Contents

Adaptive Control Architecture Guide (v1.4.0 - Fase 1)

This comprehensive guide documents the Adaptive Temperature Control system implemented in the Adlar Heat Pump Homey app, providing architectural details, implementation patterns, and integration strategies for PI-based temperature control.

Table of Contents

Overview

What is Adaptive Control?

The Adaptive Control system enables **automatic temperature regulation** using external room temperature sensors (Homey thermostats, sensors) to maintain a stable indoor temperature (±0.3°C) by dynamically adjusting the heat pump's target temperature setpoint.

Current Status: **Fase 1 MVP Complete** (PI Heating Controller only)

Key Capabilities:

- **PI-based temperature control** - Proportional-Integral algorithm for stable regulation
- **External sensor integration** - Flow card-based data input from Homey devices
- **Persistent state** - Survives app updates and Homey restarts
- **Transparency** - Flow triggers show adjustment reasons and magnitudes
- **Predictive control** - Not implemented (Fase 2: Building Model Learner)
- **Cost optimization** - Not implemented (Fase 3: Energy Price Optimizer)
- **Efficiency optimization** - Not implemented (Fase 4: COP Controller)

Why Adaptive Control?

Problem: Standard heat pump curve control (DPS 13) cannot adapt to:

- Room-specific temperature variations
- Building thermal mass differences
- Weather changes and solar gain
- Occupancy patterns and internal heat sources

Solution: External PI controller that:

1. Reads actual room temperature from Homey sensors
2. Calculates error: `targetTemp - actualTemp`
3. Applies PI algorithm for smooth correction

4. Adjusts heat pump setpoint (DPS 4) dynamically
5. Maintains stable indoor climate (±0.3°C deadband)

Benefits:

- **Comfort**: Stable room temperature without manual intervention
- **Efficiency**: Reduces overshoot/undershoot energy waste
- **Flexibility**: Works with any Homey-compatible temperature sensor
- **Transparency**: Flow cards show every adjustment decision

---

System Architecture

High-Level Architecture

Design Principles

1. **Zero Device Modifications**: External pattern - no changes to `device.ts` required
2. **Service-Oriented**: Integrates with existing ServiceCoordinator architecture
3. **Event-Driven**: Flow cards trigger updates, not polling
4. **Fail-Safe**: Graceful degradation when external data unavailable
5. **Transparent**: Every adjustment logged and exposed via flow cards
6. **Persistent**: State survives app updates and restarts

Component Catalog

1. AdaptiveControlService

File: `lib/services/adaptive-control-service.ts`

Responsibility: Main orchestrator for adaptive temperature control

Key Features:

- **5-minute control loop** (configurable via `DeviceConstants.ADAPTIVE_CONTROL_INTERVAL_MS`)
- **Enable/disable management** via settings and flow cards
- **Persistent state** - PI history, last action, accumulated adjustment
- **Flow card triggers** - Status changes and temperature adjustments
- **Throttling** - Prevents excessive DPS 4 updates (respects step:1 rounding)
- **Accumulation logic** - Fractional adjustments summed until 0.5°C threshold

Public Interface:

```
class AdaptiveControlService {
  async initialize(): Promise<void>;
  async enable(): Promise<void>;
  async disable(): Promise<void>;
  isEnabled(): boolean;
  getStatus(): AdaptiveControlStatus;
  destroy(): void;
}

interface AdaptiveControlStatus {
  enabled: boolean;
  lastControlCycle: number | null;
  controlIntervalMinutes: number;
  hasExternalTemperature: boolean;
```

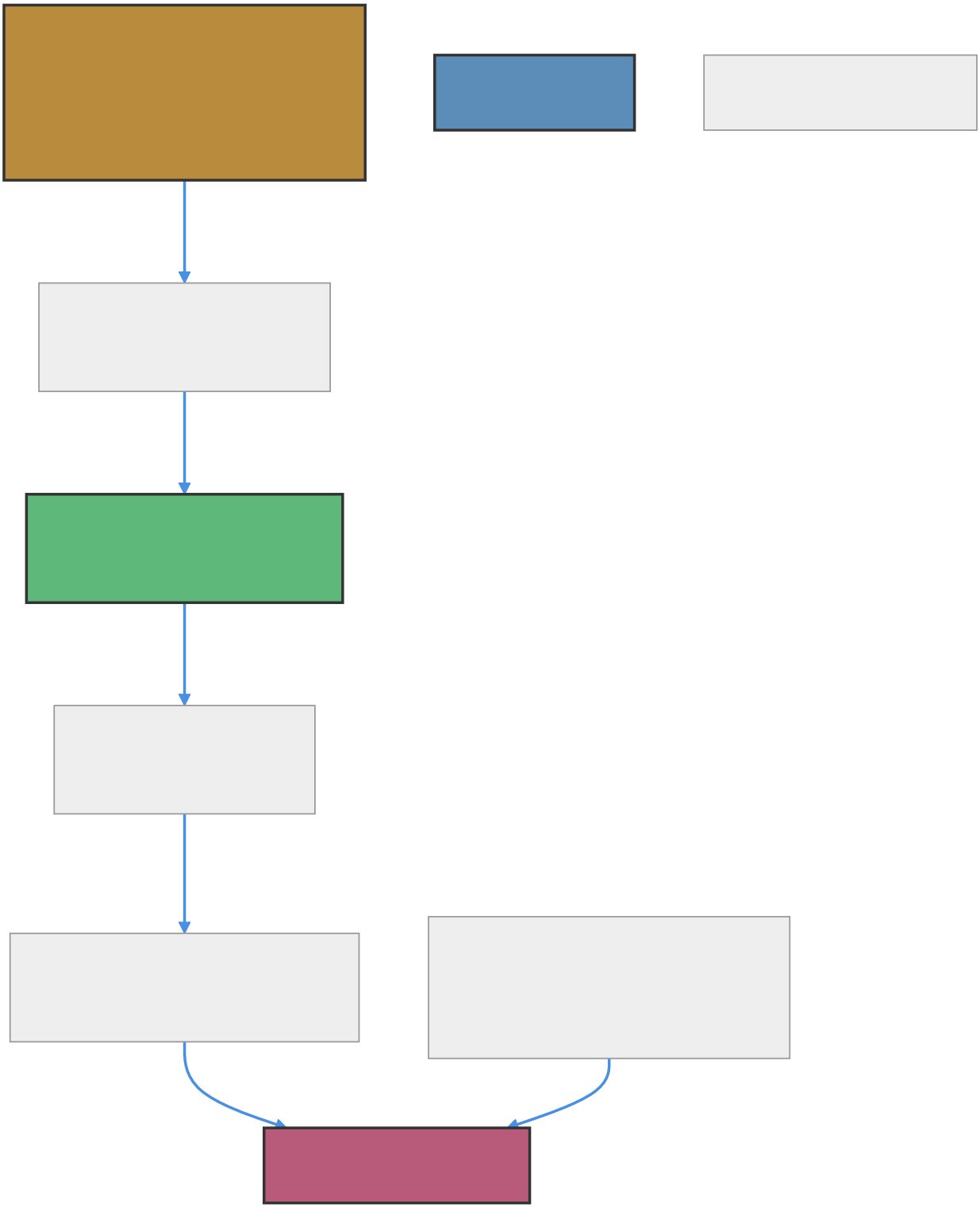

Figure 1: Diagram 1

```
  currentIndoorTemp: number | null;
  currentTargetTemp: number | null;
  piControllerStatus: PIControllerStatus;
}
```

