BodyCam Employee Management App

Development Roadmap — Phase-by-Phase Build Plan

Tech Stack	Detail
Frontend	React Native (Expo)
Streaming	LiveKit (self-hosted on Digital Ocean)
Backend / Auth / DB	Supabase (Postgres, Auth, Realtime, Edge Functions)
Storage	Digital Ocean Spaces (S3-compatible)
AI Summarization	Gemini 1.5 Flash (keyframe + transcript)
PDF Reports	Server-side generation via Edge Function

Resource	Budget	Usage
Gemini API	$25–30	~$5–15 with keyframe optimization
Digital Ocean	$100 credits	~$17–29/mo (droplet + spaces)
Supabase	Free tier	Auth, DB, Realtime, Edge Functions

Phase 1: Project Scaffolding, Auth & RBAC

Duration: 3–4 days

Objective

Stand up the foundational project structure, authentication system, and role-based access control so that managers and employees can log in and see role-appropriate screens.

Tasks

• Initialize an Expo (React Native) project with TypeScript and a clean folder structure (screens, components, services, hooks, navigation).

• Create a Supabase project. Set up the Postgres schema: **users** table (id, email, role, name, org_id), **organizations** table, and Row Level Security (RLS) policies enforcing manager vs. employee access.

• Integrate Supabase Auth into the React Native app (email/password sign-up and login).

• Build a role-selection or invite flow: managers create an org and get a join code; employees enter the code to join and are auto-assigned the employee role.

• Implement navigation guards: managers route to the Manager Dashboard; employees route to the Employee Dashboard. Both are placeholder screens for now.

• Write a seed script to populate test users (1 manager, 4 employees) for local development.

Deliverables

✓ Expo project repo with TypeScript, ESLint, folder structure.

✓ Supabase project with users, organizations tables and RLS policies.

✓ Working sign-up / login flow on iPhone (Expo Go) with role routing.

✓ Seed script for test data.

Working App State at End of Phase

Users can sign up as manager or employee, log in, and land on their respective (empty) dashboards. RBAC enforced at both the UI and database level.

Phase 2: Shift Management & Real-Time Notifications

Duration: 3–4 days

Objective

Enable the core shift lifecycle: managers start/end shifts, and employees receive real-time notifications and respond to them.

Tasks

• Add a **shifts** table in Supabase (id, org_id, manager_id, status, started_at, ended_at).

• Build the Manager Dashboard with a prominent Start Shift / End Shift toggle button and a list of employees with online/offline status indicators.

• Use Supabase Realtime (Postgres changes channel on the shifts table) to push shift-start events to all employee apps in the same org.

• Build the Employee Dashboard: when a shift-start event arrives, show an alert/modal prompting the employee to “Start Streaming.” Display shift status and a timer.

• Handle edge cases: app backgrounded during notification, manager ending shift while employees are still connecting, double-press guards on buttons.

• Add Expo push notifications as a fallback for when the app is not in the foreground.

Deliverables

✓ shifts table with RLS policies.

✓ Manager can start/end shift; button toggles state correctly.

✓ Employees receive real-time in-app notification when shift starts/ends.

✓ Push notification fallback for backgrounded apps.

✓ Employee list on manager screen shows online/offline indicators.

Working App State at End of Phase

Manager presses Start Shift → all employees instantly see a notification and a prompt to connect. Manager presses End Shift → employees are notified shift is over. No streaming yet, but the control flow is fully functional.

Phase 3: LiveKit Server Setup & Live Streaming

Duration: 5–7 days

Objective

Deploy a self-hosted LiveKit server on Digital Ocean and integrate the React Native SDK so employees stream their back camera to a manager-visible grid in real time.

Tasks

• Provision a Digital Ocean droplet (4 GB RAM / 2 vCPU recommended). Install LiveKit server using their official Docker image. Configure TURN/STUN, TLS with a subdomain (e.g., livekit.yourapp.com), and API keys.

• Write a Supabase Edge Function (**generate-livekit-token**) that mints participant tokens with the correct room name (shift ID) and participant identity (user ID + role). Manager tokens get subscribe-only permissions; employee tokens get publish permissions.

• Integrate **@livekit/react-native** SDK into the Expo app. On the employee side: when the employee presses Start Streaming, acquire the back camera, join the LiveKit room, and publish the video track.

• On the manager side: join the same room as a subscriber. Render all remote video tracks in a responsive grid (2×2 for 4 streams). Tap-to-enlarge any stream to full screen.

