Storage Manager Service

Purpose: Manages cloud storage operations, handles file lifecycle, implements storage policies, and provides storage analytics.

Responsibilities:

- Interface with GCP Storage API
- Manage storage bucket operations
- Implement file retention policies
- Handle file archival and deletion
- Calculate storage costs
- Monitor storage usage
- Implement storage quotas
- Handle storage migrations
- Provide storage analytics

Output Description:

- **Input:** Storage operations requests, file IDs, bucket names, lifecycle policies
- **Output:** Storage URLs, operation status, storage metrics, cost analytics

MongoDB Usage: Uses "storage_objects" collection mapping file_id to storage_path, bucket, size, storage_class, created_at. Implements "storage_policies" collection for retention rules with fields: policy_id, file_type_pattern, retention_days, archive_after_days, delete_after_days. Runs scheduled jobs to apply policies.

Kafka Topics Involvement:

- **Publishes to:** storage.object.created, storage.object.deleted, storage.policy.applied
- **Consumes from:** file.uploaded, file.deleted, user.deleted

WebSocket Usage: Pushes storage usage updates to admin dashboard in real-time for monitoring.

Failure Handling:

- Storage operation failures retry with exponential backoff
- Quota exceeded errors prevent operations gracefully
- Network failures queue operations for later
- Bucket access errors escalated to administrators
- Failed deletions marked for manual review
- Storage inconsistencies auto-reconciled daily

---

13. Activity Log Service

Purpose: Centralized logging service that records all system activities, user actions, and events for audit trails, analytics, and debugging.

Responsibilities:

- Consume events from all Kafka topics
- Store detailed activity logs
- Provide log search and filtering
- Generate activity reports
- Implement log retention policies
- Support audit trail requirements
- Provide analytics data
- Handle high-volume log ingestion

Input/Output Description:

- **Input:** All Kafka events, system logs, user actions
- **Output:** Log entries, search results, analytics reports, audit trails

MongoDB Usage: Stores logs in "activity_logs" collection with fields: log_id, timestamp, user_id, service_name, action_type, resource_id, details object, ip_address, user_agent. Uses time-series collection optimization. Indexes on timestamp, user_id, action_type for efficient queries. Implements TTL index for automatic old log deletion.

Kafka Topics Involvement:

- **Publishes to:** None (end consumer)
- **Consumes from:** ALL topics (comprehensive logging)

WebSocket Usage: Can push real-time activity feed to admin dashboard showing live system activity.

Failure Handling:

- Log ingestion failures don't affect source services
- Database write failures buffer logs in memory
- Buffer overflow writes to local disk temporarily
- Critical logs escalated to monitoring system
- Log processing lag triggers scaling
- Failed log writes retried with backoff

---

14. Admin Service

Purpose: Provides administrative functionality for system management, user management, analytics, and monitoring.

Responsibilities:

- Manage user accounts (activate, deactivate, delete)
- View system statistics and analytics
- Manage storage quotas
- Handle user support requests
- Monitor system health
- Configure system settings
- Generate administrative reports
- Manage content moderation
- Handle bulk operations

Input/Output Description:

- **Input:** Admin commands, user queries, report parameters, configuration changes
- **Output:** User lists, system statistics, reports, operation results, dashboard data

MongoDB Usage: Reads from all collections for analytics and reporting. Uses aggregation pipelines for complex reports. Stores admin actions in "admin_actions" collection with fields: admin_id, action_type, target_id, action_details, timestamp. Creates materialized views for dashboard statistics.

Kafka Topics Involvement:

- **Publishes to:** admin.action.executed, user.account.modified, system.config.changed
- **Consumes from:** Listens to critical events for monitoring (system errors, quota exceeded, suspicious activity)

WebSocket Usage: Receives real-time system metrics and alerts via WebSocket for live admin dashboard.

Failure Handling:

- Read-only operations fallback to cached data
- Write operations require confirmation
- Bulk operations implemented with rollback capability
- Critical admin actions require two-factor authentication
- Failed operations logged with admin context

- System-wide changes implemented gradually with monitoring

15-PHASE DEVELOPMENT PLAN

PHASE 1: Foundation & Infrastructure Setup

Objective: Set up development environment, initialize monorepo structure, configure Docker, set up MongoDB, and establish Kafka infrastructure.

Services Involved: None (infrastructure only)

Detailed Development Tasks:

1. Initialize monorepo with workspace configuration
2. Create folder structure for all 14 microservices
3. Set up shared TypeScript configuration
4. Configure Docker and Docker Compose files
5. Set up local MongoDB instance or Atlas connection
6. Install and configure Kafka locally or use Confluent Cloud
7. Create shared types and interfaces package
8. Set up environment variable management
9. Configure logging framework
10. Set up Git repository and branching strategy
11. Create development documentation

Dependencies:

- Node.js and TypeScript installed
- Docker Desktop installed
- MongoDB access configured
- Kafka instance available

Deliverables:

- Complete monorepo structure
- Docker Compose configuration running MongoDB and Kafka
- Shared types package
- Development environment documentation
- README with setup instructions

Risks:

- Kafka setup complexity may delay start
- Docker networking issues on different operating systems
- MongoDB connection configuration challenges

Acceptance Criteria:

- All developers can run `docker-compose up` successfully
- MongoDB connection verified
- Kafka topics can be created and listed
- Shared types can be imported across services
- Git workflow documented

Easy Verification/Testing Checklist:

- Run `docker-compose ps` and verify all containers running
- Connect to MongoDB using client and create test document
- Use Kafka CLI to produce and consume test message
- Import shared types in test file without errors
- Review folder structure matches design document

Expected Behavior: Docker containers start without errors, MongoDB accepts connections on configured port, Kafka responds to topic commands, TypeScript compilation succeeds.

Validation Criteria:

- Zero errors in container logs
- Successful MongoDB read/write operation
- Successful Kafka produce/consume operation
- Clean TypeScript compilation

PHASE 2: Auth Service & User Service

Objective: Implement authentication system with Telegram OAuth, JWT token generation, and basic user management functionality.

Services Involved:

- Auth Service
- User Service

Detailed Development Tasks:

1. Create Auth Service with Telegram OAuth integration
2. Implement JWT token generation and validation logic
3. Create refresh token mechanism
4. Set up MongoDB collections for auth and users
5. Implement User Service with CRUD operations
6. Create user registration flow
7. Implement storage quota management in User Service
8. Set up Kafka topics for auth and user events
9. Implement token validation middleware
10. Create health check endpoints
11. Write integration tests for auth flow

Dependencies:

- Phase 1 completed
- Telegram Bot created and token obtained
- JWT library selected

Deliverables:

- Working Auth Service with Telegram OAuth
- Working User Service with user management
- MongoDB collections for auth and users
- Kafka topics: user.registered, user.logged_in
- Token validation middleware
- API documentation for both services

Risks:

- Telegram OAuth integration complexity
- JWT security implementation errors
- Token refresh race conditions

Acceptance Criteria:

- User can authenticate via Telegram
- JWT tokens generated and validated correctly
- Refresh tokens work properly
- User profile created on registration
- Storage quota tracked accurately
- All endpoints return proper HTTP status codes

Easy Verification/Testing Checklist:

- Use Postman to simulate Telegram OAuth callback
- Verify JWT token in jwt.io decoder
- Test token refresh with expired access token
- Create user and verify in MongoDB
- Check Kafka topic for user.registered event
- Test invalid token returns 401

Expected Behavior: Telegram auth redirects to callback URL, JWT token returned in response, user document created in MongoDB, Kafka event published for user registration.

Validation Criteria:

- JWT token contains correct claims (user_id, exp, iat)
- User document has all required fields
- Token validation correctly identifies valid/invalid tokens
- Refresh token extends session properly

PHASE 3: API Gateway & Basic Routing

Objective: Create API Gateway service that routes requests to Auth and User services, implements rate limiting, and provides centralized request handling.

Services Involved:

- API Gateway
- Auth Service (integration)
- User Service (integration)

Detailed Development Tasks:

1. Create API Gateway service structure
2. Implement reverse proxy routing to services
3. Add JWT validation middleware
4. Implement rate limiting logic
5. Create request logging functionality
6. Set up error handling and standardized responses
7. Implement health check aggregation
8. Create API versioning support
9. Add request timeout handling
10. Configure CORS policies
11. Set up connection pooling for downstream services

Dependencies:

- Phase 2 completed

- Auth and User services running
- Rate limiting strategy defined

Deliverables:

- Working API Gateway service
- Route configuration for Auth and User services
- Rate limiting implementation
- Centralized error handling
- API documentation with versioning
- Health check endpoint

Risks:

- Routing complexity with microservices
- Rate limiting accuracy at high load
- Performance overhead from gateway layer

Acceptance Criteria:

- All requests route to correct services
- Rate limiting enforces limits correctly
- Invalid tokens rejected at gateway
- Timeout errors handled gracefully
- CORS configured properly
- Health checks report service status

Easy Verification/Testing Checklist:

- Send request to /api/v1/auth/login and verify routing
- Make rapid requests and verify rate limit triggers
- Send request with invalid token and verify rejection
- Check gateway logs for request logging
- Test CORS with browser fetch from different origin
- Call /health endpoint and verify all services status

Expected Behavior: Requests proxied to correct services, rate limit returns 429 after threshold, invalid auth returns 401, timeout returns 504, CORS headers present in responses.

