Matter Device UI Standard (MDUI)

A Proposal for Cross-Platform Device User Interface Rendering

Proposal Status: Draft for Review **Target:** Connectivity Standards Alliance (CSA) **Author:** Lowpan **Version:** 0.2

Executive Summary

The Problem

Matter has successfully standardized smart home device communication—discovery, commissioning, and control protocols are now interoperable across platforms. However, **Matter does not standardize device user interfaces**.

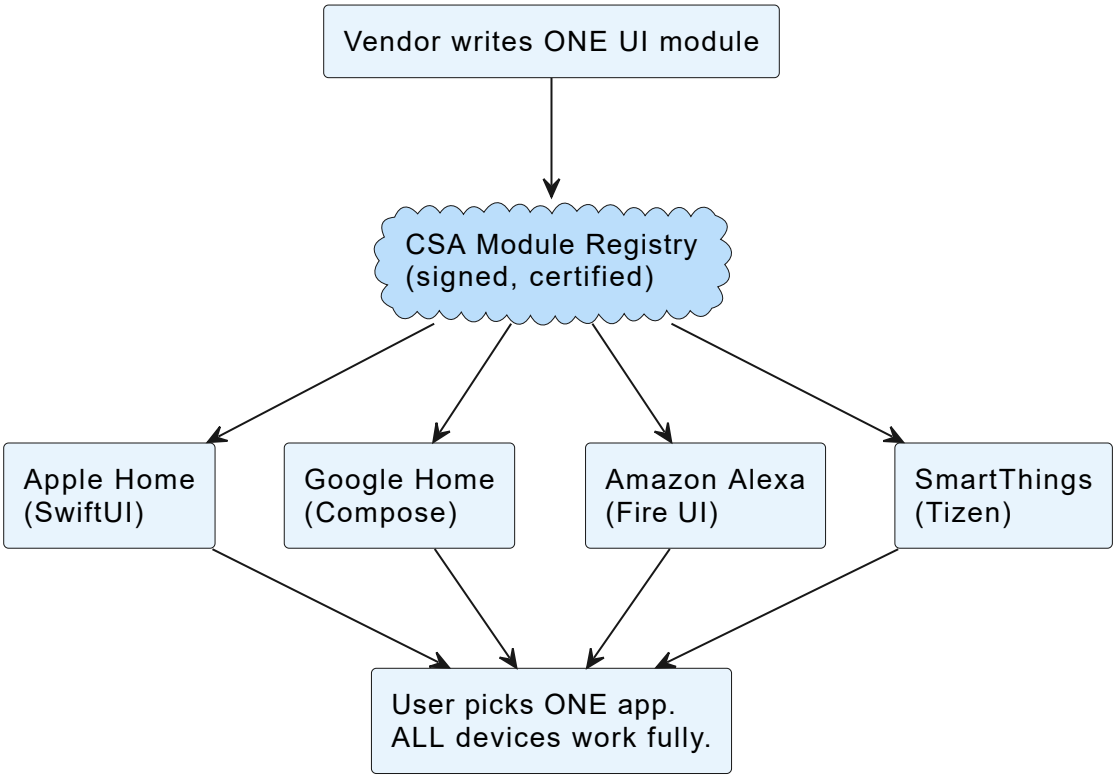
The result:

- Users must download separate vendor apps for full device functionality
- "For schedules, download the Ecobee app"
- "For color scenes, download the Philips Hue app"
- A typical smart home user has 10-15 device apps installed
- **Matter's interoperability promise remains half-fulfilled**

Platform apps (Apple Home, Google Home, Amazon Alexa) provide basic controls—on/off, brightness—but cannot render vendor-specific features, custom clusters, or branded experiences. Vendors continue building proprietary apps, fragmenting the user experience Matter was designed to unify.

The Solution

Matter Device UI Standard (MDUI): A specification for portable, declarative device user interfaces that render natively across all Matter controller platforms.

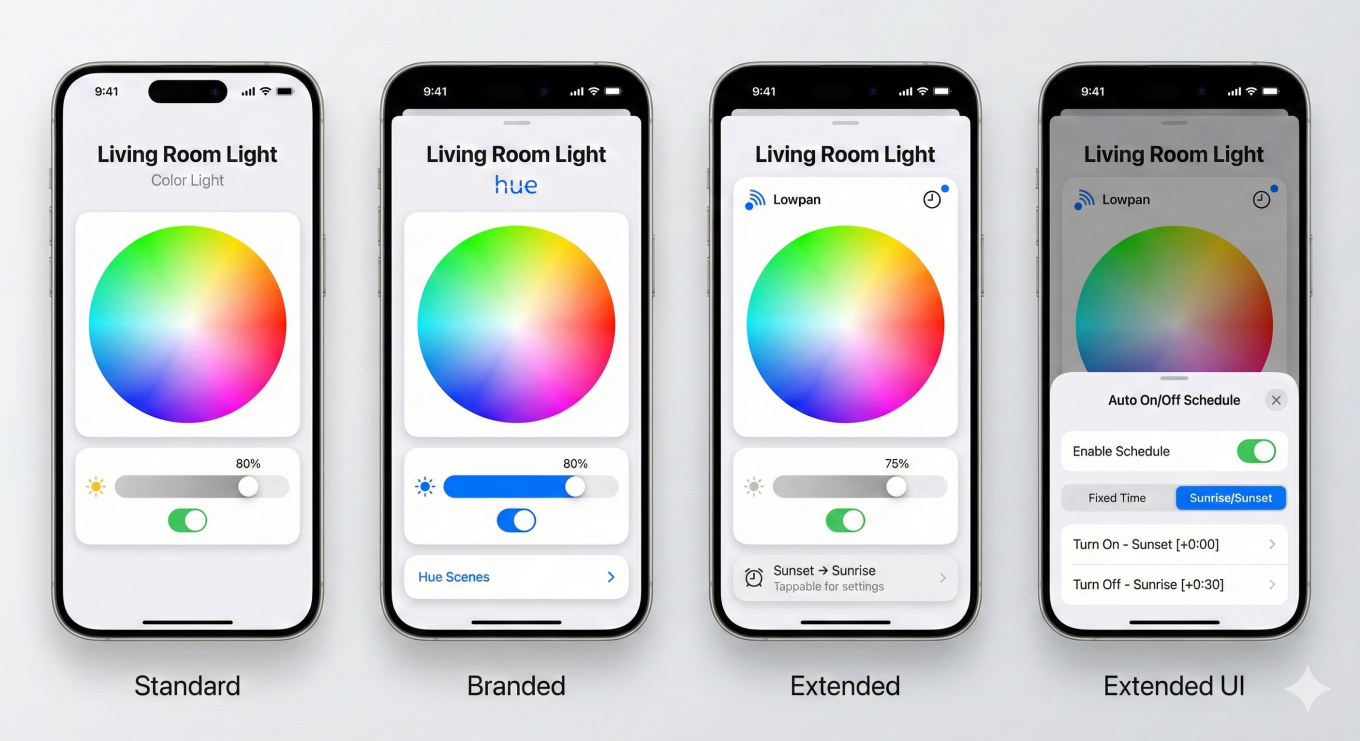


Key Innovation: UI modules are **sandboxed executable code** that produces declarative UI descriptions. Lua code runs in a restricted environment—no file/network access—and outputs a UI tree. Platforms render the tree using native frameworks. This enables conditional logic (show heat slider only in heat mode) while maintaining security.

Critical Design Decision: ALL Device UI is Lua. Platforms do not have built-in native UIs for device types. Every device—from basic lights to complex thermostats—renders through Lua modules. CSA provides standard Lua modules for all standard device types. This enables **true code inheritance**: vendors fork the standard module's actual Lua code, then add branding and features. The color wheel, sliders, and toggles in a vendor module are literally the same code as the standard module—not a reimplementation that happens to look similar.

Privacy by Design: All rendering happens locally on the user's device. No user data is sent to vendors or CSA. Modules receive only device state from the local Matter fabric.

Visual Example: Progressive UI Customization



Four levels of MDUI rendering for a color light, all using native iOS UI components:

Screen	Description
Standard	CSA-provided generic module. Basic controls, no branding. Works for any color light.
Branded	Vendor (Philips Hue) adds logo and brand colors. "Hue Scenes" button for vendor features.
Extended	Vendor (Lowpan) adds schedule icon in header. Summary card shows "Sunset → Sunrise" status.
Extended UI	Tapping schedule icon opens native bottom sheet with full scheduling controls.

