Motion Play Data Collection System - Implementation Plan

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

---

1. Executive Summary

This Implementation Plan provides a phased approach to building the Motion Play Data Collection System. The plan breaks down the technical design into actionable tasks with time estimates, dependencies, and risk mitigation strategies. The plan assumes 2-8 hours per week of development time over a 1-2 month timeline.

---

2. Project Timeline Overview

Total Estimated Duration: 6-8 weeks
Total Estimated Effort: 40-60 hours
Development Cadence: 2-8 hours per week

Milestone Summary

Phase	Duration	Key Deliverable
Phase 0: Setup & Preparation	Week 1	Development environment ready
Phase 1: Basic Connectivity	Weeks 2-3	ESP32 ↔ AWS communication working
Phase 2: Data Pipeline	Weeks 3-4	End-to-end data collection flow
Phase 3: Web Interface	Weeks 5-6	Basic frontend operational
Phase 4: Integration & Polish	Weeks 7-8	Complete system functional

---

3. Phase 0: Setup & Preparation

Duration: 1 week (4-6 hours)
Goal: Establish development environment and project structure

Tasks

Task 0.1: AWS Account Setup

Estimated Time: 1 hour
Priority: Critical
Dependencies: None

Steps:

1. Verify AWS account access
2. Install/update AWS CLI
3. Configure AWS credentials (`aws configure`)
4. Choose AWS region (recommend: us-west-2)
5. Set up billing alerts ($20/month threshold)

Acceptance Criteria:

- [ ] AWS CLI configured and working
- [ ] Can run `aws sts get-caller-identity` successfully
- [ ] Billing alert configured

Task 0.2: Project Repository Setup

Estimated Time: 30 minutes
Priority: High
Dependencies: None

Steps:

1. Create Git repository structure:

```
motion-play/
├── firmware/               # ESP32-S3 code
├── infrastructure/         # AWS CDK or CloudFormation (optional)
├── lambda/                # Lambda functions
├── frontend/              # React application
└── docs/                  # Documentation
```

```
├── requirements.md
├── technical-design.md
└── implementation-plan.md
```

2. Initialize git repository
3. Create .gitignore files
4. Initial commit with documentation

Acceptance Criteria:

- [ ] Repository structure created
- [ ] Documentation files committed
- [ ] .gitignore properly configured

Task 0.3: ESP32 Development Environment

Estimated Time: 1.5 hours
Priority: Critical
Dependencies: None

Steps:

1. Install/verify PlatformIO in VSCode
2. Create new PlatformIO project for ESP32-S3
3. Configure platformio.ini:

```
[env:esp32-s3-devkitc-1]
platform = espressif32
board = esp32-s3-devkitc-1
framework = arduino
monitor_speed = 115200
lib_deps =
    knolleary/PubSubClient@^2.8
    bblanchon/ArduinoJson@^6.21.3
    adafruit/Adafruit VCNL4040@^2.0.1
```

4. Test compile and upload simple "Hello World"
5. Verify serial monitor output

Acceptance Criteria:

- [ ] Can compile ESP32 code
- [ ] Can upload to device via USB
- [ ] Serial monitor shows output

Task 0.4: Frontend Development Environment

Estimated Time: 1 hour
Priority: High
Dependencies: None

Steps:

1. Verify Node.js and npm installed (Node 18+)
2. Create React project with Vite:

```
npm create vite@latest motion-play-frontend -- --template react-ts
cd motion-play-frontend
npm install
```

3. Install dependencies:

```
npm install axios recharts
npm install -D tailwindcss postcss autoprefixer
npx tailwindcss init -p
```

4. Configure Tailwind
5. Test dev server: `npm run dev`

Acceptance Criteria:

- [ ] Frontend builds without errors
- [ ] Dev server runs on localhost:5173
- [ ] Can see default Vite page

Task 0.5: AWS IoT Core Initial Setup

Estimated Time: 1.5 hours
Priority: Critical
Dependencies: Task 0.1

Steps:

