Motion Play Data Collection System - Technical Design Document

Project: Motion Play Cloud Platform
Version: 1.0
Date: November 7, 2025
Author: Marc
Status: Draft

---

1. Document Overview

1.1 Purpose

This Technical Design Document (TDD) defines the architecture, component designs, data structures, and interfaces for the Motion Play Data Collection System. It provides the technical blueprint for implementing the requirements defined in the Requirements Document v1.0.

1.2 Scope

This document covers Phase 1 implementation:

- ESP32-S3 firmware architecture
- AWS cloud infrastructure
- Communication protocols and data formats
- Web frontend architecture
- Security and authentication mechanisms

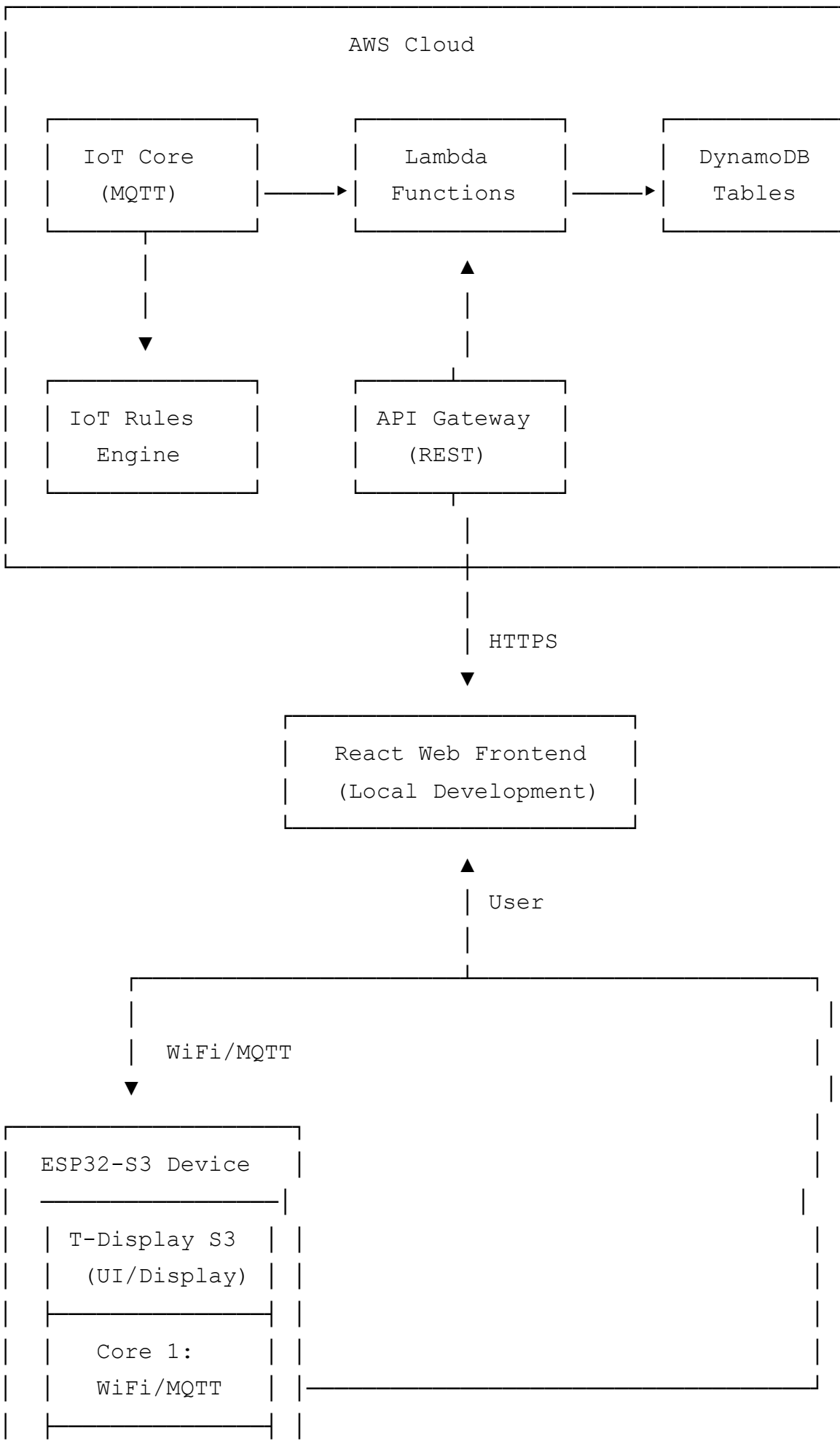
1.3 Audience

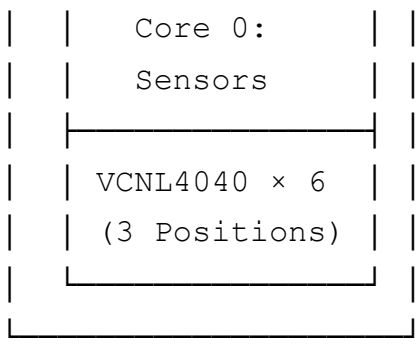
- Marc (developer/implementer)
- Future developers or contributors
- System maintainers

---

2. System Architecture

2.1 High-Level Architecture





2.2 Component Overview

Component	Technology	Purpose
ESP32-S3 Firmware	C++/Arduino, PlatformIO	Device control, sensor reading, cloud communication
AWS IoT Core	MQTT Broker	Device connectivity and message routing
IoT Rules Engine	AWS Service	Route messages to Lambda/DynamoDB
Lambda Functions	Node.js/TypeScript	Business logic and data processing
DynamoDB	NoSQL Database	Store sessions and sensor data
API Gateway	REST API	Web frontend backend interface
React Frontend	TypeScript/React	User interface for control and visualization

2.3 Data Flow

Data Collection Flow:

1. User clicks "Start Collection" in web interface
2. Frontend sends POST to API Gateway → Lambda
3. Lambda publishes command to IoT Core topic `motionplay/{device_id}/commands`
4. ESP32 receives command, begins buffering sensor data
5. User clicks "Stop Collection" in web interface
6. Frontend sends POST to API Gateway → Lambda
7. Lambda publishes stop command to device