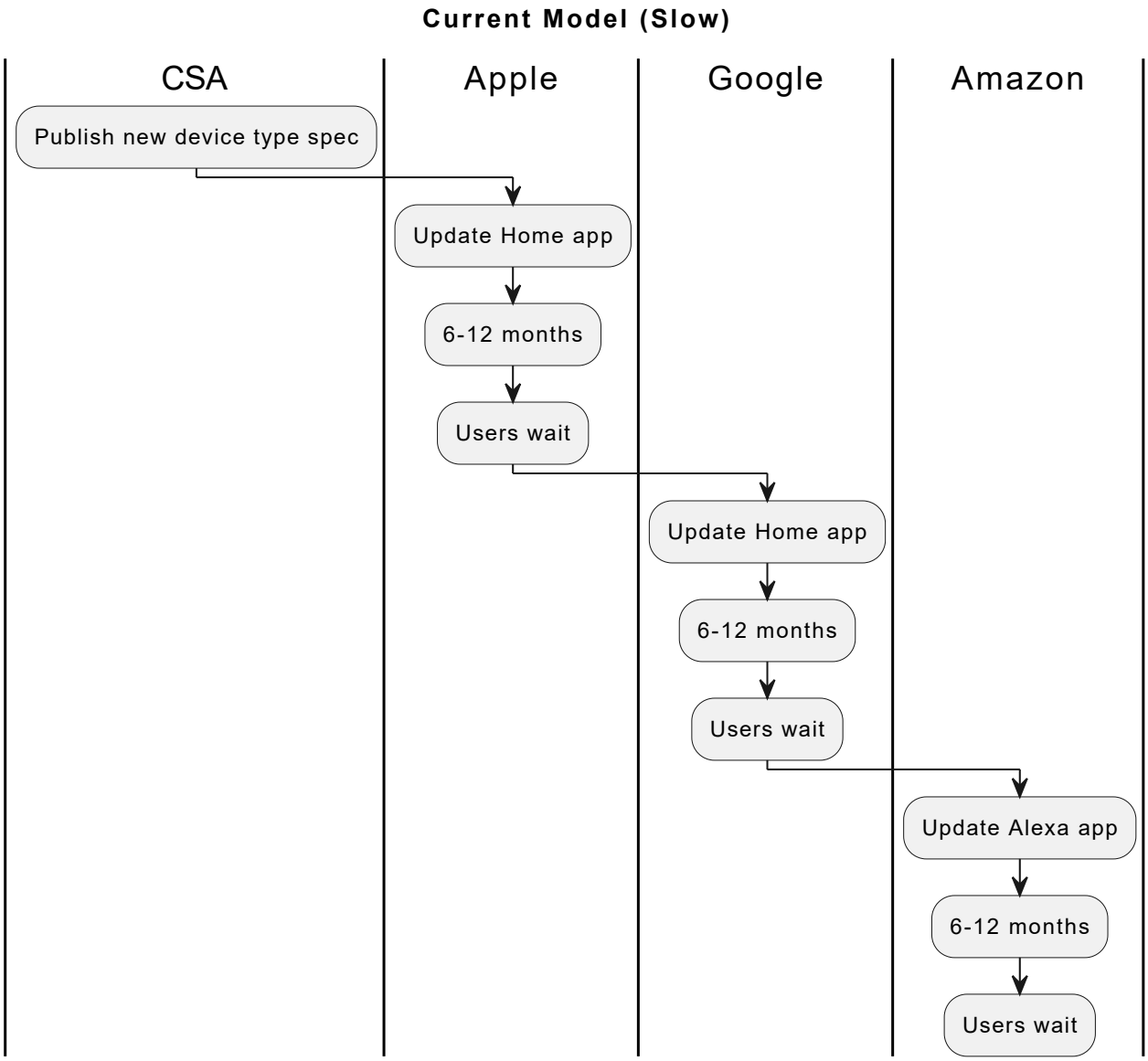
The same module renders with platform-native styling on iOS (shown), Android, and other platforms.

Value Proposition

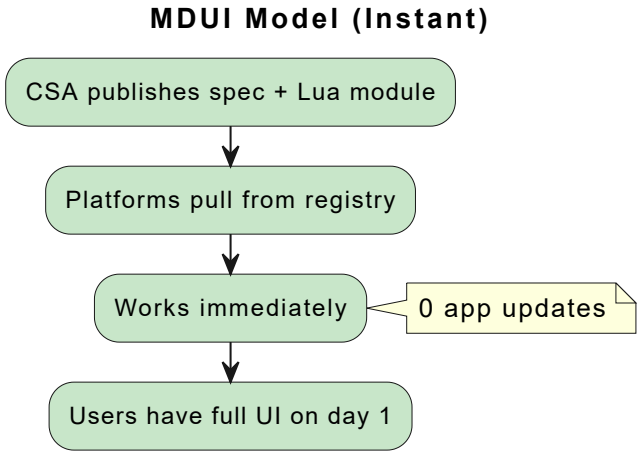
Stakeholder	Benefit
Users	One app for everything. Full device control. No more "download our app."
Device Vendors	Write once, reach all platforms. Brand presence everywhere. No app maintenance.
Platform Vendors	Richer ecosystem. Better UX. Competitive differentiator.
CSA	Fulfills Matter's interoperability promise. Drives adoption.

The Game Changer: Instant New Device Type Support

Today, when CSA adds a new device type (e.g., Robot Vacuum, Pet Feeder, Pool Controller):



With MDUI:



Example: CSA ratifies "Automated Bird Feeder" device type (0x0095):

1. CSA publishes `standard/bird_feeder` Lua module to registry
2. Apple Home, Google Home, Amazon Alexa automatically fetch module

3. User buys bird feeder, commissions it
4. Full UI appears—schedule feeding, check food level, view camera
5. **No platform app updates required. Ever.**

This fundamentally changes the Matter ecosystem dynamics:

- **CSA controls the pace of innovation**, not platform release cycles
- **New device types reach users in days**, not years
- **Platforms compete on UX quality**, not device type coverage
- **Vendors can ship new product categories immediately**

Why Now

1. **Matter adoption is accelerating** - Billions of devices expected by 2027
2. **User frustration is mounting** - App fatigue is limiting smart home adoption
3. **Vendors want relief** - Mobile app development is expensive and distracting
4. **Platforms are ready** - Declarative UI frameworks (SwiftUI, Compose) make this feasible
5. **The architecture exists** - Server-driven UI is proven at scale (Airbnb, Shopify, Netflix)

Request

We propose the CSA establish a working group to develop the Matter Device UI Standard, with:

1. Review of this technical proposal
2. Input from platform vendors (Apple, Google, Amazon, Samsung)
3. Input from device vendors (Philips, Eve, Yale, Nanoleaf, etc.)
4. Development of formal specification
5. Reference implementations for iOS and Android

Architectural Precedent

MDUI is not a novel architecture—it applies proven patterns used at scale by industry leaders.

Server-Driven UI (SDUI)

Major companies use server-driven UI to decouple UI logic from client releases:

Company	Implementation	Scale
Airbnb	Server sends JSON UI descriptions, native renderers on iOS/Android	Millions of users
Shopify	Mobile app renders merchant-customized storefronts from server data	Millions of merchants
Lyft	Dynamic ride experience UI driven by server responses	Millions of rides/day
Netflix	Personalized UI layouts delivered as data	200M+ subscribers

The pattern:

1. **Server** (or module) sends a declarative UI description
2. **Client** (or platform) has native renderers for each component type

3. **Result**: UI updates without client/platform releases

MDUI applies this exact pattern: vendor modules send UI descriptions, platform apps render natively.

Apple WidgetKit

Apple's WidgetKit is architecturally identical to MDUI:

Aspect	WidgetKit	MDUI
UI Model	Declarative SwiftUI	Declarative Lua → UINode
Primitives	Limited SwiftUI views	Limited UI primitives
Rendering	iOS renders the widget	Platform renders the module
Execution	Sandboxed, limited APIs	Sandboxed, limited APIs
Source	App provides widget	Vendor provides module
Host	iOS Home/Lock Screen	Apple Home / Google Home

```
// WidgetKit: App provides declarative UI for iOS to render
struct WeatherWidget: Widget {
    var body: some WidgetConfiguration {
        StaticConfiguration(...) { entry in
            VStack {
                Text(entry.temperature)
                Image(systemName: entry.icon)
            }
        }
    }
}
```

```
-- MDUI: Vendor provides declarative UI for platform to render
function Module.render(device, state)
    return UI.Column {
        children = {
            UI.Text { text = state.temperature },
            UI.Icon { name = state.icon }
        }
    }
end
```

WidgetKit is MDUI for the home screen. MDUI is WidgetKit for smart home devices.