Validation Criteria:

- Response times within acceptable range
- Rate limiting accurate within 5% tolerance
- Zero routing errors to wrong services
- All HTTP status codes semantically correct

PHASE 4: File Upload Service & Storage Manager

Objective: Implement file upload functionality through Telegram, integrate with GCP Storage, and create storage management capabilities.

Services Involved:

- File Upload Service
- Storage Manager Service
- User Service (integration for quota check)

Detailed Development Tasks:

1. Create File Upload Service with MTProto integration
2. Implement file reception from Telegram
3. Create file validation logic (size, type)
4. Set up GCP Storage SDK integration
5. Implement multipart upload handling
6. Create Storage Manager Service
7. Implement bucket management operations
8. Create file metadata extraction
9. Set up MongoDB collections for files
10. Implement checksum calculation
11. Create upload progress tracking
12. Set up Kafka topics for file events
13. Implement storage quota enforcement

Dependencies:

- Phase 3 completed
- GCP project created and Storage bucket configured
- Telegram bot configured to receive files
- Storage quotas defined per user tier

Deliverables:

- Working File Upload Service
- Working Storage Manager Service
- GCP Storage integration
- MongoDB files collection
- Kafka topics: file.uploaded, file.upload.failed
- Upload progress tracking
- Storage quota enforcement

Risks:

- Large file upload timeouts
- GCP Storage authentication issues
- MTProto file handling complexity
- Quota calculation race conditions

Acceptance Criteria:

- Files upload successfully to GCP Storage
- File metadata saved to MongoDB
- Upload progress tracked accurately
- Storage quota enforced before upload
- Kafka event published on successful upload
- Failed uploads cleaned up properly
- Checksum verification works

Easy Verification/Testing Checklist:

- Send file via Telegram bot
- Check GCP Storage console for uploaded file
- Verify file document in MongoDB files collection
- Check user storage_used updated in users collection
- Verify file.uploaded event in Kafka topic
- Try uploading file exceeding quota and verify rejection
- Calculate file checksum manually and compare

Expected Behavior: File received via Telegram, validated, uploaded to GCP Storage, metadata saved to MongoDB, user storage_used incremented, Kafka event published, upload confirmed to user.

Validation Criteria:

- File accessible in GCP Storage with correct path
- File size in MongoDB matches actual file size
- Checksum matches file content
- Storage quota accurately reflects uploaded files
- No orphaned files in storage

PHASE 5: File Processing Service

Objective: Implement asynchronous file processing for image thumbnail generation, video transcoding, and metadata extraction.

Services Involved:

- File Processing Service
- File Upload Service (integration via Kafka)

Detailed Development Tasks:

1. Create File Processing Service structure
2. Implement Kafka consumer for file.uploaded events
3. Set up image processing with thumbnail generation
4. Implement multiple thumbnail sizes (small, medium, large)
5. Add EXIF metadata extraction for images
6. Implement video transcoding for multiple resolutions
7. Create video thumbnail and preview generation
8. Add video metadata extraction (duration, codec)
9. Implement processing job queue
10. Create retry logic for failed jobs
11. Set up processing status tracking in MongoDB
12. Configure Kafka topics for processing events
13. Implement processing timeout handling

Dependencies:

- Phase 4 completed
- Image processing library selected (sharp, imagemagick)
- Video processing library selected (ffmpeg)
- Processing worker resources allocated

Deliverables:

- Working File Processing Service
- Image thumbnail generation
- Video transcoding pipeline
- Metadata extraction
- Processing jobs tracking
- Kafka topics: file.processing.started, file.processing.completed, file.processing.failed
- Retry mechanism for failed jobs

Risks:

- Video transcoding resource intensive
- Processing timeouts for large files
- Library compatibility issues
- Memory exhaustion with concurrent jobs

Acceptance Criteria:

- Images generate three thumbnail sizes
- Videos transcoded to 480p, 720p, 1080p
- Metadata extracted accurately
- Processing status updated in real-time
- Failed jobs retry automatically
- Processing timeout handled gracefully
- CPU/Memory usage within limits

Easy Verification/Testing Checklist:

- Upload image and verify thumbnails in GCP Storage
- Upload video and verify transcoded versions created
- Check file document for processed_variants array
- Monitor processing service logs for job progress
- Upload corrupted file and verify failure handling
- Check Kafka topic for processing.completed event
- Verify metadata extracted correctly (check EXIF data)

Expected Behavior: File.uploaded event consumed, processing job started, thumbnails/transcodes generated, variants uploaded to storage, metadata extracted, file document updated, processing.completed event published.

Validation Criteria:

- All thumbnail sizes generated correctly
- Transcoded videos playable and correct resolution
- Extracted metadata matches file properties
- Processing completes within expected timeframe
- Failed jobs marked correctly in database

PHASE 6: File Metadata Service & Search

Objective: Create metadata management service with search functionality, file listing, filtering, and organization capabilities.

Services Involved:

- File Metadata Service
- File Processing Service (integration via Kafka)

Detailed Development Tasks:

1. Create File Metadata Service structure
2. Implement file search with text indexing
3. Create pagination for file listings
4. Implement filtering by type, date, size
5. Add sorting functionality
6. Create tagging system
7. Implement folder/collection organization
8. Set up MongoDB text indexes for search
9. Create aggregation pipelines for statistics
10. Implement Kafka consumer for metadata updates
11. Add metadata validation and sanitization
12. Create file relationship tracking
13. Implement search result ranking

Dependencies:

- Phase 5 completed
- Search requirements defined
- Text indexing strategy determined

Deliverables:

- Working File Metadata Service
- Search functionality with ranking
- File listing with pagination
- Filtering and sorting
- Tagging system
- Folder organization
- MongoDB text indexes
- Aggregation queries for statistics

Risks:

- Search performance with large datasets
- Text index size and memory usage
- Complex aggregation query performance
- Search relevance accuracy

Acceptance Criteria:

- Search returns relevant results
- Pagination works correctly
- Filters apply accurately
- Tags can be added and searched
- Folders organize files logically
- Search response time under 500ms
- Statistics queries return accurate counts

Easy Verification/Testing Checklist:

- Search for filename and verify results
- Test pagination by requesting different pages
- Filter by image type and verify only images returned
- Add tags to file and search by tag
- Create folder and move files into it
- Check MongoDB indexes with explain plan
- Query file statistics and verify counts

Expected Behavior: Search queries return matching files ranked by relevance, pagination limits results per page, filters narrow results correctly, tags associated with files, folders contain assigned files.

Validation Criteria:

- Search recall and precision acceptable
- Pagination offset calculations correct
- Filter logic matches specifications
- Tag operations atomic and consistent
- Folder hierarchy maintained correctly

PHASE 7: Download Service

Objective: Implement secure file download functionality with signed URLs, permission validation, and download tracking.

Services Involved:

- Download Service
- Storage Manager Service (integration)
- File Metadata Service (integration)

Detailed Development Tasks:

1. Create Download Service structure
2. Implement permission validation logic
3. Generate GCP signed URLs for downloads
4. Configure URL expiration times
5. Implement range request support for resume
6. Create download tracking in MongoDB
7. Add rate limiting for downloads
8. Implement bandwidth throttling if needed
9. Set up Kafka topics for download events
10. Create download statistics
11. Add download link sharing functionality
12. Implement download quota enforcement
13. Create temporary download tokens

Dependencies:

- Phase 6 completed
- GCP Storage signing credentials configured
- Download policies defined

Deliverables:

- Working Download Service
- Signed URL generation
- Permission validation
- Range request support
- Download tracking
- Kafka topics: download.requested, download.completed
- Download statistics
- Rate limiting

Risks:

- Signed URL expiration timing issues
- Permission check bypassing
- Large file download timeouts
- Rate limiting accuracy

Acceptance Criteria:

- Signed URLs generated with correct expiration
- Permission checks prevent unauthorized access
- Range requests work for resume capability
- Download activity tracked in MongoDB
- Rate limiting enforces download limits
- Kafka events published correctly
- Statistics accurate

Easy Verification/Testing Checklist:

- Request download URL and verify signature
- Test URL after expiration time and verify failure
- Attempt download without permission and verify rejection
- Test range request with partial download
- Check downloads collection for tracking entry

- Make rapid download requests and verify rate limit
- Verify download.completed event in Kafka

Expected Behavior: Download request validated, signed URL generated, URL returned to user, file downloaded successfully, download tracked in database, Kafka event published.