8. ESP32 stops collection, publishes batched data to `motionplay/{device_id}/data`
9. IoT Rules Engine triggers Lambda function
10. Lambda processes and stores data in DynamoDB
11. Lambda returns session_id to device
12. Frontend polls or receives notification of completed session

---

3. ESP32-S3 Firmware Architecture

3.1 Core Assignment Strategy

Core 0 (High Priority - Time Critical):

- Sensor reading loop (1-10ms cycle time)
- I2C communication with VCNL4040 sensors
- Data buffering to local queue
- No network operations

Core 1 (Standard Priority - Asynchronous):

- WiFi connection management
- MQTT client operations
- Command processing
- Display updates (T-Display)
- Status LED control

3.2 Key Libraries and Dependencies

```
// Platform
#include <Arduino.h>
#include <WiFi.h>

// MQTT
#include <PubSubClient.h>      // MQTT client

// Display
#include <TFT_eSPI.h>          // T-Display S3 screen

// Sensors (existing)
#include <Wire.h>
#include <Adafruit_VCNL4040.h>

// Data handling
#include <ArduinoJson.h>       // JSON serialization

// Storage
#include <Preferences.h>       // NVMe storage for config
```

3.3 Data Structures

```
// Sensor reading packet
struct SensorReading {
    uint32_t timestamp_ms;      // Milliseconds since collection start
    uint8_t position;           // 1, 2, or 3
    uint8_t sensor_index;       // 0=A, 1=B
    uint16_t proximity;         // Proximity value
    uint16_t ambient;           // Ambient light value
};

// Collection session state
struct CollectionSession {
    char session_id[37];        // UUID
    uint32_t start_time;        // Unix timestamp
    uint32_t sample_count;
    bool is_active;
    QueueHandle_t data_queue;   // FreeRTOS queue
};

// Device state
enum DeviceMode {
    MODE_IDLE,
    MODE_DEBUG,
    MODE_PLAY
};

struct DeviceState {
    DeviceMode current_mode;
    bool wifi_connected;
    bool mqtt_connected;
    CollectionSession* active_session;
};
```