Dependencies:

- **HeatingController** (lib/adaptive/heating-controller.ts) - PI algorithm
- **ExternalTemperatureService** (lib/services/external-temperature-service.ts) - Sensor data
- **Device** (via Homey.Device reference) - Capability reads/writes, store operations

Events Emitted (via Flow Cards):

- `adaptive_status_change` - Trigger when enabled/disabled/error state changes
- `target_temperature_adjusted` - Trigger when DPS 4 adjusted (includes reason, magnitude)

Persistence Keys (Device Store):

```
// PI controller error history (24 data points, 2 hours)
'adaptive_pi_history': number[]

// Last controller action taken (for diagnostics)
'adaptive_last_action': ControllerAction | null

// Last time target_temperature was adjusted (throttling)
'adaptive_last_adjustment_time': number

// Enable/disable state
'adaptive_control_enabled': boolean

// Accumulated fractional adjustments (step:1 rounding)
'adaptive_accumulated_adjustment': number
```

---

2. HeatingController

File: `lib/adaptive/heating-controller.ts`

Responsibility: PI (Proportional-Integral) temperature control algorithm

Key Features:

- **Proportional gain (Kp)**: 3.0 (default, configurable)
- **Integral gain (Ki)**: 1.5 (default, configurable)
- **Deadband tolerance**: 0.3°C (prevents oscillation)
- **Error history**: 24 data points (2 hours at 5-minute intervals)
- **Safety clamp**: ±3°C maximum adjustment per cycle
- **Minimum action threshold**: 0.1°C (prevents micro-adjustments)
- **Priority classification**: Low/Medium/High based on error magnitude

Public Interface:

```
class HeatingController {
  async calculateAction(data: SensorData): Promise<ControllerAction | null>;
  updateParameters(Kp: number, Ki: number, deadband: number): void;
  getStatus(): PIControllerStatus;
  restoreHistory(history: number[]): void;
  getHistory(): number[];
  destroy(): void;
```

```
}

interface SensorData {
  indoorTemp: number;      // Current room temperature (from external sensor)
  targetTemp: number;      // Desired room temperature (user setpoint)
  timestamp: number;       // When this reading was taken
}

interface ControllerAction {
  temperatureAdjustment: number;  // °C change to apply to DPS 4
  reason: string;                 // Human-readable explanation
  priority: 'low' | 'medium' | 'high';
  controller: 'heating';
}

interface PIControllerStatus {
  Kp: number;
  Ki: number;
  deadband: number;
  historySize: number;
  currentError: number;
  averageError: number;
  maxHistorySize: number;
}
```

Dependencies: None (pure algorithm)

Algorithm Details:

```
// 1. Calculate error
error = targetTemp - indoorTemp

// 2. Check deadband (±0.3°C)
if (Math.abs(error) < deadband) {
  return null; // No action needed
}

// 3. Update error history (circular buffer, max 24 points)
errorHistory.push(error)
if (errorHistory.length > maxHistorySize) {
  errorHistory.shift()
}

// 4. Calculate PI terms
pTerm = Kp × error                                    // Proportional: immediate response
iTerm = Ki × (sum(errorHistory) / errorHistory.length)  // Integral: accumulated error

// 5. Total adjustment
adjustment = pTerm + iTerm

// 6. Safety clamp
clampedAdjustment = Math.max(-3, Math.min(3, adjustment))

// 7. Minimum threshold
if (Math.abs(clampedAdjustment) < 0.1) {
  return null; // Too small to matter
```

```
}

// 8. Priority classification
if (Math.abs(error) > 2.0) → priority = 'high'
else if (Math.abs(error) < 0.5) → priority = 'low'
else → priority = 'medium'
```

Tuning Guidelines:

Scenario	Kp	Ki	Deadband	Effect
Default (balanced)	3.0	1.5	0.3°C	Good for most homes
Fast response	4.0	2.0	0.2°C	Quick corrections, risk of overshoot
Slow/stable	2.0	1.0	0.5°C	Slower response, very stable
Massive building	5.0	3.0	0.2°C	High thermal mass requires aggressive gains
Small apartment	2.5	1.2	0.4°C	Low thermal mass, gentle control

Mathematical Background:

The PI controller is a **feedback control system** that minimizes error over time:

- **P-term (Proportional)**: Responds to current error
  - Higher Kp → faster response but potential overshoot
  - Formula: `P = Kp × error`
- **I-term (Integral)**: Eliminates steady-state error
  - Higher Ki → removes bias but can cause oscillation
  - Formula: `I = Ki × (average of recent errors)`

Why No D-term (Derivative)?:

- Temperature changes slowly (minutes/hours, not seconds)
- Derivative amplifies sensor noise
- PI sufficient for thermal control applications

---

3. ExternalTemperatureService

File: `lib/services/external-temperature-service.ts`

Responsibility: Flow card-based external temperature data integration

Key Features:

- **Flow action card**: `receive_external_indoor_temperature`
- **Data validation**: Range checks (−10°C to +50°C), null checks, type validation
- **Freshness tracking**: 5-minute timeout (data considered stale after 5 minutes)
- **Capability management**: Updates `adlar_external_indoor_temperature` capability
- **Error handling**: Graceful degradation when data invalid or stale

Public Interface:

```
class ExternalTemperatureService {
  async receiveTemperature(temperature: number): Promise<void>;
```

```
  getLastTemperature(): number | null;
  isDataFresh(): boolean;
  getLastUpdateTime(): number | null;
  destroy(): void;
}
```

Dependencies: None (receives data from flow cards)

Flow Card Integration:

Action Card: `receive_external_indoor_temperature`

```
{
  "id": "receive_external_indoor_temperature",
  "title": {
    "en": "Update external indoor temperature",
    "nl": "Externe binnentemperatuur bijwerken"
  },
  "args": [
    {
      "name": "temperature",
      "type": "number",
      "title": { "en": "Temperature", "nl": "Temperatuur" },
      "min": -10,
      "max": 50,
      "step": 0.1
    }
  ]
}
```

Example Flow:

```
WHEN: Thermostat temperature changed
THEN: Update external indoor temperature (temperature: {{temperature}})
```

Capability: `adlar_external_indoor_temperature`

- **Type**: `number`
- **Units**: `°C`
- **Range**: −10°C to +50°C
- **Decimals**: 1
- **UI Component**: `sensor` (read-only display)
- **Insights**: Enabled (historical graph)

