Architecture Overview (v0.99.99+, verified through v1.0.31)

This document provides a comprehensive overview of the Adlar Heat Pump Homey app architecture, focusing on the Service Coordinator pattern, utility libraries, and core systems that provide reliability, maintainability, and enhanced user experience through intelligent insights management and dual picker/sensor architecture for curve controls.

Service-Oriented Architecture (v0.99.23+)

ServiceCoordinator Pattern (`lib/services/service-coordinator.ts`)

The app has been refactored from a monolithic device class to a service-oriented architecture using the ServiceCoordinator pattern. This eliminates code duplication and provides clear separation of concerns.

Architecture Overview

```
class MyDevice extends Homey.Device {
  private serviceCoordinator: ServiceCoordinator | null = null;

  async onInit() {
    // Initialize ServiceCoordinator with all managed services
    this.serviceCoordinator = new ServiceCoordinator({
      device: this,
      logger: this.debugLog.bind(this),
    });

    await this.serviceCoordinator.initialize({
      deviceConfig: tuyaConfig,
      settings: this.getSettings(),
    });
  }
}
```

Service Delegation Pattern

The device class now delegates functionality to specialized services:

Service	Purpose	Eliminates Duplication
TuyaConnectionService	Device communication & reconnection	✅ Connection logic
CapabilityHealthService	Health monitoring & diagnostics	✅ Health check patterns
FlowCardManagerService	Dynamic flow card management	✅ Flow card registration
EnergyTrackingService	Energy calculations & tracking	✅ Energy computation
SettingsManagerService	Settings validation & race condition prevention	✅ Settings handling

COPCalculator	Real-time COP calculations with 8 methods	✅ Efficiency calculation logic
SCOPCalculator	Seasonal efficiency per EN 14825 standard	✅ SCOP calculation patterns
RollingCOPCalculator	Time-series COP analysis (daily/weekly/monthly)	✅ Rolling average computations

ServiceCoordinator Interface

```
export class ServiceCoordinator {
  // Service getters for delegation
  getTuyaConnection(): TuyaConnectionService;
  getCapabilityHealth(): CapabilityHealthService;
  getFlowCardManager(): FlowCardManagerService;
  getEnergyTracking(): EnergyTrackingService;
  getSettingsManager(): SettingsManagerService;
  getCOPCalculator(): COPCalculator;
  getSCOPCalculator(): SCOPCalculator;
  getRollingCOPCalculator(): RollingCOPCalculator;

  // Unified lifecycle management
  async initialize(config: ServiceConfig): Promise<void>;
  async onSettings(oldSettings: any, newSettings: any, changedKeys: string[]):
Promise<void>;
  destroy(): void;

  // Health monitoring
  getServiceHealth(): ServiceHealthStatus | null;
  getServiceDiagnostics(): ServiceDiagnostics | null;
}
```

Fallback Pattern for Backward Compatibility

The device maintains fallback methods for graceful degradation:

```
private async sendTuyaCommand(dp: number, value: string | number | boolean):
Promise<void> {
  // Try ServiceCoordinator's TuyaConnectionService first
  if (this.serviceCoordinator) {
    try {
      const tuyaService = this.serviceCoordinator.getTuyaConnection();
      await tuyaService.sendCommand({[dp]: value});
      return;
    } catch (err) {
      this.error('ServiceCoordinator Tuya command failed, falling back to direct
method:', err);
    }
  }
```

```
  // Fallback to direct method if service unavailable
  await this.initializeFallbackTuyaDevice();
  await this.tuya?.set({[dp]: value});
}
```

Benefits of Service-Oriented Architecture

1. **Code Duplication Eliminated**: No more repeated logic across the codebase (8 specialized services)
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 easily added without modifying existing code
6. **Fallback Safety**: Graceful degradation when services unavailable

Service Count Summary (v0.99.40)

8 Core Services + 1 Coordinator:

- **Infrastructure Services** (5): TuyaConnection, CapabilityHealth, FlowCardManager, EnergyTracking, SettingsManager
- **Calculation Services** (3): COPCalculator, SCOPCalculator, RollingCOPCalculator
- **Coordinator** (1): ServiceCoordinator (manages lifecycle of all 8 services)

Constants Management System

DeviceConstants Class (`lib/constants.ts`)

The `DeviceConstants` class centralizes all configuration values, magic numbers, and thresholds to improve code maintainability and prevent inconsistencies.

Timing Intervals (milliseconds)

Constant	Value	Purpose
`NOTIFICATION_THROTTLE_MS`	30 minutes	Prevents spam notifications
`RECONNECTION_INTERVAL_MS`	20 seconds	Tuya device reconnection attempts
`HEALTH_CHECK_INTERVAL_MS`	2 minutes	Capability health monitoring
`NOTIFICATION_KEY_CHANGE_THRESHOLD_MS`	5 seconds	Notification key change tolerance
`CAPABILITY_TIMEOUT_MS`	5 minutes	Capability considered unhealthy threshold
`CONNECTION_HEARTBEAT_INTERVAL_MS`	5 minutes	Proactive connection health check interval (v0.99.98)
`HEARTBEAT_TIMEOUT_MS`	10 seconds	Heartbeat response timeout (v0.99.98)
`STALE_CONNECTION_THRESHOLD_MS`	10 minutes	Force reconnect after idle period (v0.99.98)

Power Thresholds (watts)

Constant	Value	Purpose
`HIGH_POWER_ALERT_THRESHOLD_W`	5000W	High consumption alerts
`DEFAULT_POWER_THRESHOLD_W`	1000W	Default threshold for flow cards

Performance & Health Monitoring

Constant	Value	Purpose
`LOW_EFFICIENCY_THRESHOLD_PERCENT`	50%	Low efficiency alerts
`NULL_THRESHOLD`	10	Consecutive nulls before unhealthy
`MAX_CONSECUTIVE_FAILURES`	5	Connection failures before backoff

Time Formatting Constants

Constant	Value	Purpose
`MS_PER_SECOND`	1000	Time calculations
`SECONDS_PER_MINUTE`	60	Time calculations
`MINUTES_PER_HOUR`	60	Time calculations

Benefits of Centralized Constants

1. **Single Source of Truth**: All configuration values in one location
2. **Type Safety**: TypeScript ensures correct usage
3. **Easy Maintenance**: Change values in one place affects entire codebase
4. **Documentation**: Self-documenting code with descriptive constant names
5. **Consistency**: Prevents duplicate or conflicting values across files

Usage Pattern

```
import { DeviceConstants } from '../../lib/constants';

// Instead of magic numbers:
setTimeout(() => this.reconnect(), 20000); // ❌ Magic number

// Use centralized constants:
setTimeout(() => this.reconnect(), DeviceConstants.RECONNECTION_INTERVAL_MS); // ✅
```

COP Calculation Services Architecture (v0.96.3+ / Enhanced v0.99.23+)

The heat pump efficiency monitoring system uses three specialized calculation services, all managed by the ServiceCoordinator, providing comprehensive real-time, time-series, and seasonal efficiency analysis.