1. Create IoT Thing: `motionplay-device-001`
2. Create and download device certificates
3. Create IoT Policy (use policy from TDD)
4. Attach policy to certificate

5. Attach certificate to Thing
6. Save certificates securely
7. Note IoT endpoint URL

Acceptance Criteria:

- [ ] Thing created in AWS IoT Core
- [ ] Certificates downloaded and saved
- [ ] Policy attached
- [ ] IoT endpoint URL documented

Task 0.6: DynamoDB Tables Creation

Estimated Time: 45 minutes
Priority: High
Dependencies: Task 0.1

Steps:

1. Create `MotionPlaySessions` table
    - Partition key: session_id (String)
    - On-demand billing
    - GSI: DeviceTimeIndex (device_id, start_timestamp)
2. Create `MotionPlaySensorData` table
    - Partition key: session_id (String)
    - Sort key: timestamp_offset (Number)
    - On-demand billing
3. Create `MotionPlayDevices` table
    - Partition key: device_id (String)
    - On-demand billing
4. Document table ARNs

Acceptance Criteria:

- [ ] All three tables created
- [ ] GSI configured on Sessions table
- [ ] Table ARNs documented

Phase 0 Risks and Mitigation

Risk	Impact	Mitigation

Risk	Impact	Mitigation
AWS setup issues	Blocks all work	Complete first, verify thoroughly
PlatformIO driver issues	Delays firmware work	Use official ESP32 setup guide
Certificate management confusion	Security vulnerability	Follow AWS IoT documentation carefully

---

4. Phase 1: Basic Connectivity

Duration: 2 weeks (8-12 hours)
Goal: Establish bidirectional communication between ESP32 and AWS IoT Core

Tasks

Task 1.1: ESP32 WiFi Connection

Estimated Time: 2 hours
Priority: Critical
Dependencies: Task 0.3

Steps:

1. Create `wifi_manager.cpp` module
2. Implement WiFi connection with stored credentials
3. Add connection status LEDs/display feedback
4. Implement reconnection logic with exponential backoff
5. Test disconnect and reconnect scenarios

Code Structure:

```
// wifi_manager.h
class WiFiManager {
public:
    bool connect(const char* ssid, const char* password);
    bool isConnected();
    void reconnect();
    int getRSSI();
};
```

Acceptance Criteria:

- [ ] ESP32 connects to WiFi on boot
- [ ] Connection status shown on T-Display
- [ ] Automatic reconnection works
- [ ] RSSI readable

Task 1.2: MQTT Client Implementation

Estimated Time: 3 hours
Priority: Critical
Dependencies: Task 1.1, Task 0.5

Steps:

1. Load certificates from filesystem (LittleFS)
2. Configure PubSubClient for AWS IoT Core
3. Implement MQTT connection with TLS
4. Create callback for incoming messages
5. Test connection to AWS IoT Core
6. Monitor connection in AWS IoT Core console

Code Structure:

```
// mqtt_client.h
class MQTTClient {
public:
    bool connect();
    bool publish(const char* topic, const char* payload);
    bool subscribe(const char* topic);
    void loop();
    bool isConnected();
private:
    void messageCallback(char* topic, byte* payload, unsigned int length)
};
```

Acceptance Criteria:

- [ ] ESP32 connects to AWS IoT Core
- [ ] Connection visible in AWS IoT Core console
- [ ] No certificate errors
- [ ] Connection remains stable for 5+ minutes

Task 1.3: Certificate Management

Estimated Time: 1.5 hours
Priority: Critical
Dependencies: Task 0.5

Steps:

1. Set up LittleFS filesystem
2. Create data directory in firmware project
3. Copy certificates to data directory
4. Upload filesystem to ESP32
5. Implement certificate loading code
6. Verify certificates load correctly