Data Freshness Logic:

```
const FRESHNESS_TIMEOUT_MS = 5 * 60 * 1000; // 5 minutes

isDataFresh(): boolean {
  if (!this.lastUpdateTime) return false;
  const age = Date.now() - this.lastUpdateTime;
  return age < FRESHNESS_TIMEOUT_MS;
}
```

Validation Logic:

```
async receiveTemperature(temperature: number): Promise<void> {
  // Type check
  if (typeof temperature !== 'number') {
    throw new Error('Temperature must be a number');
```

```
  }

  // Range check
  if (temperature < -10 || temperature > 50) {
    throw new Error('Temperature out of range (-10°C to +50°C)');
  }

  // NaN check
  if (isNaN(temperature)) {
    throw new Error('Temperature is NaN');
  }

  // Update capability and timestamp
  await this.device.setCapabilityValue('adlar_external_indoor_temperature', temperature);
  this.lastTemperature = temperature;
  this.lastUpdateTime = Date.now();

  this.logger('External temperature received:', temperature.toFixed(1), '°C');
}
```

PI Control Theory

What is PI Control?

PI (Proportional-Integral) control is a **feedback control algorithm** that automatically adjusts a system's output to maintain a desired setpoint.

Key Concepts:

1. **Error**: Difference between desired value (target) and actual value (current)
   - `error = targetTemp - indoorTemp`
   - Example: Target 20.0°C, Indoor 19.5°C → error = +0.5°C (too cold)
2. **Proportional Term (P)**: Immediate response to current error
   - `P = Kp × error`
   - Example: Kp=3.0, error=0.5°C → P = +1.5°C adjustment
3. **Integral Term (I)**: Eliminates long-term bias
   - `I = Ki × (average of recent errors)`
   - Example: Ki=1.5, avg error=0.3°C → I = +0.45°C adjustment
4. **Total Adjustment**: Sum of P and I terms
   - `adjustment = P + I = 1.5 + 0.45 = +1.95°C`

Why PI for Temperature Control?

Advantages:

- **No overshoot** - Gentle approach to setpoint (unlike pure P control)
- **Eliminates steady-state error** - I-term removes long-term bias
- **Stable** - Temperature changes slowly, no derivative noise
- **Well-understood** - Industry standard for thermal control

Disadvantages:

- **Slower than P control** - Takes time to reach setpoint
- **Requires tuning** - Kp/Ki must match building characteristics
- **No predictive capability** - Cannot anticipate disturbances (Fase 2 feature)

Deadband Logic

Purpose: Prevent oscillation ("hunting") around setpoint

Implementation:

```
const deadband = 0.3; // °C

if (Math.abs(error) < deadband) {
  // Within acceptable range - no action needed
  return null;
}
```

Example:

- Target: 20.0°C
- Deadband: ±0.3°C
- Acceptable range: 19.7°C to 20.3°C
- Indoor temp 20.1°C → error = −0.1°C → **no action** (within deadband)
- Indoor temp 20.5°C → error = −0.5°C → **action required** (outside deadband)

Benefits:

- Prevents excessive DPS updates (wear reduction)
- Reduces energy waste from micro-adjustments
- Maintains comfort (±0.3°C imperceptible to users)

Accumulation Logic (Step:1 Rounding)

Problem: DPS 4 (target_temperature) has `step: 1` (integer only)

- PI controller calculates: +0.3°C adjustment
- Cannot apply 0.3°C directly (would round to 0°C → no effect)

Solution: Accumulate fractional adjustments

```
// Accumulator starts at 0
accumulatedAdjustment = 0

// Cycle 1: PI calculates +0.3°C
accumulatedAdjustment += 0.3  // = 0.3 (< 0.5, no action)

// Cycle 2: PI calculates +0.4°C
accumulatedAdjustment += 0.4  // = 0.7 ( 0.5, apply +1°C)
targetTemp += Math.round(0.7) // = +1°C
accumulatedAdjustment = 0.7 - 1.0 = -0.3  // Carry remainder

// Cycle 3: PI calculates +0.2°C
accumulatedAdjustment += 0.2  // = -0.1 (< 0.5, no action)
```

Code Implementation (`adaptive-control-service.ts:235-260`):

```
private accumulatedAdjustment = 0;

private async applyAdjustment(action: ControllerAction): Promise<void> {
  // Add to accumulator
  this.accumulatedAdjustment += action.temperatureAdjustment;

  // Check if accumulated adjustment significant ( 0.5°C)
  if (Math.abs(this.accumulatedAdjustment) >= 0.5) {
```

```
    // Round to nearest integer (step:1)
    const roundedAdjustment = Math.round(this.accumulatedAdjustment);

    // Read current target temperature
    const currentTarget = this.device.getCapabilityValue('target_temperature') || 20;

    // Calculate new target
    const newTarget = currentTarget + roundedAdjustment;

    // Apply to DPS 4
    await this.device.setCapabilityValue('target_temperature', newTarget);

    // Calculate remainder
    this.accumulatedAdjustment -= roundedAdjustment;

    // Persist
    await this.device.setStoreValue('adaptive_accumulated_adjustment', this.accumulatedAdjustment);

    // Trigger flow card
    await this.triggerAdjustmentFlowCard(roundedAdjustment, action.reason);
  }
}
```

Benefits:

- No fractional adjustments lost
- Smooth long-term control
- Respects DPS 4 step constraint
- Maintains PI controller accuracy

Control Flow & Timing

5-Minute Control Loop

Interval: 5 minutes (300,000ms) - configurable via `DeviceConstants.ADAPTIVE_CONTROL_INTERVAL_MS`

Rationale:

- Temperature changes slowly (thermal inertia)
- Avoids excessive DPS 4 updates (TuyAPI rate limiting)
- Balances responsiveness vs. stability

Control Cycle Steps (Every 5 Minutes):

Diagram 2

Figure 2: Diagram 2

Event-Driven Updates

External Temperature Updates (As They Arrive):

```
USER FLOW: Thermostat temperature changed


THEN: Update external indoor temperature ({{temperature}})
```

```
ExternalTemperatureService.receiveTemperature(temp)

    Validate temperature
    Update capability: adlar_external_indoor_temperature
    Update lastTemperature
    Update lastUpdateTime
    Log success

NOTE: Does NOT trigger immediate control loop
      (loop runs on fixed 5-minute timer)
```

Immediate Control Trigger (Expert Mode, Future Enhancement):

Future enhancement could add flow action card:

```
{
  "id": "adaptive_control_trigger_now",
  "title": "Trigger adaptive control cycle now"
}
```

This would allow users to force immediate control cycle after external temp update.

