Contents

Service Architecture Guide (v1.0.5)

This comprehensive guide documents the service-oriented architecture implemented in the Adlar Heat Pump Homey app, providing patterns, best practices, and implementation details for working with the 8 specialized services managed by ServiceCoordinator.

Table of Contents

Overview

Why Service-Oriented Architecture?

The Adlar Heat Pump app transitioned from a monolithic device class (v0.99.22 and earlier) to a service-oriented architecture (v0.99.23+) to address:

Problems Solved:

1. **Code Duplication**: Repeated patterns across device instances
2. **Testing Difficulty**: Monolithic class hard to unit test
3. **Maintenance Burden**: Changes required touching multiple locations
4. **Unclear Responsibilities**: Single class handling too many concerns
5. **Extension Challenges**: Adding features required modifying existing code

Benefits Achieved:

1. **Code Reusability**: Services centralize shared functionality
2. **Single Responsibility**: Each service handles one specific domain
3. **Testability**: Services can be unit tested independently
4. **Maintainability**: Changes isolated to relevant service
5. **Extensibility**: New services added without modifying existing ones
6. **Fallback Safety**: Graceful degradation when services unavailable

Architecture Principles

1. **Separation of Concerns**: Each service handles one specific domain (connection, health, calculations, etc.)
2. **Event-Driven Communication**: Services communicate via events, avoiding tight coupling
3. **Centralized Coordination**: ServiceCoordinator manages initialization, lifecycle, and events
4. **Service Independence**: Services can function with degraded dependencies
5. **Consistent Patterns**: All services follow same initialization, lifecycle, and error handling patterns

---

Service Catalog

Infrastructure Services (5)

1. TuyaConnectionService **File**: `lib/services/tuya-connection-service.ts`

Responsibility: Device communication via TuyAPI library

Key Features:

- Manages TuyAPI connection lifecycle (connect, disconnect, reconnect)
- Automatic reconnection with configurable interval (20 seconds)
- Connection health monitoring and diagnostics
- Real-time connection status tracking (v0.99.47) - 4 states: connected, disconnected, reconnecting, error
- Event-driven sensor data updates (DPS changes)
- Error categorization and recovery (via TuyaErrorCategorizer)
- Crash-proof error recovery (v0.99.46) - Triple-layer protection with unhandled promise rejection prevention
- Deep socket error interception (v0.99.49) - Intercepts TuyAPI internal socket ECONNRESET errors
- Automatic device availability status sync (unavailable during outages, available on reconnect)
- Idempotent error handler installation with listener cleanup
- **Heartbeat monitoring** (v0.99.98) - Proactive zombie connection detection every 5 minutes
- **Intelligent skip logic** (v0.99.98) - Avoids heartbeat when device active (recent data within 4 minutes)
- **Hybrid heartbeat approach** (v1.0.9) - Two-layer probing: passive get() then active set() wake-up
- **Sleep mode awareness** (v1.0.9) - Distinguishes sleeping devices from true disconnects
- **Stale connection force-reconnect** (v0.99.98) - Automatic reconnect after 10 minutes idle
- **Single-source connection truth** (v0.99.99) - Eliminates timer conflicts and race conditions
- **Persistent outage tracking** (v1.0.5) - Tracks cumulative outage duration independent of circuit breaker resets
- **Circuit breaker cycle limit** (v1.0.5) - Maximum 3 cycles (15 min) before switching to slow continuous retry
- **Internet recovery detection** (v1.0.5) - DNS probes every 30s during cooldown for immediate reconnection
- **User-visible outage timer** (v1.0.5) - Connection status shows outage duration and circuit breaker countdown
- **Time-based notifications** (v1.0.5) - Notifications at 2, 10, and 30 minutes instead of failure-count-based
- **Native heartbeat monitoring** (v1.1.2) - Layer 0 detection via TuyaAPI's built-in heartbeat events (35s timeout, fastest zombie detection)

Public Interface:

```
class TuyaConnectionService {
  async connect(deviceConfig): Promise<void>;
  async disconnect(): Promise<void>;
  async set(dps: number, value: any): Promise<void>;
  isConnected(): boolean;
```

```
  getConnectionStatus(): 'connected' | 'disconnected' | 'reconnecting' | 'error'; // v0.99.47
  getConnectionHealth(): ConnectionHealth;
  on(event: 'data' | 'connected' | 'disconnected' | 'error', handler): void;

  // Heartbeat mechanism (v0.99.98+) - Private methods called internally
  private startHeartbeat(): void;
  private stopHeartbeat(): void;
  private async performHeartbeat(): Promise<void>;
}
```

Dependencies: None (leaf service)

Events Emitted:

- `data` - DPS value changed (sensor update)
- `connected` - Connection established
- `disconnected` - Connection lost
- `error` - Connection or communication error

2. CapabilityHealthService **File**: `lib/services/capability-health-service.ts`

Responsibility: Real-time capability health tracking

Key Features:

- Tracks null value counts per capability (threshold: 10 consecutive nulls)
- Monitors data availability (timeout: 5 minutes without update)
- Classifies capabilities as healthy or unhealthy
- Provides diagnostic reports for troubleshooting
- Enables health-based flow card registration

Public Interface:

```
class CapabilityHealthService {
  startMonitoring(): void;
  stopMonitoring(): void;
  getHealthyCapabilities(): string[];
  getCapabilitiesWithRecentData(): string[];
  isCapabilityHealthy(capability: string): boolean;
  generateDiagnosticReport(): string;
}
```

Dependencies: None (monitors device capabilities)

Used By: FlowCardManagerService (auto mode registration)

3. FlowCardManagerService **File**: `lib/services/flow-card-manager-service.ts`

Responsibility: Dynamic flow card registration and management

Key Features:

- Manages 71 flow cards across 8 categories
- Three-mode control per category (disabled/auto/enabled)
- Health-based auto-registration (queries CapabilityHealthService)
- User preference management via SettingsManagerService
- Dynamic registration on settings changes

Public Interface:

```
class FlowCardManagerService {
  async registerFlowCards(settings): Promise<void>;
  async updateFlowCardRegistration(newSettings): Promise<void>;
  async shouldRegisterCategory(category, userSetting): Promise<boolean>;
  getRegisteredCategories(): string[];
}
```

Dependencies:

- CapabilityHealthService (health status for auto mode)
- SettingsManagerService (user preferences)

Flow Card Categories:

1. `flow_temperature_alerts` (11 cards)
2. `flow_voltage_alerts` (3 cards)
3. `flow_current_alerts` (3 cards)
4. `flow_power_alerts` (3 cards)
5. `flow_pulse_steps_alerts` (2 cards)
6. `flow_state_alerts` (5 cards)
7. `flow_efficiency_alerts` (3 cards)
8. `flow_expert_mode` (3 cards)

4. EnergyTrackingService   File: `lib/services/energy-tracking-service.ts`

Responsibility: External power measurement integration and validation

Key Features:

- Receives external data via flow cards (power, flow, ambient temperature)
- Validates data ranges and null checks
- Caches external data with timestamps (5-minute TTL)
- Provides fresh data to COPCalculator on request
- Manages power capability visibility

Public Interface:

```
class EnergyTrackingService {
  receiveExternalPower(power: number): void;
  receiveExternalFlow(flow: number): void;
  receiveExternalAmbient(temperature: number): void;
  getExternalPower(): { value: number; timestamp: number } | null;
  hasRecentExternalData(dataType: string): boolean;
}
```

Dependencies: None (receives flow card data)

Used By: COPCalculator (external data for Method 1: Direct Thermal)

5. SettingsManagerService   File: `lib/services/settings-manager-service.ts`

Responsibility: Settings validation, persistence, and race condition prevention

Key Features:

- Deferred settings updates pattern (prevents Homey race conditions)
- Validates settings before application
- Power settings auto-management cascade
- Seasonal data persistence (SCOP, rolling COP buffers)
- Single settings call consolidation

Public Interface:

```
class SettingsManagerService {
  validateSettings(settings): { valid: boolean; errors: string[] };
  preparePowerSettingsUpdate(enablePower: boolean): object;
  applyDeferredSettings(settings): void;
  persistSeasonalData(data): Promise<void>;
  getStoredData(key: string): any;
}
```

Dependencies: None (manages device.setSettings)

Used By: All services (settings persistence), ServiceCoordinator (race prevention)

Calculation Services (3)

6. COPCalculator **File**: `lib/services/cop-calculator.ts`

Responsibility: Real-time COP calculations with 8 methods

Key Features:

- Automatic method selection (±5% to ±30% accuracy range)
- 8 calculation methods with quality hierarchy
- Compressor operation validation (COP = 0 when idle)
- Diagnostic feedback ("No Power", "No Flow", "No Temp Δ", etc.)
- Outlier detection (< 0.5 or > 8.0 COP flagged)
- Confidence levels (high/medium/low)

Public Interface:

```
class COPCalculator {
  startCalculations(): void;
  stopCalculations(): void;
  calculateCOP(): { value: number; method: string; confidence: string };
  on(event: 'cop-calculated', handler: (data: COPData) => void): void;
}
```

Dependencies:

- TuyaConnectionService (sensor data: temperatures, flow, frequencies)
- CapabilityHealthService (sensor validation)
- EnergyTrackingService (external power data for Method 1)

Events Emitted:

- `cop-calculated` - New COP value available (consumed by RollingCOPCalculator, SCOPCalculator)

Calculation Methods (Priority Order):

1. **Direct Thermal** (±5%) - External power meter + water flow
2. **Power Module Auto-Detection** (±8%) - Internal power calculation
3. **Power Estimation** (±10%) - Physics-based power modeling
4. **Refrigerant Circuit Analysis** (±12%) - Thermodynamic analysis
5. **Carnot Estimation** (±15%) - Theoretical efficiency
6. **Valve Position Correlation** (±20%) - Valve efficiency curves
7. **Temperature Difference** (±30%) - Basic fallback method