File Structure:

```
firmware/data/
├── certs/
│   ├── device-cert.pem
│   ├── private-key.pem
│   └── root-ca.pem
└── config.json
```

Acceptance Criteria:

- [ ] Certificates stored in ESP32 filesystem
- [ ] Code successfully loads certificates
- [ ] TLS handshake succeeds
- [ ] Private key never in source code

Task 1.4: Basic MQTT Pub/Sub Test

Estimated Time: 2 hours
Priority: High
Dependencies: Task 1.2

Steps:

1. Implement status message publishing
2. Subscribe to command topic
3. Implement command callback handler
4. Test using AWS IoT Core MQTT test client
5. Send command from AWS console → verify ESP32 receives
6. Publish status from ESP32 → verify AWS receives

Test Messages:

```
// Publish from ESP32
Topic: motionplay/device-001/status
Payload: {"device_id": "device-001", "status": "online"}

// Publish from AWS (test command)
Topic: motionplay/device-001/commands
Payload: {"command": "get_status"}
```

Acceptance Criteria:

- [ ] ESP32 publishes status every 30 seconds
- [ ] Status visible in AWS IoT Core MQTT test client
- [ ] Commands sent from AWS received by ESP32
- [ ] Commands logged to serial console

Task 1.5: Display Status Updates

Estimated Time: 1.5 hours
Priority: Medium
Dependencies: Task 1.1, Task 1.2

Steps:

1. Create `display_manager.cpp` module
2. Initialize TFT_eSPI for T-Display
3. Create status screen layout
4. Display WiFi connection status
5. Display MQTT connection status
6. Update display on Core 1 (non-blocking)

Display Layout:

```
┌────────────────────────────┐
│ Motion Play v1.0           │
├────────────────────────────┤
│ WiFi: Connected (-45)      │
│ Cloud: Connected           │
│ Mode: IDLE                 │
│                            │
│ Status: Ready              │
└────────────────────────────┘
```

Acceptance Criteria:

- [ ] Display shows connection status
- [ ] Status updates in real-time
- [ ] Display doesn't block main loop
- [ ] Text is readable

Phase 1 Deliverables

- [ ] ESP32 reliably connects to WiFi
- [ ] ESP32 maintains MQTT connection to AWS IoT Core
- [ ] Bidirectional communication verified
- [ ] Status displayed on T-Display
- [ ] Code is modular and documented

---

5. Phase 2: Data Pipeline

Duration: 2 weeks (10-14 hours)
Goal: Implement complete data collection and storage flow

Tasks

Task 2.1: Sensor Data Collection (Core 0)

Estimated Time: 3 hours
Priority: Critical
Dependencies: Existing sensor code

Steps:

1. Create `sensor_manager.cpp` module for Core 0
2. Implement high-frequency sensor reading loop
3. Create FreeRTOS queue for data transfer
4. Package readings into SensorReading struct
5. Send to queue without blocking
6. Test sustained 1000 Hz sampling

Code Structure:

```
struct SensorReading {
    uint32_t timestamp_ms;
    uint8_t position;
    uint8_t sensor_index;
    uint16_t proximity;
    uint16_t ambient;
};

QueueHandle_t sensorDataQueue;
```

Acceptance Criteria:

- [ ] Sensors read at 1000 Hz consistently
- [ ] Data queued to Core 1 without blocking
- [ ] No sensor read failures
- [ ] Memory usage stable

Task 2.2: Data Buffering and Session Management

Estimated Time: 3 hours
Priority: Critical
Dependencies: Task 2.1

Steps:

1. Implement session state machine
2. Allocate buffer in PSRAM (dynamic or pre-allocated)
3. Implement start_collection command handler
4. Buffer sensor data during collection
5. Implement stop_collection command handler
6. Track session metadata (start time, count, etc.)