3.4 MQTT Topics and Message Formats

Device → Cloud (Publish):

```
Topic: motionplay/{device_id}/data
Payload: {
    "session_id": "uuid-string",
    "device_id": "device-001",
```

```
    "start_timestamp": "2025-11-07T10:30:00Z",
    "duration_ms": 5000,
    "sample_rate": 1000,
    "readings": [
        {
            "t": 0,           // timestamp offset in ms
            "p": 1,           // position 1-3
            "s": 0,           // sensor A=0, B=1
            "prox": 1024,     // proximity value
            "amb": 256        // ambient value
        },
        // ... more readings
    ]
}

Topic: motionplay/{device_id}/status
Payload: {
    "device_id": "device-001",
    "mode": "debug",
    "wifi_rssi": -45,
    "free_heap": 180000,
    "uptime_ms": 3600000
}
```

Cloud → Device (Subscribe):

```
Topic: motionplay/{device_id}/commands
Payload (Start Collection): {
    "command": "start_collection",
    "session_id": "uuid-generated-by-cloud",
    "timestamp": "2025-11-07T10:30:00Z"
}

Payload (Stop Collection): {
    "command": "stop_collection",
    "session_id": "uuid-from-start"
}

Payload (Set Mode): {
    "command": "set_mode",
    "mode": "debug" // or "play" or "idle"
}
```

```
Payload (Get Status): {
    "command": "get_status"
}

Payload (Restart): {
    "command": "restart"
}
```

3.5 WiFi and Connection Management

```
  // Connection state machine
  enum ConnectionState {
      CONN_DISCONNECTED,
      CONN_WIFI_CONNECTING,
      CONN_WIFI_CONNECTED,
      CONN_MQTT_CONNECTING,
      CONN_MQTT_CONNECTED
  };

  // Retry configuration
  const int WIFI_RETRY_INTERVAL_MS = 5000;
  const int MQTT_RETRY_INTERVAL_MS = 5000;
  const int MAX_RETRY_ATTEMPTS = 10;
  const int RETRY_BACKOFF_MULTIPLIER = 2;

  // Connection management
  - Store WiFi credentials in Preferences (NVMe)
  - Exponential backoff for reconnection attempts
  - Watchdog timer for connection health
  - Automatic reconnection on disconnect
```

3.6 Memory Management

Buffer Sizing:

- Maximum collection: 30 seconds × 1000 Hz × 6 sensors = 180,000 samples
- Per sample: ~12 bytes (timestamp, position, sensor, proximity, ambient)
- Total: ~2.1 MB raw data
- ESP32-S3 PSRAM: 8 MB available

Strategy:

- Allocate circular buffer in PSRAM for sensor readings
- Use FreeRTOS queue (1000 item capacity) for inter-core communication
- Batch data in JSON format with compression consideration
- Maximum 10 concurrent collection sessions buffered

3.7 Display Management

```
// T-Display S3 Layout
┌──────────────────────────┐
│ Motion Play              │  Header (mode, WiFi status)
├──────────────────────────┤
│                          │
│  WiFi: Connected         │  Connection status
│  Cloud: Connected        │
│                          │
│  Mode: DEBUG             │  Current mode
│                          │
│  [Ready]                 │  State indicator
│                          │
│                          │
└──────────────────────────┘
│  BTN1   │    BTN2        │  Physical buttons
└─────────┴────────────────┘

// Display update frequency: 500ms
// Non-blocking updates on Core 1
```

---

4. AWS Cloud Infrastructure

4.1 AWS IoT Core Configuration

Thing Configuration:

```
Thing Name: motionplay-device-001
Thing Type: MotionPlayDevice
Thing Group: MotionPlayDevices-Dev
```

Certificate and Policy:

```
// IoT Policy: MotionPlayDevicePolicy
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "iot:Connect",
      "Resource": "arn:aws:iot:REGION:ACCOUNT:client/motionplay-*"
    },
    {
      "Effect": "Allow",
      "Action": "iot:Publish",
      "Resource": [
        "arn:aws:iot:REGION:ACCOUNT:topic/motionplay/*/data",
        "arn:aws:iot:REGION:ACCOUNT:topic/motionplay/*/status"
      ]
    },
    {
      "Effect": "Allow",
      "Action": "iot:Subscribe",
      "Resource": "arn:aws:iot:REGION:ACCOUNT:topicfilter/motionplay/*/cc
    },
    {
      "Effect": "Allow",
      "Action": "iot:Receive",
      "Resource": "arn:aws:iot:REGION:ACCOUNT:topic/motionplay/*/commands
    }
  ]
}
```