Timing Diagram

```
Time    Event

00:00   [Control Loop] Check conditions → Action: +1°C
00:02   [User Flow] Thermostat reports 19.8°C
00:05   [Control Loop] Check conditions → Action: +0.3°C (accumulated)
00:07   [User Flow] Thermostat reports 19.9°C
00:10   [Control Loop] Check conditions → Action: +0.4°C (accumulated → +1°C applied)
00:15   [Control Loop] Check conditions → Within deadband, no action
00:20   [Control Loop] Check conditions → Within deadband, no action
...
```

Persistence Strategy

Why Persistence Matters

Problem: Without persistence, app updates/restarts lose:

- PI controller error history (integral term reset → poor control)
- Accumulated fractional adjustments (lost precision)
- Last action tracking (diagnostics impossible)
- Enable/disable state (user preference lost)

Solution: Device store-based persistence

Persistence Keys

1. PI Error History (`adaptive_pi_history`)

```
// Type: number[]
// Size: Max 24 data points (2 hours @ 5-minute intervals)
// Example: [0.5, 0.4, 0.3, 0.2, 0.1, 0.0, -0.1]
// Used by: HeatingController (integral term calculation)
```

```
await this.device.setStoreValue('adaptive_pi_history', this.heatingController.getHistory());
```

Restoration (On Initialize):

```
const restoredHistory = await this.device.getStoreValue('adaptive_pi_history');
if (restoredHistory && Array.isArray(restoredHistory)) {
  this.heatingController.restoreHistory(restoredHistory);
  this.logger('PI history restored:', restoredHistory.length, 'data points');
}
```

2. Last Action (adaptive_last_action)

```
// Type: ControllerAction | null
// Example: {
//   temperatureAdjustment: 1.8,
//   reason: 'PI Control: Error=0.5°C, P=1.5°C, I=0.3°C',
//   priority: 'medium',
//   controller: 'heating'
// }
// Used by: Diagnostics, troubleshooting

await this.device.setStoreValue('adaptive_last_action', action);
```

3. Last Adjustment Timestamp (adaptive_last_adjustment_time)

```
// Type: number (Unix timestamp milliseconds)
// Example: 1702123456789
// Used by: Throttling logic (prevent excessive updates)

this.lastAdjustmentTime = Date.now();
await this.device.setStoreValue('adaptive_last_adjustment_time', this.lastAdjustmentTime);
```

4. Enable/Disable State (adaptive_control_enabled)

```
// Type: boolean
// Example: true
// Used by: Control loop (skip if disabled)

this.isEnabled = true;
await this.device.setStoreValue('adaptive_control_enabled', this.isEnabled);
```

5. Accumulated Adjustment (adaptive_accumulated_adjustment)

```
// Type: number
// Example: 0.7 (will apply +1°C next time, carry -0.3)
// Used by: Step:1 rounding logic

this.accumulatedAdjustment = 0.7;
await this.device.setStoreValue('adaptive_accumulated_adjustment', this.accumulatedAdjustment);
```

Restoration Sequence (On Device.onInit)

```
async initialize(): Promise<void> {
  // 1. Restore enable/disable state
  const enabled = await this.device.getStoreValue('adaptive_control_enabled');
  this.isEnabled = enabled ?? false;

  // 2. Restore PI history
```

```
  const history = await this.device.getStoreValue('adaptive_pi_history');
  if (history && Array.isArray(history)) {
    this.heatingController.restoreHistory(history);
  }

  // 3. Restore accumulated adjustment
  const accumulated = await this.device.getStoreValue('adaptive_accumulated_adjustment');
  this.accumulatedAdjustment = accumulated ?? 0;

  // 4. Restore last adjustment time
  const lastTime = await this.device.getStoreValue('adaptive_last_adjustment_time');
  this.lastAdjustmentTime = lastTime ?? 0;

  // 5. Start control loop if enabled
  if (this.isEnabled) {
    this.startControlLoop();
  }

  this.logger('AdaptiveControlService initialized', {
    enabled: this.isEnabled,
    historySize: history?.length ?? 0,
    accumulated: this.accumulatedAdjustment,
  });
}
```

Cleanup (On Device.onDeleted)

```
destroy(): void {
  // Stop control loop
  if (this.controlLoopInterval) {
    clearInterval(this.controlLoopInterval);
    this.controlLoopInterval = null;
  }

  // Clean up sub-services
  this.heatingController.destroy();
  this.externalTemperature.destroy();

  // NOTE: Device store persists even after destroy()
  // This is intentional - allows restoration after app update
  this.logger('AdaptiveControlService destroyed');
}
```

Flow Card Integration

Action Cards (1)

`receive_external_indoor_temperature` **Purpose**: Receive room temperature from external Homey devices

Arguments:

```
{
  "name": "temperature",
  "type": "number",
```

```
  "title": { "en": "Temperature (°C)", "nl": "Temperatuur (°C)" },
  "min": -10,
  "max": 50,
  "step": 0.1
}
```

Example Flows:

```
WHEN: Thermostat temperature changed
THEN: Update external indoor temperature
        temperature: {{temperature}}

WHEN: Motion sensor detects motion
THEN: Update external indoor temperature
        temperature: {{sensor.temperature}}

WHEN: Zigbee sensor updates
THEN: Update external indoor temperature
        temperature: {{temp}}
```

Handler (`lib/services/external-temperature-service.ts:45-65`):

```typescript
async receiveTemperature(temperature: number): Promise<void> {
  // Validation
  if (typeof temperature !== 'number' || isNaN(temperature)) {
    throw new Error('Invalid temperature value');
  }
  if (temperature < -10 || temperature > 50) {
    throw new Error('Temperature out of range (-10°C to +50°C)');
  }

  // Update capability (visible in device UI)
  await this.device.setCapabilityValue('adlar_external_indoor_temperature', temperature);

  // Update internal state
  this.lastTemperature = temperature;
  this.lastUpdateTime = Date.now();

  this.logger('External temperature received:', temperature.toFixed(1), '°C');
}
```

Registration (`lib/services/flow-card-manager-service.ts` integration):

```typescript
// Register action card
const receiveExternalTempCard = this.device.homey.flow.getActionCard(
  'receive_external_indoor_temperature'
);

receiveExternalTempCard.registerRunListener(async (args) => {
  const service = this.device.serviceCoordinator.getAdaptiveControl();
  await service.getExternalTemperatureService().receiveTemperature(args.temperature);
  return true;
});
```

Trigger Cards (2)

`adaptive_status_change` **Purpose**: Notify when adaptive control enabled/disabled/error state changes

Tokens:

```
{
  "status": {
    "type": "string",
    "title": { "en": "Status", "nl": "Status" },
    "example": "enabled"
  },
  "message": {
    "type": "string",
    "title": { "en": "Message", "nl": "Bericht" },
    "example": "Adaptive control enabled successfully"
  }
}
```

Status Values:

- `enabled` - Control loop started
- `disabled` - Control loop stopped
- `error_no_external_temp` - External temperature data missing/stale
- `error_invalid_target` - Cannot read target_temperature capability

Example Flows:

```
WHEN: Adaptive status changed
        status = "error_no_external_temp"
THEN: Send notification
        "Adaptive control error: {{message}}"

WHEN: Adaptive status changed
        status = "enabled"
THEN: Turn on indicator LED
```