7. RollingCOPCalculator **File**: `lib/services/rolling-cop-calculator.ts`

Responsibility: Time-series COP analysis (daily/weekly/monthly)

Key Features:

- Circular buffer (1440 data points = 24h × 60min)
- Runtime-weighted averaging for accurate representation
- Trend detection (7 levels: strong improvement → significant decline)
- Idle period awareness (auto COP = 0 data points)
- Statistical outlier filtering (2.5 standard deviation threshold)
- Memory-efficient incremental updates (O(n) complexity)

Public Interface:

```
class RollingCOPCalculator {
  async initialize(): Promise<void>;
  addDataPoint(data: COPDataPoint): void;
  getDailyCOP(): number | null;
  getWeeklyCOP(): number | null;
  getMonthlyCOP(): number | null;
  getTrend(): string;
  getDiagnosticInfo(): object;
}
```

Dependencies:

- COPCalculator (subscribes to `cop-calculated` events)
- CapabilityHealthService (validates data point quality)
- SettingsManagerService (persists circular buffer)

Published Capabilities:

- `adlar_cop_daily` - 24-hour rolling average
- `adlar_cop_weekly` - 7-day rolling average
- `adlar_cop_monthly` - 30-day rolling average
- `adlar_cop_trend` - Text description (7 levels)

8. SCOPCalculator **File**: `lib/services/scop-calculator.ts`

Responsibility: Seasonal efficiency per EN 14825 European standard

Key Features:

- Temperature bin method (6 bins: -10°C to +20°C)
- Quality-weighted averaging (direct thermal = 100%, temp difference = 60%)
- Seasonal coverage tracking (Oct 1 - May 15, 228 days)
- Method contribution analysis (% per calculation method)
- Confidence levels (high/medium/low based on coverage and quality)

Public Interface:

```
class SCOPCalculator {
  async initialize(): Promise<void>;
  processCOPData(data: COPData): void;
  getSCOP(): number | null;
  getQualityScore(): string;
  getSeasonalCoverage(): number;
  getMethodContribution(): object;
}
```

Dependencies:

- COPCalculator (subscribes to `cop-calculated` events)
- SettingsManagerService (persists seasonal data)

Published Capabilities:

- `adlar_scop` - Seasonal COP average (2.0-6.0)
- `adlar_scop_quality` - Data quality indicator

ServiceCoordinator Pattern

Responsibilities

ServiceCoordinator (`lib/services/service-coordinator.ts`) is the single point of control for all services:

1. **Initialization**: Creates and initializes all 8 services in dependency order
2. **Lifecycle Management**: Coordinates startup, settings changes, and shutdown
3. **Event Wiring**: Connects service events (e.g., `cop-calculated` → RollingCOPCalculator)
4. **Service Access**: Provides getters for device class to access services
5. **Health Monitoring**: Tracks service health and diagnostics

Implementation

```typescript
export class ServiceCoordinator {
  private device: Device;
  private logger: any;

  // Infrastructure services
  private tuyaConnection: TuyaConnectionService | null = null;
  private capabilityHealth: CapabilityHealthService | null = null;
  private flowCardManager: FlowCardManagerService | null = null;
  private energyTracking: EnergyTrackingService | null = null;
  private settingsManager: SettingsManagerService | null = null;

  // Calculation services
  private copCalculator: COPCalculator | null = null;
  private rollingCOPCalculator: RollingCOPCalculator | null = null;
  private scopCalculator: SCOPCalculator | null = null;

  constructor(device: Device, logger: any) {
    this.device = device;
    this.logger = logger;
  }

  /**
   * Initialize all services in dependency order
   */
  async initialize(config: ServiceConfig): Promise<void> {
    try {
      // Initialize infrastructure services (no dependencies)
      this.tuyaConnection = new TuyaConnectionService(this.device, this.logger, config);
      this.capabilityHealth = new CapabilityHealthService(this.device, this.logger);
      this.settingsManager = new SettingsManagerService(this.device, this.logger);
      this.energyTracking = new EnergyTrackingService(this.device, this.logger);

      // Initialize calculation services (depend on infrastructure)
      this.copCalculator = new COPCalculator(this.device, this.logger, this);
      this.rollingCOPCalculator = new RollingCOPCalculator(this.device, this.logger, this);
```

```typescript
    this.scopCalculator = new SCOPCalculator(this.device, this.logger, this);

    // Initialize flow card manager (depends on health service)
    this.flowCardManager = new FlowCardManagerService(this.device, this.logger, this);

    // Connect services
    await this.tuyaConnection.connect(config.deviceConfig);
    this.capabilityHealth.startMonitoring();
    await this.copCalculator.startCalculations();
    await this.flowCardManager.registerFlowCards(config.settings);

    // Wire cross-service events
    this.wireServiceEvents();

    this.logger.log('[ServiceCoordinator] All services initialized successfully');
  } catch (error) {
    this.logger.error('[ServiceCoordinator] Initialization failed:', error);
    throw error;
  }
}

/**
 * Wire cross-service event communication
 */
private wireServiceEvents(): void {
  // COPCalculator → RollingCOPCalculator (data collection)
  this.copCalculator?.on('cop-calculated', (data) => {
    this.rollingCOPCalculator?.addDataPoint(data);
    this.scopCalculator?.processCOPData(data);
  });

  // TuyaConnectionService → Device (capability updates)
  this.tuyaConnection?.on('data', async (dps) => {
    await this.device.handleDPSUpdate(dps);
  });

  // TuyaConnectionService → Connection state
  this.tuyaConnection?.on('connected', () => {
    this.logger.log('[ServiceCoordinator] Device connected');
  });

  this.tuyaConnection?.on('disconnected', () => {
    this.logger.log('[ServiceCoordinator] Device disconnected - attempting reconnect');
  });
}

/**
 * Handle device settings changes
 */
async onSettings(oldSettings: any, newSettings: any, changedKeys: string[]): Promise<void> {
  try {
    // Power measurements toggle - coordinate across services
    if (changedKeys.includes('enable_power_measurements')) {
      const enablePower = newSettings.enable_power_measurements;
```

```
      // Prepare settings update via SettingsManagerService (race prevention)
      const settingsToUpdate = this.settingsManager?.preparePowerSettingsUpdate(enablePower);

      // Update flow card registration
      if (settingsToUpdate) {
        await this.flowCardManager?.updateFlowCardRegistration({
          ...newSettings,
          ...settingsToUpdate,
        });

        // Apply deferred settings
        this.settingsManager?.applyDeferredSettings(settingsToUpdate);
      }
    }

    // Flow card settings changed
    if (changedKeys.some((key) => key.startsWith('flow_'))) {
      await this.flowCardManager?.updateFlowCardRegistration(newSettings);
    }

    this.logger.log('[ServiceCoordinator] Settings updated successfully');
  } catch (error) {
    this.logger.error('[ServiceCoordinator] Settings update failed:', error);
    throw error;
  }
}

/**
 * Graceful shutdown of all services
 */
destroy(): void {
  this.logger.log('[ServiceCoordinator] Shutting down services...');

  this.copCalculator?.stopCalculations();
  this.capabilityHealth?.stopMonitoring();
  this.tuyaConnection?.disconnect();

  // Clear service references
  this.copCalculator = null;
  this.rollingCOPCalculator = null;
  this.scopCalculator = null;
  this.flowCardManager = null;
  this.capabilityHealth = null;
  this.energyTracking = null;
  this.settingsManager = null;
  this.tuyaConnection = null;

  this.logger.log('[ServiceCoordinator] All services destroyed');
}

/**
 * Service getters (dependency injection for device class)
 */
```

```
  getTuyaConnection(): TuyaConnectionService | null {
    return this.tuyaConnection;
  }

  getCapabilityHealth(): CapabilityHealthService | null {
    return this.capabilityHealth;
  }

  getFlowCardManager(): FlowCardManagerService | null {
    return this.flowCardManager;
  }

  getEnergyTracking(): EnergyTrackingService | null {
    return this.energyTracking;
  }

  getSettingsManager(): SettingsManagerService | null {
    return this.settingsManager;
  }

  getCOPCalculator(): COPCalculator | null {
    return this.copCalculator;
  }

  getRollingCOPCalculator(): RollingCOPCalculator | null {
    return this.rollingCOPCalculator;
  }

  getSCOPCalculator(): SCOPCalculator | null {
    return this.scopCalculator;
  }

  /**
   * Get service health diagnostics
   */
  getServiceHealth(): ServiceHealthStatus {
    return {
      tuyaConnection: this.tuyaConnection?.isConnected() ?? false,
      capabilityHealth: this.capabilityHealth !== null,
      flowCardManager: this.flowCardManager !== null,
      energyTracking: this.energyTracking !== null,
      settingsManager: this.settingsManager !== null,
      copCalculator: this.copCalculator !== null,
      rollingCOPCalculator: this.rollingCOPCalculator !== null,
      scopCalculator: this.scopCalculator !== null,
    };
  }
}
```

Cross-Service Communication

Event-Driven Pattern

Services communicate via events to avoid tight coupling:

Example: COP Calculation Event Chain

```
// 1. TuyaConnectionService receives sensor update
class TuyaConnectionService {
  private handleDPSUpdate(dps: object): void {
    this.emit('data', dps); // Emit to device
  }
}

// 2. Device updates capabilities, triggering COPCalculator
class Device {
  async handleDPSUpdate(dps: object): Promise<void> {
    await this.updateCapabilities(dps);

    // Trigger COP calculation on relevant sensor changes
    if (this.isCOPRelevantUpdate(dps)) {
      this.serviceCoordinator?.getCOPCalculator()?.calculateCOP();
    }
  }
}

// 3. COPCalculator calculates and emits event
class COPCalculator {
  calculateCOP(): void {
    const result = this.performCalculation();

    this.emit('cop-calculated', {
      timestamp: Date.now(),
      cop: result.value,
      method: result.method,
      confidence: result.confidence,
      compressorRuntime: this.getRuntime(),
    });
  }
}

// 4. RollingCOPCalculator subscribes to event
class RollingCOPCalculator {
  initialize(): void {
    this.serviceCoordinator
      .getCOPCalculator()
      .on('cop-calculated', (data) => {
        this.addDataPoint(data);
        this.updateRollingAverages();
      });
  }
}

// 5. SCOPCalculator also subscribes
class SCOPCalculator {
```

```
  initialize(): void {
    this.serviceCoordinator
      .getCOPCalculator()
      .on('cop-calculated', (data) => {
        this.processCOPData(data);
      });
  }
}
```