IoT Rules:

```
-- Rule: MotionPlayDataRouter
SELECT * FROM 'motionplay/+/data'

Action: Lambda function (processData)

-- Rule: MotionPlayStatusRouter
SELECT * FROM 'motionplay/+/status'

Action: Lambda function (processStatus)
```

4.2 DynamoDB Schema

Table 1: Sessions

```
Table Name: MotionPlaySessions
Partition Key: session_id (String)
Sort Key: N/A

Attributes:
- session_id: String (UUID)
- device_id: String
- start_timestamp: String (ISO 8601)
- end_timestamp: String (ISO 8601)
- duration_ms: Number
- mode: String (debug/play)
- sample_count: Number
- sample_rate: Number
- labels: List<String>
- notes: String
- data_s3_key: String (optional, for large datasets)
- created_at: Number (Unix timestamp)

Global Secondary Index: DeviceTimeIndex
- Partition Key: device_id
- Sort Key: start_timestamp
- Projection: ALL

TTL Attribute: ttl (optional, for auto-cleanup)
```

Table 2: SensorData

```
Table Name: MotionPlaySensorData
Partition Key: session_id (String)
Sort Key: timestamp_offset (Number)

Attributes:
- session_id: String
- timestamp_offset: Number (ms from session start)
- position: Number (1-3)
- sensor_index: Number (0-1)
- proximity: Number
- ambient: Number
```

```
Note: For large sessions (>400KB), consider storing data in S3
and keeping only metadata in DynamoDB
```

Table 3: Devices

```
Table Name: MotionPlayDevices
Partition Key: device_id (String)

Attributes:
- device_id: String
- device_name: String
- registered_at: String (ISO 8601)
- last_seen: String (ISO 8601)
- firmware_version: String
- status: String (online/offline)
- current_mode: String
```

4.3 Lambda Functions

Function 1: processData

```
// Runtime: Node.js 20.x
// Timeout: 30 seconds
// Memory: 512 MB

Input: IoT Rule trigger from motionplay/+/data topic
Output: Write to DynamoDB Sessions and SensorData tables

Responsibilities:
- Validate incoming data structure
- Parse and decompress sensor readings
- Store session metadata in Sessions table
- Store individual readings in SensorData table
- Handle large payloads (>400KB) by storing in S3
- Error handling and retry logic
```

Function 2: processStatus

```
// Runtime: Node.js 20.x
// Timeout: 10 seconds
// Memory: 256 MB
```

```
Input: IoT Rule trigger from motionplay/+/status topic
Output: Update Devices table

Responsibilities:
- Update device last_seen timestamp
- Update device status and mode
- Log status for monitoring
```

Function 3: sendCommand

```
// Runtime: Node.js 20.x
// Timeout: 10 seconds
// Memory: 256 MB

Input: API Gateway POST /devices/{device_id}/commands
Output: Publish MQTT message to device

Responsibilities:
- Validate command structure
- Generate session_id for start_collection commands
- Publish to motionplay/{device_id}/commands topic
- Return command acknowledgment
```

Function 4: getSessions

```
// Runtime: Node.js 20.x
// Timeout: 10 seconds
// Memory: 256 MB

Input: API Gateway GET /sessions?device_id=X
Output: List of sessions

Responsibilities:
- Query Sessions table by device_id
- Return paginated results
- Support filtering by date range
```

Function 5: getSessionData

```
// Runtime: Node.js 20.x
// Timeout: 15 seconds
```

```
// Memory: 512 MB

Input: API Gateway GET /sessions/{session_id}
Output: Session metadata and sensor data

Responsibilities:
- Fetch session metadata from Sessions table
- Fetch sensor readings from SensorData table or S3
- Return complete session data
- Support pagination for large datasets
```

Function 6: updateSession

```
// Runtime: Node.js 20.x
// Timeout: 10 seconds
// Memory: 256 MB

Input: API Gateway PATCH /sessions/{session_id}
Output: Updated session metadata

Responsibilities:
- Update labels, notes, or other metadata
- Validate update permissions
- Return updated session
```