Trigger Call (`adaptive-control-service.ts:180-195`):

```
private async triggerStatusChange(status: string, message: string): Promise<void> {
  try {
    const statusCard = this.device.homey.flow.getTriggerCard('adaptive_status_change');
    await statusCard.trigger(
      this.device,
      {
        status,
        message,
      }
    ).catch((err) => this.logger('Failed to trigger status change:', err));

    this.logger('Status change triggered:', status, '-', message);
  } catch (error) {
    this.logger('Error triggering status change:', error);
  }
}
```

`target_temperature_adjusted` **Purpose**: Notify when heat pump target temperature adjusted by PI controller

Tokens:

```
{
  "adjustment": {
    "type": "number",
    "title": { "en": "Adjustment (°C)", "nl": "Aanpassing (°C)" },
    "example": 2
  },
  "new_target": {
    "type": "number",
    "title": { "en": "New Target (°C)", "nl": "Nieuw Doel (°C)" },
    "example": 22
  },
  "reason": {
    "type": "string",
    "title": { "en": "Reason", "nl": "Reden" },
    "example": "PI Control: Error=0.8°C, P=2.4°C, I=0.6°C"
  },
  "priority": {
    "type": "string",
    "title": { "en": "Priority", "nl": "Prioriteit" },
    "example": "medium"
  }
}
```

Example Flows:

```
WHEN: Target temperature adjusted
        priority = "high"
THEN: Send notification
        "Large temperature correction: {{adjustment}}°C ({{reason}})"

WHEN: Target temperature adjusted
THEN: Log to Insights
        "Adaptive control: {{reason}}"

WHEN: Target temperature adjusted
        adjustment > 2
THEN: Send critical alert
        "Unusual adjustment: {{adjustment}}°C"
```

Trigger Call (adaptive-control-service.ts:265-285):

```
private async triggerAdjustmentFlowCard(
  adjustment: number,
  reason: string,
  priority: 'low' | 'medium' | 'high',
  newTarget: number
): Promise<void> {
  try {
    const adjustmentCard = this.device.homey.flow.getTriggerCard('target_temperature_adjusted');
    await adjustmentCard.trigger(
      this.device,
      {
```

```
        adjustment,
        new_target: newTarget,
        reason,
        priority,
      }
    ).catch((err) => this.logger('Failed to trigger adjustment card:', err));

    this.logger('Adjustment flow card triggered:', {
      adjustment: adjustment.toFixed(1),
      newTarget,
      reason,
      priority,
    });
  } catch (error) {
    this.logger('Error triggering adjustment card:', error);
  }
}
```

ServiceCoordinator Integration

Registering AdaptiveControlService

File: `lib/services/service-coordinator.ts`

Import:

```
import { AdaptiveControlService } from './adaptive-control-service';
```

Property:

```
private adaptiveControl!: AdaptiveControlService;
```

Initialize (`initialize()` method):

```
// Initialize AdaptiveControlService (v1.3.0+)
this.adaptiveControl = new AdaptiveControlService(serviceOptions);
await this.adaptiveControl.initialize();
this.logger('ServiceCoordinator: AdaptiveControlService initialized');
```

Getter:

```
getAdaptiveControl(): AdaptiveControlService {
  return this.adaptiveControl;
}
```

Destroy (`destroy()` method):

```
// Destroy AdaptiveControlService
if (this.adaptiveControl) {
  this.adaptiveControl.destroy();
}
```

Settings Integration (`onSettings()` method):

```
async onSettings(
  oldSettings: Record<string, unknown>,
  newSettings: Record<string, unknown>,
  changedKeys: string[]
): Promise<void> {
```

```
  // ... existing settings logic ...

  // Adaptive control enable/disable (future enhancement)
  if (changedKeys.includes('adaptive_control_enable')) {
    const enabled = newSettings.adaptive_control_enable as boolean;
    if (enabled) {
      await this.adaptiveControl.enable();
    } else {
      await this.adaptiveControl.disable();
    }
  }

  // Delegate to SettingsManagerService
  await this.settingsManager.onSettings(oldSettings, newSettings, changedKeys);
}
```

Cross-Service Dependencies

AdaptiveControlService interacts with:

1. **Device** (via Homey.Device reference)
   - `getCapabilityValue('target_temperature')`
   - `setCapabilityValue('target_temperature', newValue)`
   - `getCapabilityValue('adlar_external_indoor_temperature')`
   - `setStoreValue(key, value)` / `getStoreValue(key)`
2. **FlowCardManagerService** (indirectly via Homey.flow)
   - Registers flow action/trigger cards
   - No direct service-to-service calls
3. **TuyaConnectionService** (indirectly via device capabilities)
   - DPS 4 updates flow through device capability system
   - No direct service-to-service calls
4. **SettingsManagerService** (future)
   - Will manage `adaptive_control_enable` setting
   - Will persist PI tuning parameters (Kp, Ki, deadband)

Architectural Benefits:

- **Loose coupling** - AdaptiveControl doesn't directly depend on other services
- **Event-driven** - Flow cards trigger updates, not polling
- **Fail-safe** - Graceful degradation if external data unavailable
- **Extensible** - Easy to add Fase 2-4 components later

Implementation Roadmap

Fase 1: Heating Controller ✅ COMPLETE (v1.3.0)

Goal: Basic PI temperature control using external sensors

Components:

- **HeatingController** - PI algorithm with Kp=3.0, Ki=1.5, deadband=0.3°C
- **ExternalTemperatureService** - Flow card data input, validation, freshness
- **AdaptiveControlService** - 5-minute control loop, persistence, flow triggers
- **Flow Cards** - 1 action, 2 triggers
- **Capability** - `adlar_external_indoor_temperature`
- **ServiceCoordinator Integration** - getAdaptiveControl() getter

- **Documentation** - Setup guide, architecture guide (this document)

Status: **Production-ready** (v1.3.0)

Limitations:

- No predictive control (reactive only)
- No cost optimization (energy prices not considered)
- No efficiency optimization (COP not used)
- Fixed PI gains (no auto-tuning)

Fase 2: Building Model Learner NOT IMPLEMENTED

Goal: Learn building thermal characteristics for predictive control

Components (Planned):

- **BuildingModelLearner** (`lib/adaptive/building-model-learner.ts`)
  - Recursive Least Squares (RLS) algorithm
  - 4 parameters: C (thermal mass), UA (heat loss), g (solar gain), P_int (internal gains)
  - 4×4 covariance matrix (P-matrix)
  - Forgetting factor: 0.98 (2-day adaptation)

Data Requirements:

- Sensor data every 5 minutes:
  - Indoor temperature (external sensor)
  - Outdoor temperature (DPS 26)
  - Heat pump power (from EnergyTrackingService)
  - Solar irradiance (Weer API integration needed)