Service Dependency Graph

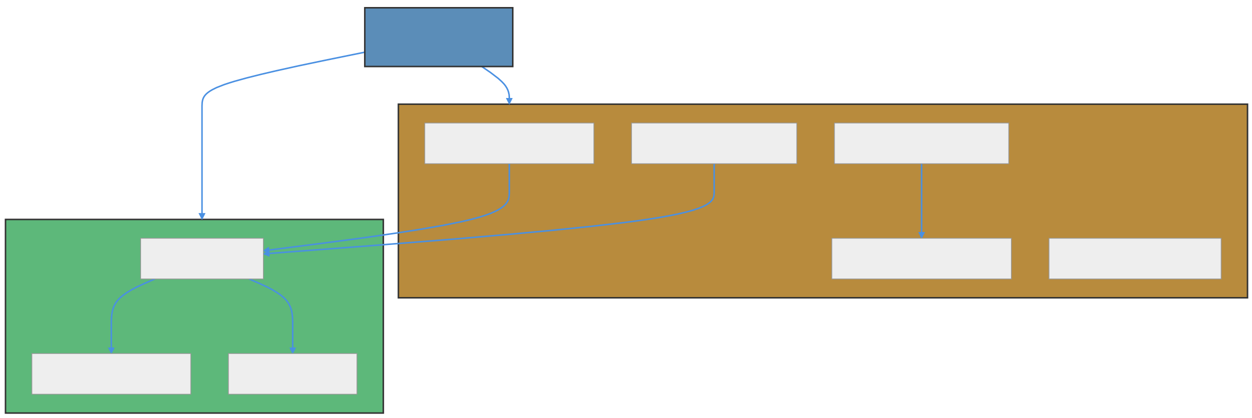

Figure 1: Diagram 1

Service Lifecycle Management

Initialization Sequence

1. **ServiceCoordinator** created in `device.onInit()`
2. **Infrastructure Services** initialized first (no dependencies)
3. **Calculation Services** initialized second (depend on infrastructure)
4. **FlowCardManager** initialized last (depends on CapabilityHealth)
5. **Cross-Service Events** wired after all services initialized
6. **TuyaConnection** connect called to establish device communication

Settings Changes

1. User changes setting in Homey app
2. Homey calls `device.onSettings(oldSettings, newSettings, changedKeys)`
3. Device delegates to `serviceCoordinator.onSettings()`
4. ServiceCoordinator identifies affected services
5. **SettingsManagerService** prepares deferred updates (race prevention)
6. Affected services updated (e.g., FlowCardManager re-registers cards)
7. **SettingsManagerService** applies deferred settings after completion

Device Shutdown

1. Device removed or app restarted
2. Device calls `this.serviceCoordinator.destroy()`
3. Services shut down in reverse order:

    - COPCalculator stops calculations
    - CapabilityHealth stops monitoring
    - TuyaConnection disconnects
4. Service references cleared (garbage collection)

Adding New Services

Step 1: Create Service File

File: `lib/services/my-new-service.ts`

```typescript
import Homey from 'homey';
import { Device } from '../device';
import { ServiceCoordinator } from './service-coordinator';

export class MyNewService extends Homey.SimpleClass {
  private device: Device;
  private logger: any;
  private serviceCoordinator: ServiceCoordinator | null;

  constructor(device: Device, logger: any, serviceCoordinator?: ServiceCoordinator) {
    super();
    this.device = device;
    this.logger = logger;
    this.serviceCoordinator = serviceCoordinator || null;
  }

  /**
   * Initialize service
   */
  async initialize(): Promise<void> {
    this.logger.log('[MyNewService] Initializing...');

    // Service initialization logic here

    this.logger.log('[MyNewService] Initialized successfully');
  }

  /**
   * Graceful shutdown
   */
  destroy(): void {
    this.logger.log('[MyNewService] Shutting down...');

    // Cleanup logic here
  }

  /**
   * Public service methods
   */
  public doSomething(): void {
    // Implementation
  }
}
```

Step 2: Add to ServiceCoordinator

```typescript
// Add private property
private myNewService: MyNewService | null = null;

// Initialize in initialize() method
this.myNewService = new MyNewService(this.device, this.logger, this);
await this.myNewService.initialize();

// Add getter
getMyNewService(): MyNewService | null {
  return this.myNewService;
}

// Clear in destroy() method
this.myNewService = null;
```

Step 3: Wire Events (If Needed)

```typescript
// In wireServiceEvents() method
this.someService?.on('some-event', (data) => {
  this.myNewService?.handleEvent(data);
});
```

Step 4: Update Device Class

```typescript
// Use service via ServiceCoordinator
class Device extends Homey.Device {
  someMethod(): void {
    this.serviceCoordinator?.getMyNewService()?.doSomething();
  }
}
```

Service Testing

Unit Testing Pattern

File: `test/services/my-new-service.test.ts`

```typescript
import { MyNewService } from '../../lib/services/my-new-service';

describe('MyNewService', () => {
  let service: MyNewService;
  let mockDevice: any;
  let mockLogger: any;
  let mockServiceCoordinator: any;

  beforeEach(() => {
    mockDevice = {
      // Mock device methods
    };

    mockLogger = {
      log: jest.fn(),
      error: jest.fn(),
```

```
    };

    mockServiceCoordinator = {
      // Mock service coordinator methods
      getCapabilityHealth: jest.fn(),
    };

    service = new MyNewService(mockDevice, mockLogger, mockServiceCoordinator);
  });

  afterEach(() => {
    service.destroy();
  });

  describe('initialize', () => {
    it('should initialize service successfully', async () => {
      await service.initialize();

      expect(mockLogger.log).toHaveBeenCalledWith(
        expect.stringContaining('Initialized successfully'),
      );
    });
  });

  describe('doSomething', () => {
    it('should perform action correctly', () => {
      // Test implementation
    });
  });
});
```

Integration Testing

Test cross-service communication via ServiceCoordinator:

```
describe('ServiceCoordinator Integration', () => {
  it('should wire COPCalculator to RollingCOPCalculator', async () => {
    const coordinator = new ServiceCoordinator(mockDevice, mockLogger);
    await coordinator.initialize(config);

    const copCalculator = coordinator.getCOPCalculator();
    const rollingCalculator = coordinator.getRollingCOPCalculator();

    // Trigger COP calculation
    copCalculator.calculateCOP();

    // Verify RollingCOPCalculator received event
    expect(rollingCalculator.getDataPoints().length).toBeGreaterThan(0);
  });
});
```

Best Practices

Service Design

1. **Single Responsibility**: Each service handles ONE domain (connection, health, calculations, etc.)
2. **Minimal Dependencies**: Services depend only on what they absolutely need
3. **Event-Driven**: Use events for cross-service communication, not direct method calls
4. **Null-Safe**: Always check service availability (`this.serviceCoordinator?.getService()`)
5. **Graceful Degradation**: Service unavailability shouldn't crash the app

Error Handling

```
//   Good: Graceful degradation
const healthService = this.serviceCoordinator?.getCapabilityHealth();
if (healthService) {
  const healthyCapabilities = healthService.getHealthyCapabilities();
  // Use healthy capabilities
} else {
  // Fallback: assume all capabilities available
  this.logger.warn('[Service] CapabilityHealth unavailable - using fallback');
}

//   Bad: Assumes service availability
const healthyCapabilities = this.serviceCoordinator
  .getCapabilityHealth() // Could be null!
  .getHealthyCapabilities();
```

Constants Integration

Always use `DeviceConstants` instead of magic numbers:

```
//   Good: Use centralized constants
import { DeviceConstants } from '../constants';

setTimeout(() => {
  this.reconnect();
}, DeviceConstants.RECONNECTION_INTERVAL_MS);

//   Bad: Magic numbers
setTimeout(() => {
  this.reconnect();
}, 20000); // What does 20000 mean?
```

Service Communication

```
//   Good: Event-driven communication
this.copCalculator.on('cop-calculated', (data) => {
  this.rollingCOPCalculator.addDataPoint(data);
});

//   Bad: Direct service calls (tight coupling)
class COPCalculator {
  calculateCOP(): void {
    const result = this.performCalculation();
    this.serviceCoordinator.getRollingCOPCalculator().addDataPoint(result); // Too coupled!
  }
}
```

Troubleshooting

Service Not Initialized

Symptom: `Cannot read property 'method' of null`

Cause: Accessing service before ServiceCoordinator initialization

Solution:

```
// Always check service availability
const service = this.serviceCoordinator?.getMyService();
if (!service) {
  this.logger.warn('[Device] MyService not available');
  return;
}
service.method();
```

Events Not Firing

Symptom: Cross-service events don't trigger subscribed handlers

Causes:

1. Event wiring not called in `wireServiceEvents()`
2. Service not initialized before wiring events
3. Event name mismatch

Solution:

```
// Verify event wiring in ServiceCoordinator.wireServiceEvents()
private wireServiceEvents(): void {
  this.copCalculator?.on('cop-calculated', (data) => {
    this.logger.log('[ServiceCoordinator] COP event received:', data);
    this.rollingCOPCalculator?.addDataPoint(data);
  });
}
```