Function 7: deleteSession

```
// Runtime: Node.js 20.x
// Timeout: 10 seconds
// Memory: 256 MB

Input: API Gateway DELETE /sessions/{session_id}
Output: Confirmation

Responsibilities:
- Delete session from Sessions table
- Delete associated data from SensorData table or S3
- Return confirmation
```

4.4 API Gateway Configuration

API Type: REST API (for simpler CORS handling during development)

Endpoints:

```
POST   /devices/{device_id}/commands
  Body: { "command": "start_collection" | "stop_collection" | "set_mode" |
"get_status" | "restart", ... }
  Response: { "command_id": "uuid", "status": "sent" }

GET    /sessions
  Query: ?device_id=X&start_date=Y&end_date=Z&limit=N
  Response: { "sessions": [...], "next_token": "..." }

GET    /sessions/{session_id}
  Response: { "session": {...}, "data": [...] }

PATCH  /sessions/{session_id}
  Body: { "labels": [...], "notes": "..." }
  Response: { "session": {...} }

DELETE /sessions/{session_id}
  Response: { "deleted": true }

GET    /devices/{device_id}/status
  Response: { "device": {...}, "online": true, "last_seen": "..." }
```

CORS Configuration:

```
  {
    "AllowOrigins": ["http://localhost:3000", "http://localhost:5173"],
    "AllowMethods": ["GET", "POST", "PATCH", "DELETE", "OPTIONS"],
    "AllowHeaders": ["Content-Type", "Authorization"],
    "MaxAge": 3600
  }
```

---

5. Web Frontend Architecture

5.1 Technology Stack

```
Framework: React 18
Language: TypeScript
Build Tool: Vite
```

```
Styling: TailwindCSS
Charts: Recharts
HTTP Client: Axios
State Management: React Context + useState (simple for Phase 1)
```

5.2 Component Architecture

```
src/
├── components/
│   ├── DeviceStatus.tsx        // Connection status display
│   ├── ModeSelector.tsx        // Debug/Play mode toggle
│   ├── CollectionControl.tsx   // Start/Stop buttons
│   ├── SessionList.tsx         // List of past sessions
│   ├── SessionDetail.tsx       // View session data
│   ├── SensorChart.tsx         // Data visualization
│   ├── LabelEditor.tsx         // Add/edit labels
│   └── ExportButton.tsx        // Export data
├── services/
│   ├── api.ts                  // API Gateway client
│   └── types.ts                // TypeScript interfaces
├── contexts/
│   └── DeviceContext.tsx       // Device state management
├── hooks/
│   ├── useDevice.ts            // Device control hook
│   ├── useSessions.ts          // Session management hook
│   └── usePolling.ts           // Status polling hook
├── App.tsx
└── main.tsx
```

5.3 Key TypeScript Interfaces

```typescript
interface Device {
  device_id: string;
  device_name: string;
  status: 'online' | 'offline';
  current_mode: 'idle' | 'debug' | 'play';
  last_seen: string;
}

interface Session {
  session_id: string;
```

```
  device_id: string;
  start_timestamp: string;
  end_timestamp: string;
  duration_ms: number;
  mode: string;
  sample_count: number;
  sample_rate: number;
  labels: string[];
  notes: string;
}

interface SensorReading {
  timestamp_offset: number;
  position: number;
  sensor_index: number;
  proximity: number;
  ambient: number;
}

interface SessionData {
  session: Session;
  readings: SensorReading[];
}

interface Command {
  command: 'start_collection' | 'stop_collection' | 'set_mode' | 'get_sta
  mode?: 'debug' | 'play' | 'idle';
  session_id?: string;
}
```

5.4 API Client Service

```
// src/services/api.ts
import axios from 'axios';

const API_BASE_URL = import.meta.env.VITE_API_BASE_URL || 'http://localhc

class MotionPlayAPI {
  private client = axios.create({
    baseURL: API_BASE_URL,
    timeout: 10000,
  });
```

```
  async sendCommand(deviceId: string, command: Command) {
    return this.client.post(`/devices/${deviceId}/commands`, command);
  }

  async getSessions(deviceId?: string, limit?: number) {
    return this.client.get('/sessions', {
      params: { device_id: deviceId, limit }
    });
  }

  async getSessionData(sessionId: string) {
    return this.client.get(`/sessions/${sessionId}`);
  }

  async updateSession(sessionId: string, updates: Partial<Session>) {
    return this.client.patch(`/sessions/${sessionId}`, updates);
  }

  async deleteSession(sessionId: string) {
    return this.client.delete(`/sessions/${sessionId}`);
  }

  async getDeviceStatus(deviceId: string) {
    return this.client.get(`/devices/${deviceId}/status`);
  }
}

export default new MotionPlayAPI();
```