Validation Criteria:

- Signed URLs work before expiration
- Expired URLs return 403 Forbidden
- Unauthorized requests return 403
- Range requests return correct byte ranges
- Download counts accurate

PHASE 8: Video Streaming Service

Objective: Implement adaptive video streaming with HLS protocol, multiple quality levels, and playback tracking.

Services Involved:

- Video Streaming Service
- File Processing Service (integration for video variants)
- Storage Manager Service (integration)

Detailed Development Tasks:

1. Create Video Streaming Service structure
2. Implement HLS playlist generation
3. Create adaptive bitrate streaming logic
4. Set up video segment serving
5. Implement quality level switching
6. Create playback session tracking
7. Add seek functionality
8. Implement resume playback across devices
9. Create buffering optimization
10. Set up Kafka topics for streaming events
11. Add playback analytics
12. Implement bandwidth detection
13. Create video player API endpoints

Dependencies:

- Phase 7 completed
- Video transcoding from Phase 5 working
- HLS protocol understanding
- Video player client-side integration plan

Deliverables:

- Working Video Streaming Service
- HLS playlist generation
- Adaptive bitrate streaming
- Playback session management
- Seek and resume functionality
- Kafka topics: video.streaming.started, video.streaming.stopped
- Playback analytics
- Quality switching logic

Risks:

- HLS implementation complexity
- Buffering issues with slow connections
- Quality switching delays
- Playback session sync across devices

Acceptance Criteria:

- HLS playlists generated correctly
- Video streams without buffering on good connection
- Quality switches based on bandwidth
- Playback position saved and restored
- Seek operations fast and accurate
- Analytics track viewing patterns
- Multiple devices sync playback position

Easy Verification/Testing Checklist:

- Request HLS playlist and verify m3u8 format
- Play video in HLS player and verify streaming
- Throttle network and observe quality downgrade
- Pause video, reload page, verify resume from same position
- Seek to middle of video and verify quick load
- Check playback_sessions collection for tracking
- Monitor Kafka for streaming.started event

Expected Behavior: Playlist requested, HLS manifest returned, video segments stream sequentially, quality adapts to bandwidth, playback position saved every 10 seconds, resume works across sessions.

Validation Criteria:

- HLS playlist follows specification
- Video segments stream without gaps
- Quality switches within 5 seconds of bandwidth change
- Playback position accuracy within 2 seconds
- Seek operations complete within 1 second

PHASE 9: Preview Service

Objective: Create preview and thumbnail serving service with caching, lazy loading support, and responsive image delivery.

Services Involved:

- Preview Service
- File Processing Service (integration for generated previews)

Detailed Development Tasks:

1. Create Preview Service structure
2. Implement thumbnail serving endpoints
3. Create preview caching logic
4. Add lazy loading support with placeholders
5. Implement responsive image serving
6. Create on-demand preview generation
7. Add image optimization for web delivery
8. Implement cache eviction policies
9. Set up CDN integration for preview delivery
10. Create preview card generation for sharing
11. Add watermarking capability if needed
12. Implement preview access logging

Dependencies:

- Phase 8 completed
- Thumbnail generation from Phase 5 working
- CDN configuration (optional)

Deliverables:

- Working Preview Service
- Thumbnail serving with multiple sizes
- Preview caching implementation
- Lazy loading support
- Responsive image delivery
- On-demand generation
- Cache eviction logic
- CDN integration

Risks:

- Cache memory usage growth
- On-demand generation latency
- CDN caching synchronization
- Concurrent generation of same preview

Acceptance Criteria:

- Thumbnails served quickly from cache
- Cache misses trigger on-demand generation
- Lazy loading provides low-quality placeholder
- Responsive images match requested size
- Cache eviction prevents memory exhaustion
- CDN serves previews when configured
- Access logging tracks popular previews

Easy Verification/Testing Checklist:

- Request thumbnail and verify quick response
- Request non-existent preview and verify generation
- Test lazy loading endpoint for placeholder
- Request different sizes and verify responsive delivery
- Check preview_cache collection for entries
- Monitor cache memory usage
- Verify CDN headers if configured

Expected Behavior: Thumbnail request checks cache, returns cached image if exists, generates on-demand if missing, caches for future requests, serves appropriate size based on request.

Validation Criteria:

- Cache hit rate above 80% for popular files
- On-demand generation completes within 3 seconds
- Correct image sizes returned for responsive requests
- Cache eviction maintains memory under limit
- No duplicate generation for concurrent requests

PHASE 10: Notification Service

Objective: Implement multi-channel notification system with Telegram messages, in-app notifications, and notification preferences.

Services Involved:

- Notification Service
- All other services (integration via Kafka for triggering notifications)

Detailed Development Tasks:

1. Create Notification Service structure
2. Implement Telegram notification delivery via MTProto
3. Create notification template system
4. Implement notification preferences management
5. Add notification queuing and batching
6. Create retry logic for failed deliveries
7. Set up notification channels (Telegram, in-app)
8. Implement notification read status tracking
9. Create notification history
10. Add priority-based delivery
11. Implement rate limiting to prevent spam
12. Set up Kafka consumers for all notification triggers
13. Create notification aggregation for similar events

Dependencies:

- Phase 9 completed
- Telegram bot configured for sending messages
- Notification templates designed

Deliverables:

- Working Notification Service
- Telegram notification delivery
- In-app notification support
- Notification preferences
- Template system
- Retry mechanism
- Kafka topics: notification.sent, notification.delivered
- Notification history
- Read status tracking

Risks:

- Telegram API rate limits
- Notification spam overwhelming users
- Delivery failures not properly retried
- Template rendering errors

Acceptance Criteria:

- Notifications delivered via Telegram successfully
- Users can configure notification preferences
- Templates render correctly with data
- Failed deliveries retried appropriately
- Read status tracked accurately
- Rate limiting prevents spam
- History shows all past notifications

Easy Verification/Testing Checklist:

- Upload file and verify Telegram notification received
- Check notification preferences in database
- Test notification with template variables
- Simulate Telegram API failure and verify retry
- Mark notification as read and verify status update
- Check notifications collection for history
- Verify multiple similar events batched together

Expected Behavior: Kafka event consumed, notification template rendered with data, notification queued, delivered via Telegram, delivery status updated, read receipts tracked.

Validation Criteria:

- Telegram notifications arrive within 5 seconds
- Template variables replaced correctly
- Retry attempts follow exponential backoff
- Read status updates instantly
- Notification history complete and searchable

PHASE 11: WebSocket Realtime Service

Objective: Implement WebSocket server for real-time updates, connection management, and event broadcasting.

Services Involved:

- WebSocket Realtime Service
- API Gateway (integration for WebSocket upgrade)
- All other services (integration via Kafka for events)

Detailed Development Tasks:

1. Create WebSocket Realtime Service structure
2. Implement WebSocket server with connection handling
3. Create authentication for WebSocket connections
4. Implement heartbeat/ping-pong mechanism
5. Set up Kafka consumers for all real-time events
6. Create event routing to appropriate connections
7. Implement room-based broadcasting
8. Add connection registry in MongoDB
9. Create connection scaling across multiple instances
10. Implement reconnection handling
11. Add message queuing for offline clients
12. Create presence tracking
13. Implement connection statistics

Dependencies:

- Phase 10 completed
- WebSocket protocol understanding
- Load balancing strategy for WebSocket connections

Deliverables:

- Working WebSocket Realtime Service
- Connection authentication
- Event broadcasting
- Room-based messaging
- Connection registry
- Reconnection handling
- Presence tracking
- Connection statistics

Risks:

- Connection scaling complexity
- Message delivery guarantees
- Stale connections consuming resources
- Cross-instance message routing

Acceptance Criteria:

- WebSocket connections establish successfully
- Authentication required before accepting connection
- Events broadcast to correct connections
- Heartbeat keeps connections alive
- Reconnection restores missed events
- Multiple server instances route correctly
- Presence tracking shows online users

Easy Verification/Testing Checklist:

- Connect to WebSocket endpoint from browser
- Verify authentication challenge
- Upload file and verify real-time update received
- Disconnect and reconnect, verify missed events delivered
- Check websocket_connections collection
- Connect from multiple tabs and verify each receives events
- Monitor connection count in service metrics

Expected Behavior: WebSocket connection upgraded after authentication, heartbeat sent every 30 seconds, Kafka events consumed and pushed to relevant connections, disconnected clients removed from registry.