Race Conditions in Settings

Symptom: "Cannot set Settings while this.onSettings is still pending"

Cause: Multiple `setSettings()` calls during `onSettings()` lifecycle

Solution: Use SettingsManagerService deferred update pattern:

```
async onSettings(oldSettings, newSettings, changedKeys): Promise<void> {
  // Prepare deferred updates
  const settingsToUpdate = this.settingsManager?.preparePowerSettingsUpdate(
    newSettings.enable_power_measurements,
  );

  // Update services synchronously
  await this.updateServices(newSettings);

  // Apply deferred settings AFTER onSettings completes
  this.settingsManager?.applyDeferredSettings(settingsToUpdate);
}
```

Service Memory Leaks

Symptom: Memory usage grows over time

Causes:

1. Event listeners not removed on destroy
2. Timers not cleared
3. Circular buffer not cleaned

Solution:

```
class MyService {
  private timerId: NodeJS.Timeout | null = null;

  initialize(): void {
    this.timerId = setInterval(() => {
      this.doWork();
    }, 60000);

    this.someService.on('event', this.handleEvent);
  }

  destroy(): void {
    // Clear timers
    if (this.timerId) {
      clearInterval(this.timerId);
      this.timerId = null;
    }

    // Remove event listeners
    this.someService.off('event', this.handleEvent);
  }
}
```

Dual Picker/Sensor Architecture (v0.99.54+)

Overview

The app implements a **dual picker/sensor architecture** for curve control capabilities, enabling a single DPS to update multiple capabilities simultaneously. This architecture resolves the iPhone picker bug while providing enhanced UX through always-visible status displays with optional user controls.

Multi-Capability DPS Mapping

Traditional Approach (Pre-v0.99.54):

```
// One DPS → One capability (allArraysSwapped pattern)
DPS 11 → adlar_enum_capacity_set (picker only)
DPS 13 → adlar_enum_countdown_set (sensor only)
```

New Multi-Capability Mapping (v0.99.54+):

```
// One DPS → Multiple capabilities (dpsToCapabilities pattern)
DPS 11 → adlar_enum_capacity_set (picker) + adlar_sensor_capacity_set (sensor)
DPS 13 → adlar_enum_countdown_set (sensor) + adlar_picker_countdown_set (picker)
```

AdlarMapping Enhancement

File: `lib/definitions/adlar-mapping.ts`

New Primary Mapping System (Lines 102-133):

```typescript
/**
 * Multi-capability DPS mapping (v0.99.54+)
 *
 * Maps each DPS ID to an array of ALL capabilities that should be updated when that DPS changes.
 * This enables dual picker/sensor architecture where one DPS updates multiple capabilities.
 *
 * IMPORTANT: This is the PRIMARY mapping for DPS-to-capability updates.
 * Use this instead of allArraysSwapped for multi-capability support.
 */
static dpsToCapabilities: Record<number, string[]> = (() => {
  const mapping: Record<number, string[]> = {};

  // Build mapping from allCapabilities - each DPS gets an array of capabilities
  Object.entries(AdlarMapping.allCapabilities).forEach(([capability, dpsArray]) => {
    const dpsId = dpsArray[0];

    if (!mapping[dpsId]) {
      mapping[dpsId] = [];
    }

    // Add capability to array (allows multiple capabilities per DPS)
    mapping[dpsId].push(capability);
  });

  return mapping;
})();
```

Key Features:

- **Auto-Generated**: Mapping built automatically from `allCapabilities`
- **Backward Compatible**: Single-capability DPS have arrays with one element
- **Type-Safe**: TypeScript ensures correct DPS ID and capability name matching
- **Extensible**: New dual capabilities added by declaring them in `adlarCapabilities`

Device Update Logic

Enhanced `updateCapabilitiesFromDps()` (device.ts:2140-2175):

```typescript
private updateCapabilitiesFromDps(dpsFetched: Record<number, unknown>): void {
  Object.entries(dpsFetched).forEach(([dpsIdStr, value]) => {
    const dpsId = Number(dpsIdStr);

    // Use NEW multi-capability mapping (v0.99.54+)
    const capabilities = AdlarMapping.dpsToCapabilities[dpsId];

    if (!capabilities || capabilities.length === 0) {
      this.log(`No capability mapping for DPS ${dpsId}`);
      return;
    }

    // Update ALL capabilities mapped to this DPS
```

```
    capabilities.forEach((capability) => {
      if (this.hasCapability(capability)) {
        this.setCapabilityValue(capability, value)
          .then(() => {
            this.log(` Updated ${capability} to ${value} (DPS ${dpsId})`);

            // Notify CapabilityHealthService about update
            this.serviceCoordinator
              ?.getCapabilityHealth()
              ?.updateCapabilityHealth(capability, value);
          })
          .catch((err) => {
            this.error(`Failed to update ${capability}:`, err);
          });
      }
    });
  });
}
```

Flow:

1. **DPS Change Received** from Tuya device (e.g., DPS 11 = "H2")
2. **Multi-Capability Lookup** via `dpsToCapabilities[11]`
3. **Returns Array** `['adlar_enum_capacity_set', 'adlar_sensor_capacity_set']`
4. **Updates Both Capabilities** with same value from single DPS
5. **Health Tracking** notifies CapabilityHealthService for each capability
6. **Data Consistency** guaranteed - both capabilities always synchronized

User Control Setting

Setting Definition (driver.settings.compose.json):

```json
{
  "id": "enable_curve_controls",
  "type": "checkbox",
  "label": {
    "en": "Show curve picker controls in device UI",
    "nl": "Toon curve picker besturing in apparaat UI"
  },
  "value": false,
  "hint": {
    "en": "Show picker controls for heating and hot water curves in device UI. When disabled, only senso
    "nl": "Toon picker besturing voor verwarmings- en warmwatercurves in apparaat UI. Wanneer uitgeschak
  }
}
```

Capability Visibility Matrix:

Setting State	Sensor Capabilities (Always Visible)	Picker Capabilities (Conditional)	Flow Cards
Disabled (Default)	`adlar_enum_countdown_setadlar_sensor_capacity_set`	Hidden	Active
Enabled	`adlar_enum_countdown_setadlar_sensor_capacity_set`	`adlar_picker_countdown_setadlar_enum_capacity_set`	Active

Architecture Benefits

1. **Always-Visible Status**: Users always see current curve settings via sensor capabilities

2. **Optional Control**: Advanced users can enable picker controls when needed
3. **Data Consistency**: Single DPS update maintains perfect sync between sensor and picker
4. **Flow Card Independence**: Automation works regardless of UI picker visibility setting
5. **Reduced UI Clutter**: Default installation shows read-only values only (cleaner interface)
6. **User Choice**: Power users can enable full control via device settings
7. **iPhone Bug Resolution**: Solves picker crash issue by making pickers optional
8. **Backward Compatible**: Existing devices upgrade automatically with migration logic

Automatic Capability Migration

Migration Logic (device.ts:2489-2510):

```typescript
// Add missing curve sensor capabilities for existing devices (v0.99.54 migration)
if (!this.hasCapability('adlar_sensor_capacity_set')) {
  await this.addCapability('adlar_sensor_capacity_set');
  this.log('  Added adlar_sensor_capacity_set capability (hot water curve sensor)');
}

if (!this.hasCapability('adlar_picker_countdown_set')) {
  await this.addCapability('adlar_picker_countdown_set');
  this.log('  Added adlar_picker_countdown_set capability (heating curve picker)');
}

// Initialize values from existing capabilities
const currentHotWater = this.getCapabilityValue('adlar_enum_capacity_set');
if (currentHotWater !== null) {
  await this.setCapabilityValue('adlar_sensor_capacity_set', currentHotWater);
}

const currentHeating = this.getCapabilityValue('adlar_enum_countdown_set');
if (currentHeating !== null) {
  await this.setCapabilityValue('adlar_picker_countdown_set', currentHeating);
}
```

Migration Features:

- Detects missing capabilities during `onInit()`
- Adds new sensor/picker capabilities automatically
- Copies current values from existing capabilities
- Zero user intervention required
- Preserves existing curve settings during upgrade

Usage Pattern for Developers

Adding New Dual Capability:

1. **Define Both Capabilities** in `adlarCapabilities` (adlar-mapping.ts):

```typescript
static adlarCapabilities: Record<string, number[]> = {
  // Sensor capability (always visible)
  my_sensor_capability: [42],

  // Picker capability (conditional visibility)
  my_picker_capability: [42],  // Same DPS ID!
};
```

2. **Define Capability JSON Files** in `.homeycompose/capabilities/`:

```
// my_sensor_capability.json
{
  "type": "enum",
  "title": { "en": "My Sensor" },
  "getable": true,
  "setable": false,  // Read-only sensor
  "uiComponent": "sensor",
  "values": [...]
}

// my_picker_capability.json
{
  "type": "enum",
  "title": { "en": "My Control" },
  "getable": true,
  "setable": true,   // User can change
  "uiComponent": "picker",
  "values": [...]
}
```

3. **`dpsToCapabilities` Auto-Generates** the mapping:

```
// Automatic result:
dpsToCapabilities[42] = ['my_sensor_capability', 'my_picker_capability']
```

4. **Device Update Logic Handles** the rest automatically!

Testing Multi-Capability Updates

```
describe('Multi-Capability DPS Updates', () => {
  it('should update both sensor and picker when DPS 11 changes', async () => {
    const device = new MyDevice();
    await device.onInit();

    // Simulate DPS 11 change from Tuya device
    device.updateCapabilitiesFromDps({ 11: 'H3' });

    // Verify BOTH capabilities updated
    expect(device.getCapabilityValue('adlar_enum_capacity_set')).toBe('H3');
    expect(device.getCapabilityValue('adlar_sensor_capacity_set')).toBe('H3');
  });

  it('should maintain data consistency across capabilities', () => {
    // Both capabilities should always have identical values
    const sensorValue = device.getCapabilityValue('adlar_sensor_capacity_set');
    const pickerValue = device.getCapabilityValue('adlar_enum_capacity_set');

    expect(sensorValue).toBe(pickerValue);
  });
});
```

Production-Ready Enhancements (v0.99.46-v0.99.49)

TuyaConnectionService Updates

The TuyaConnectionService has been significantly enhanced with production-ready features for crash prevention and real-time connection monitoring.

