Captain's Log System - Complete Implementation Plan

Table of Contents

System Overview

Captain's Log is a voice recording application inspired by Star Trek, featuring AI-powered transcription and social sharing capabilities. Users can record audio logs, automatically transcribe them, search through their content, and share with friends.

Key Features: - Voice recording with playback - AI-powered speech-to-text transcription - Friend management and log sharing - Natural language search - Cross-platform synchronization

---

Android Application Architecture

Activities

MainActivity - Purpose: Entry point and navigation host - Responsibilities: Handle permissions, initialize Compose UI

UI Components (Composables)

Component	Purpose	Key Features
MainScreen	Navigation container	Bottom navigation, screen routing
LogListScreen	Display user's audio logs	List view, search bar, expandable items
RecordScreen	Audio recording interface	Record/stop controls, duration display
LogDetailScreen	Individual log view	Playback controls, transcription display, share button
FriendsScreen	Social features	Friend list, add/remove functionality
SearchScreen	Log search interface	Query input, filtered results
SharedLogsScreen	View shared logs	Logs from friends

ViewModels

ViewModel	State Management	Core Functions
LogViewModel	logs: StateFlow> isLoading: StateFlow	loadLogs() deleteLog(id) refreshLogs()
RecordViewModel	isRecording: StateFlow duration: StateFlow	startRecording() stopRecording() saveLog(title)
LogDetailViewModel	currentLog: StateFlow playbackState: StateFlow	loadLogDetail(id) playAudio() pauseAudio() shareWithFriends(ids)
FriendsViewModel	friends: StateFlow> requests: StateFlow>	loadFriends() sendRequest(username) acceptRequest(id)
SearchViewModel	results: StateFlow> query: StateFlow	performSearch(query) clearSearch()

Repositories

Repository	Purpose	Main Operations
LogRepository	Log data management	• Create/read/update/delete logs • Handle audio file uploads • Sync with backend
AudioRepository	Audio operations	• Record audio • Play audio files • Manage local storage
UserRepository	User management	• Authentication • Profile management • Session handling
FriendRepository	Social features	• Friend list operations • Request handling • Sharing permissions
SearchRepository	Search functionality	• Query processing • Result filtering • Cache management

Data Models

Core Entities - `LogEntry` : id, userId, title, audioUrl, transcription, timestamp, duration - `User` : id, username, displayName, profileImageUrl - `Friend` : userId, friendId, status, sharedLogsCount - `FriendRequest` : id, fromUserId, toUserId, status, timestamp - `PlaybackState` : isPlaying, currentPosition, duration

Service Classes

Service	Function
AudioRecordingService	Handle audio capture using MediaRecorder
AudioPlaybackService	Manage playback with ExoPlayer
TranscriptionService	Interface with backend transcription API

Navigation Structure

```
MainScreen
├── LogListScreen (default)
├── RecordScreen
├── FriendsScreen
├── SearchScreen
└── LogDetailScreen (parameterized by logId)
```

Dependency Injection Setup

Using Hilt for dependency injection: - `AppModule` : Provides singleton instances - `NetworkModule` : API service configuration - `DatabaseModule` : Local storage setup

---

Backend System Architecture

API Gateway Routes

Authentication Service

Method	Endpoint	Purpose	Request Body	Response
POST	/auth/register	Create new account	{username, email, password}	{userId, token}
POST	/auth/login	User authentication	{email, password}	{token, refreshToken}
POST	/auth/refresh	Token renewal	{refreshToken}	{token}
GET	/auth/verify	Session validation	Authorization header	{valid, userId}

Log Management Service

Method	Endpoint	Purpose	Request/Params	Response
GET	/logs	Retrieve user logs	?page=1&limit=20	{logs[], total}
POST	/logs	Create new log	Multipart: audio file + metadata	{logId, status}
GET	/logs/{id}	Get specific log	Path parameter	{log details}
PUT	/logs/{id}	Update log info	{title, notes}	{success}
DELETE	/logs/{id}	Remove log	Path parameter	{success}
POST	/logs/{id}/share	Share with friends	{friendIds[]}	{shareIds[]}