Validation Criteria:

- Connection establishment within 1 second
- Events delivered within 100ms of Kafka consumption
- Heartbeat prevents premature disconnections
- Reconnection delivers all missed events
- Zero memory leaks from stale connections

PHASE 12: Activity Log Service

Objective: Create centralized logging service that captures all system activities for audit trails, analytics, and debugging.

Services Involved:

- Activity Log Service
- All other services (integration via Kafka for logging events)

Detailed Development Tasks:

1. Create Activity Log Service structure
2. Set up Kafka consumers for all topics

3. Implement high-throughput log ingestion
4. Create MongoDB time-series collection for logs
5. Implement log search functionality
6. Add log filtering and aggregation
7. Create retention policies with automatic cleanup
8. Implement log level configuration
9. Add analytics query capabilities
10. Create log export functionality
11. Implement structured logging format
12. Set up critical event alerting
13. Create log visualization data preparation

Dependencies:

- Phase 11 completed
- All Kafka topics established
- Log retention policies defined

Deliverables:

- Working Activity Log Service
- Comprehensive event logging
- Log search and filtering
- Analytics queries
- Retention policies
- Time-series optimized storage
- Critical event alerting
- Export functionality

Risks:

- High log volume overwhelming system
- Storage growth exceeding limits
- Query performance on large datasets
- Missing events due to processing lag

Acceptance Criteria:

- All events logged without loss
- Log ingestion keeps pace with event rate
- Search returns results within 2 seconds
- Retention policy cleans old logs automatically
- Analytics queries provide accurate insights
- Critical events trigger alerts
- Export generates complete log files

Easy Verification/Testing Checklist:

- Perform various actions and verify logs captured
- Search logs by user_id and verify results
- Filter logs by date range
- Check log count before and after retention cleanup
- Query analytics for upload statistics
- Verify critical event (like failed login) triggers alert
- Export logs and verify completeness

Expected Behavior: Kafka events consumed continuously, log documents inserted into MongoDB, indexes updated for search, retention policy runs daily to remove old logs, queries execute efficiently.

Validation Criteria:

- Zero event loss during ingestion
- Log search returns relevant results
- Analytics accurate within 1% margin
- Storage growth matches expected rate
- Query performance acceptable for dataset size

PHASE 13: Admin Service & Dashboard

Objective: Create administrative service with system management, user administration, analytics, and monitoring capabilities.

Services Involved:

- Admin Service
- All other services (integration for management operations)
- WebSocket Realtime Service (integration for live dashboard)

Detailed Development Tasks:

1. Create Admin Service structure
2. Implement user management endpoints
3. Create system statistics aggregation
4. Add storage management interface
5. Implement file moderation capabilities
6. Create system health monitoring
7. Add configuration management
8. Implement bulk operations

9. Create administrative reporting
10. Add role-based admin access control
11. Implement audit trail for admin actions
12. Create real-time dashboard data feed
13. Add system alert management

Dependencies:

- Phase 12 completed
- Admin access control policies defined
- Dashboard requirements specified

Deliverables:

- Working Admin Service
- User management endpoints
- System statistics API
- Storage management
- Content moderation tools
- Health monitoring
- Configuration management
- Administrative reports
- Audit trail

Risks:

- Admin privilege escalation vulnerabilities
- Bulk operations affecting system performance
- Configuration changes breaking system
- Dashboard data aggregation overhead

Acceptance Criteria:

- Admins can manage user accounts
- System statistics accurate and real-time
- Storage management operations work correctly
- Content can be moderated effectively
- Health monitoring shows service status
- Configuration changes applied safely
- Bulk operations complete successfully
- Admin actions audited completely

Easy Verification/Testing Checklist:

- Login as admin and access admin endpoints
- View user list and modify user account
- Check system statistics dashboard
- Perform storage cleanup operation
- Moderate a file (delete/flag)
- View service health status
- Change system configuration
- Check admin_actions collection for audit trail

Expected Behavior: Admin authenticated with elevated privileges, management operations executed with proper authorization, statistics aggregated from all services, changes logged to audit trail, dashboard updated in real-time.

Validation Criteria:

- Only authorized admins can access endpoints
- User modifications persist correctly
- Statistics match actual system state
- Bulk operations atomic and safe
- Configuration changes validated before applying
- Audit trail complete and tamper-proof

PHASE 14: Integration, Testing & Bug Fixes

Objective: Integrate all services, perform comprehensive end-to-end testing, fix discovered bugs, and ensure system stability.

Services Involved:

- All 14 microservices

Detailed Development Tasks:

1. Verify all service-to-service integrations
2. Test complete user workflows end-to-end
3. Perform load testing on all services
4. Test Kafka message flow completeness
5. Verify WebSocket event delivery
6. Test error handling and recovery
7. Validate database consistency
8. Test concurrent operations
9. Verify security measures
10. Check rate limiting effectiveness
11. Test failover scenarios
12. Validate monitoring and logging

13. Fix all identified bugs
14. Optimize performance bottlenecks
15. Create comprehensive test documentation

Dependencies:

- Phase 13 completed
- All services deployable
- Testing environment configured

Deliverables:

- Complete integration test suite
- End-to-end test scenarios
- Load test results
- Bug fix documentation
- Performance optimization report
- Security audit results
- Test coverage report
- Known issues log

Risks:

- Integration issues between services
- Performance problems under load
- Security vulnerabilities discovered
- Data consistency issues
- Time required for thorough testing

Acceptance Criteria:

- All user workflows complete successfully
- Load tests meet performance targets
- No critical or high-severity bugs
- Error handling works in all scenarios
- Data remains consistent under concurrent load
- Security measures effective
- Monitoring captures all important metrics
- Documentation complete

Easy Verification/Testing Checklist:

- Execute complete user journey: register → upload → view → stream → download
- Run load test with 100 concurrent users
- Verify all Kafka topics have messages flowing
- Test WebSocket disconnection and reconnection
- Simulate service failure and verify recovery
- Check database for orphaned records
- Review logs for errors and warnings
- Test rate limiting across all endpoints
- Verify admin operations work correctly
- Check all API endpoints return proper status codes

Expected Behavior: All workflows complete without errors, system handles load gracefully, errors handled with proper messages, Kafka guarantees message delivery, WebSocket reconnects automatically, services recover from failures, data consistent across all operations.

Validation Criteria:

- Zero critical bugs in production path
- Load test sustains target concurrent users
- Response times within SLA
- Error rate below 0.1%
- Data integrity maintained
- Security tests pass
- Test coverage above 70%

PHASE 15: GCP Deployment & Production Launch

Objective: Deploy all services to Google Cloud Platform, configure production environment, set up monitoring, and launch the system.

Services Involved:

- All 14 microservices
- Infrastructure components

Detailed Development Tasks:

1. Create GCP project and configure billing
2. Set up Cloud Run services for each microservice
3. Configure GCP Storage buckets with proper permissions
4. Set up MongoDB Atlas production cluster
5. Deploy Kafka to GKE or configure Cloud Pub/Sub
6. Configure Cloud Load Balancer as API Gateway
7. Set up Cloud CDN for static assets
8. Configure IAM roles and service accounts
9. Set up Cloud Build for CI/CD pipeline
10. Configure environment variables and secrets

11. Set up Cloud Monitoring and Logging
12. Configure alerting policies
13. Set up auto-scaling policies
14. Create backup and disaster recovery plan
15. Perform production smoke tests
16. Document deployment procedures
17. Create runbook for operations
18. Launch to production

Dependencies:

- Phase 14 completed
- GCP account with appropriate quotas
- Production domain and SSL certificates
- Monitoring tools configured

Deliverables:

- All services deployed to GCP
- Production environment configured
- CI/CD pipeline operational
- Monitoring and alerting active
- Backup strategy implemented
- Deployment documentation
- Operations runbook
- Production launch checklist

Risks:

- GCP quota limitations
- Configuration errors in production
- Performance differences from development
- Cost overruns
- Deployment rollback requirements
- DNS propagation delays

Acceptance Criteria:

- All services running on GCP without errors
- Auto-scaling responds to load changes
- Monitoring captures all metrics
- Alerts trigger for critical issues
- CI/CD pipeline deploys successfully
- Backup and restore tested and working
- Security hardened for production
- Cost tracking configured
- Documentation complete
- System publicly accessible

Easy Verification/Testing Checklist:

- Access production URL and verify landing page
- Perform complete user workflow in production
- Upload file and verify storage in GCP bucket
- Check Cloud Monitoring for service metrics
- Trigger alert condition and verify notification
- Deploy code change via CI/CD pipeline
- Check Cloud Logging for application logs
- Verify auto-scaling by simulating load
- Test backup restore procedure
- Review IAM permissions for security
- Verify SSL certificate valid
- Check CDN serving static assets
- Monitor costs in billing dashboard
- Test disaster recovery procedure

Expected Behavior: Services deploy successfully to Cloud Run, respond to HTTP requests, scale automatically under load, logs appear in Cloud Logging, metrics visible in Cloud Monitoring, alerts fire when conditions met, CI/CD pipeline deploys on commit.