React Native / Flutter

Cross-platform frameworks that render native UI from a single codebase:

Framework	How it works	MDUI parallel
React Native	JavaScript describes UI, native renderers on each platform	Lua describes UI, native renderers on each platform
Flutter	Dart describes UI, Skia renders consistently	Lua describes UI, platform renders natively

MDUI differs in one key way: **modules produce declarative output**. The Lua code runs, but it only outputs a UI tree—no direct rendering, no platform API access. This makes it safer (sandboxed execution) and more consistent (platforms control rendering).

Why This Matters

These patterns prove:

1. **Declarative UI descriptions work** - Billions of users interact with SDUI daily
2. **Platform-native rendering works** - WidgetKit, React Native prove the model
3. **Sandboxed execution works** - WidgetKit runs untrusted code safely
4. **The industry is ready** - Developers understand these patterns

MDUI simply applies these proven patterns to smart home device interfaces.

Why Lua?

Q: Why not pure JSON/YAML for declarative UI?

A: Device UIs need logic:

- Show heat slider only when mode is "Heat"
- Calculate percentage from raw sensor value
- Format temperature based on locale
- Conditionally render based on device capabilities

Pure data formats can't express this. We need a language.

Q: Why Lua specifically?

Requirement	Why Lua Fits
Easy to sandbox	Lua was designed for embedding; trivial to remove dangerous APIs
Small runtime	~200KB, fits in any app
Fast startup	No JIT warmup, instant execution
Simple syntax	Vendors learn it in hours, not days
Battle-tested	Powers Roblox (70M+ daily users), Redis, Nginx, game engines
No dependencies	Single library, no package managers

Luau (Roblox's typed Lua variant) adds type safety and is MIT-licensed.

Q: Why not JavaScript?

JavaScript runtimes (V8, JavaScriptCore) are large, have complex sandbox requirements, and bring ecosystem expectations (npm, bundlers) that add friction. Lua is purpose-built for embedding.

Part 1: Problem Statement

Current State of Matter

Matter 1.0+ provides:

- ☑ Unified device discovery (mDNS/DNS-SD)
- ☑ Standardized commissioning (PASE, CASE)
- ☑ Interoperable control protocol (TLV over TCP/UDP)
- ☑ Common data model (clusters, attributes, commands)
- ☑ Multi-admin support (devices work with multiple platforms)
- ❌ **No standard for user interface**

The Fragmentation Problem

For Users

A user with a typical smart home (lights, thermostat, locks, sensors) experiences:

Device	In Apple Home	Full Features
Philips Hue	On/Off, Brightness, Color	Scenes, Entertainment, Routines → **Requires Hue app**
Ecobee Thermostat	Temperature, Mode	Schedules, Sensors, Reports → **Requires Ecobee app**
Yale Lock	Lock/Unlock	User codes, History, Auto-lock → **Requires Yale app**
Nanoleaf	On/Off, Color	Rhythm, Scenes, Touch actions → **Requires Nanoleaf app**

Result: 4 devices = 5 apps (platform + 4 vendor apps)

This directly contradicts Matter's value proposition: "Buy any device, use any app."

For Device Vendors

Vendors face a difficult choice:

Option A: Basic Matter support only

- Limited to standard cluster controls
- No differentiation from competitors
- Features that make their product special are inaccessible

Option B: Build and maintain mobile apps

- iOS app (Swift, yearly iOS updates)
- Android app (Kotlin, device fragmentation)
- Ongoing maintenance, bug fixes, feature parity
- App store fees, review processes
- Support burden

Most vendors choose Option B, perpetuating fragmentation.

For Platform Vendors

Platform apps (Apple Home, Google Home) cannot render:

- Vendor-specific clusters (custom features)
- Vendor branding (logo, colors, messaging)
- Advanced configurations (schedules, automations, settings)
- Device-specific UI patterns (color wheels, thermostat arcs)

Platforms are limited to generic controls, making their apps feel incomplete.

The Opportunity

What if vendors could ship a UI definition with their device—one that renders natively on every platform?

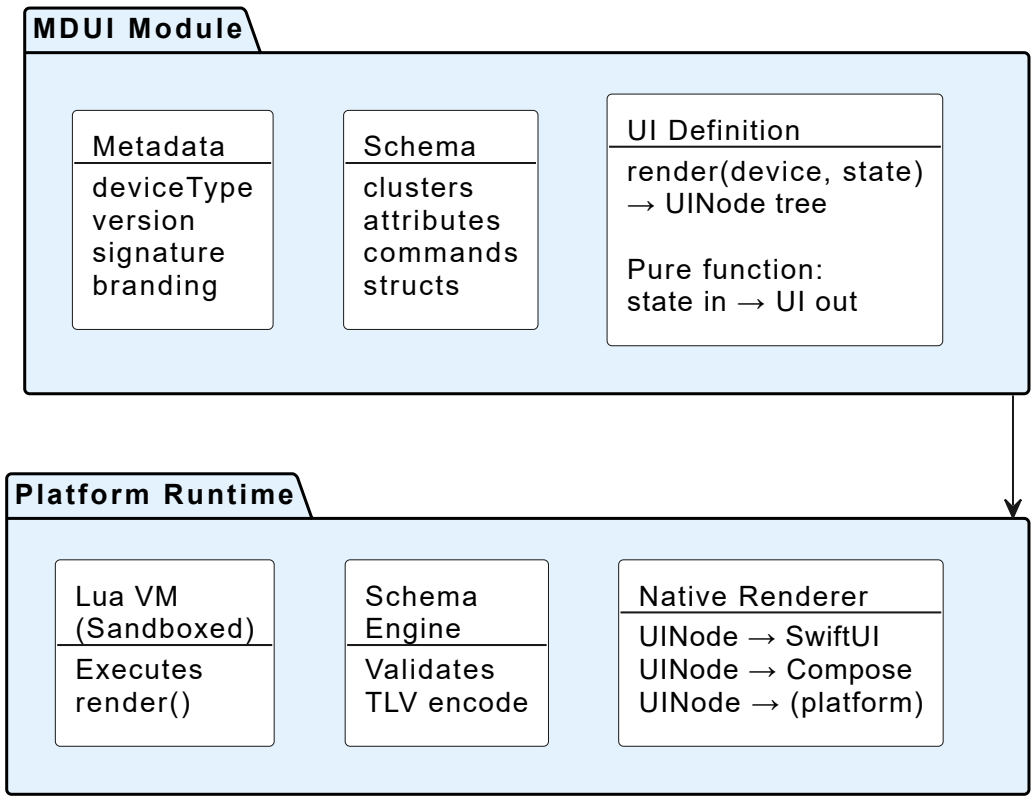
- Users get full functionality in their preferred app
- Vendors reach all platforms without building apps
- Platforms offer richer experiences
- Matter's promise is fulfilled