Social Features Service

Method	Endpoint	Purpose	Request/Params	Response
GET	/friends	List all friends	-	{friends[]}
POST	/friends/request	Send friend request	{targetUsername}	{requestId}
GET	/friends/requests	Pending requests	-	{requests[]}
PUT	/friends/request/{id}	Accept/reject request	{action: accept/reject}	{success}
DELETE	/friends/{id}	Remove friend	Path parameter	{success}
GET	/shared	Logs shared with me	?page=1&limit=20	{sharedLogs[]}

Search Service

Method	Endpoint	Purpose	Request/Params	Response
GET	/search	Search logs	?q=query&filter=date	{results[]}
GET	/search/suggest	Auto-complete	?q=partial	{suggestions[]}

Microservice Architecture

Service Components

Service	Technology	Responsibilities	Communication
API Gateway	Node.js/Express	• Request routing • Authentication middleware • Rate limiting • CORS handling	REST endpoints
Auth Service	Node.js	• JWT token management • User registration • Password handling • Session control	Internal HTTP
Log Service	Node.js	• CRUD operations • File management • Metadata storage • Share logic	Internal HTTP
Audio Service	Python/FastAPI	• File validation • Format conversion • Storage handling • Compression	Internal HTTP
Transcription Service	Python	• Speech-to-text • Job queuing • Result caching	Message Queue
Search Service	Node.js	• Full-text search • Index management • Query optimization	Internal HTTP

Data Storage

Database	Purpose	Schema Overview
PostgreSQL	Primary data	• users table • logs table • friendships table • log_shares table
Redis	Caching & Sessions	• Session tokens • Search cache • Transcription queue
MinIO/S3	File storage	• Audio files • User avatars

Inter-Service Communication

Synchronous (REST) - API Gateway → All services - Services use internal DNS names - JWT tokens passed in headers

Asynchronous (Message Queue) - Log creation → Transcription queue - Transcription complete → Search indexing - Share events → Notification queue

Container Configuration

Docker Services

`yaml services: - api-gateway (port: 3000) - auth-service (port: 3001) - log-service (port: 3002) - audio-service (port: 3003) - transcr`

Security Measures

Aspect	Implementation
Authentication	JWT with 24h expiration
File Upload	50MB limit, audio formats only
Rate Limiting	100 requests/minute per user
Data Validation	Input sanitization on all endpoints
HTTPS	TLS for all external communication

AI Models Implementation

Speech-to-Text Model Selection

Primary Model: OpenAI Whisper

Attribute	Details
Version	whisper-large-v3
Model Size	1.5GB
Framework	PyTorch
Language Support	99+ languages
Output Format	JSON with timestamps

Alternative Models (Fallback Options)

Model	Size	Pros	Cons
whisper-base	74MB	Fast, lightweight	Lower accuracy
whisper-small	244MB	Good balance	Moderate accuracy
Google Speech API	Cloud	High accuracy	Requires internet, costs

Transcription Service Architecture

API Endpoints

Route	Method	Purpose	Input	Output
/transcribe	POST	Process audio file	Audio file (MP3/WAV)	{text, segments, duration}
/status/{jobId}	GET	Check job status	Job ID	{status, progress}
/health	GET	Service health check	-	{status, modelLoaded}

Processing Pipeline

1. **Audio Reception** → Validate format and size
2. **Preprocessing** → Normalize audio, convert to 16kHz
3. **Transcription** → Process through Whisper model
4. **Post-processing** → Format output, add punctuation
5. **Result Delivery** → Return JSON with text and metadata

Search Enhancement Model

Semantic Search Implementation

Component	Technology	Purpose
Embedding Model	Sentence-BERT (all-MiniLM-L6-v2)	Convert text to vectors
Vector Storage	FAISS or pgvector	Efficient similarity search
Query Processing	Custom pipeline	Handle natural language queries

Search Pipeline

1. Query embedding generation
2. Vector similarity calculation
3. Result ranking and filtering
4. Context extraction for highlights

Model Deployment Strategy

Container Configuration

```
transcription-service/
├── Dockerfile
├── requirements.txt
├── model_loader.py
├── transcription_api.py
└── audio_processor.py
```

Resource Requirements

Environment	CPU	RAM	GPU	Processing Speed
Development	4 cores	8GB	Optional	~2x real-time
Production	8 cores	16GB	Recommended	~10x real-time
High-load	16 cores	32GB	Required	~20x real-time

Performance Optimization

Strategy	Implementation	Impact
Model Caching	Keep model in memory	Eliminate reload time
Batch Processing	Queue multiple files	Increase throughput
Audio Chunking	Split long recordings	Reduce memory usage
Result Caching	Redis cache layer	Avoid re-processing