Validation Criteria:

- All health checks green
- Response times meet SLA
- No deployment errors
- Auto-scaling maintains performance
- Monitoring complete and accurate
- Alerts tested and working
- Backup completes successfully
- Cost within budget projections
- Security scan passes
- Documentation accurate and complete

DATABASE DESIGN

MongoDB Collections

1. users

Stores user account information, profile data, and storage quota details.

Schema Explanation:

- **telegram_id**: Unique Telegram user identifier (unique index)
- **username**: Telegram username for display
- **email**: User email if provided (optional)
- **profile_picture**: URL to profile picture
- **storage_quota**: Allocated storage in bytes
- **storage_used**: Current storage usage in bytes
- **preferences**: Embedded document with user settings (theme, notifications, language)
- **created_at**: Account creation timestamp
- **updated_at**: Last profile update timestamp
- **is_active**: Account status flag
- **last_login_at**: Last login timestamp

Indexing Strategy:

- Unique index on telegram_id for authentication lookups
- Index on email for email-based queries
- Compound index on (storage_used, storage_quota) for quota enforcement
- Index on created_at for analytics

---

2. auth_sessions

Manages authentication sessions and refresh tokens.

Schema Explanation:

- **user_id**: Reference to users collection
- **refresh_token_hash**: Hashed refresh token for security
- **access_token_hash**: Hashed access token (optional storage)
- **issued_at**: Token issuance timestamp
- **expires_at**: Token expiration timestamp
- **device_info**: Embedded document with device details (user_agent, ip_address, device_id)
- **is_revoked**: Revocation status
- **revoked_at**: Revocation timestamp if applicable

Indexing Strategy:

- Index on user_id for user session queries
- Unique index on refresh_token_hash for validation
- TTL index on expires_at for automatic cleanup
- Index on (user_id, is_revoked) for active sessions

---

3. files

Central collection for all file metadata and processing status.

Schema Explanation:

- **file_id**: Unique file identifier (UUID)
- **user_id**: Owner reference
- **filename**: Original file name
- **size**: File size in bytes
- **mime_type**: File MIME type
- **storage_path**: GCP Storage path
- **checksum**: File hash for integrity
- **upload_status**: Status enum (uploading, completed, failed)
- **processing_status**: Processing state (pending, processing, completed, failed)
- **processed_variants**: Array of processed versions with URLs and metadata
- **metadata**: Embedded document with extracted metadata (dimensions, duration, EXIF, etc.)
- **tags**: Array of tag strings
- **folder_id**: Reference to folders collection (optional)
- **is_public**: Public sharing status
- **upload_started_at**: Upload start time
- **upload_completed_at**: Upload completion time
- **created_at**: Record creation timestamp
- **updated_at**: Last modification timestamp

Indexing Strategy:

- Unique index on file_id
- Compound index on (user_id, created_at) for listing user files
- Compound index on (user_id, mime_type) for filtering by type
- Text index on filename for search
- Index on tags for tag-based queries
- Index on folder_id for folder listings
- Index on processing_status for queue management

4. processing_jobs

Tracks file processing tasks and their status.

Schema Explanation:

- **job_id**: Unique job identifier
- **file_id**: Reference to files collection
- **processing_type**: Type enum (thumbnail, transcode, metadata_extraction)
- **status**: Status enum (queued, processing, completed, failed)
- **priority**: Priority number for queue ordering
- **retry_count**: Number of retry attempts
- **max_retries**: Maximum retry limit
- **started_at**: Processing start time
- **completed_at**: Processing completion time
- **error_message**: Error details if failed
- **worker_id**: Processing worker identifier

Indexing Strategy:

- Index on file_id for job lookups
- Compound index on (status, priority) for queue processing
- Index on worker_id for worker monitoring
- TTL index for cleanup of old completed jobs

---

5. storage_objects

Maps files to cloud storage locations.

Schema Explanation:

- **file_id**: Reference to files collection
- **bucket_name**: GCP Storage bucket
- **object_key**: Object path in bucket
- **storage_class**: Storage class (standard, nearline, coldline)
- **region**: Storage region
- **size**: Object size
- **created_at**: Storage creation timestamp
- **last_accessed_at**: Last access time for lifecycle management

Indexing Strategy:

- Unique compound index on (bucket_name, object_key)
- Index on file_id
- Index on last_accessed_at for archival policies

---

6. downloads

Tracks file download history and analytics.

Schema Explanation:

- **download_id**: Unique download identifier
- **file_id**: Downloaded file reference
- **user_id**: User who downloaded
- **downloaded_at**: Download timestamp
- **ip_address**: Request IP address
- **user_agent**: Request user agent
- **download_size**: Bytes downloaded (for partial downloads)
- **is_complete**: Completion status

Indexing Strategy:

- Index on file_id for file download statistics
- Index on user_id for user download history
- Index on downloaded_at for time-based analytics
- Compound index on (file_id, downloaded_at) for file analytics

---

7. playback_sessions

Manages video streaming sessions and progress.

Schema Explanation:

- **session_id**: Unique session identifier
- **file_id**: Video file reference
- **user_id**: Viewing user reference
- **current_position**: Playback position in seconds
- **duration**: Total video duration
- **quality_level**: Current quality setting

- **started_at**: Session start time
- **last_updated_at**: Last progress update time
- **device_info**: Device details
- **is_active**: Active session flag

Indexing Strategy:

- Unique compound index on (file_id, user_id) for single active session
- Index on user_id for user's viewing history
- TTL index on last_updated_at for stale session cleanup

---

8. notifications

Stores all system notifications.

Schema Explanation:

- **notification_id**: Unique notification identifier
- **user_id**: Recipient user reference
- **type**: Notification type enum (upload_complete, processing_complete, quota_warning, etc.)
- **title**: Notification title
- **message**: Notification content
- **priority**: Priority level (low, medium, high, critical)
- **status**: Delivery status (pending, sent, delivered, failed)
- **channels**: Array of delivery channels (telegram, in_app, email)
- **data**: Additional data payload as embedded document
- **created_at**: Notification creation time
- **sent_at**: Delivery time
- **read_at**: Read timestamp

Indexing Strategy:

- Compound index on (user_id, read_at) for unread notifications
- Index on status for delivery queue
- Index on created_at for notification history
- TTL index for old notification cleanup

---

9. notification_preferences

User notification preferences and settings.

Schema Explanation:

- **user_id**: User reference (unique)
- **channels_enabled**: Array of enabled channels
- **notification_types**: Object mapping notification types to enabled status
- **quiet_hours**: Embedded document with start and end times
- **updated_at**: Last preference update

Indexing Strategy:

- Unique index on user_id

---

10. activity_logs

Comprehensive system activity logs.

Schema Explanation:

- **log_id**: Unique log identifier
- **timestamp**: Event timestamp
- **user_id**: Acting user reference (null for system events)
- **service_name**: Originating service
- **action_type**: Action category (upload, download, auth, admin, etc.)
- **resource_id**: Affected resource identifier
- **resource_type**: Resource type (file, user, session, etc.)
- **details**: Embedded document with additional context
- **ip_address**: Request IP if applicable
- **user_agent**: Request user agent if applicable
- **level**: Log level (info, warning, error, critical)

Indexing Strategy:

- Index on timestamp for time-based queries
- Index on user_id for user activity
- Compound index on (action_type, timestamp) for analytics
- Index on service_name for service-specific logs
- TTL index on timestamp for automatic cleanup per retention policy

---

11. websocket_connections

Active WebSocket connection registry.

Schema Explanation:

- **connection_id**: Unique connection identifier
- **user_id**: Connected user reference
- **server_instance**: Server instance identifier for routing
- **connected_at**: Connection establishment time
- **last_ping**: Last heartbeat timestamp
- **device_info**: Device and client information
- **subscribed_rooms**: Array of room identifiers

Indexing Strategy:

- Index on user_id for user connection lookup
- Index on server_instance for instance-specific queries
- TTL index on last_ping for stale connection cleanup

---

12. upload_sessions

Multipart upload session tracking.