Part 2: Technical Architecture

Design Principles

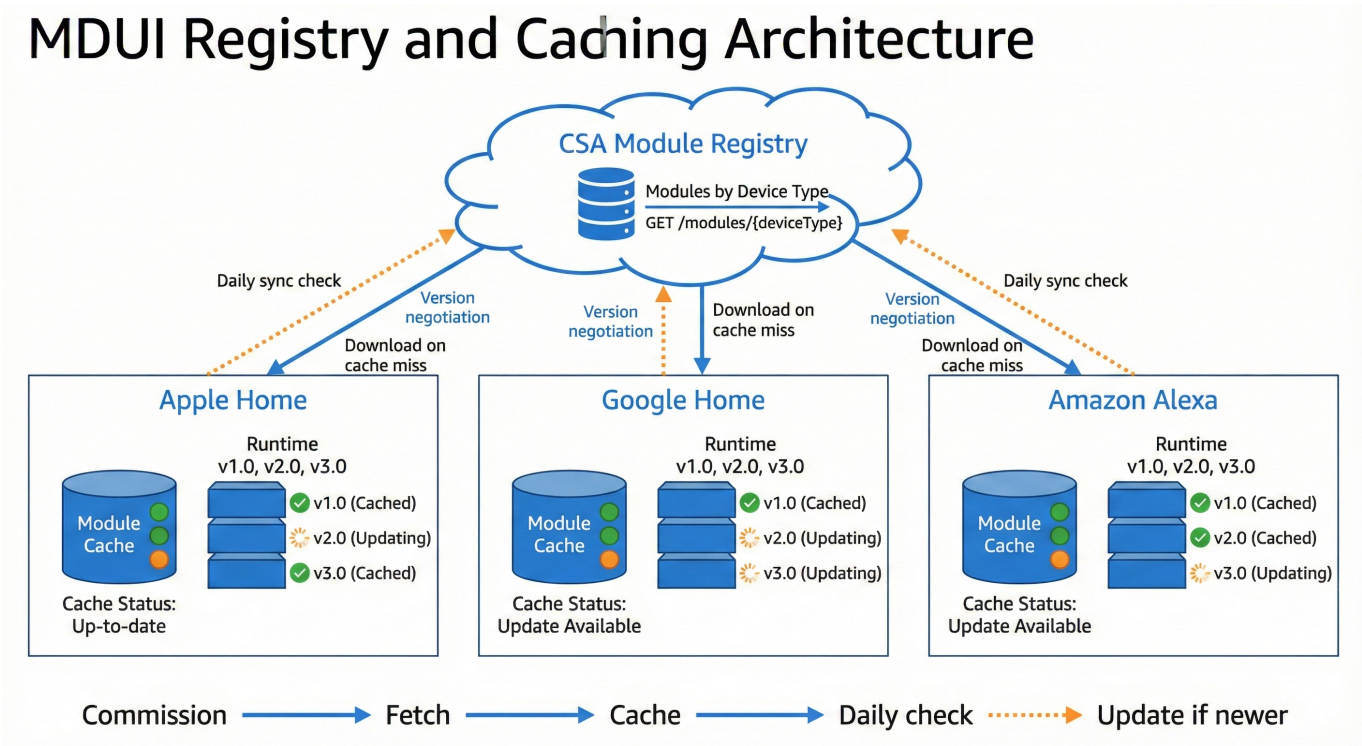
1. **Declarative, not imperative** - Modules describe UI structure, not rendering logic
2. **Platform-native rendering** - Each platform uses its own UI framework
3. **Sandboxed execution** - Modules cannot access device resources beyond Matter
4. **Reactive state model** - UI updates automatically when device state changes
5. **Vendor extensibility** - Custom clusters and UIs without platform updates
6. **Graceful degradation** - Platforms render what they support, ignore what they don't

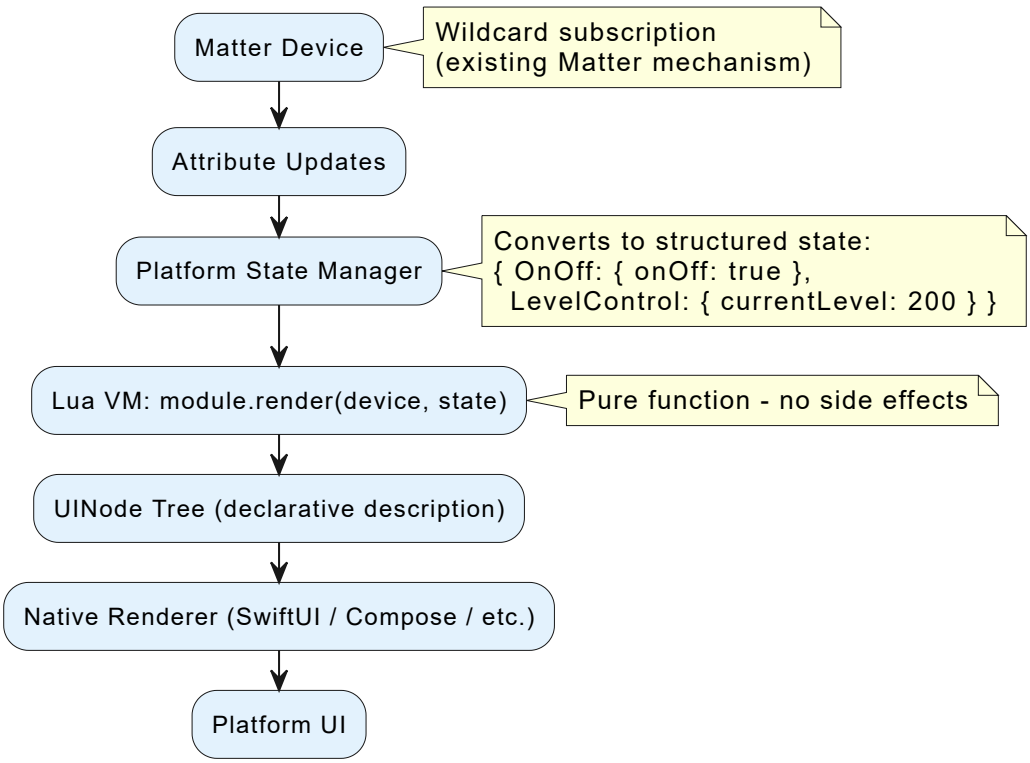
Architecture Overview



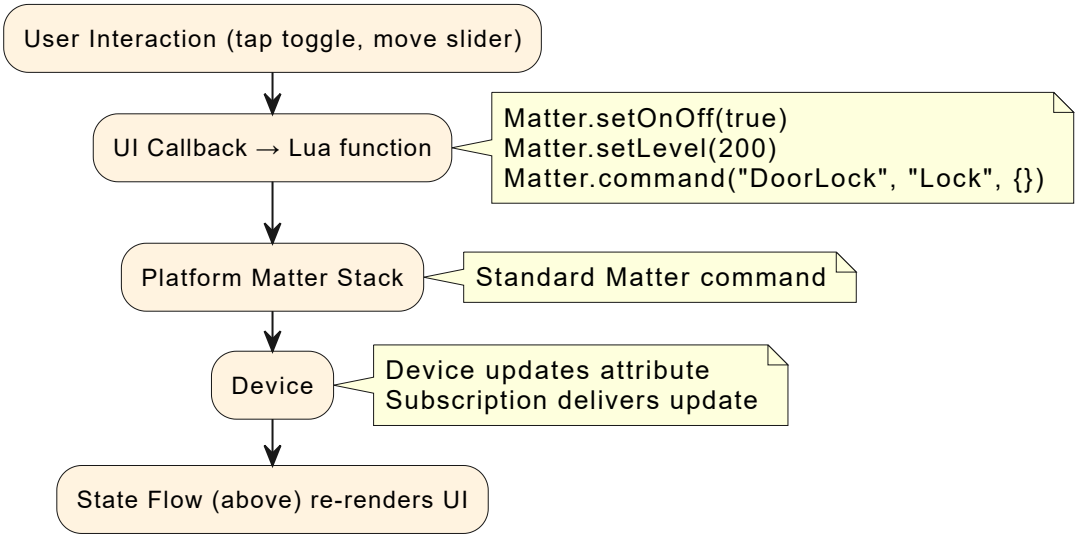
Execution Model

State Flow (Reactive)