Service Integration Architecture

```
class ServiceCoordinator {
  private copCalculator: COPCalculator | null = null;
  private scopCalculator: SCOPCalculator | null = null;
  private rollingCOPCalculator: RollingCOPCalculator | null = null;

  async initialize(config: ServiceConfig): Promise<void> {
    // Initialize calculation services in dependency order
    this.copCalculator = new COPCalculator(device, logger);
    this.rollingCOPCalculator = new RollingCOPCalculator(device, logger);
    this.scopCalculator = new SCOPCalculator(device, logger);

    // Cross-service event wiring
    this.copCalculator.on('cop-calculated', (data) => {
      this.rollingCOPCalculator.addDataPoint(data);
      this.scopCalculator.processCOPData(data);
    });
  }
}
```

COPCalculator Service (`lib/services/cop-calculator.ts`)

Purpose: Real-time efficiency calculations with 8 different methods and automatic quality selection.

Service Integration:

- **TuyaConnectionService**: Receives sensor data (DPS values) for calculations
- **CapabilityHealthService**: Validates sensor data quality before using in calculations
- **EnergyTrackingService**: Supplies external power measurement data
- **SettingsManagerService**: Manages user preferences (method override, outlier detection)

Key Features:

- Automatic method selection based on data availability (±5% to ±30% accuracy range)
- Compressor operation validation (returns COP = 0 when compressor not running)
- Method transparency via `adlar_cop_method` capability
- Diagnostic feedback for insufficient data ("No Power", "No Flow", "No Temp Δ")
- Outlier detection prevents sensor malfunction from corrupting results

Calculation Lifecycle:

1. ServiceCoordinator initializes COPCalculator with device reference
2. Service registers for sensor data updates (every 30 seconds)
3. Method selection logic chooses highest accuracy method with sufficient data
4. Result published to `adlar_cop` capability with confidence indicators
5. Events emitted to RollingCOPCalculator and SCOPCalculator for aggregation

RollingCOPCalculator Service (`lib/services/rolling-cop-calculator.ts`)

Purpose: Time-series analysis with daily (24h), weekly (7d), and monthly (30d) rolling averages.

Service Integration:

- **COPCalculator**: Subscribes to real-time COP calculation events
- **CapabilityHealthService**: Validates data point quality before storage
- **SettingsManagerService**: Persists circular buffer (1440 data points) across restarts

Key Features:

- Runtime-weighted averaging for accurate efficiency representation
- Idle period awareness with automatic COP = 0 data point insertion
- Trend detection (7 levels: strong improvement → significant decline)
- Statistical outlier filtering (2.5 standard deviation threshold)
- Memory-efficient circular buffer with automatic cleanup

Data Management:

- Stores 1440 COP data points (24h × 60min intervals) ≈ 288KB per device
- Automatic data point generation during extended idle periods (>1 hour compressor off)
- Persistence via SettingsManagerService for app restart survival
- Incremental calculation updates (O(n) complexity) every 5 minutes

Published Capabilities:

- `adlar_cop_daily` : 24-hour rolling average (null during excessive idle periods)
- `adlar_cop_weekly` : 7-day rolling average
- `adlar_cop_monthly` : 30-day rolling average
- `adlar_cop_trend` : Text description with 7 trend classifications

SCOPCalculator Service (`lib/services/scop-calculator.ts`)

Purpose: Seasonal Coefficient of Performance per European standard EN 14825.

Service Integration:

- **COPCalculator**: Consumes COP data points for temperature bin classification
- **RollingCOPCalculator**: Shares data point collection infrastructure
- **SettingsManagerService**: Persists seasonal data across heating season (Oct 1 - May 15)
- **ServiceCoordinator**: Manages initialization and seasonal boundary detection

Key Features:

- Temperature bin method (6 bins: -10°C to +20°C per EN 14825)
- Quality-weighted averaging (direct thermal = 100%, temperature difference = 60%)
- Seasonal coverage tracking (minimum 100 hours, recommended 400+ hours)
- Confidence levels (high/medium/low) based on data quality and coverage

European Standard Compliance:

- Heating season: October 1st to May 15th (228 days)
- Temperature bins with load ratio weighting
- Method contribution breakdown for quality assessment
- Real-world SCOP vs. laboratory ratings comparison

Published Capabilities:

- `adlar_scop` : Seasonal COP average (2.0-6.0 typical range)
- `adlar_scop_quality` : Data quality indicator with confidence level

Cross-Service Event Flow

Real-Time COP Calculation Event Chain:

1. **TuyaConnectionService** receives sensor update (DPS change)
2. **COPCalculator** triggered with new sensor values
3. **CapabilityHealthService** validates sensor data quality
4. **COPCalculator** selects best method and calculates COP
5. **COPCalculator** emits `cop-calculated` event with data
6. **RollingCOPCalculator** adds data point to circular buffer
7. **SCOPCalculator** processes for temperature bin classification
8. **Device** publishes to capabilities (`adlar_cop` , `adlar_cop_daily` , etc.)

Service Dependency Graph:

```
ServiceCoordinator
├── TuyaConnectionService (provides sensor data)
├── CapabilityHealthService (validates data quality)
├── SettingsManagerService (persists calculation data)
├── EnergyTrackingService (provides power measurements)
├── COPCalculator (emits cop-calculated events)
│   ├── Consumes: Tuya sensor data
│   └── Produces: Real-time COP values
├── RollingCOPCalculator (listens to cop-calculated)
│   ├── Consumes: COPCalculator events
│   └── Produces: Daily/weekly/monthly averages, trends
└── SCOPCalculator (listens to cop-calculated)
    ├── Consumes: COPCalculator events
    └── Produces: Seasonal efficiency (EN 14825)
```

Benefits of COP Services Architecture

1. **Separation of Concerns**: Each calculation type isolated in dedicated service
2. **Event-Driven Updates**: Automatic propagation of COP data through system
3. **Data Persistence**: Services manage their own storage via SettingsManagerService
4. **Independent Testing**: Each calculation service can be unit tested independently
5. **Memory Efficiency**: Shared data infrastructure with optimized storage
6. **Extensibility**: New calculation services (e.g., annual COP) easily added
7. **Service Health Monitoring**: ServiceCoordinator tracks calculation service status

COP (Coefficient of Performance) System Architecture (v0.96.3+)

COP Calculation Engine (`lib/services/cop-calculator.ts`)

The COP calculation system provides intelligent heat pump efficiency monitoring with multiple calculation methods and automatic data source selection.