Schema Explanation:

- **session_id**: Unique session identifier
- **file_id**: Target file reference
- **user_id**: Uploading user
- **total_parts**: Expected number of parts
- **parts_uploaded**: Array of uploaded part numbers
- **upload_urls**: Embedded document mapping part numbers to signed URLs
- **created_at**: Session creation time
- **expires_at**: Session expiration time

Indexing Strategy:

- Unique index on session_id
- Index on file_id
- TTL index on expires_at for automatic cleanup

---

13. tags

Tag definitions and metadata.

Schema Explanation:

- **tag_id**: Unique tag identifier
- **user_id**: Tag owner (null for system tags)
- **tag_name**: Tag text (lowercase)
- **usage_count**: Number of files with this tag
- **created_at**: Tag creation time

Indexing Strategy:

- Unique compound index on (user_id, tag_name)
- Index on tag_name for tag search

---

14. folders

Folder structure for file organization.

Schema Explanation:

- **folder_id**: Unique folder identifier
- **user_id**: Owner reference
- **folder_name**: Folder display name
- **parent_folder_id**: Parent folder reference (null for root)
- **created_at**: Creation timestamp

Indexing Strategy:

- Compound index on (user_id, parent_folder_id) for folder hierarchy
- Index on folder_id

---

15. api_keys

API key management for programmatic access.

Schema Explanation:

- **api_key_hash**: Hashed API key
- **user_id**: Key owner reference
- **name**: Key description/name
- **permissions**: Array of allowed operations
- **created_at**: Key creation time
- **last_used_at**: Last usage timestamp
- **expires_at**: Expiration timestamp (optional)
- **is_active**: Active status

Indexing Strategy:

- Unique index on api_key_hash
- Index on user_id for user's keys
- Index on last_used_at for inactive key detection

---

16. rate_limits

Rate limiting counters.

Schema Explanation:

- **key**: Rate limit key (user_id, IP, or API key)
- **endpoint**: API endpoint pattern
- **request_count**: Request count in current window
- **window_start**: Window start timestamp
- **window_end**: Window end timestamp

Indexing Strategy:

- Unique compound index on (key, endpoint)
- TTL index on window_end for automatic cleanup

---

17. admin_actions

Audit trail for administrative operations.

Schema Explanation:

- **action_id**: Unique action identifier
- **admin_id**: Admin user reference
- **action_type**: Type of admin action
- **target_type**: Type of affected resource
- **target_id**: Affected resource identifier
- **action_details**: Embedded document with details
- **ip_address**: Admin's IP address
- **timestamp**: Action timestamp

Indexing Strategy:

- Index on admin_id for admin activity history
- Index on timestamp for chronological audit
- Index on target_id for resource history

---

18. preview_cache

Generated preview metadata and tracking.

Schema Explanation:

- **preview_id**: Unique preview identifier
- **file_id**: Source file reference
- **preview_type**: Type (thumbnail, preview_card, etc.)
- **size**: Size variant (small, medium, large)
- **format**: Output format (jpeg, webp, etc.)
- **storage_path**: Preview location in storage
- **generated_at**: Generation timestamp
- **access_count**: Access frequency counter
- **last_accessed_at**: Last access time

Indexing Strategy:

- Compound index on (file_id, preview_type, size) for lookup
- Index on last_accessed_at for cache eviction
- Index on access_count for popularity tracking

Relationship Explanation

One-to-Many Relationships:

- users → files: One user owns many files
- users → auth_sessions: One user has multiple sessions
- files → processing_jobs: One file has multiple processing jobs
- files → downloads: One file tracked in multiple download records
- users → notifications: One user receives many notifications
- users → folders: One user creates many folders

Many-to-Many Relationships:

- files ↔ tags: Implemented with tags array in files collection
- folders ↔ files: Files reference folder_id

Referenced Documents: Most relationships use referenced documents (storing IDs) rather than embedded documents to maintain flexibility and allow independent updates. Embedded documents used only for tightly coupled data that doesn't need independent queries (preferences, device_info, metadata).

KAFKA DESIGN

Kafka Topics

1. user.registered

Published when new user account created.

Publishers: User Service

Consumers:

- Auth Service (create initial auth credentials)
- Notification Service (send welcome notification)
- Activity Log Service (log registration)

Event Schema (Conceptual):

- event_id
- timestamp
- user_id
- telegram_id
- username
- registration_source

Event Lifecycle: User Service validates registration → Creates user in MongoDB → Publishes event → Auth Service sets up credentials → Notification Service sends welcome message → Activity Log Service records event

---

2. user.logged_in

Published on successful user authentication.

Publishers: Auth Service

Consumers:

- Activity Log Service (log login)
- WebSocket Realtime Service (trigger presence update)
- User Service (update last_login_at)

Event Schema:

- event_id
- timestamp
- user_id
- session_id
- device_info
- ip_address

Event Lifecycle: User authenticates → Auth Service validates → JWT issued → Event published → Activity logged → Presence updated → Last login timestamp updated

---

3. user.logged_out

Published when user logs out or session revoked.

Publishers: Auth Service

Consumers:

- Activity Log Service (log logout)
- WebSocket Realtime Service (disconnect user sessions)

Event Schema:

- event_id
- timestamp
- user_id
- session_id
- logout_reason

Event Lifecycle: Logout requested → Auth Service revokes token → Event published → WebSocket connections closed → Activity logged

---

4. file.uploaded

Published when file successfully uploaded to storage.

Publishers: File Upload Service

Consumers:

- File Processing Service (queue processing job)
- User Service (update storage_used)
- Notification Service (send upload confirmation)
- File Metadata Service (index file)
- Activity Log Service (log upload)
- WebSocket Realtime Service (notify user)

Event Schema:

- event_id
- timestamp
- file_id
- user_id
- filename
- size
- mime_type
- storage_path
- checksum

Event Lifecycle: File uploaded → Storage confirmed → Event published → Processing queued → Storage quota updated → User notified → File indexed → Activity logged

---

5. file.upload.failed

Published when file upload fails.

Publishers: File Upload Service

Consumers:

- Notification Service (notify user of failure)
- Activity Log Service (log failure)

Event Schema:

- event_id
- timestamp
- user_id
- filename
- error_reason
- error_details

Event Lifecycle: Upload attempt → Validation or storage failure → Event published → User notified → Failure logged

---

6. file.processing.started

Published when file processing begins.

Publishers: File Processing Service

Consumers:

- File Metadata Service (update processing status)
- WebSocket Realtime Service (push status to user)
- Activity Log Service (log processing start)

Event Schema:

- event_id
- timestamp
- file_id
- processing_type
- job_id

Event Lifecycle: Processing job dequeued → Processing begins → Event published → Status updated in database → User sees "Processing..." → Activity logged

---

7. file.processing.completed

Published when file processing successfully completes.

Publishers: File Processing Service

Consumers:

- File Metadata Service (update metadata with results)
- Notification Service (notify user of completion)
- Video Streaming Service (prepare streaming if video)
- Preview Service (cache generated previews)
- WebSocket Realtime Service (push completion to user)
- Activity Log Service (log completion)

Event Schema:

- event_id
- timestamp
- file_id
- processing_type
- processed_variants (array of generated versions)
- extracted_metadata

Event Lifecycle: Processing completes → Variants uploaded → Event published → Metadata updated → User notified → Streaming prepared → Previews cached → Activity logged

---

8. file.processing.failed

Published when file processing fails.

Publishers: File Processing Service

Consumers:

- File Metadata Service (update status to failed)
- Notification Service (notify user of failure)
- Activity Log Service (log failure)

Event Schema:

- event_id
- timestamp
- file_id
- processing_type
- error_reason
- retry_count

Event Lifecycle: Processing fails → Retry logic exhausted → Event published → Status updated → User notified → Failure logged

---

9. file.deleted

Published when file deleted by user or system.

Publishers: File Metadata Service, Admin Service

Consumers:

- Storage Manager Service (delete from cloud storage)
- User Service (decrease storage_used)
- Activity Log Service (log deletion)
- WebSocket Realtime Service (update user's file list)

Event Schema:

- event_id
- timestamp
- file_id
- user_id
- deleted_by (user_id or "system")
- deletion_reason

Event Lifecycle: Delete requested → Database record soft deleted → Event published → Storage object deleted → Quota updated → Activity logged → UI updated

---

10. download.requested

Published when user requests file download.

Publishers: Download Service

Consumers:

- Activity Log Service (log download request)

- Notification Service (notify file owner if shared file)

Event Schema:

- event_id
- timestamp
- file_id
- user_id
- download_token

Event Lifecycle: Download requested → Permissions validated → Signed URL generated → Event published → Request logged → Owner notified if applicable

---

11. download.completed

Published when download successfully completes.