• Handle connectivity edge cases: employee loses connection mid-stream (auto-reconnect), manager sees a placeholder for disconnected employees, graceful room teardown on End Shift.

• Test on your iPhone via Expo Dev Client (camera access requires a dev build, not Expo Go). Verify latency, resolution, and stability over Wi-Fi.

Deliverables

✓ LiveKit server running on Digital Ocean with TLS and TURN configured.

✓ Edge Function minting scoped LiveKit tokens per role.

✓ Employee streams back camera into LiveKit room on pressing Start Streaming.

✓ Manager sees live grid of all employee streams with tap-to-enlarge.

✓ Reconnection and disconnect handling tested.

Working App State at End of Phase

Full shift flow works end-to-end: Manager starts shift → employees notified → employees press Start Streaming → manager sees a live multi-stream grid. No recording or AI yet, but real-time video surveillance is operational.

Phase 4: Video Recording, Chunking & Cloud Storage

Duration: 4–5 days

Objective

Record each employee's stream on the server side, chunk it into 15-minute segments, and store all footage in Digital Ocean Spaces — never on employee devices.

Tasks

• Enable LiveKit's **Egress API** (built-in server-side recording). When an employee's track is published, trigger a TrackComposite egress that records to a file. Configure it to segment output into 15-minute chunks automatically using LiveKit's segment output mode.

• Set up a **Digital Ocean Space** (S3-compatible bucket). Configure LiveKit egress to upload segments directly to the Space using S3 credentials (endpoint, key, secret, bucket).

• Create a **recordings** table in Supabase (id, shift_id, employee_id, chunk_index, storage_url, started_at, ended_at, summary). Insert a row for each chunk as it is finalized.

• Write a Supabase Edge Function or a lightweight webhook listener that LiveKit calls when each segment upload completes, inserting the metadata into the recordings table.

• Verify that files appear in DO Spaces with a predictable path structure: **/org_id/shift_id/employee_id/chunk_001.mp4**.

• Ensure nothing is saved to employee devices. Confirm that if an employee force-quits the app, the server-side recording continues for any buffered data and the chunk is still saved.

Deliverables

✓ LiveKit Egress configured for 15-min segment recording to DO Spaces.

✓ Digital Ocean Space provisioned with correct CORS and access policies.

✓ recordings table in Supabase auto-populated per chunk via webhook.

✓ Storage path structure verified; files downloadable from Spaces.

✓ Zero data stored on employee phones confirmed.

Working App State at End of Phase

Everything from Phase 3 plus: all streams are silently recorded server-side, chunked every 15 minutes, and stored in the cloud. Manager can verify recordings exist in DO Spaces. The database tracks every chunk with metadata.

Phase 5: AI Summarization Pipeline (Gemini)

Duration: 4–5 days

Objective

Build the automated pipeline that processes each 15-minute video chunk — extracting keyframes and audio transcripts — and sends them to Gemini 1.5 Flash for summarization, storing results in Supabase.

Tasks

• Write a processing worker (Supabase Edge Function or a small Node.js service on the DO droplet) that triggers when a new recording row is inserted.

• The worker downloads the 15-min chunk from DO Spaces, extracts keyframes (1 frame every 5–10 seconds using FFmpeg) and the audio track. Transcribe audio using Whisper (tiny model, runs locally on the droplet) or Gemini's native audio input.

• Assemble a multimodal prompt for Gemini 1.5 Flash: include the keyframe images (as base64 or uploaded via File API) and the transcript text. Prompt it to produce a structured summary: key activities observed, notable events, timestamps of significant moments, and any safety or compliance flags.

• Save the returned summary JSON into the **recordings.summary** column in Supabase.

• Build a simple summary viewer on the Manager Dashboard: during an active shift, the manager can tap an employee and see a rolling list of 15-minute summaries as they come in.

• Add error handling and retry logic: if Gemini returns an error or times out, queue the chunk for retry. Log failures to a **processing_errors** table.

Deliverables

✓ Processing worker triggered automatically per new chunk.

✓ Keyframe extraction + audio transcription pipeline working.

✓ Gemini 1.5 Flash producing structured summaries per chunk.

✓ Summaries stored in Supabase and viewable in-app by manager.

✓ Error handling with retry queue for failed processing.

Working App State at End of Phase

Full live pipeline: employee streams → server records and chunks → each chunk is auto-processed by Gemini → manager sees rolling AI summaries per employee during the shift. Still no final report yet.

Phase 6: End-of-Shift Report Generation

Duration: 3–4 days

Objective

When the manager ends a shift, compile all per-employee summaries into a comprehensive data report PDF for each employee, generated by Gemini and stored in the cloud.