State Machine:

```
IDLE → (start_collection) → COLLECTING → (stop_collection) → UPLOADING → IDLE
```

Acceptance Criteria:

- [ ] Can start/stop collection via command
- [ ] Data buffered correctly during collection
- [ ] Session metadata tracked
- [ ] Buffer doesn't overflow (<30s sessions)

Task 2.3: Data Serialization and Transmission

Estimated Time: 2.5 hours
Priority: Critical
Dependencies: Task 2.2

Steps:

1. Implement JSON serialization using ArduinoJson
2. Create batch payload structure (from TDD)
3. Implement MQTT publish for data payload
4. Handle large payloads (may need chunking)
5. Test with various session lengths (5s, 15s, 30s)

Payload Format:

```
{
    "session_id": "uuid",
    "device_id": "device-001",
    "start_timestamp": "2025-11-07T10:30:00Z",
    "duration_ms": 5000,
    "sample_rate": 1000,
    "readings": [...]
}
```

Acceptance Criteria:

- [ ] Data correctly serialized to JSON
- [ ] Published to MQTT successfully
- [ ] No data corruption
- [ ] Handles 30-second sessions without memory issues

Task 2.4: Lambda Function - Data Processor

Estimated Time: 3 hours
Priority: Critical
Dependencies: Task 0.6

Steps:

1. Create Lambda function in Node.js/TypeScript
2. Set up local development environment
3. Implement data validation logic

4. Parse incoming MQTT payload
5. Write session metadata to Sessions table
6. Write readings to SensorData table
7. Handle errors and logging
8. Deploy to AWS

Function Structure:

```
export async function handler(event: IoTEvent) {
    // Validate payload
    // Generate session_id if needed
    // Write to DynamoDB
    // Return success/error
}
```

Acceptance Criteria:

- [ ] Lambda function deploys successfully
- [ ] Validates incoming data
- [ ] Writes to DynamoDB correctly
- [ ] Logs errors to CloudWatch
- [ ] Handles malformed payloads gracefully

Task 2.5: IoT Rules Configuration

Estimated Time: 1 hour
Priority: Critical
Dependencies: Task 2.4

Steps:

1. Create IoT Rule for data topic
2. Configure SQL query: `SELECT * FROM 'motionplay/+/data'`
3. Add Lambda action to trigger processData function
4. Create IoT Rule for status topic
5. Test rules fire correctly
6. Monitor CloudWatch logs

Acceptance Criteria:

- [ ] IoT Rules created and enabled
- [ ] Lambda triggered on MQTT publish
- [ ] Data flows ESP32 → IoT → Lambda → DynamoDB

- [ ] No rule errors in console

Task 2.6: End-to-End Data Flow Test

Estimated Time: 2 hours
Priority: High
Dependencies: Tasks 2.1-2.5

Steps:

1. Manually trigger start_collection via AWS IoT console
2. Wave hand in front of sensors
3. Manually trigger stop_collection
4. Verify data published to MQTT
5. Verify Lambda invoked
6. Verify data in DynamoDB
7. Query and inspect stored data

Test Scenarios:

- 5-second collection
- 15-second collection
- 30-second collection
- Collection with no sensor data (edge case)

Acceptance Criteria:

- [ ] Complete data flow working
- [ ] Data accuracy verified
- [ ] Timestamps correct
- [ ] All 6 sensors reporting

Phase 2 Deliverables

- [ ] ESP32 collects and buffers sensor data
- [ ] Data transmitted to AWS on command
- [ ] Lambda processes and stores data
- [ ] Data queryable in DynamoDB
- [ ] End-to-end flow tested and working

---

6. Phase 3: Web Interface

Duration: 2 weeks (12-16 hours)

Goal: Create functional web interface for device control and data viewing

Tasks

Task 3.1: API Gateway Setup

Estimated Time: 2 hours
Priority: Critical
Dependencies: Task 0.1

Steps:

1. Create REST API in API Gateway
2. Configure CORS for localhost
3. Create resources and methods (from TDD)
4. Set up Lambda proxy integrations
5. Deploy to dev stage
6. Test endpoints with Postman/curl
7. Document API base URL