Fine-tuning Considerations

When to Fine-tune

- Domain-specific vocabulary (Star Trek terms)
- Consistent accuracy issues with certain accents
- Need for custom formatting rules

Data Requirements

Purpose	Data Needed	Format
General improvement	50-100 hours audio	WAV + accurate transcripts
Domain adaptation	10-20 hours themed audio	Aligned text files
Accent optimization	5-10 hours per accent	Native speaker recordings

Monitoring and Metrics

Key Performance Indicators

- Transcription accuracy rate
- Processing time per minute of audio
- Queue length and wait times
- Error rates by file type
- User correction frequency

Health Checks

- Model loading status
- GPU/CPU utilization
- Memory consumption
- Queue backlog size
- API response times

Cost Optimization

Approach	Description	Savings
Model Quantization	Reduce precision to INT8	75% model size
Selective Processing	Only transcribe on demand	Reduce compute
Tiered Service	Different models by user tier	Balance cost/quality
Edge Caching	Cache common phrases	Reduce API calls

Fallback Strategy

If primary model fails: 1. Switch to smaller Whisper model 2. Use cloud API (Google/Azure) temporarily 3. Queue for later processing 4. Notify user of delay

---

Risk Assessment

Technical Risks Matrix

High Priority Risks

Risk	Impact	Probability	Mitigation Strategy
Audio transcription latency	User dissatisfaction	High	• Use smaller Whisper model • Implement progress indicators • Add queue status display
Large file processing	System crash/timeout	Medium	• 50MB file size limit • Audio chunking for 30+ min files • Background job processing
Model memory requirements	Server overload	Medium	• Use model quantization • Implement model caching • Scale horizontally with K8s

Medium Priority Risks

Risk	Impact	Probability	Mitigation Strategy
Search accuracy	Poor user experience	Medium	• Combine keyword + semantic search • User feedback mechanism • Continuous improvement
Storage costs	Budget overrun	Medium	• Audio compression (MP3 128kbps) • Retention policies • Tiered storage options
Concurrent users	Service degradation	Low	• Load balancing • Queue management • Rate limiting

Low Priority Risks

Risk	Impact	Probability	Mitigation Strategy
Android fragmentation	Feature inconsistency	Low	• Target API 24+ • Extensive device testing • Graceful degradation
Privacy concerns	User trust loss	Low	• End-to-end encryption • Clear privacy policy • GDPR compliance

Technology Unfamiliarity

Technology	Current Knowledge	Learning Plan	Timeline
Kubernetes	Basic	• Official tutorials • Local Minikube practice	Week 1-2
WebSocket	Limited	• Socket.io documentation • Build test chat app	Week 1
Elasticsearch	Query basics only	• Official guide • Use managed service initially	Week 2
GPU optimization	None	• PyTorch GPU tutorials • Cloud GPU instances	Week 3

Development Phases Risk

Phase 1: Setup (Days 1-3)

Potential Blockers - Environment configuration - Model download times
- Docker networking issues

Mitigation - Pre-download all models - Use docker-compose templates - Have fallback local setup

Phase 2: Core Features (Days 4-10)

Potential Blockers - API integration complexity - Audio recording permissions - Cross-service communication

Mitigation - Start with mock data - Test on real devices early - Use Postman for API testing

Phase 3: AI Integration (Days 11-14)

Potential Blockers - Whisper performance issues - Transcription accuracy - Resource constraints

Mitigation - Start with whisper-base model - Have cloud API backup - Pre-optimize audio files

Contingency Plans

If Whisper is too slow

1. Use whisper-tiny model initially
2. Implement cloud API fallback
3. Offer async processing option
4. Cache common phrases

If search is inadequate

1. Start with simple SQL LIKE queries
2. Add basic filters (date, duration)
3. Implement semantic search later
4. Collect user feedback for improvement

If real-time features fail

1. Use polling instead of WebSocket
2. Email notifications as backup
3. "Pull to refresh" pattern
4. Batch update notifications

Success Metrics

Metric	Target	Measurement
Transcription speed	< 2x audio duration	Processing time logs
Search response	< 500ms	API monitoring
App crash rate	< 1%	Crash reporting
User satisfaction	> 4.0/5.0	In-app feedback