Crash Prevention (v0.99.46)

Triple-Layer Error Protection:

```
// Layer 1: Specific .catch() handlers on async setTimeout callbacks
setTimeout(async () => {
  try {
    await this.reconnect();
  } catch (err) {
    this.logger('Reconnection failed:', err);
  }
}, DeviceConstants.RECONNECTION_INTERVAL_MS).catch((err) => {
  // CRITICAL: Prevents unhandled promise rejection crashes
  this.logger('  Async setTimeout error caught:', err);
});

// Layer 2: Device status sync (5 consecutive failures)
if (this.consecutiveFailures >= DeviceConstants.MAX_CONSECUTIVE_FAILURES) {
  await this.device.setUnavailable('Connection lost - attempting reconnection');
}

// On successful reconnection:
await this.device.setAvailable();
this.consecutiveFailures = 0;

// Layer 3: Global process handlers (app.ts)
process.on('unhandledRejection', (reason) => {
  this.error('  UNHANDLED PROMISE REJECTION prevented app crash:', reason);
});
```

Real-Time Connection Status (v0.99.47)

Four Connection States:

```
type ConnectionStatus = 'connected' | 'disconnected' | 'reconnecting' | 'error';

class TuyaConnectionService {
  private currentStatus: ConnectionStatus = 'disconnected';

  // Status updates at all transition points
  async connectTuya(): Promise<void> {
    this.currentStatus = 'reconnecting';

    try {
      await this.tuya.connect();
      this.currentStatus = 'connected';
    } catch (err) {
      this.currentStatus = 'error';
    }
  }

  getConnectionStatus(): ConnectionStatus {
    return this.currentStatus;
```

```
  }
}
```

Device Integration:

```
// Device polls connection status every 5 seconds
setInterval(() => {
  const status = this.serviceCoordinator?.getTuyaConnection()?.getConnectionStatus();

  if (status && this.hasCapability('adlar_connection_status')) {
    this.setCapabilityValue('adlar_connection_status', status);
  }
}, 5000);
```

Deep Socket Error Handler (v0.99.49) CRITICAL FIX for ECONNRESET errors:

```
/**
 * Install deep socket error handler (v0.99.49)
 *
 * TIMING CRITICAL: Must be called AFTER this.tuya.connect()
 * TuyAPI only creates the internal .device object DURING connect(), not in constructor
 */
private installDeepSocketErrorHandler(): void {
  if (!this.tuya || !(this.tuya as any).device) {
    this.logger(' Cannot install socket handler - TuyAPI .device not created yet');
    return;
  }

  const tuyaDevice = (this.tuya as any).device;

  // Remove existing error listeners (idempotent installation)
  tuyaDevice.removeAllListeners('error');

  // Install new handler with crash protection
  tuyaDevice.on('error', (err: Error) => {
    this.logger(' Deep socket error intercepted:', err.message);

    // Categorize and handle error
    const categorizedError = TuyaErrorCategorizer.categorize(err, 'Socket');

    if (categorizedError.shouldReconnect) {
      this.currentStatus = 'reconnecting';
      this.scheduleReconnection();
    }
  });

  this.logger(' Deep socket error handler installed');
}
```

Installation Points:

1. After initial connection in `initialize()`
2. After every successful reconnection in `connectTuya()`

Why v0.99.48 Failed:

```
//  v0.99.48 - WRONG: Handler installed BEFORE connect
```

```
this.tuya = new TuyAPI({ ... });
this.installDeepSocketErrorHandler();  // .device doesn't exist yet!
await this.tuya.connect();

//  v0.99.49 - CORRECT: Handler installed AFTER connect
this.tuya = new TuyAPI({ ... });
await this.tuya.connect();             // .device created HERE
this.installDeepSocketErrorHandler();   // Now .device exists
```

Updated TuyaConnectionService Interface

```
class TuyaConnectionService {
  async connect(deviceConfig): Promise<void>;
  async disconnect(): Promise<void>;
  async set(dps: number, value: any): Promise<void>;

  // Connection state (v0.99.47)
  isConnected(): boolean;
  getConnectionStatus(): 'connected' | 'disconnected' | 'reconnecting' | 'error';

  // Connection health (v0.99.46)
  getConnectionHealth(): ConnectionHealth;

  // Events
  on(event: 'data' | 'connected' | 'disconnected' | 'error', handler): void;

  // Heartbeat mechanism (v0.99.98-v0.99.99) - Private methods (internal use only)
  private startHeartbeat(): void;                     // Start 5-minute heartbeat timer
  private stopHeartbeat(): void;                      // Stop heartbeat timer
  private async performHeartbeat(): Promise<void>;   // Execute heartbeat probe with intelligent skip lo
}
```

Layer 0: Native Heartbeat Monitoring (v1.1.2)

The fastest disconnection detection mechanism, leveraging TuyaAPI's built-in heartbeat events for immediate zombie connection detection.

Architecture **Location**: `lib/services/tuya-connection-service.ts:966-972, 1063-1115`

Core Components:

```
// Event listener - registers TuyaAPI's native heartbeat events
this.tuya.on('heartbeat', (): void => {
  this.lastNativeHeartbeatTime = Date.now();
  this.logger(' Native heartbeat received');
});

// Monitoring - started automatically on connection
private startNativeHeartbeatMonitoring(): void {
  // Clear any existing monitor
  this.stopNativeHeartbeatMonitoring();

  // Initialize timestamp
  this.lastNativeHeartbeatTime = Date.now();

  this.logger(' Starting Layer 0 native heartbeat monitoring (timeout: 35s)');
```

```
  // Check every 10 seconds if native heartbeats have stopped
  this.nativeHeartbeatMonitorInterval = this.device.homey.setInterval(() => {
    if (!this.isConnected) return; // Skip if already disconnected

    const timeSinceLastHeartbeat = Date.now() - this.lastNativeHeartbeatTime;

    if (timeSinceLastHeartbeat > this.NATIVE_HEARTBEAT_TIMEOUT_MS) { // 35 seconds
      this.logger(`  Layer 0: Native heartbeat timeout (${Math.round(timeSinceLastHeartbeat / 1000)}s)`)
      this.logger('  Layer 0: Zombie connection detected - forcing reconnect');

      // Mark as disconnected and trigger reconnection
      this.isConnected = false;
      this.lastDisconnectSource = `layer0_native_heartbeat_timeout`;
      this.consecutiveFailures++;
      this.scheduleNextReconnectionAttempt();
    }
  }, 10000); // Check every 10 seconds
}

private stopNativeHeartbeatMonitoring(): void {
  if (this.nativeHeartbeatMonitorInterval) {
    clearInterval(this.nativeHeartbeatMonitorInterval);
    this.nativeHeartbeatMonitorInterval = null;
  }
}
```

Key Features

1. **TuyaAPI Integration**: Listens to TuyaAPI's built-in `'heartbeat'` events sent every ~10 seconds
2. **Passive Monitoring**: No active network queries required (zero overhead)
3. **Fast Detection**: Detects zombie connections within 35 seconds
4. **Automatic Lifecycle**:
   - Started automatically on `'connected'` event (line 958)
   - Stopped automatically on `disconnect()` (line 1111-1115)
5. **Complementary Operation**: Works alongside hybrid heartbeat (Layer 1-3) and DPS refresh

Detection Speed Comparison

Layer	Detection Time	Method	Network Overhead	Status
Layer 0	**35 seconds**	Native TuyaAPI heartbeat events	None (passive)	v1.1.2
Layer 1	5 minutes	Hybrid heartbeat (get/set probes)	Low (conditional)	v1.0.9
Layer 2	5 minutes	DPS refresh (NAT keep-alive)	Low (periodic)	v1.0.3

Layer	Detection Time	Method	Network Overhead	Status
Layer 3	10 minutes	Stale connection force-reconnect	None (check only)	v0.99.98

Why Layer 0 is Critical **Speed Advantage**: 5-8x faster detection than Layer 1-3 mechanisms

- Pre-v1.1.2: 5-10 minute detection window
- Post-v1.1.2: 35-second detection window

Zero False Positives: If TuyaAPI heartbeats stop, connection is definitively dead

- No need for multi-layer probing
- No wake-up commands required
- Direct protocol-level signal

No Network Impact: Piggybacks on TuyaAPI's existing heartbeat protocol

- No additional `get()` or `set()` queries
- No bandwidth consumption
- No device wake-up side effects

Complements TCP Keep-Alive: TuyaAPI heartbeats align with 5-minute TCP keep-alive strategy

- OS-level: TCP keep-alive prevents NAT timeout
- Protocol-level: TuyaAPI heartbeats maintain application-level awareness
- App-level: Layer 0 monitors for gaps in protocol heartbeats

Integration with v1.0.31 Architecture Layer 0 fits seamlessly into the synergistic connection recovery architecture:

```
TCP Keep-Alive (5min, OS-level)
   Prevents NAT timeout at router level
  ↓
Layer 0: Native Heartbeat (35s detection) ← FASTEST
   TuyaAPI protocol-level heartbeat monitoring
   Zero overhead, immediate zombie detection
   Signals: isConnected = false → triggers reconnection loop
  ↓
Layer 1-3: App-Level Monitoring (5-10 min)
   Backup detection for edge cases
   Hybrid heartbeat (get/set) handles sleeping devices
   DPS refresh maintains NAT mapping
  ↓
Reconnection Loop (single source of truth)
   Handles all reconnection attempts
   Exponential backoff + circuit breaker
   Always reschedules (never breaks loop)
```