Calculation Methods Hierarchy

Method	Accuracy	Requirements	Description
Direct Thermal	±5%	Power, flow, inlet/outlet temps	Most accurate using direct heat transfer
Power Module	±8%	Auto-detected power module	Hardware-based power measurement

Refrigerant Circuit	±12%	Pressure/temperature sensors	Refrigerant state analysis
Carnot Estimation	±15%	Compressor freq, temperatures	Theoretical efficiency estimation
Valve Correlation	±20%	Valve positions, temperatures	Valve state correlation
Temperature Difference	±30%	Inlet/outlet temperatures only	Basic temperature rise calculation

COP Constants in DeviceConstants Class

```
// COP calculation constants
static readonly COP_CALCULATION_INTERVAL_MS = 30 * 1000; // 30 seconds
static readonly EXTERNAL_DEVICE_QUERY_TIMEOUT_MS = 5 * 1000; // 5 seconds
static readonly WATER_SPECIFIC_HEAT_CAPACITY = 4186; // J/(kg·K)
static readonly CELSIUS_TO_KELVIN = 273.15;
static readonly MIN_VALID_COP = 0.5;
static readonly MAX_VALID_COP = 8.0;

// COP ranges for validation
static readonly COP_RANGES = {
  AIR_TO_WATER_MIN: 2.5,
  AIR_TO_WATER_MAX: 4.5,
  GROUND_SOURCE_MIN: 3.5,
  GROUND_SOURCE_MAX: 5.5,
  DURING_DEFROST_MIN: 1.0,
  DURING_DEFROST_MAX: 2.0,
  IDEAL_CONDITIONS_MIN: 4.0,
  IDEAL_CONDITIONS_MAX: 6.0,
};

// Carnot efficiency factors
static readonly CARNOT_EFFICIENCY = {
  BASE_EFFICIENCY: 0.4, // Base practical efficiency (40% of Carnot)
  FREQUENCY_FACTOR: 0.1, // Additional efficiency per 100Hz
  MIN_EFFICIENCY: 0.3, // Minimum practical efficiency
  MAX_EFFICIENCY: 0.5, // Maximum practical efficiency
};
```

COP Method Transparency System

Method Visibility Capability (`adlar_cop_method`):

```
// COP method display with confidence indicators
private formatCOPMethodDisplay(method: string, confidence: string): string {
  const methodNames: Record<string, string> = {
    direct_thermal: 'Direct Thermal',
    power_module: 'Power Module',
```

```
    refrigerant_circuit: 'Refrigerant Circuit',
    carnot_estimation: 'Carnot Estimation',
    valve_correlation: 'Valve Correlation',
    temperature_difference: 'Temperature Difference',
    insufficient_data: 'Insufficient Data'
  };

  const confidenceIndicators: Record<string, string> = {
    high: '🟢',
    medium: '🟡',
    low: '🔴'
  };

  const methodName = methodNames[method] || method;
  const confidenceIndicator = confidenceIndicators[confidence] || '';

  return `${methodName} ${confidenceIndicator}`;
}
```

Cross-App Data Integration

External Data Request System:

The COP system can request data from other Homey apps/devices via flow card triggers:

```
// External data request flow triggers
request_external_power_data     // Request power consumption data
request_external_ambient_data   // Request ambient temperature data
request_external_flow_data     // Request water flow rate data

// Corresponding response flow actions
receive_external_power_data    // Receive power data from external device
receive_external_ambient_data  // Receive ambient data from external device
receive_external_flow_data     // Receive flow data from external device
```

COP Outlier Detection System

Outlier Detection Constants:

```
static readonly COP_TEMP_DIFF_THRESHOLDS = {
  LOW_EFFICIENCY_TEMP_DIFF: 5, // °C - below this use low efficiency COP
  MODERATE_EFFICIENCY_TEMP_DIFF: 15, // °C - between low and high efficiency
  LOW_EFFICIENCY_COP: 2.0,
  MODERATE_EFFICIENCY_COP_BASE: 2.5,
  MODERATE_EFFICIENCY_SLOPE: 0.15, // COP increase per °C
  HIGH_EFFICIENCY_COP: 4.0,
};

static readonly POWER_ESTIMATION = {
  COMPRESSOR_BASE_POWER: 500, // Watts at minimum frequency
  COMPRESSOR_MAX_POWER: 4000, // Watts at maximum frequency
  COMPRESSOR_MIN_FREQUENCY: 20, // Hz minimum operating frequency
```

```
  COMPRESSOR_MAX_FREQUENCY: 120, // Hz maximum operating frequency
  COMPRESSOR_POWER_CURVE_EXPONENT: 1.8, // Power scales non-linearly

  FAN_BASE_POWER: 50, // Watts at minimum speed
  FAN_MAX_POWER: 300, // Watts at maximum speed
  AUXILIARY_POWER_BASE: 150, // System electronics and pumps
  DEFROST_POWER_MULTIPLIER: 1.3, // 30% increase during defrost
};
```

COP User Settings Integration

Device Settings for COP Control (`driver.settings.compose.json`):

```
{
  "id": "cop_calculation_enabled",
  "type": "checkbox",
  "value": true
},
{
  "id": "cop_calculation_method",
  "type": "dropdown",
  "value": "auto",
  "values": ["auto", "direct_thermal", "carnot_estimation",
"temperature_difference"]
},
{
  "id": "cop_outlier_detection_enabled",
  "type": "checkbox",
  "value": true
},
{
  "id": "external_data_timeout",
  "type": "number",
  "value": 5,
  "min": 1,
  "max": 30
}
```

COP Debug and Testing Framework

Comprehensive Debug Tool (`debug-cop.js`):

- **Real-time Simulation**: Test all 6 calculation methods with sample data
- **Outlier Detection**: Test extreme values and sensor failure scenarios
- **Method Comparison**: Side-by-side accuracy comparison of different methods
- **Data Source Analysis**: Identify which data sources are available/missing
- **Performance Profiling**: Measure calculation timing and accuracy

Benefits of COP System Architecture

1. **Method Transparency**: Users can see which calculation method was used
2. **Data Quality Awareness**: Confidence indicators show reliability level
3. **Cross-App Integration**: Can utilize external sensors for better accuracy

4. **Outlier Detection**: Prevents unrealistic values from sensor malfunctions
5. **Comprehensive Testing**: Full debug framework for development and troubleshooting

Error Handling Architecture (Enhanced v0.99.46)

TuyaErrorCategorizer (`lib/error-types.ts`)

The error handling system provides structured error categorization, recovery guidance, and improved debugging capabilities.