Tasks

• Write an Edge Function (**generate-shift-report**) triggered when a shift's status changes to "ended." For each employee in the shift, fetch all summary records ordered by chunk_index.

• Send the concatenated summaries to Gemini 1.5 Flash with a report-generation prompt: produce a structured shift report including an executive summary, timeline of activities, notable events, compliance observations, and performance notes.

• Use a server-side PDF library (e.g., **pdf-lib** in the Edge Function, or **puppeteer** on the DO droplet rendering an HTML template) to format Gemini's output into a branded, professional PDF.

• Upload the PDF to DO Spaces under **/org_id/shift_id/employee_id/report.pdf**. Store the URL in a new **shift_reports** table (id, shift_id, employee_id, report_url, generated_at).

• On the Manager Dashboard, add a "Shift History" section where past shifts are listed. Tapping a shift shows the list of employees with download/view links for each report PDF.

• Test with realistic data: run a mock shift with all 4 employees streaming for 30–45 minutes, verify summaries and final reports are accurate and well-formatted.

Deliverables

✓ Edge Function compiles summaries and calls Gemini for final report text.

✓ PDF generated server-side with professional formatting.

✓ PDFs stored in DO Spaces; URLs saved in shift_reports table.

✓ Manager can view/download per-employee PDF reports from Shift History.

✓ End-to-end tested with multi-employee mock shift.

Working App State at End of Phase

The complete workflow is now functional: Start Shift → employees stream → live grid + rolling summaries → End Shift → per-employee PDF reports generated and downloadable. This is a feature-complete MVP.

Phase 7: Polish, Hardening & Bodycam Migration Prep

Duration: 4–6 days

Objective

Harden the app for presentation, optimize performance, polish the UI, and prepare the codebase for bodycam integration.

Tasks

• UI/UX polish: consistent theming, loading skeletons, error toasts, empty states, and smooth transitions. Make the stream grid performant (limit resolution to 480p for grid view, switch to 720p on full-screen).

• Add a settings screen for managers: configure chunk duration, toggle AI summarization, manage employee roster (invite/remove).

• Implement data retention controls: auto-delete recordings older than X days from DO Spaces (lifecycle policy) to manage storage costs.

• Abstract the video source layer: create a **StreamSource** interface that the app consumes. The phone camera is one implementation; a bodycam RTMP/WebRTC feed is another. This makes swapping to a bodycam a configuration change rather than a code rewrite.

• Test bodycam integration path: if you have a Wi-Fi bodycam that outputs RTMP, write an RTMP-to-WebRTC bridge (LiveKit has an ingress service for this) so the bodycam feed enters the same LiveKit room as phone streams.

• Security audit: ensure all Edge Functions validate auth tokens, all Spaces URLs are pre-signed (not public), and RLS policies cover every table.

• Performance and stress testing: simulate 5 concurrent streams, monitor droplet CPU/memory, and tune LiveKit settings (bitrate caps, simulcast).

Deliverables

✓ Polished UI ready for demo/presentation.

✓ StreamSource abstraction layer for bodycam migration.

✓ LiveKit Ingress configured for RTMP bodycam input (if hardware available).

✓ Security audit completed: signed URLs, validated tokens, RLS coverage.

✓ Stress test passed with 5 concurrent streams on current droplet.

✓ Data retention lifecycle policy active on DO Spaces.

Working App State at End of Phase

A production-grade, presentation-ready app. Managers and employees experience a polished, reliable workflow. The codebase is structured so that swapping a phone camera for a bodycam requires minimal changes. Ready for live demo with real or simulated bodycam hardware.

Timeline Summary

Phase	Focus	Duration	Cumulative
1	Auth & RBAC	3–4 days	~4 days
2	Shift Mgmt & Notifications	3–4 days	~8 days
3	LiveKit & Live Streaming	5–7 days	~14 days
4	Recording & Cloud Storage	4–5 days	~19 days
5	AI Summarization (Gemini)	4–5 days	~24 days
6	Report Generation	3–4 days	~28 days
7	Polish & Bodycam Prep	4–6 days	~33 days

Total estimated timeline: **5–6 weeks** of focused solo development. Each phase yields a working, testable app that builds on the previous one.

Key risk areas: Phase 3 (LiveKit networking/TLS setup can be tricky) and Phase 5 (Gemini cost optimization requires tuning keyframe density). Budget buffer is healthy — the keyframe approach keeps Gemini costs well under the $25–30 API budget.