5.5 Main Application Views

1. Dashboard View

- Device connection status
- Current mode display
- Quick action buttons (start/stop collection)
- Recent sessions list

2. Sessions View

- Searchable/filterable session list
- Session cards with metadata
- Quick label editing

- Delete functionality

3. Session Detail View

- Full session metadata
- Multi-line chart of sensor readings
- Zoom/pan capabilities
- Label management
- Export buttons (JSON, CSV)
- Notes editor

4. Settings View (Future)

- Device configuration
- Threshold adjustments
- WiFi settings

---

6. Security Architecture

6.1 Device Authentication

Certificate-Based Authentication:

- Each device has unique X.509 certificate
- Private key stored securely on device
- Certificate signed by AWS IoT CA
- Mutual TLS (mTLS) for all IoT Core connections

Certificate Provisioning Flow:

1. Generate device certificate using AWS IoT Core
2. Download certificate, private key, and root CA
3. Flash to ESP32-S3 filesystem (LittleFS or SPIFFS)
4. Device loads certificates on boot
5. Establishes mTLS connection to IoT Core

6.2 Data Security

In Transit:

- TLS 1.2+ for all MQTT connections
- HTTPS for all API Gateway calls
- WSS (WebSocket Secure) if real-time updates added

At Rest:

- DynamoDB encryption enabled (AWS managed keys)
- S3 server-side encryption (if used)
- No sensitive data stored in plaintext

6.3 API Security (Phase 1 - Development)

Current:

- No authentication (local development only)
- CORS restricted to localhost

Future (Phase 2+):

- API keys for basic protection
- AWS Cognito for user authentication
- JWT tokens for API authorization
- Rate limiting via API Gateway

---

7. Error Handling and Resilience

7.1 ESP32 Error Handling

WiFi Disconnection:

```
- Buffer sensor data locally during disconnection
- Attempt reconnection with exponential backoff
- Maximum buffer size: 50 sessions (~100MB)
- Oldest data discarded if buffer full
```

MQTT Publish Failure:

```
- Retry up to 3 times with 1s delay
- Queue failed messages for later retry
- Log error to serial console
- Display error state on T-Display
```

Sensor Read Failure:

```
- Log error count per sensor
- Continue reading other sensors
- If all sensors fail: restart I2C bus
- If persistent: require device restart
```

7.2 Lambda Error Handling

DynamoDB Write Failure:

```
- Retry with exponential backoff (3 attempts)
- Log error to CloudWatch
- Store failed payload in DLQ (Dead Letter Queue)
- Alert on repeated failures
```

Data Validation Failure:

```
- Log invalid payload to CloudWatch
- Return 400 error with details
- Do not store invalid data
```

7.3 Frontend Error Handling

API Call Failure:

```
- Display user-friendly error message
- Retry button for transient failures
- Log error to console
- Timeout after 10 seconds
```

Network Disconnection:

```
- Show offline indicator
- Queue actions for retry
- Poll for reconnection
```

---

8. Monitoring and Logging

8.1 CloudWatch Metrics

Custom Metrics:

- Device connection/disconnection events
- Data collection session count
- Lambda invocation duration
- DynamoDB read/write capacity usage
- API Gateway request count and latency

Alarms:

- Lambda error rate > 5%
- DynamoDB throttling events
- Device offline > 5 minutes
- API Gateway 5xx errors

8.2 Logging Strategy

ESP32:

```
- Serial output for development debugging
- Log levels: ERROR, WARN, INFO, DEBUG
- Structured logging format: [LEVEL] [Component] Message
```

Lambda:

```
- CloudWatch Logs for all functions
- Structured JSON logging
- Include request ID, device ID, session ID in all logs
- Log retention: 7 days (development), 30 days (production)
```

Frontend:

```
- Console logging for development
- Error boundary for React components
- Optional: Integration with error tracking (Sentry, etc.)
```