Mathematical Model:

```
dT/dt = (1/C) × [P_heating - UA×(T_in - T_out) + g×Solar + P_int]

Where:
  C       = Thermal mass (Wh/°C) - heat capacity of building
  UA      = Heat loss coefficient (W/°C) - insulation quality
  g       = Solar gain factor (W/(W/m²)) - window efficiency
  P_int   = Internal gains (W) - occupants, appliances, etc.
```

Features (Planned):

- Temperature prediction (24-48 hours ahead)
- Pre-heating strategy (heat before prices spike)
- Disturbance anticipation (weather changes, occupancy patterns)
- Auto-tuning of PI gains based on learned model

Complexity: **HIGH** - Requires:

- RLS algorithm implementation
- Numerical stability (matrix inversions)
- Outdoor temperature history
- Solar data API integration (e.g., OpenWeatherMap)
- Extensive testing (multiple building types)

Estimated Effort: 2-3 weeks development + 1 week testing

Fase 3: Energy Price Optimizer NOT IMPLEMENTED

Goal: Minimize energy costs using day-ahead electricity prices

Components (Planned):

- **EnergyPriceOptimizer** (`lib/adaptive/energy-price-optimizer.ts`)
  - Day-ahead price API integration (Dutch market: ENTSO-E, EPEX SPOT)
  - Price categorization (zeer laag, laag, normaal, hoog, zeer hoog)
  - Pre-heating schedule optimization
  - Cost-benefit analysis (comfort vs. savings)

Price Thresholds (Dutch Market):

Category	Price Range	Strategy
Zeer Laag	$<$ €0.10/kWh	Pre-heat MAX (+2°C target bump)
Laag	€0.10-0.15	Pre-heat (+1°C target bump)
Normaal	€0.15-0.25	Standard (PI control only)
Hoog	€0.25-0.35	Reduce (−1°C target bump)
Zeer Hoog	$>$ €0.35	Minimal (−2°C target bump, within comfort bounds)

Features (Planned):

- Automatic price fetching (daily at 14:00 CET)
- 24-hour price graph in device UI
- User-configurable price thresholds
- Cost savings tracking (Insights capability)
- Balance comfort vs. cost (user-defined priority)

Dependencies:

- **Fase 2 (Building Model Learner)** - Required for accurate pre-heating timing
- Energy price API integration (ENTSO-E Transparency Platform or commercial provider)
- Price data persistence (store 7-day history)

Complexity: **MEDIUM** - Requires:

- External API integration (rate limiting, authentication)
- Price data validation and fallback handling
- Optimization algorithm (when to pre-heat, how much)
- User preference management (comfort priority slider)

Estimated Savings: €400-600/year (based on dynamic contract, 3000 kWh heating consumption)

Estimated Effort: 1-2 weeks development + 1 week testing

Fase 4: COP Controller NOT IMPLEMENTED

Goal: Maximize heat pump efficiency (COP) through flow temperature optimization

Components (Planned):

- **COPController** (`lib/adaptive/cop-controller.ts`)
  - Historical COP database (flow temp vs. outdoor temp vs. COP)
  - Flow temperature tuning (DPS 4 adjustment)
  - Efficiency-based control (lower flow temp = higher COP)

- Iterative search for optimal setpoint

COP Optimization Strategy:

```
For given outdoor temperature:
  1. Look up historical COP data at different flow temperatures
  2. Find lowest flow temp that still meets heating demand
  3. Adjust DPS 4 (target temperature) to achieve that flow temp
  4. Monitor actual COP vs. predicted COP
  5. Update database with new measurement
  6. Repeat every hour (slower than PI loop)
```

Features (Planned):

- COP prediction model (based on outdoor temp, flow temp)
- Flow temperature learning (iterative search)
- Safety limits (min flow temp 35°C, max 55°C)
- Integration with RollingCOPCalculator (use existing COP data)
- User-configurable priority (efficiency vs. comfort)

Dependencies:

- **Existing COPCalculator service** - Provides real-time COP measurements
- **RollingCOPCalculator service** - Provides historical COP trends
- **Outdoor temperature** (DPS 26) - Key input for COP prediction

Complexity: **MEDIUM-HIGH** - Requires:

- COP database design (store flow temp, outdoor temp, COP)
- Optimization algorithm (find optimal flow temp)
- Safety constraints (prevent freezing, overheating)
- Balancing efficiency vs. comfort (some users prefer warmer)

Estimated Savings: €200-300/year (10-15% COP improvement)

Estimated Effort: 2 weeks development + 2 weeks testing (requires real-world tuning)

Weighted Decision System NOT IMPLEMENTED

Goal: Combine all 4 controllers with user-defined priorities

Architecture (Planned):

```
class AdaptiveControlService {
  private heatingController: HeatingController;       // Comfort (60% weight)
  private buildingLearner: BuildingModelLearner;      // Prediction (info only)
  private energyOptimizer: EnergyPriceOptimizer;      // Cost (15% weight)
  private copController: COPController;               // Efficiency (25% weight)

  async calculateFinalAdjustment(): Promise<number> {
    // Get recommendations from each controller
    const comfortAdj = await this.heatingController.calculateAction(sensorData);
    const priceAdj = await this.energyOptimizer.calculateAdjustment(currentPrice);
    const efficiencyAdj = await this.copController.calculateAdjustment(currentCOP);

    // Weighted average (user-configurable)
    const weights = this.getUserWeights(); // e.g., { comfort: 0.6, cost: 0.15, efficiency: 0.25 }

    const finalAdj =
```

```
      comfortAdj.adjustment * weights.comfort +
      priceAdj.adjustment * weights.cost +
      efficiencyAdj.adjustment * weights.efficiency;

    return finalAdj;
  }
}
```

User Settings (Planned):

```
{
  "adaptive_priority_comfort": 60,      // % weight
  "adaptive_priority_cost": 15,         // % weight
  "adaptive_priority_efficiency": 25    // % weight
  // Total must equal 100%
}
```

Example Scenarios:

User Profile	Comfort	Cost	Efficiency	Behavior
Default (balanced)	60%	15%	25%	Prioritize comfort, some savings
Cost-focused	40%	40%	20%	Aggressive price optimization
Eco-focused	40%	10%	50%	Maximize COP, minimal cost focus
Comfort-focused	90%	5%	5%	Ignore cost/efficiency, stable temp

Complexity: **LOW** (once Fase 2-4 complete) - Just weighted averaging

Estimated Effort: 1 day (assuming all controllers implemented)

Troubleshooting

Common Issues

Issue 1: "No external temperature data" Error **Symptoms**:

- Flow trigger: `adaptive_status_change` with status `error_no_external_temp`
- Control loop skips cycles
- Log: "External temperature data missing or stale"

Causes:

1. **No flow configured** - User hasn't created flow to send temperature
2. **Flow inactive** - Flow disabled or trigger condition not met
3. **Stale data** - Last temperature update > 5 minutes ago
4. **Invalid data** - Temperature out of range (−10 to +50°C)