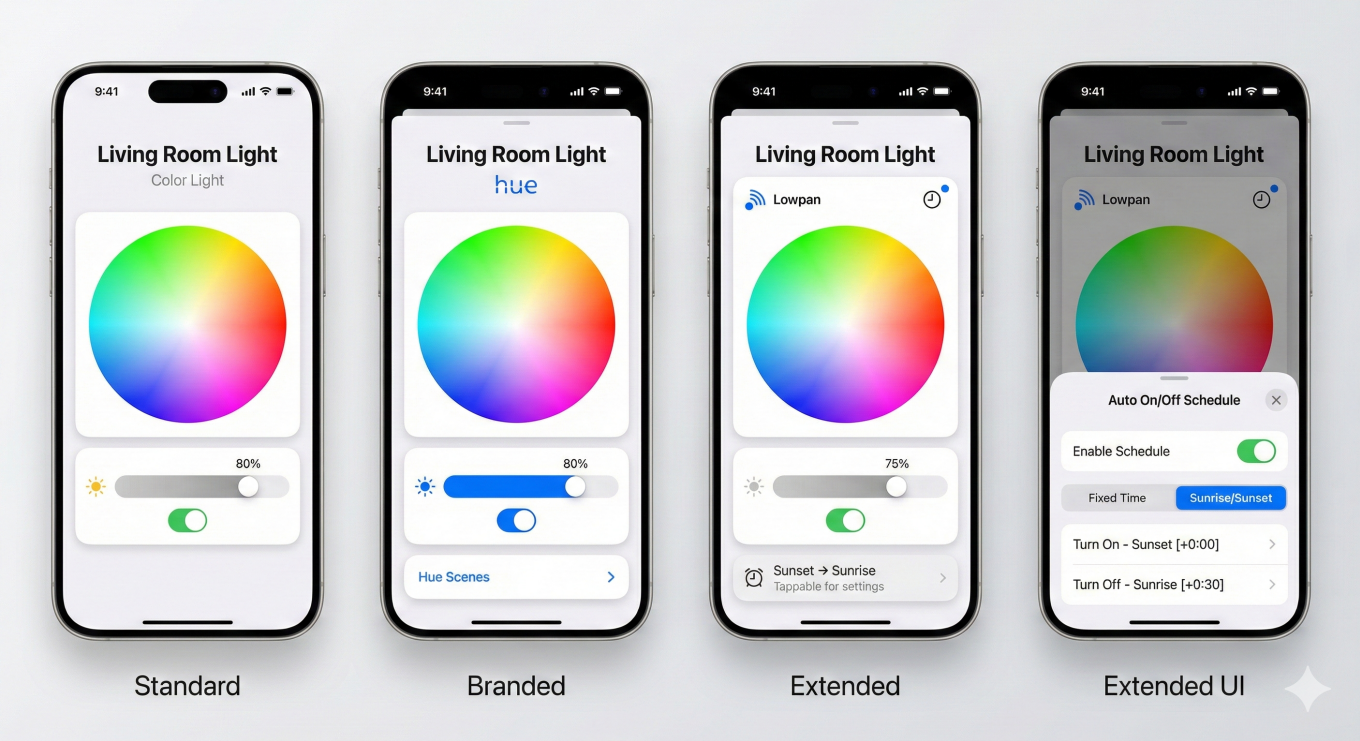


Command Flow (Fire-and-Forget)



Key insight: Commands are fire-and-forget. State updates flow back through the subscription. No callbacks, no async handling in Lua.

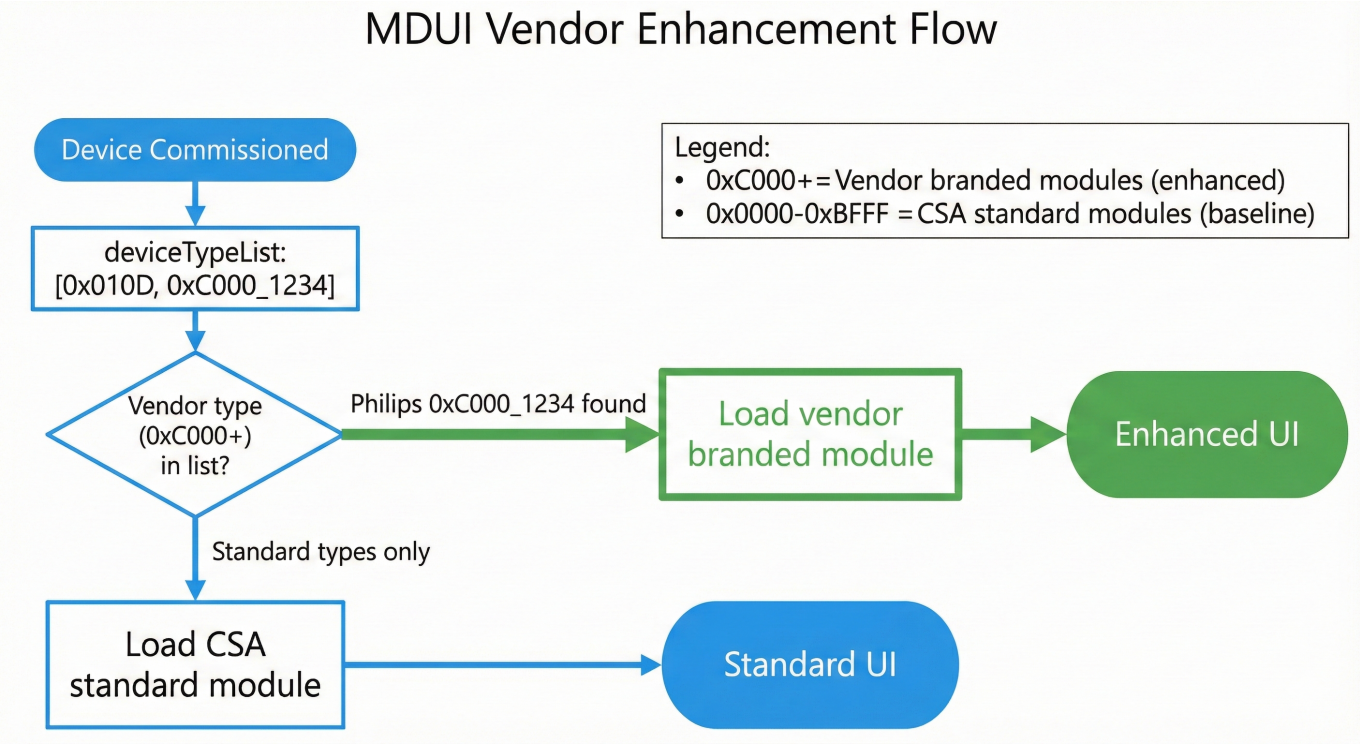
Branded UI Enhancement



The image above shows how vendors progressively enhance the standard UI:

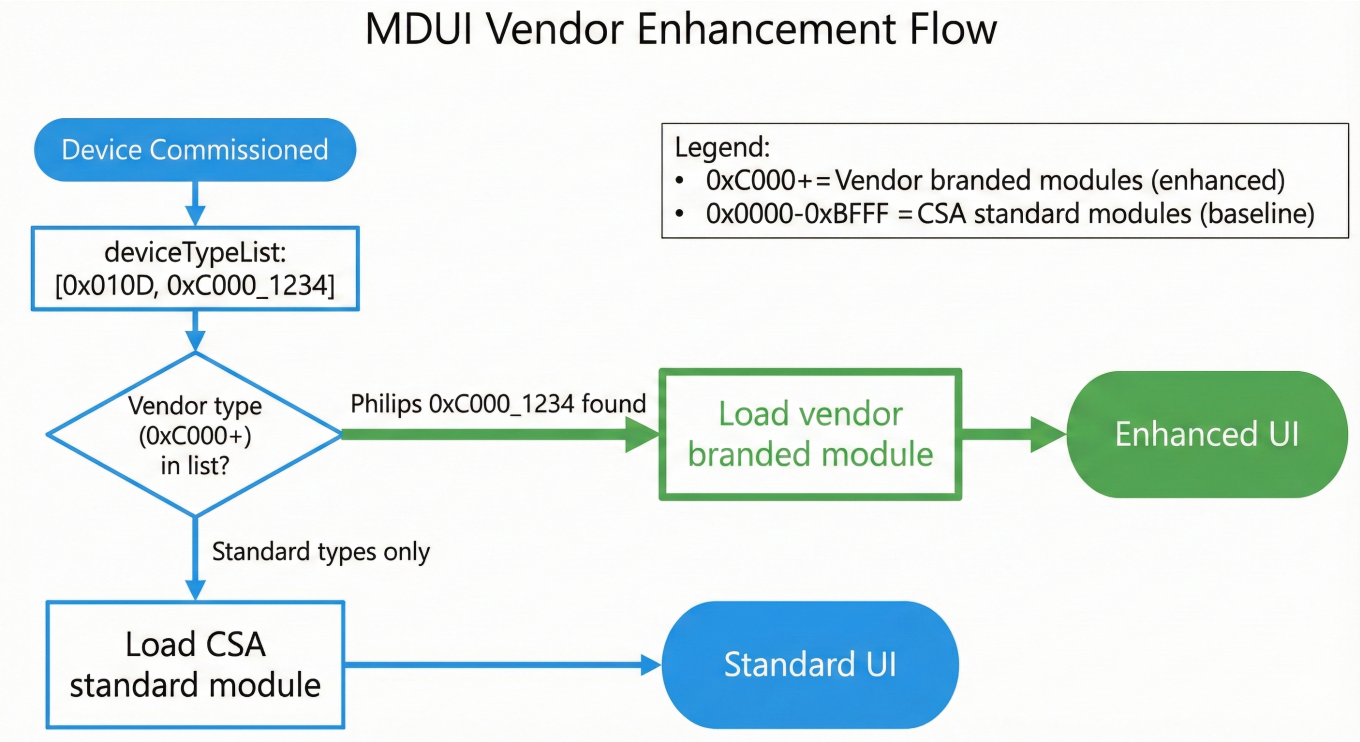
1. **Standard** - CSA standard module renders basic device controls (color wheel, brightness, power toggle)
2. **Branded** - Vendor module adds branding (logo) and vendor-specific features ("Hue Scenes" button)
3. **Extended** - Vendor module surfaces additional capabilities (schedule preview: "Sunset → Sunrise")
4. **Extended UI** - Tapping the schedule row opens a sheet with full configuration options

Vendors build ON TOP of the standard UI patterns—same color wheel, same slider, same toggle—but with their branding and extended features. Users get a consistent experience across devices while vendors can differentiate.