---

9. Development and Deployment

9.1 Development Environment Setup

ESP32-S3:

```
# PlatformIO project structure
motion-play-firmware/
├── platformio.ini
├── src/
│   ├── main.cpp
│   ├── sensor_manager.cpp
│   ├── mqtt_client.cpp
│   ├── display_manager.cpp
│   └── config.h
├── lib/
└── data/
    ├── certs/
    │   ├── device-cert.pem
    │   ├── private-key.pem
    │   └── root-ca.pem
    └── config.json
```

AWS Infrastructure:

```
# Recommended: Infrastructure as Code
# Option 1: AWS CDK (TypeScript)
motion-play-infrastructure/
├── bin/
│   └── motion-play-stack.ts
├── lib/
│   ├── iot-stack.ts
│   ├── lambda-stack.ts
│   ├── dynamodb-stack.ts
│   └── api-stack.ts
└── package.json

# Option 2: Manual AWS Console setup for Phase 1
```

Frontend:

```
motion-play-frontend/
├── src/
│   ├── components/
```

```
│   ├── services/
│   ├── contexts/
│   ├── hooks/
│   ├── App.tsx
│   └── main.tsx
├── public/
├── package.json
├── vite.config.ts
└── tsconfig.json
```

9.2 Deployment Strategy

Phase 1 (Development):

- ESP32: USB upload via PlatformIO
- AWS: Manual setup or CDK deploy
- Frontend: Local development server (Vite)

Phase 2 (Testing):

- ESP32: OTA updates via AWS IoT Jobs
- AWS: Automated deployment via CDK
- Frontend: S3 + CloudFront hosting

9.3 Configuration Management

ESP32 Configuration:

```
// config.h
#define DEVICE_ID "motionplay-device-001"
#define WIFI_SSID "your-ssid"
#define WIFI_PASSWORD "your-password"
#define MQTT_BROKER "xxxxx.iot.us-west-2.amazonaws.com"
#define MQTT_PORT 8883
#define SAMPLE_RATE_HZ 1000
```

Frontend Configuration:

```
// .env.development
VITE_API_BASE_URL=https://your-api-id.execute-api.us-west-2.amazonaws.com
VITE_DEFAULT_DEVICE_ID=motionplay-device-001
```

```
// .env.production
VITE_API_BASE_URL=https://your-api-id.execute-api.us-west-2.amazonaws.com
```

---

10. Testing Strategy

10.1 ESP32 Testing

Unit Tests:

- Sensor reading accuracy
- Data buffering and queue operations
- MQTT message formatting

Integration Tests:

- WiFi connection and reconnection
- MQTT publish/subscribe
- Full collection workflow

Manual Testing:

- Display rendering
- Button interactions
- End-to-end data collection

10.2 Backend Testing

Lambda Unit Tests:

```
- Data validation logic
- DynamoDB operations
- Error handling
```

Integration Tests:

```
- IoT Rule → Lambda trigger
- API Gateway → Lambda invoke
- End-to-end data flow
```

10.3 Frontend Testing

Component Tests:

- Button clicks trigger correct actions
- API calls made with correct parameters
- Error states displayed properly

E2E Tests (Optional):

- Complete collection workflow
- Session management flow

---

11. Performance Optimization

11.1 ESP32 Optimizations

Memory:

- Use PSRAM for large buffers
- Minimize heap fragmentation
- Pre-allocate buffers where possible

Power:

- WiFi sleep mode when idle
- Display dimming after inactivity
- Optimized polling intervals

Network:

- Batch data transmission
- MQTT QoS 1 for reliability without overhead
- Compress large payloads (gzip)

11.2 Backend Optimizations

DynamoDB:

- On-demand billing for unpredictable workload
- GSI for efficient queries by device_id
- Consider S3 for large datasets (>400KB per session)

Lambda:

- Provisioned concurrency not needed (low volume)

- Optimize cold start with minimal dependencies
- Stream large responses from S3 if needed

API Gateway:

- Enable caching for GET /sessions (short TTL: 30s)
- Response compression enabled

11.3 Frontend Optimizations

React:

- Lazy load Session Detail view
- Memoize expensive computations
- Virtual scrolling for large session lists

Network:

- Debounce API calls
- Pagination for large datasets
- Local caching of session list