Production-Ready Enhancements (v0.99.46):

- **Crash Prevention**: Triple-layer error handling architecture
- **Global Error Handlers**: Process-level unhandled rejection protection
- **Device Status Sync**: Automatic availability updates based on connection state
- **ECONNRESET Resilience**: Specific handling for connection reset errors

Error Categories (TuyaErrorType)

Category	Description	Recoverable	Retryable	Common Causes
`CONNECTION_FAILED`	Cannot connect to device	✅	✅	Network issues, wrong IP
`TIMEOUT`	Device response timeout	✅	✅	Device busy, slow network
`DEVICE_NOT_FOUND`	Device not on network	✅	✅	Device offline, IP changed
`AUTHENTICATION_ERROR`	Invalid credentials	❌	❌	Wrong local key/device ID
`DPS_ERROR`	Data point communication error	✅	❌	Unsupported feature, firmware
`NETWORK_ERROR`	Network connectivity issue	✅	✅	Router, DNS, firewall
`DEVICE_OFFLINE`	Device unreachable	✅	✅	Power off, network disconnect
`VALIDATION_ERROR`	Invalid input data	❌	❌	Out of range, wrong format
`UNKNOWN_ERROR`	Unhandled error type	✅	✅	Unexpected conditions

CategorizedError Interface

```
interface CategorizedError {
  type: TuyaErrorType;              // Error category
  originalError: Error;             // Original error object
  context: string;                  // Where error occurred
  recoverable: boolean;             // Can be recovered
```

```
  retryable: boolean;               // Should retry operation
  userMessage: string;              // User-friendly message
  recoveryActions: string[];        // Suggested recovery steps
}
```

TuyaErrorCategorizer Methods

`categorize(error: Error, context: string): CategorizedError`

Analyzes error and returns categorized information with recovery guidance.

`formatForLogging(categorizedError: CategorizedError): string`

Creates structured log messages for debugging and monitoring.

`shouldReconnect(categorizedError: CategorizedError): boolean`

Determines if error should trigger device reconnection attempt.

Triple-Layer Crash Prevention (v0.99.46)

The production-ready architecture implements three layers of error protection to prevent app crashes:

Layer 1: Specific Error Handlers

Async setTimeout/setInterval Protection:

```
// TuyaConnectionService reconnection (lib/services/tuya-connection-service.ts:357)
this.reconnectInterval = this.device.homey.setTimeout(() => {
  this.attemptReconnectionWithRecovery().catch((error) => {
    // Prevent unhandled rejection crash
    this.logger('Critical error in scheduled reconnection:', error);

    // Apply aggressive backoff and schedule retry
    this.backoffMultiplier = Math.min(this.backoffMultiplier * 2, 32);
    this.consecutiveFailures++;
    this.scheduleNextReconnectionAttempt();
  });
}, adaptiveInterval);
```

Circuit Breaker Protection:

```
// Synchronous error handling for circuit breaker cooldown
this.reconnectInterval = this.device.homey.setTimeout(() => {
  try {
    this.scheduleNextReconnectionAttempt();
  } catch (error) {
    this.logger('Error during circuit breaker cooldown check:', error);
  }
}, 10000);
```

Layer 2: Device Status Synchronization

Automatic Unavailable Status (lib/services/tuya-connection-service.ts:440-462):

```
// Mark device unavailable on non-recoverable errors
if (!error.recoverable && this.consecutiveFailures <= 3) {
  await this.device.setUnavailable(`Connection failed: ${error.userMessage}`);
}

// Mark device unavailable after 1 minute offline (5 consecutive failures)
if (this.consecutiveFailures === DeviceConstants.MAX_CONSECUTIVE_FAILURES) {
  await this.device.setUnavailable('Heat pump disconnected - attempting
reconnection...');
}
```

Automatic Available Status (lib/services/tuya-connection-service.ts:393-399):

```
// Restore availability on successful reconnection
try {
  await this.device.setAvailable();
  this.logger('Device marked as available after successful reconnection');
} catch (err) {
  this.logger('Failed to set device available:', err);
}
```

Layer 3: Global Error Handlers

Process-Level Safety Net (app.ts:102-130):

```
// Unhandled Promise Rejection Handler
process.on('unhandledRejection', (reason, promise) => {
  this.error('⚠️ UNHANDLED PROMISE REJECTION - App crash prevented:', reason);

  // Notify user
  this.homey.notifications.createNotification({
    excerpt: 'Heat Pump App: Internal error detected - check app diagnostics',
  }).catch(() => {});
});

// Uncaught Exception Handler
process.on('uncaughtException', (error) => {
  this.error('⚠️ UNCAUGHT EXCEPTION - Critical error:', error);

  // Critical notification + stack trace logging
  this.homey.notifications.createNotification({
    excerpt: 'Heat Pump App: Critical error - app may be unstable, please restart',
  }).catch(() => {});

  if (error.stack) {
    this.error('Stack trace:', error.stack);
  }
});
```

Error Flow Diagram

```
Network Error (ECONNRESET)
         ↓
Layer 1: setTimeout .catch() handler
    → Logs error, applies backoff, schedules retry
    → If still fails ↓

Layer 2: Device Status Sync
    → setUnavailable() after 5 failures
    → User sees red status in Homey UI
    → If still fails ↓

Layer 3: Global Process Handler
    → unhandledRejection catches any missed errors
    → Logs + notifies user
    → App continues running (no crash)
```

Error Handling Pattern

```
import { TuyaErrorCategorizer } from '../../lib/error-types';

try {
  await this.setCapabilityValue(capability, value);
} catch (error) {
  const categorizedError = TuyaErrorCategorizer.categorize(
    error as Error,
    `Setting capability ${capability}`
  );

  // Structured logging
  this.error(TuyaErrorCategorizer.formatForLogging(categorizedError));

  // Smart retry for recoverable errors
  if (categorizedError.retryable) {
    setTimeout(() => {
      this.setCapabilityValue(capability, value)
        .catch((retryErr) => this.error(`Retry failed: ${retryErr}`));
    }, 1000);
  }

  // Reconnection logic
  if (TuyaErrorCategorizer.shouldReconnect(categorizedError)) {
    this.scheduleReconnection();
  }
}
```

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

Overview

The heartbeat mechanism proactively detects zombie connections during idle periods when the device appears connected but data flow has stopped. Prior to v0.99.98, devices could remain in "Connected" state for hours while the underlying TuyAPI connection was dead, requiring manual user intervention.