Module Discovery Flow

MDUI uses **vendor-specific device types** to trigger branded UI modules. Matter already supports endpoints with multiple device types in the `deviceTypeList`. Vendors add their vendor-specific device type alongside the standard device type to enable their enhanced UI.



Device Type ID Ranges:

- `0x0000` - `0xBFFF`: Standard CSA device types (e.g., 0x010D = Extended Color Light)
- `0xC000` - `0xFFFE`: Manufacturer-specific device types (vendor prefix encoded in ID)

How It Works:

1. User commissions new device
2. Platform reads device descriptor cluster:
    - deviceTypeList: [0x010D, 0xC000_1234] │ │ └── Philips-specific UI device type (triggers branded module) │ └── Standard Extended Color Light (baseline) └── Endpoint supports BOTH device types
3. Platform scans deviceTypeList for vendor-specific types (0xC000+):
    - Found 0xC000_1234 → GET /modules/0xC000_1234
    - Registry returns Philips branded module
4. Module loaded: { "deviceType": "0xC000_1234", "module": "philips-hue-color-light", "version": "2.1.0", "url": "https://..." }
5. Platform downloads, verifies signature, caches, renders branded UI

Standard baseline (no vendor enhancement):

If device only has standard types (no 0xC000+ types in deviceTypeList):

- deviceTypeList: [0x010D] → Only standard color light
- GET /modules/0x010D → CSA standard module
- Render using standard module (full-featured, unbranded)

Error handling:

If registry unavailable or module fails verification:

- Platform renders basic controls from cluster list
- (Rare case - modules are cached locally)

Example deviceTypeLists:

Standard light (no vendor UI): deviceTypeList: [0x010D] → Uses standard/color_light module

Philips Hue light (vendor UI): deviceTypeList: [0x010D, 0xC000_1234] → Uses Philips module

Ecobee thermostat (vendor UI): deviceTypeList: [0x0301, 0xC000_5678] → Uses Ecobee module

Benefits of Device Type-Based Discovery:

1. **No vendorId matching** - Device type ID already encodes vendor identity
2. **Explicit opt-in** - Vendors choose which products get branded UI
3. **Standard baseline always works** - CSA modules for all standard device types ensure every device has full UI
4. **Already standardized** - Uses existing Matter descriptor cluster mechanism

Module Structure

Lua Module Format

Modules are registered by **device type ID**. Vendor modules use vendor-specific device type IDs (0xC000-0xFFFE range), while CSA standard modules use standard device type IDs (0x0000-0xBFFF range).

```
-- Module metadata
local Module = {
    name = "Philips Hue Color Light",
    version = "1.2.0",

    -- Device type this module handles
    -- Vendor-specific device type (0xC000+ range)
    deviceType = 0xC000_1234,

    -- Standard device type this module extends (for validation)
    -- Module only loads if device also has this standard type
    baseDeviceType = 0x010D,  -- Extended Color Light

    -- Minimum platform runtime version
    minRuntimeVersion = "1.0",

    -- Required UI primitives
    requiredPrimitives = { "Toggle", "Slider", "ColorWheel", "Tabs" },
```

```
    -- Vendor branding
    branding = {
        logo = "https://cdn.philips-hue.com/logo.png",
        accentColor = "#0066FF"  -- Suggestion; platform may override
    },

    -- Custom cluster schemas (if any)
    schemas = {
        -- Inline or URL reference
    }
}

-- Pure render function
-- Called whenever device state changes
function Module.render(device, state)
    local onOff = state.OnOff or {}
    local level = state.LevelControl or {}
    local color = state.ColorControl or {}

    return UI.Column {
        spacing = 16,
        children = {
            -- Vendor logo (optional)
            Module.branding.logo and UI.Image {
                src = Module.branding.logo,
                height = 24
            },

            -- Power toggle
            UI.Toggle {
                value = onOff.onOff or false,
                label = "Power",
                onChange = function(v)
                    Matter.setOnOff(v)
                end
            },

            -- Brightness slider
            UI.Slider {
                value = level.currentLevel or 0,
                min = 1,
                max = 254,
                label = "Brightness",
                onChange = function(v)
                    Matter.setLevel(v)
                end
            },

            -- Color controls (tabbed)
            UI.Tabs {
                tabs = {
                    {
                        label = "Color",
```

```
                        content = UI.ColorWheel {
                            hue = color.currentHue or 0,
                            saturation = color.currentSaturation or 0,
                            onChange = function(h, s)
                                Matter.setColor({ hue = h, saturation = s })
                            end
                        }
                    },
                    {
                        label = "Temperature",
                        content = UI.Slider {
                            value = color.colorTemperatureMireds or 370,
                            min = 153,
                            max = 500,
                            label = "Color Temperature",
                            onChange = function(m)
                                Matter.setColorTemperature(m)
                            end
                        }
                    }
                }
            }
        }
    }
end

return Module
```

UI Primitive Specification

Core Primitives (Required)

All conforming platforms MUST implement these primitives:

Primitive	Purpose	Key Properties
`Column`	Vertical layout	children, spacing, padding, alignment
`Row`	Horizontal layout	children, spacing, alignment
`Text`	Labels, values	text, size, weight, color
`Toggle`	On/Off switch	value, label, onChange
`Slider`	Range input	value, min, max, step, label, onChange
`Button`	Actions	label, onClick, variant, icon
`Card`	Grouped content	children, title
`Icon`	Semantic icons	name, size, color
`Spacer`	Flexible space	-

Primitive	Purpose	Key Properties
`Divider`	Visual separator	-

Extended Primitives (Recommended)

Platforms SHOULD implement these for richer experiences:

Primitive	Purpose	Key Properties
`ColorWheel`	Hue/Saturation picker	hue, saturation, onChange
`ColorTemperature`	Warm/Cool slider	mireds, min, max, onChange
`SegmentedControl`	Mode selection	value, options, onChange
`Tabs`	Tabbed content	tabs, selected
`Image`	Logos, icons	src, width, height
`ProgressBar`	Loading, levels	value, max

Navigation Primitives (Extended UI Screens)

Many devices have extended features beyond basic controls—schedules, automation rules, advanced settings. MDUI supports secondary screens via navigation primitives:

Primitive	Purpose	Key Properties
`Sheet`	Modal bottom sheet for secondary content	content, title, icon, onClose
`IconButton`	Compact button to trigger navigation	icon, label, onPress
`TimePicker`	Time selection	value, onChange, mode ("time", "duration")
`DatePicker`	Date selection	value, onChange
`ListItem`	Row in a list with optional controls	title, subtitle, trailing, onPress
`Switch`	Inline toggle for settings	value, onChange