Implementation Timeline **v0.99.98-v1.0.30**: Only Layer 1-3 mechanisms existed

- Detection window: 5-10 minutes
- User impact: Extended "stuck connected" status

v1.1.2: Layer 0 added as primary detection

- Detection window: 35 seconds

- User impact: Near-immediate zombie detection
- Backward compatible: Layers 1-3 remain as backup

Lifecycle Management **Startup Sequence**:

```
// On successful connection (tuya.on('connected'))
this.isConnected = true;
this.startNativeHeartbeatMonitoring(); // ← Layer 0 activated
this.startHeartbeat();                  // ← Layer 1-3 activated
this.startPeriodicDpsRefresh();         // ← NAT keep-alive
```

Shutdown Sequence:

```
// On disconnect() or destroy()
this.stopNativeHeartbeatMonitoring(); // ← Layer 0 cleanup
this.stopHeartbeat();                  // ← Layer 1-3 cleanup
this.stopPeriodicDpsRefresh();         // ← NAT keep-alive cleanup
```

Practical Example **Scenario**: Router temporarily loses internet connectivity for 2 minutes

```
T+0s   - Device connected, heartbeats arriving every ~10s
T+10s  - Heartbeat received → lastNativeHeartbeatTime updated
T+20s  - Heartbeat received
T+30s  - Router loses internet → heartbeats stop
T+40s  - Layer 0 monitor checks: 10s since last heartbeat (OK)
T+50s  - Layer 0 monitor checks: 20s since last heartbeat (OK)
T+60s  - Layer 0 monitor checks: 30s since last heartbeat (OK)
T+70s  - Layer 0 monitor checks: 40s > 35s threshold
         → Zombie detected!
         → isConnected = false
         → scheduleNextReconnectionAttempt() triggered
T+75s  - Reconnection attempt begins
```

Without Layer 0: First detection would occur at T+5min (Layer 1) or T+10min (Layer 3)

With Layer 0: Detection at T+70s (35-second timeout after last heartbeat)

Testing Layer 0

```
describe('Layer 0: Native Heartbeat Monitoring', () => {
  it('should update timestamp on heartbeat event', () => {
    const service = new TuyaConnectionService(config);
    const beforeTime = Date.now();

    // Simulate TuyaAPI heartbeat event
    service.tuya.emit('heartbeat');

    expect(service.lastNativeHeartbeatTime).toBeGreaterThanOrEqual(beforeTime);
  });

  it('should detect zombie after 35s timeout', async () => {
    const service = new TuyaConnectionService(config);
    service.isConnected = true;

    // Simulate last heartbeat 40 seconds ago
    service.lastNativeHeartbeatTime = Date.now() - 40000;
```

```
    // Wait for monitor to check (runs every 10s)
    await new Promise(resolve => setTimeout(resolve, 11000));

    expect(service.isConnected).toBe(false);
    expect(service.lastDisconnectSource).toContain('layer0_native_heartbeat_timeout');
  });

  it('should stop monitoring on disconnect', () => {
    const service = new TuyaConnectionService(config);
    service.startNativeHeartbeatMonitoring();

    expect(service.nativeHeartbeatMonitorInterval).not.toBeNull();

    service.stopNativeHeartbeatMonitoring();

    expect(service.nativeHeartbeatMonitorInterval).toBeNull();
  });
});
```

Layer 0 Benefits

1. **Speed**: 5-8x faster detection (35s vs 5-10 min)
2. **Reliability**: Zero false positives (protocol-level signal)
3. **Efficiency**: No network overhead (event-driven)
4. **Simplicity**: Single event listener + timer (minimal complexity)
5. **Compatibility**: Complements existing Layer 1-3 mechanisms

---

Heartbeat Mechanism (v0.99.98-v0.99.99, Enhanced v1.0.9)

The heartbeat mechanism is a critical enhancement to TuyaConnectionService that proactively detects and resolves zombie connections during idle periods.

v1.0.9 Enhancement: Hybrid approach distinguishes between **sleeping devices** (responsive to commands but not queries) and **true disconnects** (unresponsive to all operations).

Problem Solved **Pre-v0.99.98 User Experience**: - Device status shows "Connected" in Homey UI - No sensor data updates for hours - TuyAPI connection silently failed (zombie state) - User must manually use "Force Reconnect" button - Unacceptable downtime for heating/cooling control

Root Cause: TuyAPI connections can enter zombie state where socket appears open but no data flows and no error events are emitted.

v1.0.9 Additional Problem: Devices entering sleep mode ignored passive `get()` queries but responded to active `set()` commands, causing false positive disconnects and unnecessary reconnection cascades.

Implementation Architecture **Three-Layer Detection System** (v1.0.9):

1. **Layer 1: Passive Query Probe**
   - Timer: Every 5 minutes (`CONNECTION_HEARTBEAT_INTERVAL_MS`)
   - Skip logic: Heartbeat skipped if device sent data within last 4 minutes
   - Probe method: `tuya.get({ schema: true })` with 10-second timeout
   - On success: Connection healthy, exit
   - On failure: Proceed to Layer 2
2. **Layer 2: Active Wake-Up Command** (NEW v1.0.9)
   - Trigger: Only when Layer 1 fails

  - Probe method: `tuya.set({ dps: 1, set: currentOnOffValue })` with 10-second timeout
  - Idempotent: Writes current value back (no side effects)
  - On success: Device was sleeping, now awake
  - On failure: True disconnect detected
3. **Layer 3: Stale Connection Force-Reconnect**
  - Backup detection in `scheduleNextReconnectionAttempt()`
  - Checks: If connected but no data for 10+ minutes
  - Action: Force disconnect and reconnect with moderate backoff

Heartbeat Probe Implementation (v1.0.9 Hybrid Approach)

```
private async performHeartbeat(): Promise<void> {
  // Early returns for efficiency
  if (!this.isConnected) return;
  if (this.heartbeatInProgress) return;

  // Intelligent skip: Avoid heartbeat if device recently active
  const timeSinceLastData = Date.now() - this.lastDataEventTime;
  if (timeSinceLastData < DeviceConstants.CONNECTION_HEARTBEAT_INTERVAL_MS * 0.8) {
    this.logger('Heartbeat skipped - device active (data within 4 min)');
    return;
  }

  // Device idle - probe connection health with hybrid approach
  this.heartbeatInProgress = true;

  try {
    // LAYER 1: Try passive get() first (network-friendly)
    try {
      await Promise.race([
        this.tuya.get({ schema: true }),
        new Promise((_, reject) =>
          setTimeout(() => reject(new Error('Heartbeat get() timeout')),
          DeviceConstants.HEARTBEAT_TIMEOUT_MS)
        )
      ]);

      // Success with get() - device is responsive
      this.logger(' Heartbeat (get) successful - connection healthy');
      this.lastDataEventTime = Date.now();
      return; // Exit early - connection is healthy

    } catch (getError) {
      // LAYER 2: get() failed - try active set() wake-up
      this.logger(' Heartbeat get() failed, attempting wake-up set()...');

      // Get current onoff state for idempotent write
      const currentOnOff = this.device.getCapabilityValue('onoff') || false;

      try {
        await Promise.race([
          this.tuya.set({ dps: 1, set: currentOnOff }), // Idempotent write
          new Promise((_, reject) =>
            setTimeout(() => reject(new Error('Heartbeat set() timeout')),
```

```
          DeviceConstants.HEARTBEAT_TIMEOUT_MS)
        )
      ]);

      // Success with set() - device was sleeping but is now awake
      this.logger('  Heartbeat (wake-up set) successful - device was sleeping');
      this.lastDataEventTime = Date.now();
      return; // Recovery successful!

    } catch (setError) {
      // Both layers failed - true disconnect
      throw new Error(`Both get() and set() failed - true disconnect`);
    }
  }

} catch (error) {
  // Both heartbeat layers failed - mark as disconnected
  this.logger('  Heartbeat completely failed - true disconnect detected');

  // Use standard error categorization
  const categorizedError = TuyaErrorCategorizer.categorize(
    error as Error,
    'Heartbeat probe'
  );

  // Trigger reconnection
  this.isConnected = false;
  this.consecutiveFailures++;
  this.scheduleNextReconnectionAttempt();

} finally {
  this.heartbeatInProgress = false;
}
}
```

v1.0.9 Key Changes: - Nested try-catch structure for two-layer probing - Layer 1 (get) executes first for efficiency - Layer 2 (set) only executes if Layer 1 fails - Idempotent write ensures no device state changes - DPS 1 (onoff) chosen for universal availability and safety

Connection Health Tracking **Three Timestamps Track Activity**:

```
private lastDataEventTime: number = Date.now();      // Last DPS update from device
private lastHeartbeatTime: number = Date.now();      // Last successful heartbeat
private lastStatusChangeTime: number = Date.now();   // Last status transition
```

Activity Detection:

```
const isDeviceActive =
  (Date.now() - this.lastDataEventTime < 4 * 60 * 1000) ||  // Data < 4 min
  (Date.now() - this.lastHeartbeatTime < 5 * 60 * 1000);    // Heartbeat < 5 min
```

Single-Source Connection Truth (v0.99.99 Critical Fix) **Problem in v0.99.98**: - Heartbeat timer and reconnection timer could both try to reconnect - Race conditions caused extended disconnection periods (20+ minutes) - Multiple concurrent reconnection attempts confused TuyAPI state machine

Solution:

```
private scheduleNextReconnectionAttempt(): void {
  // CRITICAL: Clear ALL existing timers first
  if (this.reconnectionTimer) {
    clearTimeout(this.reconnectionTimer);
    this.reconnectionTimer = null;
  }

  if (this.heartbeatTimer) {
    clearTimeout(this.heartbeatTimer);
    this.heartbeatTimer = null;
  }

  // Only schedule if actually disconnected
  if (this.isConnected) return;

  // ... schedule single new reconnection timer
}
```

Key Principle: Only ONE timer manages reconnection at any time. All existing timers cleared before scheduling new one.