---

12. Risks and Mitigations

Risk	Impact	Likelihood	Mitigation
ESP32 memory overflow with large sessions	High	Medium	Implement buffer limits, move to S3 storage
WiFi instability causing data loss	High	Low	Local buffering and retry logic
DynamoDB throttling	Medium	Low	On-demand billing, optimize write patterns
MQTT message size limits	Medium	Medium	Split large payloads, use S3 presigned URLs
Certificate provisioning complexity	Low	Medium	Clear documentation, automated scripts
AWS cost overruns	Medium	Low	CloudWatch billing alarms, cost monitoring

---

13. Future Architecture Considerations

13.1 Multi-Device Support (Phase 2)

MQTT Topic Changes:

```
motionplay/group/{group_id}/devices
motionplay/group/{group_id}/commands
motionplay/device/{device_id}/data
```

Additional Components:

- Device coordinator Lambda function
- DynamoDB table: DeviceGroups
- WebSocket API for real-time updates

13.2 ML Pipeline (Phase 3)

Components:

- S3 bucket for training data export
- SageMaker for model training
- Lambda for model deployment
- ESP32 TensorFlow Lite integration

13.3 Production Features (Phase 4)

Authentication:

- AWS Cognito user pools
- Device registration portal
- Multi-tenant support

Monitoring:

- Custom dashboard
- Real-time alerts
- Usage analytics

---

14. Technical Decisions and Rationale

14.1 Why MQTT over HTTP?

Decision: Use MQTT for device-to-cloud communication
Rationale:

- Persistent connection reduces latency
- Bidirectional communication (cloud can push commands)
- More efficient for frequent small messages
- AWS IoT Core provides managed MQTT broker
- Industry standard for IoT devices

Trade-offs:

- Slightly more complex than HTTP REST
- Requires maintaining persistent connection
- Certificate management overhead

14.2 Why DynamoDB over RDS?

Decision: Use DynamoDB for data storage
Rationale:

- Serverless, scales automatically
- Pay-per-request pricing good for unpredictable load
- Low latency for key-value lookups
- Integrates well with Lambda
- No server management

Trade-offs:

- Less flexible querying than SQL
- No complex joins
- Requires careful schema design

14.3 Why React over Vue/Angular?

Decision: React with TypeScript for frontend
Rationale:

- Marc has TypeScript experience
- Large ecosystem and component libraries
- Vite provides fast development experience
- Easy to integrate charting libraries
- Good for single-page applications

14.4 Why Lambda over EC2/Containers?

Decision: Lambda for backend logic
Rationale:

- Serverless, no server management
- Scales automatically
- Pay only for execution time
- Fast cold starts with Node.js
- Native integration with AWS services

Trade-offs:

- Cold start latency (minimal for Node.js)
- 15-minute execution limit (sufficient for this use case)
- Less control over environment

---

15. Open Questions and TODOs

15.1 Open Questions

1. Should we implement data compression for large payloads?
    - Recommendation: Start without, add if >400KB sessions common
2. Real-time vs polling for session completion notification?
    - Recommendation: Start with polling (simpler), add WebSocket later
3. Device provisioning workflow for multiple devices?
    - Recommendation: Manual for Phase 1, automated script for Phase 2

15.2 Implementation TODOs

Before Implementation:

- [ ] Set up AWS account and configure AWS CLI
- [ ] Choose AWS region (recommend us-west-2)
- [ ] Create GitHub/Git repository structure
- [ ] Set up development environment

Phase 1 Priorities:

- [ ] Basic ESP32 WiFi and MQTT connection
- [ ] Single Lambda function for data ingestion
- [ ] Minimal DynamoDB schema
- [ ] Simple React UI with device control

---

16. Glossary

Term	Definition
MQTT	Message Queuing Telemetry Transport - lightweight pub/sub protocol
mTLS	Mutual TLS - both client and server authenticate with certificates
DLQ	Dead Letter Queue - storage for failed message processing
GSI	Global Secondary Index - additional query pattern for DynamoDB
IoT Thing	AWS IoT Core representation of a physical device
Device Shadow	Virtual representation of device state in AWS IoT
OTA	Over-The-Air - remote firmware update capability

Document Approval:

Developer: Marc (Self-approved for hobby project)
Date: November 7, 2025

Revision History:

Version	Date	Author	Changes
1.0	Nov 7, 2025	Marc	Initial technical design document