Example: Lowpan Light with Auto On/Off Scheduling

```
local Module = {
    name = "Lowpan Smart Light",
    version = "1.0.0",
    deviceType = 0xC000_ABCD,  -- Lowpan vendor device type
    standardDeviceTypes = { 0x010D },  -- Extended Color Light
}

function Module.render(device, state)
    local onOff = state.OnOff or {}
    local level = state.LevelControl or {}
    local schedule = state.LowpanSchedule or {}  -- Custom cluster
```

```
    return UI.Column {
        spacing = 16,
        children = {
            -- Header row with settings icon
            UI.Row {
                alignment = "spaceBetween",
                children = {
                    UI.Image { src = "https://lowpan.com/logo.png", height = 24 },

                    -- Icon button opens the schedule sheet
                    UI.IconButton {
                        icon = "schedule",
                        label = "Schedule",
                        onPress = function()
                            Navigation.openSheet("schedule")
                        end
                    }
                }
            },

            -- Standard controls
            UI.Toggle {
                value = onOff.onOff or false,
                label = "Power",
                onChange = function(v) Matter.setOnOff(v) end
            },

            UI.Slider {
                value = level.currentLevel or 0,
                min = 1, max = 254,
                label = "Brightness",
                onChange = function(v) Matter.setLevel(v) end
            },

            -- Schedule status indicator (on main screen)
            schedule.enabled and UI.Card {
                children = {
                    UI.Row {
                        children = {
                            UI.Icon { name = "schedule", color = "primary" },
                            UI.Text {
                                text = formatSchedule(schedule),
                                color = "secondary"
                            }
                        }
                    }
                }
            }
        },

        -- Define sheets (rendered when opened)
        sheets = {
            schedule = UI.Sheet {
```

```
                title = "Auto On/Off Schedule",
                content = renderScheduleSheet(state)
            }
        }
    }
end

-- Separate function for schedule sheet content
function renderScheduleSheet(state)
    local schedule = state.LowpanSchedule or {}

    return UI.Column {
        spacing = 16,
        children = {
            -- Enable/disable toggle
            UI.ListItem {
                title = "Enable Schedule",
                trailing = UI.Switch {
                    value = schedule.enabled or false,
                    onChange = function(v)
                        Matter.writeAttribute("LowpanSchedule", "enabled", v)
                    end
                }
            },

            UI.Divider {},

            -- Schedule mode selector
            UI.SegmentedControl {
                value = schedule.mode or 0,
                options = {
                    { label = "Fixed Time", value = 0 },
                    { label = "Sunrise/Sunset", value = 1 }
                },
                onChange = function(m)
                    Matter.writeAttribute("LowpanSchedule", "mode", m)
                end
            },

            -- Fixed time mode
            (schedule.mode == 0) and UI.Column {
                children = {
                    UI.ListItem {
                        title = "Turn On At",
                        trailing = UI.TimePicker {
                            value = schedule.onTime or "07:00",
                            onChange = function(t)
                                Matter.writeAttribute("LowpanSchedule", "onTime",
t)
                            end
                        }
                    },
                    UI.ListItem {
                        title = "Turn Off At",
```

```
                        trailing = UI.TimePicker {
                            value = schedule.offTime or "23:00",
                            onChange = function(t)
                                Matter.writeAttribute("LowpanSchedule", "offTime",
t)
                            end
                        }
                    }
                }
            },

            -- Sunrise/sunset mode
            (schedule.mode == 1) and UI.Column {
                children = {
                    UI.ListItem {
                        title = "Turn On",
                        subtitle = "Relative to sunset",
                        trailing = UI.TimePicker {
                            value = schedule.sunsetOffset or "00:00",
                            mode = "offset",  -- Shows +/- offset picker
                            onChange = function(t)
                                Matter.writeAttribute("LowpanSchedule",
"sunsetOffset", t)
                            end
                        }
                    },
                    UI.ListItem {
                        title = "Turn Off",
                        subtitle = "Relative to sunrise",
                        trailing = UI.TimePicker {
                            value = schedule.sunriseOffset or "00:00",
                            mode = "offset",
                            onChange = function(t)
                                Matter.writeAttribute("LowpanSchedule",
"sunriseOffset", t)
                            end
                        }
                    }
                }
            }
        }
    }
end

function formatSchedule(schedule)
    if schedule.mode == 1 then
        return "Sunset → Sunrise"
    else
        return (schedule.onTime or "07:00") .. " → " .. (schedule.offTime or
"23:00")
    end
end

return Module
```

How Navigation Works:

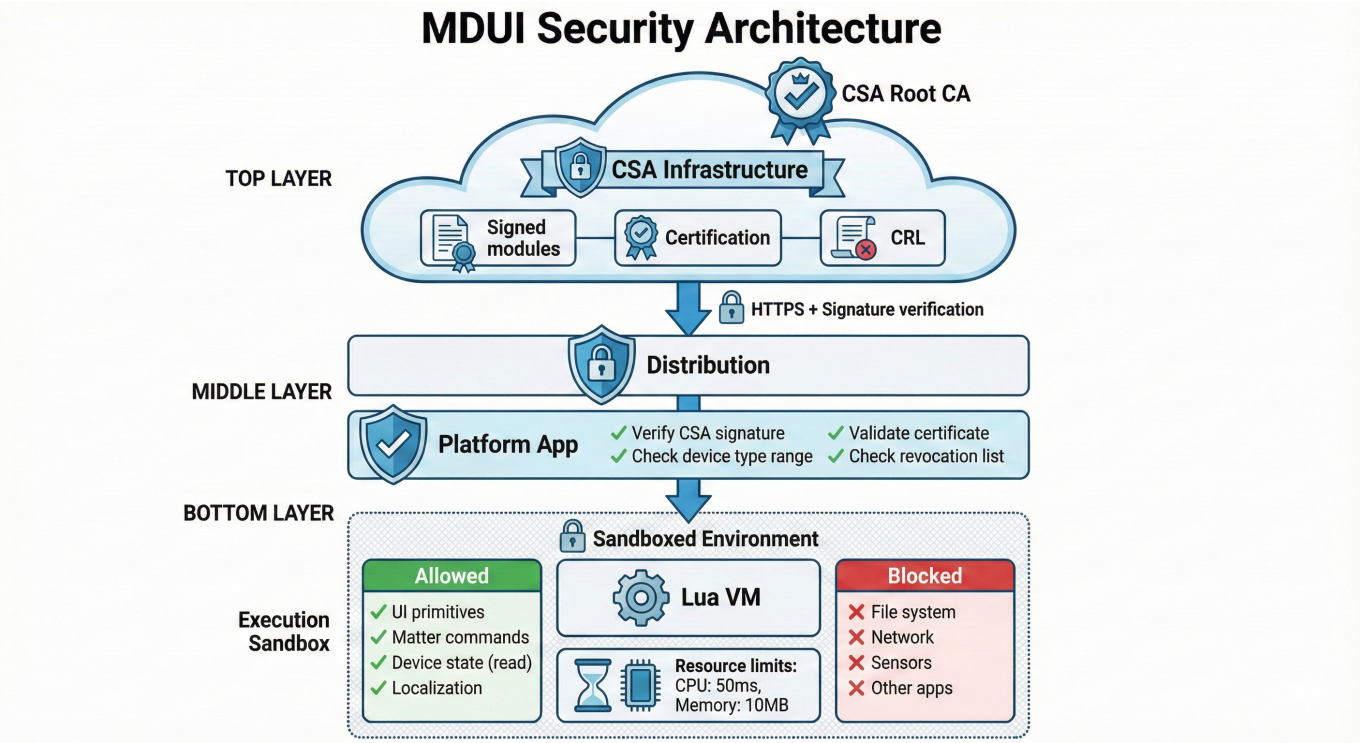
Main Screen
Logo Schedule Icon
Power Toggle: ON
Brightness: ====*====
Schedule: 7:00→23:00
← **tap icon**
Sheet (slides up)
Auto On/Off Schedule [X]
Enable Schedule [✓]
(●) Fixed ( ) Sunrise
Turn On: [07:00] Turn Off: [23:00]
Done

Platform Rendering of Sheets:

Platform	Sheet Rendering
iOS	`UISheetPresentationController` (native bottom sheet)
Android	`ModalBottomSheet` (Material 3)
Web	Modal dialog or slide-over panel

Sheets are platform-native, so they feel natural on each platform while the content inside is defined by the module.