Diagnosis:

```
// Check capability value
const externalTemp = device.getCapabilityValue('adlar_external_indoor_temperature');
console.log('External temp capability:', externalTemp); // null = no data
```

```
// Check service status
const status = serviceCoordinator.getAdaptiveControl().getStatus();
console.log('Has external temp:', status.hasExternalTemperature); // false = problem
console.log('Current indoor temp:', status.currentIndoorTemp); // null = no data

// Check ExternalTemperatureService
const extService = serviceCoordinator.getAdaptiveControl().getExternalTemperatureService();
console.log('Last temp:', extService.getLastTemperature());
console.log('Data fresh:', extService.isDataFresh());
console.log('Last update:', extService.getLastUpdateTime());
```

Solutions:

1. **Create flow**:

   ```
   WHEN: [Your thermostat] temperature changed
   THEN: Update external indoor temperature
           temperature: {{temperature}}
   ```

2. **Check flow execution**:
   - Open Homey app → Flows
   - Find the flow
   - Check "Last run" timestamp (should be recent)
   - Test flow manually
3. **Verify thermostat working**:
   - Open thermostat device
   - Check temperature capability updates regularly
4. **Check temperature range**:
   - Ensure thermostat reports reasonable values (10-30°C typical)
   - Values outside −10 to +50°C rejected

Issue 2: Target Temperature Not Adjusting Symptoms:

- External temperature data arriving
- No `target_temperature_adjusted` flow triggers
- DPS 4 (target_temperature) not changing

Causes:

1. **Within deadband** - Temperature error < 0.3°C (no action needed)
2. **Accumulation not reached** - Fractional adjustments < 0.5°C threshold
3. **Adaptive control disabled** - Control loop not running
4. **PI calculation returns null** - Error too small or other condition

Diagnosis:

```
// Check if enabled
const status = serviceCoordinator.getAdaptiveControl().getStatus();
console.log('Enabled:', status.enabled); // false = disabled

// Check error magnitude
const indoorTemp = status.currentIndoorTemp; // e.g., 20.1
const targetTemp = status.currentTargetTemp; // e.g., 20.0
```

```
const error = targetTemp - indoorTemp;       // = -0.1°C
console.log('Error:', error, 'Deadband: 0.3°C');
// If |error| < 0.3 → within deadband, no action

// Check PI controller status
const piStatus = status.piControllerStatus;
console.log('Current error:', piStatus.currentError);
console.log('Average error:', piStatus.averageError);
console.log('History size:', piStatus.historySize);

// Check accumulated adjustment
const accumulated = await device.getStoreValue('adaptive_accumulated_adjustment');
console.log('Accumulated adjustment:', accumulated);
// If |accumulated| < 0.5 → not enough to apply
```

Solutions:

1. **Deadband behavior** - This is normal! ±0.3°C prevents oscillation
   - Wait for error to exceed 0.3°C
   - If comfort issue, reduce deadband (Expert Mode setting, future)
2. **Accumulation** - Be patient
   - Small adjustments accumulate over multiple cycles
   - Eventually 0.5°C threshold triggers DPS 4 update
   - Check store: `adaptive_accumulated_adjustment`
3. **Enable adaptive control**:
   - Check device settings (future: enable/disable toggle)
   - Manually call `serviceCoordinator.getAdaptiveControl().enable()`
4. **Verify PI calculation**:
   - Check logs for "PI calculation" entries
   - Look for "Within deadband, no action needed" (normal)
   - Look for "Adjustment too small (<0.1°C), no action" (normal)

Issue 3: Oscillation / Instability Symptoms:

- Target temperature adjusting frequently (every cycle)
- Temperature swings > ±1°C
- Flow triggers: `target_temperature_adjusted` with alternating +/− adjustments

Causes:

1. **PI gains too high** - Kp and/or Ki set too aggressively
2. **Deadband too small** - Triggers action too frequently
3. **Heat pump response slow** - Building thermal inertia not considered
4. **External sensor lag** - Sensor location poor (near heat source, drafts)

Diagnosis:

```
// Check PI parameters
const piStatus = status.piControllerStatus;
console.log('Kp:', piStatus.Kp); // > 5.0 likely too high
console.log('Ki:', piStatus.Ki); // > 3.0 likely too high
console.log('Deadband:', piStatus.deadband); // < 0.2 likely too small

// Check adjustment history
const lastAction = await device.getStoreValue('adaptive_last_action');
console.log('Last adjustment:', lastAction.temperatureAdjustment);
```

```
// Rapid +/- swings indicate oscillation

// Check error history
const history = await device.getStoreValue('adaptive_pi_history');
console.log('Error history:', history);
// Should converge toward zero, not oscillate
```

Solutions:

1. **Reduce PI gains** (Expert Mode, future):
   - Lower Kp: 3.0 → 2.0 (slower response, more stable)
   - Lower Ki: 1.5 → 1.0 (less integral windup)
2. **Increase deadband** (Expert Mode, future):
   - 0.3°C → 0.5°C (wider acceptable range)
3. **Improve sensor placement**:
   - Move sensor away from heat sources (radiators, appliances)
   - Avoid drafts (doors, windows)
   - Place in representative location (living room center)
4. **Wait for settling**:
   - Initial cycles may oscillate as PI controller learns
   - After 2-4 hours (24-48 cycles), should stabilize

Issue 4: Persistent Store Errors Symptoms:

- Logs: "Failed to save adaptive state"
- PI history not persisting across app updates
- Accumulated adjustment resets to 0

Causes:

1. **Device store corrupted** - Rare, but possible after Homey crash
2. **Store key conflicts** - Multiple devices using same keys (shouldn't happen)
3. **Homey storage full** - Disk space exhausted

Diagnosis:

```
// Check store values directly
const history = await device.getStoreValue('adaptive_pi_history');
const accumulated = await device.getStoreValue('adaptive_accumulated_adjustment');
const enabled = await device.getStoreValue('adaptive_control_enabled');

console.log('History type:', typeof history, 'Value:', history);
console.log('Accumulated type:', typeof accumulated, 'Value:', accumulated);
console.log('Enabled type:', typeof enabled, 'Value:', enabled);

// Try writing test value
try {
  await device.setStoreValue('test_key', { test: 'value' });
  console.log('Store write successful');
} catch (error) {
  console.error('Store write failed:', error);
}
```

Solutions:

1. **Clear device store** (nuclear option):
   - Remove device from Homey

- Re-pair device
- Reconfigure adaptive control
- **WARNING**: Loses all persistent state!