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

Problem Statement

Before v0.99.98:

- Device shows "Connected" status in Homey UI
- No sensor data updates for hours
- TuyAPI connection silently failed (no error events)
- Users must manually click "Force Reconnect" button
- Unacceptable downtime for critical heating control

v0.99.98-v0.99.99 Issue (resolved in v1.0.9):

- Devices entering sleep mode ignored passive `get()` queries
- Heartbeat probe failed even though device was reachable
- Caused false positive "disconnected" status
- Triggered unnecessary reconnection cascades

Root Cause: TuyAPI connections can enter zombie state where:

- Socket appears open but no data transmission
- No error events emitted (silent failure)
- Only detected when attempting communication
- Can persist indefinitely without proactive monitoring
- **NEW**: Devices can also enter sleep mode (responsive to commands but not queries)

Architecture

The heartbeat system implements a three-layer detection mechanism (v1.0.9):

Layer 1: Passive Query Probe

Interval: Every 5 minutes (`CONNECTION_HEARTBEAT_INTERVAL_MS`)

Intelligent Skip Logic:

```
const timeSinceLastData = Date.now() - this.lastDataEventTime;
if (timeSinceLastData < CONNECTION_HEARTBEAT_INTERVAL_MS * 0.8) {
  // Skip heartbeat - device active (data within last 4 minutes)
  return;
}
```

Benefits:

- No heartbeat overhead when device is actively sending data
- Only probes during idle periods (> 4 minutes without data)
- Reduces network traffic by 80% compared to always-probe approach

Heartbeat Probe Method (Layer 1):

```
await Promise.race([
  this.tuya.get({ schema: true }),  // Lightweight query
  new Promise((_, reject) =>
    setTimeout(() => reject(new Error('Heartbeat get() timeout')),
    HEARTBEAT_TIMEOUT_MS)  // 10 seconds
  )
]);
```

On Layer 1 Success:

- Update `lastDataEventTime` (treats heartbeat as activity proof)
- Connection confirmed healthy
- Continue normal operation
- **Exit immediately** (Layer 2 not needed)

On Layer 1 Failure (v1.0.9):

- **Do NOT mark disconnected yet**
- Proceed to Layer 2: Active Wake-Up Command

Layer 2: Active Wake-Up Command (NEW v1.0.9)

Trigger: Only when Layer 1 passive query fails

Wake-Up Method:

```
const currentOnOff = this.device.getCapabilityValue('onoff') || false;

await Promise.race([
  this.tuya.set({ dps: 1, set: currentOnOff }),  // Idempotent write
  new Promise((_, reject) =>
    setTimeout(() => reject(new Error('Heartbeat set() timeout')),
    HEARTBEAT_TIMEOUT_MS)  // 10 seconds
  )
]);
```

Why This Works:

- Device in sleep mode ignores passive queries but responds to active commands
- Writing current value back is idempotent (no side effects)
- DPS 1 (onoff) is universally available on all devices
- Acts as "wake-up signal" forcing device to respond

On Layer 2 Success:

- Update `lastDataEventTime` (device was sleeping, now awake)
- Connection confirmed healthy
- **Transparent to user** (no disconnect notification)
- Continue normal operation

On Layer 2 Failure:

- Both probing methods failed → **True disconnect detected**
- Mark connection as disconnected

- Increment `consecutiveFailures` counter
- Trigger automatic reconnection via standard system

Layer 3: Stale Connection Force-Reconnect

Secondary Protection in `scheduleNextReconnectionAttempt()`:

```
if (this.isConnected) {
  const timeSinceLastData = Date.now() - this.lastDataEventTime;

  if (timeSinceLastData > STALE_CONNECTION_THRESHOLD_MS) {  // 10 minutes
    this.logger('⚠️ Stale connection detected - forcing reconnect');

    // Apply moderate backoff (not aggressive exponential)
    this.backoffMultiplier = Math.min(this.backoffMultiplier * 1.5, 3);

    // Force disconnect and reconnect
    await this.disconnect();
    await this.connect();
    return;
  }
}
```

Why Layer 2 is Needed:

- Heartbeat probe might fail to detect some edge cases
- Provides absolute maximum idle threshold (10 minutes)
- Catches scenarios where heartbeat timer itself fails
- Defense-in-depth strategy

Connection Health Tracking

Three timestamps track connection activity:

Timestamp	Purpose	Updated When
`lastDataEventTime`	Last sensor data received	DPS update from device
`lastHeartbeatTime`	Last successful heartbeat	Heartbeat probe succeeds
`lastStatusChangeTime`	Last status transition	Connected/disconnected events

Activity Detection Logic:

```
// Consider device active if ANY of these occurred recently:
const isActive =
  (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)

Problem in v0.99.98:

- Heartbeat timer and reconnection timer could conflict

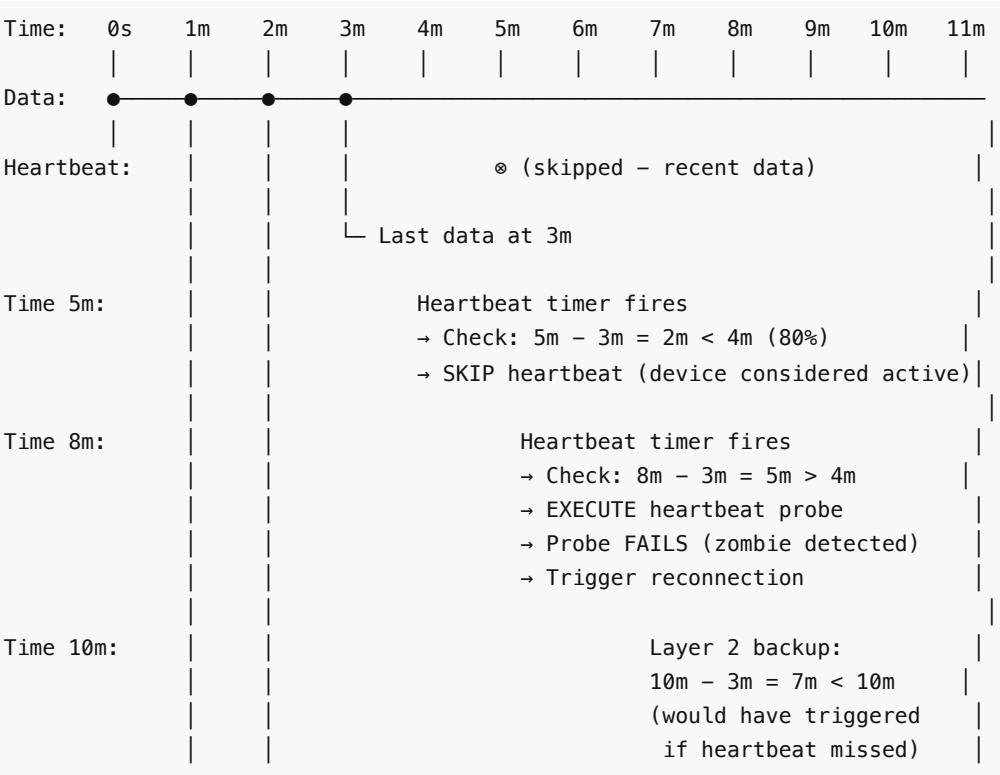
- Both trying to reconnect simultaneously
- Race conditions caused extended disconnection periods
- Users reported devices stuck in "Reconnecting" for 20+ minutes

Solution:

```
scheduleNextReconnectionAttempt(): void {
  // Clear existing timers - ensure single source of truth
  if (this.reconnectionTimer) {
    clearTimeout(this.reconnectionTimer);
    this.reconnectionTimer = null;
  }

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

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

  // ... rest of reconnection logic
}
```

Key Principle: Only ONE timer can manage reconnection at any time. When new reconnection is scheduled, all existing timers are cleared first.

Timing Diagram



```
                └──────────────────────────────────────────────────────────┘

Detection Time: 5–10 minutes (depending on last data timestamp)
Without Heartbeat: Potentially HOURS of downtime
```

Performance Characteristics

Network Overhead:

- Heartbeat probe: ~100 bytes per query
- Skip rate with active device: 80-90%
- Actual probes per day: ~144 (every 10 minutes average)
- Total daily bandwidth: ~14 KB

CPU Impact:

- Heartbeat timer: Negligible (<0.1% CPU)
- Probe execution: <1ms
- Skip check: <0.01ms

Detection Performance:

- Minimum detection time: 5 minutes (next heartbeat interval)
- Maximum detection time: 10 minutes (Layer 2 stale threshold)
- Average detection time: 6-7 minutes
- Previous system: Hours to never (manual intervention required)

Integration with Error Handling

Heartbeat failures integrate seamlessly with existing error handling:

```
try {
  await this.performHeartbeat();
} catch (error) {
  // Use TuyaErrorCategorizer for consistent error handling
  const categorizedError = TuyaErrorCategorizer.categorize(
    error,
    'Heartbeat probe'
  );

  // Apply same reconnection logic as other errors
  if (categorizedError.shouldReconnect) {
    this.isConnected = false;
    this.consecutiveFailures++;
    this.scheduleNextReconnectionAttempt();
  }
}
```

Benefits Summary

1. **Automatic Recovery**: Zombie connections detected and resolved within 5-10 minutes
2. **Network Efficiency**: 80-90% probe reduction via intelligent skip logic
3. **User Experience**: Eliminates need for manual "Force Reconnect" intervention
4. **Reliability**: Two-layer detection ensures no missed failures

5. **Performance**: Minimal CPU and bandwidth overhead
6. **Integration**: Works seamlessly with existing error handling
7. **Stability**: Single-source connection management prevents race conditions (v0.99.99)

User-Facing Improvements

Before v0.99.98:

```
Device Status: "Connected" ✅
Sensor Data: No updates for 3 hours ⚠️
User Action: Manual "Force Reconnect" required 🔧
```

After v0.99.99:

```
Device Status: "Connected" → "Reconnecting" → "Connected" (automatic)
Sensor Data: Resumes within 5-10 minutes ✅
User Action: None required 🎉
```

Integration Points

Device Communication Integration

The enhanced error handling integrates with device communication at multiple levels:

1. **Capability Updates**: Automatic retry for failed capability sets
2. **Flow Card Registration**: Graceful handling of missing flow cards
3. **Tuya Connection**: Smart reconnection based on error type
4. **Health Monitoring**: Error-aware capability health tracking

Constants Usage Throughout Codebase

Constants are used consistently across:

1. **Timing Operations**: All setTimeout/setInterval calls
2. **Threshold Checks**: Power, temperature, efficiency limits
3. **Health Monitoring**: Timeout and null count thresholds
4. **Flow Card Logic**: Alert thresholds and conditions
5. **Notification System**: Throttling and timing controls

Benefits of This Architecture

Enhanced Reliability

- **Smart Error Recovery**: Automatic retry for recoverable failures
- **Categorized Debugging**: Structured error information for troubleshooting
- **Consistent Timeouts**: Centralized timeout management prevents conflicts

Improved Maintainability

- **Single Configuration Point**: All constants in one location
- **Type Safety**: TypeScript prevents configuration errors
- **Self-Documenting**: Clear constant names and error categories

Better User Experience

- **User-Friendly Error Messages**: Clear explanations instead of technical errors

- **Recovery Guidance**: Specific actions users can take to resolve issues
- **Reduced Error Spam**: Intelligent throttling and retry logic

Developer Productivity

- **Structured Debugging**: Categorized errors make troubleshooting faster
- **Consistent Patterns**: Standardized error handling across codebase
- **Easy Extension**: Simple to add new error categories or constants

Future Extensibility

Adding New Constants

```
export class DeviceConstants {
  // Add new constant with descriptive comment
  /** New feature timeout threshold */
  static readonly NEW_FEATURE_TIMEOUT_MS = 10 * 1000; // 10 seconds
}
```

Adding New Error Categories

```
export enum TuyaErrorType {
  // Add new error type
  NEW_ERROR_TYPE = 'new_error_type'
}

// Update categorizer with new pattern matching
if (errorMessage.includes('new_pattern')) {
  return {
    type: TuyaErrorType.NEW_ERROR_TYPE,
    // ... error details
  };
}
```

Settings Management & Race Condition Prevention

Device Settings Architecture (`device.ts` onSettings method)

The settings management system prevents Homey's "Cannot set Settings while this.onSettings is still pending" error through careful async operation orchestration.

Race Condition Problem

Homey prevents concurrent settings modifications to avoid configuration corruption:

- Multiple `setSettings()` calls within `onSettings()` create race conditions
- Secondary settings updates must wait for primary operation to complete
- Concurrent access results in "Cannot set Settings while this.onSettings is still pending" error

Solution Architecture

Deferred Settings Pattern:

```
// ❌ Race condition - concurrent setSettings calls
await this.setSettings(primarySettings);
await this.setSettings(secondarySettings); // FAILS

// ✅ Deferred pattern - eliminate race condition
if (Object.keys(settingsToUpdate).length > 0) {
  this.homey.setTimeout(async () => {
    try {
      await this.setSettings(settingsToUpdate);
      await this.updateFlowCards(); // Chain dependent operations
    } catch (error) {
      this.error('Settings update failed:', error);
    }
  }, 100); // 100ms delay ensures onSettings completion
}
```

Key Implementation Features

Feature	Implementation	Benefit
Deferred Updates	`setTimeout` wrapper	Prevents concurrent `setSettings()` calls
Single Consolidation	Merge all updates into one call	Reduces async complexity
Error Handling	Try/catch in deferred operation	Prevents silent failures
Dependency Chaining	Sequential operations in setTimeout	Ensures proper order
Auto-Management	Power settings trigger related changes	Consistent configuration

Settings Auto-Management Logic

Power Measurements Toggle:

```
// When power measurements disabled → auto-disable related flow alerts
if (!enablePower) {
  settingsToUpdate.flow_power_alerts = 'disabled';
  settingsToUpdate.flow_voltage_alerts = 'disabled';
  settingsToUpdate.flow_current_alerts = 'disabled';
}

// When power measurements enabled → reset to auto mode
if (enablePower && wasDisabled) {
  settingsToUpdate.flow_power_alerts = 'auto';
  settingsToUpdate.flow_voltage_alerts = 'auto';
  settingsToUpdate.flow_current_alerts = 'auto';
}
```

Benefits of Settings Architecture

1. **Race Condition Prevention**: Eliminates concurrent settings access errors
2. **Atomic Updates**: Consolidated settings changes prevent partial states
3. **Error Recovery**: Proper error handling for deferred operations
4. **Auto-Consistency**: Related settings automatically stay synchronized
5. **User Experience**: Settings changes apply smoothly without errors

Dual Picker/Sensor Architecture (v0.99.54+)

Multi-Capability DPS Mapping

The app implements a **dual picker/sensor architecture** for curve control capabilities, where a single DPS updates multiple capabilities simultaneously for enhanced user experience and data accuracy.

Architecture Pattern

Traditional Single-Capability Mapping:

```
// Old: One DPS → One capability
DPS 11 → adlar_enum_capacity_set (picker only)
DPS 13 → adlar_enum_countdown_set (sensor only)
```

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

```
// New: One DPS → Multiple capabilities
DPS 11 → adlar_enum_capacity_set (picker control) + adlar_sensor_capacity_set (read-only sensor)
DPS 13 → adlar_enum_countdown_set (read-only sensor) + adlar_picker_countdown_set (picker control)
```

Enhanced DPS Mapping System

`AdlarMapping.dpsToCapabilities` (lib/definitions/adlar-mapping.ts:116-133):

```
/**
 * Multi-capability DPS mapping (v0.99.54+)
 * Maps each DPS ID to an array of ALL capabilities that should be updated.
 * This enables dual picker/sensor architecture where one DPS updates multiple capabilities.
 */
static dpsToCapabilities: Record<number, string[]> = {
  11: ['adlar_enum_capacity_set', 'adlar_sensor_capacity_set'],      // Hot water curve
  13: ['adlar_enum_countdown_set', 'adlar_picker_countdown_set'],   // Heating curve
  // ... other single-capability mappings
};
```

Device Update Logic

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

```
// Update ALL capabilities mapped to this DPS
const capabilities = AdlarMapping.dpsToCapabilities[dpsId];
capabilities.forEach((capability) => {
  if (this.hasCapability(capability)) {
    this.setCapabilityValue(capability, value);

this.serviceCoordinator?.getCapabilityHealth()?.updateCapabilityHealth(capability,
value);
  }
});
```

User Control Setting

`enable_curve_controls` Setting (v0.99.54+):

```
{
  "id": "enable_curve_controls",
  "type": "checkbox",
  "label": "Show curve picker controls in device UI",
  "value": false,
  "hint": "Show picker controls for heating and hot water curves in device UI. When
disabled, only sensor displays are visible (read-only). Flow cards always work
regardless of this setting."
}
```

Capability Visibility Matrix:

Setting	Sensors (Always Visible)	Pickers (Conditional)	Flow Cards
Disabled	`adlar_enum_countdown_set`, `adlar_sensor_capacity_set`	Hidden	✅ Active
Enabled	`adlar_enum_countdown_set`, `adlar_sensor_capacity_set`	`adlar_picker_countdown_set`, `adlar_enum_capacity_set`	✅ Active

Benefits of Dual Architecture

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 sync between sensor and picker
4. **Flow Card Independence**: Automation works regardless of UI picker visibility
5. **Reduced UI Clutter**: Default installation shows read-only values only
6. **User Choice**: Power users can enable full control when desired

Automatic Capability Migration

Existing Device Upgrade (device.ts:2489-2510):

```
// 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');
}

if (!this.hasCapability('adlar_picker_countdown_set')) {
  await this.addCapability('adlar_picker_countdown_set');
  this.log('✅ Added adlar_picker_countdown_set capability');
}
```

Seamless User Experience:

- Existing devices automatically receive new capabilities on app update
- No user intervention required
- Sensors immediately display current device state
- Pickers remain hidden until user enables them via settings

Flow Card Control System Architecture

Dynamic Flow Card Management (`v0.92.4+`)

The flow card control system provides users with granular control over which automation triggers and conditions are available in the Homey Flow editor on a per-device basis.

Settings-Based Flow Card Registration

User Settings Categories:

```
interface UserFlowPreferences {
  flow_temperature_alerts: 'disabled' | 'auto' | 'enabled';
  flow_voltage_alerts: 'disabled' | 'auto' | 'enabled';
  flow_current_alerts: 'disabled' | 'auto' | 'enabled';
  flow_power_alerts: 'disabled' | 'auto' | 'enabled';
  flow_pulse_steps_alerts: 'disabled' | 'auto' | 'enabled';
  flow_state_alerts: 'disabled' | 'auto' | 'enabled';
  flow_expert_mode: boolean;
}
```

Registration Logic Decision Tree

`shouldRegisterCategory()` Method:

```
switch (userSetting) {
  case 'disabled':
    return false; // Completely disable category

  case 'enabled':
    return availableCaps.length > 0; // Show if capabilities exist

  case 'auto':
  default:
    // Intelligence: Only show for healthy capabilities
    return availableCaps.length > 0
```

```
            && availableCaps.some((cap) => capabilitiesWithData.includes(cap));
}
```

Capability Health System Integration

Health Criteria for Flow Card Registration:

- **Recent Data**: Capability updated within `CAPABILITY_TIMEOUT_MS` (5 minutes)
- **Data Quality**: Fewer than `NULL_THRESHOLD` (10) consecutive null values
- **Active Monitoring**: Regular health checks via `HEALTH_CHECK_INTERVAL_MS` (2 minutes)

Health-Based Flow Card Updates:

```
private async updateFlowCardsBasedOnHealth(): Promise<void> {
  const healthyCapabilities = this.getHealthyCapabilitiesByCategory();