Performance Characteristics Network Efficiency: - Skip rate with active device: 80-90% - Actual probes per day: ~144 (average every 10 minutes) - Probe size: ~100 bytes - Total daily bandwidth: ~14 KB

Detection Performance: - Minimum detection: 5 minutes (next heartbeat) - Maximum detection: 10 minutes (Layer 2 backup) - Average detection: 6-7 minutes - Pre-v0.99.98: Hours to never (manual intervention)

CPU Impact: - Timer overhead: <0.1% CPU - Probe execution: <1ms - Skip check: <0.01ms

Integration with ServiceCoordinator Automatic Lifecycle Management:

```
// ServiceCoordinator.initialize()
this.tuyaConnection = new TuyaConnectionService(device, logger, config);
await this.tuyaConnection.connect(config.deviceConfig);
// → Heartbeat started automatically after successful connection

// ServiceCoordinator.destroy()
this.tuyaConnection?.disconnect();
// → Heartbeat stopped automatically during disconnect
```

Developer Usage: - Heartbeat is fully automatic - no manual intervention needed - Started automatically after `connect()` succeeds - Stopped automatically during `disconnect()` - Private methods - not exposed in public interface - Integrated with existing error handling and reconnection systems

Testing Heartbeat Mechanism Unit Test Example:

```
describe('TuyaConnectionService Heartbeat', () => {
  it('should skip heartbeat when device is active', async () => {
    service.lastDataEventTime = Date.now() - (2 * 60 * 1000);  // 2 min ago

    await service.performHeartbeat();

    expect(service.heartbeatInProgress).toBe(false);
    expect(mockLogger.log).toHaveBeenCalledWith(
      expect.stringContaining('Heartbeat skipped')
    );
  });
```

```
  it('should detect zombie connection after idle period', async () => {
    service.lastDataEventTime = Date.now() - (6 * 60 * 1000);  // 6 min ago
    mockTuya.get.mockRejectedValue(new Error('ETIMEDOUT'));

    await service.performHeartbeat();

    expect(service.isConnected).toBe(false);
    expect(service.consecutiveFailures).toBeGreaterThan(0);
  });
});
```

Benefits Summary

1. **Automatic Recovery**: 5-10 minute detection vs hours (or never) previously
2. **Network Efficient**: 80-90% probe reduction via intelligent skip logic
3. **User Experience**: Eliminates manual "Force Reconnect" button usage
4. **Reliability**: Three-layer detection ensures no missed failures (v1.0.9)
5. **Integration**: Seamless with existing error handling
6. **Stability**: Single-source connection management prevents race conditions (v0.99.99)
7. **Performance**: Minimal CPU and bandwidth overhead
8. **Sleep Mode Aware** (v1.0.9): Distinguishes sleeping devices from true disconnects
9. **Transparent Wake-Up** (v1.0.9): Sleeping devices resume without user awareness
10. **False Positive Prevention** (v1.0.9): Avoids unnecessary reconnection cascades

v1.0.9 Specific Benefits **Scenario Comparison**:

Scenario	v0.99.99 Behavior	v1.0.9 Hybrid Behavior
Active Device	get() succeeds → (0s)	get() succeeds → (0s)
Sleeping Device	get() fails → Reconnect (20s+)	get() fails → set() succeeds → Wake-up (10s)
True Disconnect	get() fails → Reconnect (20s+)	get() fails → set() fails → Reconnect (20s)

User Impact: - **Before v1.0.9**: Sleeping device → "Disconnected" status → Reconnection cascade → User sees intermittent connectivity - **After v1.0.9**: Sleeping device → Wake-up transparent → "Connected" status maintained → Zero user awareness

Technical Impact: - False positive disconnects: Eliminated - Unnecessary reconnections: Prevented - Network efficiency: Improved (Layer 2 only runs when Layer 1 fails) - Device battery life: Better (wake-up targeted vs full reconnection)

Updated Flow Card Count (v0.99.56)

Total: 71 Flow Cards (Updated from 64):

- **Triggers**: 36 cards (was 35)
- **Conditions**: 23 cards (was 19)
- **Actions**: 12 cards (unchanged)

Categories:

1. `flow_temperature_alerts` - 11 trigger cards
2. `flow_voltage_alerts` - 3 trigger cards
3. `flow_current_alerts` - 3 trigger cards
4. `flow_power_alerts` - 3 trigger cards

5. `flow_pulse_steps_alerts` - 2 trigger cards
6. `flow_state_alerts` - 5 trigger cards
7. `flow_efficiency_alerts` - 3 trigger cards
8. `flow_expert_mode` - 3 trigger cards

TuyaConnectionService v1.0.5 Reconnection Improvements

Version 1.0.5 introduces comprehensive improvements to the reconnection mechanism, eliminating the need for manual intervention during extended internet outages.

Problem Statement (Pre-v1.0.5)

Reported Issue: Device remained disconnected for 5+ hours after internet outage without auto-recovery: - Circuit breaker entered infinite loop (fail 10x → 5min cooldown → reset → fail 10x → repeat) - No notifications after initial "Connection Lost" at failure #5 - Notification #15 ("Extended Outage") was unreachable due to counter resets - Manual `force_reconnect` button required to restore connection - User experience: "intolerable"

Root Cause: Circuit breaker pattern designed for *transient failures* (brief hiccups), not *sustained outages* (hours-long internet downtime).

Solution Architecture: 5 Integrated Proposals

Proposal 1: Persistent Outage Tracking **Implementation**: `tuya-connection-service.ts:57-58`

```
// Persistent outage tracking (v1.0.5 - Proposal 1)
private outageStartTime = 0; // When current outage began
private totalOutageDuration = 0; // Cumulative outage duration
```

Behavior: - Starts tracking when disconnect detected - Persists through circuit breaker resets - Only resets on successful reconnection - Enables accurate outage duration reporting

Use Cases: - Time-based notifications (Proposal 5) - User-visible outage timer (Proposal 4) - Diagnostic logging and analytics

---

Proposal 2: Circuit Breaker Cycle Limit **Implementation**: `tuya-connection-service.ts:61-62`, `tuya-connection-service.ts:973-989`

```
// Circuit breaker cycle limit (v1.0.5 - Proposal 2)
private circuitBreakerCycles = 0;
private readonly MAX_CIRCUIT_BREAKER_CYCLES = 3; // 3 cycles = 15 min total

// In scheduleNextReconnectionAttempt():
if (this.circuitBreakerCycles >= this.MAX_CIRCUIT_BREAKER_CYCLES) {
  // Max cycles reached - switch to continuous slow retry
  this.logger(' Max circuit breaker cycles reached - switching to slow continuous retry');
  this.circuitBreakerOpen = false;
  this.backoffMultiplier = 8; // 2.5 min retry interval (20s * 8 = 160s)
  // Keep trying indefinitely at slower rate
} else {
  // Reset circuit breaker for another cycle
  this.circuitBreakerOpen = false;
  this.consecutiveFailures = 0;
```

```
  this.backoffMultiplier = 1; // Reset backoff for fresh cycle
}
```

Behavior: - Allows **maximum 3 circuit breaker cycles** (3 × 5min = 15 minutes total) - After 3 cycles, switches to **slow continuous retry** mode: - Retry interval: 2.5 minutes (160 seconds) - No more circuit breaker cooldowns - Continues indefinitely until connection restored

Benefits: - Eliminates infinite cooldown loop - Guarantees continuous recovery attempts - Prevents hours-long disconnection periods

---

Proposal 3: Internet Recovery Detection During Cooldown **Implementation**: `tuya-connection-service.ts:947`

```
// Proposal 3: Lightweight connectivity probe every 30 seconds during cooldown
if (timeSinceOpen % 30000 < 10000) {
  dnsPromises.resolve('google.com')
    .then(() => {
      this.logger('  Internet recovered during cooldown - attempting immediate reconnection');
      this.circuitBreakerOpen = false;
      this.circuitBreakerCycles = 0;
      this.consecutiveFailures = 0;
      this.scheduleNextReconnectionAttempt();
    })
    .catch(() => {
      // Still offline, continue cooldown
    });
}
```

Behavior: - **Every 30 seconds** during circuit breaker cooldown: lightweight DNS probe - **If internet restored**: immediately exits cooldown and attempts reconnection - **If internet still down**: continues cooldown period

Benefits: - Recovery within seconds instead of waiting for full 5-minute cooldown - Minimal network overhead (DNS query only) - Works during any circuit breaker cycle

Example Timeline:

```
T+0:00  - Internet restored
T+0:15  - DNS probe detects recovery
T+0:16  - Circuit breaker reset
T+0:17  - Reconnection attempt starts
T+0:20  - Device connected
```

Versus old behavior: wait until T+5:00 for cooldown to expire.

Proposal 4: User-Visible Outage Timer **Implementation**: `tuya-connection-service.ts:399-409`

```
// Add context for circuit breaker or outage duration (v1.0.5)
let contextInfo = '';
if (this.currentStatus === 'reconnecting' && this.circuitBreakerOpen) {
  const remainingCooldown = Math.ceil(
    (this.circuitBreakerResetTime - (Date.now() - this.circuitBreakerOpenTime)) / 1000,
  );
  contextInfo = ` [retry in ${remainingCooldown}s]`;
} else if ((this.currentStatus === 'disconnected' || this.currentStatus === 'error') && this.outageStart
  const outageMinutes = Math.floor((Date.now() - this.outageStartTime) / 60000);
```

```
  contextInfo = ` [${outageMinutes} min]`;
}

return `${statusLabel}${contextInfo} (${timeString})`;
```

Display Examples:

```
"Connected (14:25:30)"
"Disconnected [5 min] (14:20:15)"
"Reconnecting [retry in 245s] (14:25:30)"
"Error [30 min] (14:00:00)"
```

Benefits: - Users see **real-time outage duration** - Circuit breaker countdown shows **when next retry happens** - Provides transparency into reconnection process - Reduces support requests ("how long has it been down?")