Endpoints to Create:

- POST /devices/{device_id}/commands
- GET /sessions
- GET /sessions/{session_id}
- PATCH /sessions/{session_id}
- DELETE /sessions/{session_id}

Acceptance Criteria:

- [ ] API Gateway deployed
- [ ] All endpoints created
- [ ] CORS configured correctly
- [ ] Can call endpoints from Postman

Task 3.2: Lambda Functions - API Handlers

Estimated Time: 4 hours
Priority: Critical
Dependencies: Task 3.1, Task 0.6

Steps:

1. Create Lambda function: sendCommand
2. Create Lambda function: getSessions
3. Create Lambda function: getSessionData
4. Create Lambda function: updateSession
5. Create Lambda function: deleteSession
6. Implement DynamoDB queries
7. Implement IoT publish for commands
8. Deploy all functions
9. Connect to API Gateway

Acceptance Criteria:

- [ ] All Lambda functions deployed
- [ ] Functions connected to API Gateway
- [ ] DynamoDB operations working
- [ ] MQTT commands published successfully
- [ ] Error handling implemented

Task 3.3: Frontend Project Structure

Estimated Time: 1.5 hours
Priority: High
Dependencies: Task 0.4

Steps:

1. Set up folder structure (from TDD)
2. Create TypeScript interfaces
3. Create API client service
4. Set up React Router (if needed)
5. Configure environment variables
6. Create basic layout component

Acceptance Criteria:

- [ ] Folder structure organized
- [ ] TypeScript types defined
- [ ] API client configured
- [ ] Environment variables working

Task 3.4: Device Control Component

Estimated Time: 3 hours
Priority: High
Dependencies: Task 3.2, Task 3.3

Steps:

1. Create DeviceStatus component
2. Create ModeSelector component
3. Create CollectionControl component
4. Implement start/stop collection buttons
5. Display connection status
6. Handle loading and error states
7. Test with real device

UI Elements:

- Connection indicator (green/red)
- Mode display (Debug/Play/Idle)
- Start Collection button
- Stop Collection button
- Status messages

Acceptance Criteria:

- [ ] Can send start_collection command
- [ ] Can send stop_collection command
- [ ] Device responds to commands
- [ ] UI shows loading states
- [ ] Errors displayed to user

Task 3.5: Session List Component

Estimated Time: 2.5 hours
Priority: High
Dependencies: Task 3.2, Task 3.3

Steps:

1. Create SessionList component
2. Fetch sessions from API
3. Display sessions in table/list
4. Show session metadata (time, duration, labels)
5. Implement click to view details
6. Add delete functionality

7. Add basic filtering (by date)

Display Fields:

- Session ID (truncated)
- Start time
- Duration
- Sample count
- Labels (tags)
- Actions (view, delete)

Acceptance Criteria:

- [ ] Sessions load from API
- [ ] List displays correctly
- [ ] Can click to view details
- [ ] Can delete sessions
- [ ] Loading and empty states handled

Task 3.6: Session Detail Component

Estimated Time: 3 hours
Priority: High
Dependencies: Task 3.5

Steps:

1. Create SessionDetail component
2. Fetch session data from API
3. Display session metadata
4. Create SensorChart component using Recharts
5. Plot proximity values over time
6. Add label editing functionality
7. Add notes editing
8. Implement export (JSON/CSV)

Chart Features:

- Line chart with 6 lines (one per sensor)
- X-axis: time (ms)
- Y-axis: proximity value
- Legend identifying sensors
- Zoom/pan (if time permits)

Acceptance Criteria:

- [ ] Session details load correctly
- [ ] Chart displays sensor data
- [ ] All 6 sensors visible
- [ ] Can add/edit labels
- [ ] Can add/edit notes
- [ ] Can export data

Task 3.7: Styling and Polish

Estimated Time: 2 hours
Priority: Medium
Dependencies: Tasks 3.4-3.6

Steps:

1. Apply Tailwind styling consistently
2. Improve layout and spacing
3. Add loading spinners
4. Improve error messages
5. Make responsive (desktop focus)
6. Add keyboard shortcuts (optional)
7. Test in Chrome/Firefox/Safari

Acceptance Criteria:

- [ ] UI looks clean and professional
- [ ] Consistent spacing and colors
- [ ] Loading states clear
- [ ] Works in major browsers
- [ ] No console errors

Phase 3 Deliverables

- [ ] Web interface operational
- [ ] Can control device remotely
- [ ] Can view session list
- [ ] Can view session details with chart
- [ ] Can label and annotate sessions
- [ ] Can export data

---

7. Phase 4: Integration & Polish

Duration: 2 weeks (8-12 hours)
Goal: Complete integration, testing, and polish for Phase 1 release

Tasks

Task 4.1: End-to-End Integration Testing

Estimated Time: 3 hours
Priority: Critical
Dependencies: All previous phases

Test Scenarios:

1. Complete collection workflow
2. Multiple sessions in sequence
3. Session labeling and editing
4. Data export and verification
5. Error scenarios (WiFi disconnect, etc.)
6. Concurrent operations (if applicable)

Test Plan:

```
1. Power on device
2. Verify connection status in UI
3. Start collection from UI
4. Wave hand in front of sensors
5. Stop collection from UI
6. Verify data appears in session list
7. View session details
8. Add labels
9. Export data
10. Verify exported data accuracy
```

Acceptance Criteria:

- [ ] All test scenarios pass
- [ ] No critical bugs found
- [ ] Data accuracy verified
- [ ] Performance acceptable

Task 4.2: Error Handling Improvements

Estimated Time: 2 hours
Priority: High
Dependencies: Task 4.1

Focus Areas:

1. Network disconnection during collection
2. AWS service failures (Lambda timeout, DynamoDB throttling)
3. Invalid user inputs
4. Large session handling
5. Concurrent collection attempts

Improvements:

- Better error messages
- Retry logic where appropriate
- Graceful degradation
- User guidance on errors

Acceptance Criteria:

- [ ] Key error scenarios handled
- [ ] Error messages are actionable
- [ ] System recovers gracefully
- [ ] No crashes or hangs

Task 4.3: Documentation Updates

Estimated Time: 2 hours
Priority: Medium
Dependencies: Task 4.1

Documents to Update:

1. README.md with setup instructions
2. ESP32 firmware README
3. Frontend README
4. AWS setup guide
5. API documentation
6. Troubleshooting guide

Acceptance Criteria:

- [ ] Setup instructions complete
- [ ] All READMEs updated
- [ ] Common issues documented
- [ ] API endpoints documented

Task 4.4: Performance Optimization

Estimated Time: 2 hours
Priority: Medium
Dependencies: Task 4.1

Optimization Areas:

1. ESP32 memory usage
2. Lambda cold start time
3. DynamoDB query efficiency
4. Frontend bundle size
5. Chart rendering performance

Measurements:

- Baseline performance metrics
- Target improvements
- Final measurements

Acceptance Criteria:

- [ ] No memory leaks on ESP32
- [ ] Lambda cold starts <3s
- [ ] Session list loads <3s
- [ ] Chart renders smoothly

Task 4.5: Security Hardening

Estimated Time: 1.5 hours
Priority: High
Dependencies: All previous phases

Security Checklist:

1. Verify certificates not in source code
2. Verify no secrets in environment variables
3. Review IAM permissions (least privilege)
4. Test certificate rotation process

5. Review error messages (no sensitive info leaking)
6. API input validation

Acceptance Criteria:

- [ ] No secrets in code repository
- [ ] IAM policies follow least privilege
- [ ] Certificates properly secured
- [ ] Input validation in place