Publishers: Download Service (based on callback or tracking)

Consumers:

- Activity Log Service (log completion)
- File Metadata Service (update download count)

Event Schema:

- event_id
- timestamp
- file_id
- user_id
- download_size

Event Lifecycle: Download completes → Event published → Statistics updated → Activity logged

---

12. video.streaming.started

Published when video streaming session begins.

Publishers: Video Streaming Service

Consumers:

- Activity Log Service (log streaming start)
- File Metadata Service (update view count)

Event Schema:

- event_id
- timestamp
- file_id
- user_id
- session_id
- quality_level

Event Lifecycle: Stream requested → Session created → Event published → View count incremented → Activity logged

---

13. video.streaming.stopped

Published when streaming session ends.

Publishers: Video Streaming Service

Consumers:

- Activity Log Service (log streaming end with duration)

Event Schema:

- event_id
- timestamp
- file_id
- user_id
- session_id
- watch_duration
- completion_percentage

Event Lifecycle: Stream ends → Session closed → Event published → Watch analytics recorded → Activity logged

---

14. notification.sent

Published when notification dispatched.

Publishers: Notification Service

Consumers:

- Activity Log Service (log notification)
- WebSocket Realtime Service (push to in-app)

Event Schema:

- event_id
- timestamp
- notification_id
- user_id
- notification_type
- channels

Event Lifecycle: Notification created → Queued for delivery → Sent via channels → Event published → Delivery logged → In-app push sent

---

15. admin.action.executed

Published when admin performs system action.

Publishers: Admin Service

Consumers:

- Activity Log Service (log admin action)
- Affected services (execute admin command)

Event Schema:

- event_id
- timestamp
- admin_id
- action_type
- target_type
- target_id
- action_details

Event Lifecycle: Admin executes action → Permission validated → Action performed → Event published → Audit trail created

---

16. storage.object.created

Published when new storage object created.

Publishers: Storage Manager Service

Consumers:

- Activity Log Service (log storage operation)

Event Schema:

- event_id
- timestamp
- file_id
- bucket_name
- object_key
- size

Event Lifecycle: Object written to storage → Event published → Activity logged

---

17. storage.object.deleted

Published when storage object removed.

Publishers: Storage Manager Service

Consumers:

- Activity Log Service (log deletion)

Event Schema:

- event_id
- timestamp
- file_id
- bucket_name
- object_key

Event Lifecycle: Delete command executed → Object removed → Event published → Activity logged

---

Event Ordering & Partitioning Strategy

Partitioning by user_id: Events for the same user routed to same partition to maintain order. Critical for file operations to ensure processing happens sequentially.

Partitioning by file_id: File-related events partitioned by file_id so all events for a file are ordered.

Retention Policy: Events retained for 7 days in Kafka for debugging and event replay. Long-term storage in Activity Log Service.

Consumer Groups: Each service has its own consumer group to receive all events independently. Multiple instances of same service share consumer group for parallel processing.

Error Handling: Failed message processing retried with exponential backoff. After max retries, message sent to dead letter topic for manual investigation.

REALTIME (WEBSOCKET)

WebSocket Events List

1. file.upload.progress

Trigger: File Upload Service tracks upload progress

Broadcast Behavior: Private (only to uploading user)

Payload:

- file_id
- filename
- bytes_uploaded
- total_bytes
- percentage
- estimated_time_remaining

Usage: Shows real-time progress bar during file upload in UI

---

2. file.upload.complete

Trigger: file.uploaded Kafka event consumed

Broadcast Behavior: Private (only to file owner)

Payload:

- file_id
- filename
- size
- mime_type
- message: "Upload completed successfully"

Usage: Instantly notifies user when upload finishes without page refresh

---

3. file.processing.status

Trigger: file.processing.started, file.processing.completed, file.processing.failed Kafka events

Broadcast Behavior: Private (only to file owner)

Payload:

- file_id
- status: "processing" | "completed" | "failed"
- processing_type
- progress_percentage (if available)
- message

Usage: Updates user on processing status in real-time, shows when file is ready

---

4. notification.new

Trigger: notification.sent Kafka event consumed

Broadcast Behavior: Private (only to notification recipient)

Payload:

- notification_id

- type
- title
- message
- priority
- created_at

Usage: Delivers in-app notifications instantly without polling

---

5. storage.quota.updated

Trigger: User Service updates storage_used after file operations

Broadcast Behavior: Private (only to user whose quota changed)

Payload:

- user_id
- storage_used
- storage_quota
- percentage_used

Usage: Updates storage meter in UI in real-time

---

6. file.list.updated

Trigger: file.uploaded, file.deleted Kafka events

Broadcast Behavior: Private (only to file owner)

Payload:

- action: "added" | "removed"
- file_id
- file_summary (basic file info)

Usage: Automatically updates file list without refresh when files added or removed

---

7. session.terminated

Trigger: user.logged_out Kafka event with reason "forced"

Broadcast Behavior: Private (to specific user session)

Payload:

- session_id
- reason
- message

Usage: Forces logout on all devices when session revoked by admin or security event

---

8. system.announcement

Trigger: Admin Service broadcasts system message

Broadcast Behavior: Broadcast (to all connected users)

Payload:

- announcement_id
- message
- severity: "info" | "warning" | "critical"
- display_duration

Usage: Shows system-wide messages like maintenance windows, new features, or urgent alerts

---

9. admin.metrics.update

Trigger: Periodic aggregation (every 10 seconds) for admin dashboard

Broadcast Behavior: Room-based (only to admin room subscribers)

Payload:

- active_users
- total_uploads_today
- total_storage_used
- active_streams

- timestamp

Usage: Updates admin dashboard with live system metrics

---

10. presence.update

Trigger: User connects or disconnects from WebSocket

Broadcast Behavior: Room-based (to users in shared spaces)

Payload:

- user_id
- status: "online" | "offline"
- last_seen

Usage: Shows online status for collaborative features (future enhancement)

Connection Lifecycle

Connection Establishment:

1. Client initiates WebSocket connection to /ws endpoint
2. Server validates JWT token from connection URL or initial message
3. Server registers connection in MongoDB websocket_connections collection
4. Server subscribes connection to user-specific channel
5. Server sends connection confirmation message
6. Heartbeat mechanism activated (ping every 30 seconds)

Message Routing:

1. Kafka event consumed by WebSocket Realtime Service
2. Service determines target connections based on user_id or broadcast type
3. Service queries websocket_connections for target users
4. Message pushed to appropriate WebSocket connections
5. Delivery confirmed or failure logged

Heartbeat Mechanism:

- Server sends ping every 30 seconds
- Client responds with pong within 10 seconds
- Missing pong triggers connection cleanup
- Client automatically reconnects on disconnection

Disconnection Handling:

1. Client disconnects or heartbeat fails
2. Server removes connection from registry
3. Server logs disconnection event
4. Missed messages queued based on Kafka offset
5. On reconnection, queued messages delivered

Room-Based Broadcasting

User Rooms: Each user automatically subscribed to private room based on user_id. All personal events pushed to this room.

Admin Room: Admin users subscribe to "admin" room for system-wide metrics and monitoring events.

Broadcast Room: All connected users implicitly in broadcast room for system announcements.

File-Specific Rooms (Future): For collaborative features, users viewing same file can join file-specific room for shared cursor, comments, etc.

Message Format

Standard Message Structure:

- event_type: Event identifier
- payload: Event-specific data
- timestamp: Event timestamp
- message_id: Unique message identifier for deduplication

Client Message Structure (for bidirectional):

- action: Requested action
- payload: Action parameters
- request_id: For matching response to request

Scaling Considerations

Multi-Instance Routing: When multiple WebSocket server instances run, connection registry in MongoDB tracks which instance holds each connection. Kafka messages include routing information to ensure events reach correct instance.

Load Balancing: Sticky sessions ensure user reconnects to same instance when possible, reducing cross-instance routing. Load balancer uses IP hash or cookie-based routing.

Connection Limits: Each instance handles up to 10,000 concurrent connections. Auto-scaling adds instances when connection count exceeds threshold.

Token Refresh Flow:
When access token expires, client sends refresh token to `/api/v1/auth/refresh` endpoint. Auth Service validates refresh token against stored hash, checks revocation status, and issues new access token. Refresh tokens are single-use; new refresh token issued with each refresh to prevent token replay attacks.

Telegram OAuth Integration:
Uses Telegram Login Widget for web authentication. Users authenticate via Telegram, receive `auth_data` string with user ID, hash, and timestamp. Backend validates `auth_data` using bot token to prevent forgery. Session established via JWT after validation.

Role-Based Access Control (RBAC):

- **User:** Default role with file upload/download, profile management.
- **Admin:** Elevated role with user management, system monitoring, content moderation.
- **System:** Internal service role for inter-service communication.