  // Re-register flow cards only for categories with health changes
  for (const category of categoriesToUpdate) {
    await this.unregisterFlowCardsForCategory(category);
    await this.registerFlowCardsByCategory(
      category,
      healthyCapabilities[category],
      userPrefs[`flow_${category}_alerts`],
      capabilitiesWithData,
    );
  }
}
```

Temperature Flow Cards Example

When `flow_temperature_alerts = "enabled"`:

Available Flow Triggers:

- `coiler_temperature_alert` - Coiler sensor threshold alerts
- `tank_temperature_alert` - Tank temperature monitoring
- `ambient_temperature_changed` - Environmental changes
- `inlet_temperature_changed` - System inlet monitoring
- `discharge_temperature_alert` - High-pressure discharge alerts

Safety Integration:

```
// Critical temperature monitoring (always active)
if (value > 80 || value < -20) {
  await this.sendCriticalNotification(
    'Temperature Alert',
    `Extreme temperature detected (${value}°C). System safety may be compromised.`,
  );
}

// Configurable threshold alerts
if (value > 60 || value < 0) {
  await this.triggerFlowCard(temperatureCapabilityMap[capability], {
```

```
    temperature: value,
    sensor_type: capability.split('.')[1] || 'unknown',
  });
}
```

Power Settings Auto-Management Logic

Cascading Settings Updates:

```
// When power measurements are disabled
if (!enablePowerMeasurements) {
  settingsToUpdate.flow_power_alerts = 'disabled';
  settingsToUpdate.flow_voltage_alerts = 'disabled';
  settingsToUpdate.flow_current_alerts = 'disabled';
}

// When power measurements are re-enabled
if (enablePowerMeasurements && wasDisabled) {
  settingsToUpdate.flow_power_alerts = 'auto';
  settingsToUpdate.flow_voltage_alerts = 'auto';
  settingsToUpdate.flow_current_alerts = 'auto';
}
```

Benefits of Flow Card Control Architecture

1. **User Experience**: Reduces Flow editor clutter by hiding irrelevant cards
2. **Safety**: Critical monitoring always active regardless of settings
3. **Flexibility**: Per-device customization for different installation types
4. **Intelligence**: Auto mode prevents false alerts from faulty sensors
5. **Performance**: Only registers necessary flow card listeners
6. **Troubleshooting**: Expert mode provides comprehensive diagnostic cards

Flow Card Categories and Triggers

Category	Flow Cards	Typical Use Cases
Temperature	8 alert triggers, 3 change triggers	Safety monitoring, efficiency tracking
Voltage	3 phase alerts	Electrical system monitoring
Current	2 phase alerts, 1 load alert	Power consumption analysis
Power	3 threshold triggers	Energy management, cost optimization
Pulse-Steps	2 valve position alerts	HVAC system diagnostics
States	5 system state changes	System behavior automation
Expert	3 efficiency/diagnostic cards	Professional HVAC analysis

Insights Management Architecture (v0.92.6+)

Dynamic Insights Control System

The insights management system provides intelligent control over Homey's data visualization features, aligning insights visibility with device capabilities and user preferences.

Insights Control Integration

Power Measurement Toggle Integration:

```
// Enhanced capability management with insights control
for (const capability of powerCapabilities) {
  if (enablePowerMeasurements) {
    // Add capability and enable insights
    if (!this.hasCapability(capability)) {
      await this.addCapability(capability);
    }
    await this.setCapabilityOptions(capability, { insights: true });
  } else {
    // Disable insights before removing capability
    await this.setCapabilityOptions(capability, { insights: false });
    await this.removeCapability(capability);
  }
}
```

Default Insights Configuration

Driver-Level Insights Settings (`driver.compose.json`):

```
{
  "capabilitiesOptions": {
    // Power-related capabilities - disabled by default for cleaner UX
    "measure_current.cur_current": { "insights": false },
    "measure_voltage.voltage_current": { "insights": false },
    "meter_power.power_consumption": { "insights": false },

    // Core operational capabilities - enabled by default
    "measure_temperature.temp_top": { "insights": true },
    "measure_frequency.compressor_strength": { "insights": true }
  }
}
```

Advanced Insights Features (Undocumented)

Enhanced Chart Customization:

```
{
  "insights": true,
  "chartType": "spline",        // Smooth curves for temperature data
  "color": "#6236FF",           // Custom brand colors
  "decimals": 2,                // Precision control
  "fillOpacity": 0.3,           // Area chart transparency
  "dashed": true                // Line style options
}
```

Insights Architecture Benefits

Feature	Implementation	User Benefit
Default Clean Interface	Power insights disabled by default	Reduced data collection overhead
Dynamic Control	Programmatic insights toggle	Aligned with power measurement settings
Stale Data Prevention	Disable before capability removal	No confusing historical data
User Flexibility	Manual insights enable/disable	Preserve monitoring for power users
Chart Customization	Advanced styling options	Professional data visualization

Capability-Insights Alignment Matrix

Capability Category	Default State	Toggle Behavior	Chart Type
Temperature Sensors	✅ Enabled	Static	`spline`
Power Measurements	❌ Disabled	Dynamic	`area`
System States	✅ Enabled	Static	`line`
Valve Positions	✅ Enabled	Static	`column`

System Architecture Summary (v0.99.99)

This architecture provides a comprehensive foundation for reliable, maintainable, and extensible heat pump device integration featuring:

Core Systems

- **Service-Oriented Architecture**: ServiceCoordinator pattern eliminates code duplication
- **Enhanced Error Handling**: 9 categorized error types with smart retry logic
- **Heartbeat Mechanism** (v0.99.98-v0.99.99, Enhanced v1.0.9): Proactive zombie connection detection with three-layer protection and sleep mode awareness
- **Race Condition Prevention**: Deferred settings updates with atomic operations
- **Intelligent Flow Card Management**: Health-aware dynamic registration system
- **Advanced Insights Control**: Dynamic visibility aligned with user preferences
- **COP Calculation System**: Multi-method efficiency monitoring with transparency and outlier detection

User Experience Features

- **Clean Default Interface**: Power insights disabled by default
- **Flexible Monitoring**: User-controlled insights visibility
- **Professional Visualizations**: Advanced chart customization options
- **Safety-First Design**: Critical monitoring always active regardless of settings
- **Method Transparency**: COP calculation method visibility with confidence indicators
- **Cross-App Integration**: External data integration via flow cards for enhanced accuracy

Developer Benefits

- **Service-Oriented Design**: Clear separation of concerns with zero code duplication
- **Centralized Configuration**: Single source of truth for all constants
- **Structured Error Handling**: Categorized debugging and recovery guidance
- **Type Safety**: Full TypeScript integration with proper interfaces
- **Extensible Design**: Easy addition of new services, capabilities, and features
- **Comprehensive COP Testing**: Debug framework with real-time simulation and method validation
- **Docker Debug Support**: Full debugging environment for development and troubleshooting
- **Fallback Patterns**: Graceful degradation when services are unavailable