Task 4.6: Monitoring and Logging Setup

Estimated Time: 1.5 hours
Priority: Medium
Dependencies: All previous phases

Setup:

1. CloudWatch dashboard for key metrics
2. Lambda function logs organized
3. DynamoDB metrics visible
4. IoT Core connection monitoring
5. Cost monitoring

Key Metrics:

- Device connection status
- Data collection frequency
- Lambda invocation count and errors
- DynamoDB read/write units
- Estimated monthly cost

Acceptance Criteria:

- [ ] CloudWatch dashboard created
- [ ] Key metrics visible
- [ ] Logs easily searchable
- [ ] Cost tracking enabled

Task 4.7: User Testing and Feedback

Estimated Time: 2 hours
Priority: Medium
Dependencies: Task 4.1

Testing Activities:

1. Perform complete workflows as end user
2. Document any confusion or issues
3. Identify UX improvements
4. Test edge cases
5. Verify acceptance criteria from Requirements Doc

Focus Areas:

- Ease of use
- Clarity of interface
- Speed of operations
- Reliability

Acceptance Criteria:

- [ ] All acceptance criteria from Requirements Doc met
- [ ] User workflow smooth and intuitive
- [ ] No blocking issues
- [ ] System ready for regular use

Phase 4 Deliverables

- [ ] Fully integrated and tested system
- [ ] Documentation complete
- [ ] Performance acceptable
- [ ] Monitoring in place
- [ ] Ready for Phase 1 production use

---

8. Risk Management

High-Priority Risks

Risk	Probability	Impact	Mitigation Strategy	Contingency Plan
MQTT connection unstable	Medium	High	Test thoroughly, implement reconnection	Add local data buffering
ESP32 memory overflow	Medium	High	Monitor memory usage, test limits	Reduce buffer size or sample rate

Risk	Probability	Impact	Mitigation Strategy	Contingency Plan
Lambda cold starts too slow	Low	Medium	Minimize dependencies	Consider provisioned concurrency
DynamoDB costs exceed budget	Low	Medium	Monitor costs weekly	Implement data cleanup
Certificate provisioning issues	Medium	Medium	Follow AWS docs carefully	Use AWS support resources
Time estimate too optimistic	High	Low	Build buffer into schedule	Reduce scope if needed

Risk Response Plan

Weekly Risk Review:

- Check AWS costs
- Monitor system stability
- Review outstanding issues
- Adjust plan if needed

Escalation Criteria:

- AWS costs >$30/month
- Critical bug blocking progress >3 days
- Schedule slip >2 weeks

---

9. Resource Requirements

Development Tools

Required:

- VSCode with PlatformIO extension
- AWS CLI
- Node.js 18+ and npm
- Git
- Web browser (Chrome/Firefox)

Optional but Helpful:

- Postman or similar API testing tool
- Serial monitor tool (alternative to PlatformIO)

- DynamoDB GUI tool (NoSQL Workbench)

AWS Resources (Estimated Costs)

Service	Monthly Cost (est.)
IoT Core	$5
Lambda	<$1 (free tier)
DynamoDB	$3-5
API Gateway	$1-2
CloudWatch	<$1
Data Transfer	<$1
Total	**$10-15**

Hardware Requirements

- ESP32-S3 T-Display (already owned)
- USB-C cable for programming
- Reliable WiFi network
- VCNL4040 sensors (already on PCB)

---

10. Quality Metrics

Success Criteria

Technical Metrics:

- Device uptime: >95%
- Data collection success rate: >98%
- End-to-end latency: <10 seconds
- Frontend load time: <3 seconds

Functional Metrics:

- All requirements from Requirements Doc met
- All acceptance criteria passed
- Zero critical bugs
- Documentation complete

User Experience:

- Can complete collection workflow in <60 seconds
- Clear feedback on all actions
- Intuitive interface requiring no training

---

11. Communication and Reporting

Weekly Check-In (Self)

Questions to Answer:

1. What did I complete this week?
2. What blockers did I encounter?
3. What's planned for next week?
4. Are we on track for timeline?
5. Any risks or concerns?