2. **Check Homey storage**:
   - Settings → System → Storage
   - Ensure > 100 MB free space
3. **Restart Homey**:
   - Settings → System → Reboot
   - Sometimes clears transient store issues
4. **Contact support**:
   - If persistent, may indicate Homey platform bug
   - Provide logs and reproduction steps

Best Practices

Sensor Placement

Ideal Location:

- **Living room center** - Representative of main living space
- **1.5m height** - Average room temperature (not floor/ceiling)
- **Away from heat sources** - Not near radiators, appliances, sunlight
- **Avoid drafts** - Not near doors, windows, vents

Poor Locations:

- **Kitchen** - Cooking heat spikes (not representative)
- **Bathroom** - High humidity, temperature swings
- **Near thermostat** - Already has sensor (redundant)
- **Bedroom** - Different heating zone (if doors closed)

Flow Configuration

Best Practice Flow:

```
WHEN: [Living Room Thermostat] temperature changed
AND:  [Living Room Thermostat] is available
      (prevents stale data if thermostat offline)
THEN: Update external indoor temperature
        temperature: {{temperature}}
```

Avoid:

```
WHEN: Every 5 minutes (polling)
THEN: Update external indoor temperature
        temperature: (some API call)
```

Why: Event-driven updates are more efficient and accurate than polling.

PI Tuning

Conservative (Default):

- Kp = 3.0, Ki = 1.5, Deadband = 0.3°C
- **Use for**: Most homes, first installation
- **Behavior**: Slow, stable, minimal overshoot

Aggressive (Fast Response):

- Kp = 4.5, Ki = 2.5, Deadband = 0.2°C
- **Use for**: High thermal mass buildings (concrete, brick)
- **Behavior**: Fast response, risk of overshoot

Gentle (Ultra-Stable):

- Kp = 2.0, Ki = 1.0, Deadband = 0.5°C
- **Use for**: Low thermal mass buildings (wood, well-insulated)
- **Behavior**: Very slow, no overshoot, wide comfort range

Monitoring

Recommended Insights:

1. **External Indoor Temperature** - Track actual room temp
2. **Target Temperature** - See PI adjustments over time
3. **Adaptive Status** - Enabled/disabled history
4. **Adjustment Magnitude** - How often and how much PI adjusts

Flow Logging (Optional):

```
WHEN: Target temperature adjusted
THEN: Log to Insights
        "Adaptive: {{adjustment}}°C - {{reason}}"

WHEN: Target temperature adjusted
        priority = "high"
THEN: Send notification
        "Large correction: {{adjustment}}°C"
```

Maintenance

Weekly:

- Check external temperature capability (should update regularly)
- Verify flow triggers (at least 1-2 adjustments per day)

Monthly:

- Review Insights graphs (temperature stability)
- Check for oscillation patterns (rapid +/− adjustments)

After App Update:

- Verify adaptive control still enabled
- Check PI history restored (status.piControllerStatus.historySize > 0)
- Confirm accumulated adjustment carried over

Future Enhancements

Expert Mode Settings (Planned)

Device Settings (`driver.settings.compose.json`):

```
{
  "type": "group",
  "label": { "en": "Adaptive Control (Expert)", "nl": "Adaptieve Regeling (Expert)" },
  "children": [
    {
```

```
      "id": "adaptive_control_enable",
      "type": "checkbox",
      "label": { "en": "Enable Adaptive Control", "nl": "Adaptieve Regeling Inschakelen" },
      "value": false
    },
    {
      "id": "adaptive_kp",
      "type": "number",
      "label": { "en": "Proportional Gain (Kp)", "nl": "Proportionele Versterking (Kp)" },
      "value": 3.0,
      "min": 0.5,
      "max": 10.0,
      "step": 0.1
    },
    {
      "id": "adaptive_ki",
      "type": "number",
      "label": { "en": "Integral Gain (Ki)", "nl": "Integrale Versterking (Ki)" },
      "value": 1.5,
      "min": 0.1,
      "max": 10.0,
      "step": 0.1
    },
    {
      "id": "adaptive_deadband",
      "type": "number",
      "label": { "en": "Deadband (°C)", "nl": "Deadband (°C)" },
      "value": 0.3,
      "min": 0.1,
      "max": 1.0,
      "step": 0.1
    }
  ]
}
```

Auto-Tuning (Planned)

Ziegler-Nichols Method:

1. Set Ki = 0 (proportional-only control)
2. Increase Kp until oscillation occurs (critical gain Ku)
3. Measure oscillation period (Tu)
4. Calculate optimal gains:
   - Kp = 0.6 × Ku
   - Ki = 1.2 × Ku / Tu

Implementation Complexity: MEDIUM (requires oscillation detection)

Predictive Pre-Heating (Fase 2+)

Concept: Heat before temperature drops (proactive vs. reactive)

Example:

```
Current: 08:00, Indoor 20.0°C, Target 20.0°C
Prediction (Building Model): At 09:00, indoor will drop to 19.2°C (weather cooling)
Action: Pre-heat NOW to 21.0°C to compensate
```

`Result: At 09:00, indoor stabilizes at 20.0°C (user sees no drop)`

Requires: Fase 2 (Building Model Learner)

Glossary

Adaptive Control: Automatic adjustment of heat pump setpoint based on room temperature feedback

PI Controller: Proportional-Integral feedback control algorithm

Deadband: Temperature range where no control action taken (prevents oscillation)

Error: Difference between target and actual temperature

Proportional Term (P): Control response proportional to current error

Integral Term (I): Control response based on accumulated error history

Thermal Mass: Building's heat capacity (how long to heat/cool)

Heat Loss Coefficient (UA): Rate of heat loss through walls/windows (insulation quality)

DPS 4: Tuya data point for target temperature (steltemperatuur)

DPS 13: Tuya data point for heating curve (stooklijn)

Step:1: Capability constraint allowing only integer values (no decimals)

Accumulation Logic: Summing fractional adjustments until rounding threshold reached

Persistence: Storing data in device store to survive app updates/restarts

---

Version History

v1.3.0 (December 2024) - Fase 1 MVP Complete

- AdaptiveControlService implemented
- HeatingController (PI algorithm) implemented
- ExternalTemperatureService implemented
- Flow cards registered (1 action, 2 triggers)
- Capability added: `adlar_external_indoor_temperature`
- ServiceCoordinator integration complete
- Documentation: Setup guide, architecture guide
- Persistence strategy implemented
- Accumulation logic for step:1 rounding

Future Versions:

- **v1.4.0** (Planned) - Expert Mode settings (enable/disable, PI tuning)
- **v1.5.0** (Planned) - Fase 2: Building Model Learner
- **v1.6.0** (Planned) - Fase 3: Energy Price Optimizer
- **v1.7.0** (Planned) - Fase 4: COP Controller
- **v2.0.0** (Planned) - Weighted Decision System (all 4 controllers)

References

Internal Documentation:

- CLAUDE.md - Service Architecture
- Setup Guide - Adaptive Control
- Service Architecture Guide

External Resources:

- PI Controller Theory (Wikipedia)
- Ziegler-Nichols Tuning Method
- Homey Apps SDK - Flow Cards
- Homey Apps SDK - Device Store

Academic Papers:

- EN 14825:2018 - Air conditioners, liquid chilling packages and heat pumps (SCOP standard)
- ISO 13790:2008 - Energy performance of buildings (thermal mass calculation)

End of Document