Permission Enforcement:
Each service validates user permissions for resource access. File operations check user_id matches file owner. Admin endpoints require `admin` role in JWT. Service-to-service calls use internal API keys.

Data Encryption

Encryption at Rest:

- **MongoDB:** Uses encrypted storage volumes (AWS EBS encryption or GCP persistent disk encryption). Sensitive fields (tokens, API keys) encrypted with AES-256 before storage.
- **GCP Storage:** Server-side encryption with Google-managed keys. Option for customer-managed keys (CMEK) for compliance.

Encryption in Transit:

- **TLS 1.3+** for all external communications (HTTPS, WSS).
- **MTProto** for Telegram communication (Telegram's proprietary encryption).
- **Service Mesh** (optional) with mutual TLS for inter-service communication.

Field-Level Encryption:

- Refresh tokens hashed with bcrypt before storage.
- API keys hashed with SHA-256.
- User emails encrypted if required for compliance.

API Security

Rate Limiting:
Implemented at API Gateway using sliding window algorithm. Limits per user/IP:

- 100 requests/minute for general API
- 10 requests/minute for authentication endpoints
- 5 concurrent uploads per user
- 20 downloads/minute per user

Request Validation:
All endpoints validate input with strict schema validation (using Zod). File uploads validated for MIME type, size limits (2GB max). SQL injection prevention via MongoDB driver parameterization.

CORS Policy:
Restricted to trusted domains only (web app domain). Preflight requests cached for 1 hour. No wildcard origins allowed.

File Security

Access Control Lists (ACLs):
Each file in GCP Storage has ACL restricting access to file owner and system services. Signed URLs provide temporary access with expiration (default 1 hour for downloads, 24 hours for streaming).

Malware Scanning:
File Processing Service integrates with ClamAV or VirusTotal API for virus scanning. Infected files quarantined in isolated bucket and user notified.

Content-Type Enforcement:
Files served with correct Content-Type headers. Direct file access prevented via bucket policies; all access through services.

Infrastructure Security

Service Isolation:
Each microservice runs in separate container with minimal required permissions. Network policies restrict inter-service communication to required ports only.

Secrets Management:
Environment variables stored in GCP Secret Manager. Rotated automatically every 90 days. Services fetch secrets at startup with automatic refresh.

VPC & Firewall:
Services deployed in private VPC with no public IPs. API Gateway and Load Balancer only public endpoints. Ingress/egress rules restrict traffic to required CIDR ranges.

DDoS Protection:
Cloud Load Balancer with Cloud Armor provides DDoS protection. Rate limiting at edge with geographic blocking for known malicious IP ranges.

Audit & Monitoring

Security Logging:
All authentication attempts, token refreshes, admin actions, and file deletions logged to Activity Log Service with immutable storage. Logs retained for 365 days.

Intrusion Detection:
Anomaly detection on login patterns, upload frequencies, and API usage. Alerts triggered for:

- Multiple failed login attempts (>5 in 5 minutes)
- Unusual download patterns (e.g., entire file library in short period)
- Access from unusual geographic locations

Regular Security Audits:
Monthly vulnerability scanning of containers. Quarterly penetration testing. Annual third-party security audit for compliance.

Incident Response Plan:
Documented procedures for security incidents including data breach, DDoS attack, and compromised credentials. 24/7 on-call rotation for critical security alerts.

Compliance Considerations

GDPR Compliance:
User data deletion endpoint fully removes user data across all services and storage. Data processing agreements with third-party providers. Privacy policy detailing data usage.

Data Residency:
Option to configure storage region based on user preference. Metadata stored in MongoDB region matching user's country when required.

Backup Encryption:
Automated daily backups encrypted with separate keys. Backup retention for 30 days with point-in-time recovery capability.

Security Headers

HTTP Security Headers:
All responses include:

- `Strict-Transport-Security: max-age=31536000; includeSubDomains`
- `Content-Security-Policy: default-src 'self'`
- `X-Content-Type-Options: nosniff`
- `X-Frame-Options: DENY`
- `X-XSS-Protection: 1; mode=block`

Cookie Security:
All cookies use `Secure`, `HttpOnly`, `SameSite=Strict` flags. Session cookies expire after 24 hours of inactivity.

DEPLOYMENT & SCALING

(Continued from earlier GCP Deployment section)

Auto-scaling Configuration

Cloud Run Auto-scaling:

```
 min-instances: 1
max-instances: 50
cpu-utilization: 70%
concurrency: 80  # max requests per instance
```

Kafka Partition Strategy:

- `file.uploaded` : Partition by user_id for ordered processing per user
- `file.processing.completed` : Partition by file_id for ordered updates
- General topics: Partition count = 3 × expected consumer instances

Database Scaling:

- MongoDB Atlas M30 tier (minimum)
- Read replicas for File Metadata Service queries
- Sharding by user_id for files collection at 10M+ documents

Monitoring Stack

GCP Cloud Monitoring:

- Custom metrics for upload success rate, processing latency
- Alert policies for 95th percentile latency > 2s
- Service-level objectives (SLOs): 99.9% availability for core services

Log Analytics: Cloud Logging with Log-based metrics for error rates. BigQuery export for long-term analytics.

Health Checks: Each service exposes `/health` endpoint checking:

- Database connectivity
- Kafka connectivity
- GCP Storage connectivity
- Memory usage (<80%)

CI/CD Pipeline

Cloud Build Triggers:

- Push to main → build, test, deploy to staging
- Tag creation → deploy to production
- Pull request → build and run integration tests

Deployment Strategy: Blue-green deployment for API Gateway and stateless services. Database migrations applied with backward compatibility.

Rollback Procedure: Automatic rollback if health check fails for 5 minutes. Manual rollback via Cloud Build triggers to previous container image.

COST OPTIMIZATION

Estimated Monthly Costs (Projected 10K Users)

```
 GCP Cloud Run: $200-400
MongoDB Atlas: $150-300
GCP Storage: $50-100 (based on 1TB storage)
Kafka (Confluent Cloud): $100-200
Cloud CDN: $20-50
Total: $520-1050/month
```

Cost-Saving Measures

- Cold storage class for old files (>90 days unused)
- Spot instances for background processing jobs
- CDN caching for frequently accessed thumbnails
- Auto-scaling minimum instances reduced during off-peak hours
- Multi-region deployment only for premium tier users

TESTING STRATEGY

Test Pyramid

```
         E2E Tests (10%)
        Integration Tests (20%)
        Unit Tests (70%)
```

Unit Tests:
Each service tested in isolation with mocked dependencies. 80% coverage minimum.

Integration Tests:
Service-to-service communication tested with test containers. Kafka events verified end-to-end.

E2E Tests:
Complete user journeys tested with Cypress for web UI and Telegram bot simulation.

Load Tests:
Locust scripts simulating 1000 concurrent users for performance validation.

MAINTENANCE & OPERATIONS

Daily Operations

- Monitor error rates and latency percentiles
- Review security logs for suspicious activity
- Check storage usage and auto-scaling events
- Validate backup completion

Weekly Tasks

- Rotate internal API keys
- Update dependencies (security patches)
- Review cost reports and optimize resources
- Clean up test data and expired sessions

Monthly Tasks

- Full security scan of containers
- Performance review and bottleneck analysis
- Update compliance documentation
- Disaster recovery drill

FUTURE ENHANCEMENTS

Phase 2 Features

1. **Collaborative Features:** File sharing, real-time collaborative editing
2. **Advanced Search:** OCR for images, audio transcription search
3. **Mobile Apps:** iOS and Android native applications
4. **Workflow Automation:** File processing pipelines with user-defined rules
5. **Enterprise Features:** SAML SSO, custom retention policies, audit exports

Technical Improvements

1. **Service Mesh:** Istio for advanced traffic management
2. **GraphQL Gateway:** Unified API with flexible querying
3. **Edge Computing:** Lambda@Edge for global low-latency processing
4. **Machine Learning:** Automated tagging, content classification
5. **Blockchain:** Immutable audit trail for compliance-sensitive documents

---

CONCLUSION

TeleDrive demonstrates a production-ready microservices architecture for cloud storage with Telegram integration. The system balances complexity with practical implementation, providing a realistic learning platform for distributed systems. Each service has clear responsibilities, communication patterns are well-defined, and scaling strategies are implemented from the start.

The 15-phase development plan ensures manageable progression from infrastructure setup to production deployment. Security considerations are integrated throughout the design, and cost optimization measures keep the project affordable for learning purposes.

This architecture provides flexibility for future enhancements while maintaining stability and performance for core functionality. The use of industry-standard technologies (TypeScript, MongoDB, Kafka, GCP) ensures relevance to real-world production systems.

Document Version: 1.0
Last Updated: 2024-03-20
Author: TeleDrive Architecture Team