Milestone Reviews

After Each Phase:

- Review deliverables against plan
- Document lessons learned
- Adjust subsequent phases if needed
- Celebrate progress!

---

12. Maintenance Plan (Post Phase 1)

Regular Maintenance Tasks

Weekly:

- Review AWS costs
- Check system health metrics
- Review CloudWatch logs for errors

Monthly:

- Review and clean up old test sessions

- Update dependencies (security patches)
- Review and optimize costs

Quarterly:

- Firmware updates if needed
- Review architecture for Phase 2 readiness
- Backup critical data

---

13. Phase 2 Preview

Future Enhancements (Not in Phase 1)

1. **Multi-Device Support**
    - Device clustering
    - Coordinated data collection
    - Group management UI
2. **ML Pipeline**
    - Automated training data export
    - Model training workflow
    - On-device inference
3. **Advanced Features**
    - Real-time streaming visualization
    - WebSocket for live updates
    - Mobile app
    - User authentication
4. **Production Features**
    - OTA firmware updates
    - Advanced monitoring
    - Alerting system
    - User management

Estimated Timeline: 2-3 months additional

---

14. Appendix

Appendix A: Task Dependencies Graph

```
Phase 0: Setup
├── 0.1 AWS Account → 0.5, 0.6, 2.4, 3.1
├── 0.3 ESP32 Env → 1.1
├── 0.4 Frontend Env → 3.3
├── 0.5 IoT Core → 1.2, 1.3
└── 0.6 DynamoDB → 2.4, 3.2

Phase 1: Connectivity
├── 1.1 WiFi → 1.2, 1.5
├── 1.2 MQTT → 1.4
├── 1.3 Certificates → 1.2
└── 1.4 Pub/Sub → 2.2

Phase 2: Data Pipeline
├── 2.1 Sensor Reading → 2.2
├── 2.2 Buffering → 2.3
├── 2.3 Transmission → 2.6
├── 2.4 Lambda → 2.5, 2.6
├── 2.5 IoT Rules → 2.6
└── 2.6 E2E Test → 3.2

Phase 3: Web Interface
├── 3.1 API Gateway → 3.2
├── 3.2 Lambda APIs → 3.4, 3.5
├── 3.3 Frontend Structure → 3.4
├── 3.4 Device Control → 4.1
├── 3.5 Session List → 3.6
├── 3.6 Session Detail → 4.1
└── 3.7 Styling → 4.1

Phase 4: Integration
All tasks depend on Phase 3 completion
```

Appendix B: Time Tracking Template

```
Task: [Task Number and Name]
Estimated: [X hours]
Actual: [Y hours]
Variance: [Y-X hours]
```

```
Notes:
- What went well
- What took longer than expected
- Lessons learned
```

Appendix C: Testing Checklist

Pre-Deployment:

- [ ] Code compiles without warnings
- [ ] All unit tests pass
- [ ] Manual testing completed
- [ ] Documentation updated
- [ ] No secrets in code
- [ ] Code committed to git

Post-Deployment:

- [ ] Smoke test passes
- [ ] Monitoring shows normal behavior
- [ ] No errors in logs
- [ ] Cost within budget

Appendix D: Useful Commands

ESP32:

```
# Build
pio run

# Upload
pio run --target upload

# Monitor
pio device monitor

# Clean
pio run --target clean
```

AWS:

```
# List IoT things
aws iot list-things

# Test MQTT
aws iot-data publish --topic "motionplay/device-001/commands" --payload '

# Query DynamoDB
aws dynamodb scan --table-name MotionPlaySessions --max-items 10

# View Lambda logs
aws logs tail /aws/lambda/processData --follow
```

Frontend:

```
# Development
npm run dev

# Build
npm run build

# Preview build
npm run preview
```

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 implementation plan