Proposal 5: Time-Based Outage Notifications **Implementation**: `tuya-connection-service.ts:906-936`

Old System (Failure-Count-Based):

```
//  PROBLEM: Unreachable after circuit breaker resets counter
if (consecutiveFailures === 5) {
  notify("Device Connection Lost");
}
if (consecutiveFailures === 15) {  // NEVER REACHED!
  notify("Extended Device Outage");
}
```

New System (Time-Based):

```
//  SOLUTION: Time-based notifications independent of circuit breaker
const outageDuration = Date.now() - this.outageStartTime;

if (outageDuration >= 2 * 60 * 1000 && !this.notificationSent2Min) {
  await this.sendCriticalNotification(
    'Device Connection Lost',
    'Heat pump has been offline for 2 minutes. Automatic recovery in progress.',
  );
  this.notificationSent2Min = true;
}

if (outageDuration >= 10 * 60 * 1000 && !this.notificationSent10Min) {
  await this.sendCriticalNotification(
    'Extended Device Outage',
    'Heat pump has been offline for 10 minutes. Please check network connectivity.',
  );
  this.notificationSent10Min = true;
}

if (outageDuration >= 30 * 60 * 1000 && !this.notificationSent30Min) {
  await this.sendCriticalNotification(
    'Critical Outage',
    'Heat pump has been offline for 30 minutes. Manual intervention may be required.',
  );
  this.notificationSent30Min = true;
}
```

Notification Timeline: - **T+2 min**: "Device Connection Lost - Automatic recovery in progress" - **T+10 min**: "Extended Device Outage - Please check network connectivity" - **T+30 min**: "Critical Outage - Manual intervention may be required"

Benefits: - Notifications **guaranteed** to arrive at specific outage durations - Works **independently** of circuit breaker state - Users stay informed during extended outages - Progressive escalation (info → warning → critical)

Integration & Interaction

The 5 proposals work together as a **unified system**:

Example Timeline: 5-Hour Internet Outage

Pre-v1.0.5 Behavior:

```
T+0:00  - Disconnect
T+4:23  - Notification #5 "Connection Lost"
T+4:40  - Circuit breaker (10 failures)
T+9:40  - Circuit breaker reset, counter → 0
T+14:03 - Circuit breaker again (10 failures)
T+19:03 - Circuit breaker reset, counter → 0
... infinite loop, no more notifications ...
T+5:00:00 - Still disconnected, user must force_reconnect
```

v1.0.5 Behavior:

```
T+0:00  - Disconnect, outage tracking started
T+2:00  - Notification "Connection Lost" (time-based)
T+4:40  - Circuit breaker cycle 1 (10 failures)
T+4:45  - DNS probe detects internet still down
T+5:15  - DNS probe detects internet still down
T+9:40  - Circuit breaker cycle 2
T+10:00 - Notification "Extended Outage" (time-based)
T+14:40 - Circuit breaker cycle 3
T+19:40 - Max cycles reached → slow continuous retry (2.5 min)
T+22:10 - Retry attempt (2.5 min interval)
T+24:40 - Retry attempt
T+27:10 - Retry attempt
T+29:40 - Retry attempt
T+30:00 - Notification "Critical Outage" (time-based)
T+32:10 - Retry attempt
...continues until internet restored...
T+5:00:00 - Internet restored
T+5:00:15 - DNS probe detects recovery
T+5:00:20 - Device reconnected automatically
```

Key Improvements: - 3 notifications instead of 1 - Continuous retry every 2.5 minutes after 15 minutes - Automatic recovery when internet restored - No manual intervention required

---

Performance & Resource Impact

Memory: - Additional tracking variables: ~24 bytes total - `outageStartTime`: 8 bytes (number) - `totalOutageDuration`: 8 bytes (number) - `circuitBreakerCycles`: 8 bytes (number) - Notification flags (3x boolean): negligible

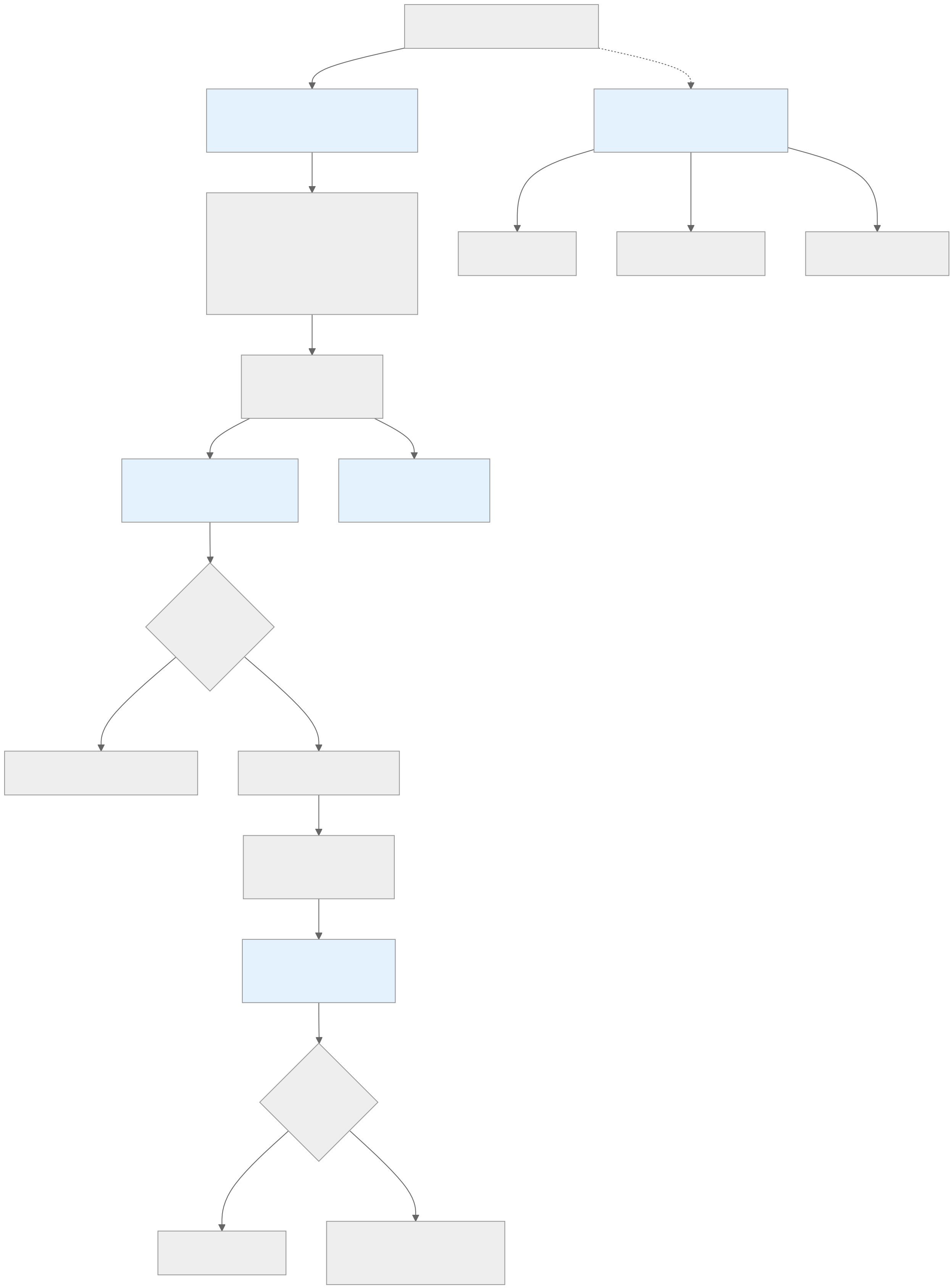

Figure 2: Diagram 2

Network Overhead: - DNS probes: ~100 bytes per query, every 30s during cooldown only - Max overhead during 15-min cooldown: ~30 probes = 3KB total - Negligible impact

CPU: - DNS queries: async, non-blocking - Time calculations: nanosecond-level operations - No measurable CPU impact

---

Testing & Validation

Unit Tests Required: 1. Persistent outage tracking across circuit breaker resets 2. Circuit breaker cycle limit enforcement 3. DNS probe success/failure handling 4. Notification timing accuracy 5. Status display format correctness

Integration Tests Required: 1. Full reconnection cycle with simulated internet outage 2. Circuit breaker → slow retry transition 3. Internet recovery during various cooldown phases 4. Notification delivery at correct timestamps

Manual Testing Scenarios: 1. **Short outage** (< 2 min): Should auto-recover without notifications 2. **Medium outage** (5-10 min): Should receive 1-2 notifications, auto-recover 3. **Extended outage** (30+ min): Should receive all 3 notifications, continuous retry 4. **Internet restoration during cooldown**: Should detect within 30s and reconnect

---

Migration Notes

Breaking Changes: None - fully backward compatible

Automatic Migration: - New tracking variables initialize to 0 on first run - Existing reconnection logic preserved - No settings changes required

User Impact: - Improved UX: status now shows outage duration - Better notification system: time-based instead of count-based - No action required from users

---

Conclusion

The service-oriented architecture provides a robust, maintainable, and extensible foundation for the Adlar Heat Pump Homey app. By following the patterns and best practices outlined in this guide, you can:

- Add new services without modifying existing code
- Test services independently with clear contracts
- Maintain clear separation of concerns
- Achieve graceful degradation when services are unavailable
- Coordinate complex cross-service interactions via ServiceCoordinator

For more information on specific services, see:

- Architecture Overview - High-level architecture summary
- COP Calculation - COPCalculator service details
- SCOP Calculation - SCOPCalculator service details
- Rolling COP - RollingCOPCalculator service details