Platform Rendering

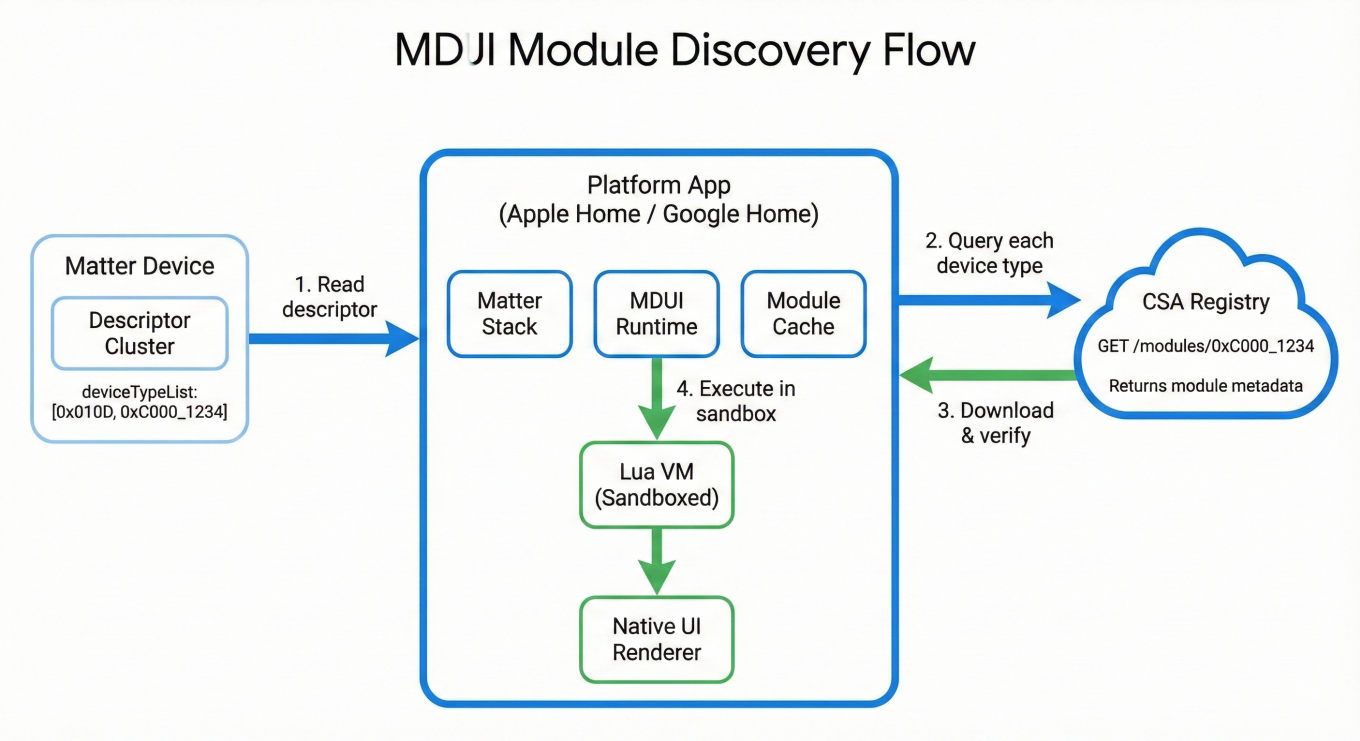


Each platform renders primitives using its native UI framework:

Primitive	SwiftUI (iOS)	Compose (Android)
`Column`	`VStack`	`Column`
`Row`	`HStack`	`Row`
`Text`	`Text`	`Text`
`Toggle`	`Toggle`	`Switch`
`Slider`	`Slider`	`Slider`
`Button`	`Button`	`Button` / `FilledTonalButton`
`Card`	`GroupBox`	`Card`
`Icon`	`Image(systemName:)`	`Icon` (Material)
`ColorWheel`	Custom View	Custom Composable
`Tabs`	`TabView`	`TabRow` + `Pager`

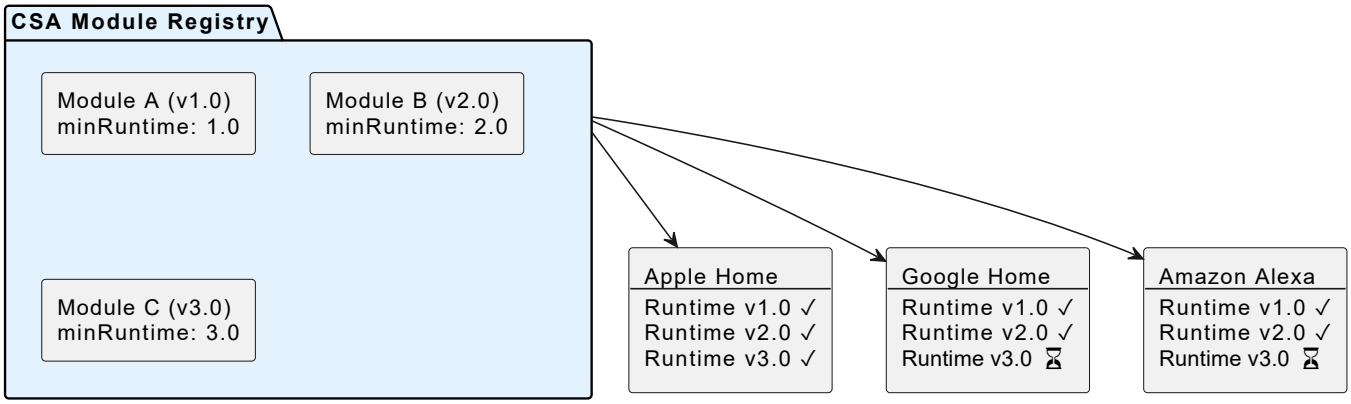
Styling: Platforms apply their own design language. A `Toggle` looks like an iOS toggle on Apple Home, a Material switch on Google Home. Modules do NOT control visual styling—only structure and behavior.

Versioning Model



MDUI uses a **runtime version** system that allows the specification to evolve while maintaining backward compatibility.

Version Architecture



Module Version Declaration

```
local Module = {
    name = "Philips Hue Color Light",
    version = "2.1.0",              -- Module's own version
    deviceType = 0xC000_1234,       -- Vendor-specific device type
    minRuntimeVersion = "2.0",      -- Minimum MDUI runtime required

    -- Optional: Declare required primitives for graceful degradation
    requiredPrimitives = { "Toggle", "Slider", "ColorWheel" },
    optionalPrimitives = { "ColorTemperatureArc" }  -- Falls back if missing
}
```

Platform Runtime Versions

Each MDUI runtime version defines:

1. **Core primitives** - Required for all platforms
2. **Extended primitives** - Recommended but optional
3. **Matter API surface** - Available commands and helpers
4. **Lua API surface** - Available globals and functions

Runtime	Release	Key Additions
1.0	Initial	Core primitives (Toggle, Slider, Button, etc.), basic Matter API
2.0	+1 year	ColorWheel, ColorTemperature, Tabs, enhanced Matter API
3.0	+2 years	Animation support, conditional visibility, nested navigation
4.0	+3 years	(future expansion)

Backward Compatibility Guarantees

For Platforms:

- Platforms MUST support at least 3 major runtime versions simultaneously
- Platforms MAY drop support for versions older than 5 years
- When loading a module, platform uses the runtime version the module requests

For Modules:

- Modules targeting runtime 1.0 will work on runtime 2.0, 3.0, etc. (forward compatible)
- Modules can declare `minRuntimeVersion` to require newer features
- Registry stores modules with version metadata for platform queries

Version Negotiation Flow

1. Platform discovers device with deviceTypeList: [0x010D, 0xC000_1234]
2. Platform queries registry for each device type: GET /modules/0xC000_1234?runtimeVersions=1.0,2.0,3.0
3. Registry returns best match: { "deviceType": "0xC000_1234", "module": "philips-hue-color-light", "moduleVersion": "2.1.0", "minRuntimeVersion": "2.0", "url": "https://..." }
4. Platform loads module with runtime v2.0 context

Multi-Version Module Support

Vendors MAY publish multiple module versions for different runtimes:

philips-hue-color-light/ ├── v1.0/ # Works with runtime 1.0+ │ └── module.lua # Basic UI (no ColorWheel) ├── v2.0/ # Works with runtime 2.0+ │ └── module.lua # Enhanced UI (ColorWheel, Tabs) └── v3.0/ # Works with runtime 3.0+ └── module.lua # Full UI (animations, transitions)

Registry serves the best version based on platform capabilities.

Primitive Evolution

New primitives are added in new runtime versions:

Primitive	Introduced	Notes
Toggle, Slider, Button	1.0	Core, never removed
ColorWheel	2.0	Can use in 2.0+ modules
AnimatedValue	3.0	Smooth transitions

Graceful Degradation: If a module uses a primitive the platform doesn't support:

1. Platform checks `optionalPrimitives` - skip if optional
2. Platform substitutes fallback (e.g., `ColorWheel` → `Slider` + `Slider`)
3. Platform shows warning if required primitive missing

CSA Governance of Versions

- **Annual major versions** - New primitives, API additions
- **Patch versions** - Bug fixes, clarifications (1.0.1, 1.0.2)
- **Working group approval** - All primitive additions require WG vote
- **Platform vendor input** - "Can you implement this?" before standardizing

Schema System

Purpose

Modules for devices with custom clusters need schema definitions for:

1. Attribute name → ID mapping
2. Command name → ID mapping
3. Struct/enum definitions for TLV encoding
4. Type information for validation

Schema Format

Schemas are derived from ZCL XML (the source of truth for Matter clusters) and distributed as JSON:

```
{
  "clusters": {
    "PhilipsHueEntertainment": {
      "id": 4294115000,
      "attributes": {
        "entertainmentMode": { "id": 0, "type": "enum8" },
        "syncActive": { "id": 1, "type": "boolean" }
      },
      "commands": {
        "StartSync": {
          "id": 0,
          "request": {
```

```
            "groupId": { "id": 0, "type": "uint16" }
          }
        },
        "StopSync": {
          "id": 1,
          "request": {}
        }
      },
      "enums": {
        "EntertainmentModeEnum": {
          "Off": 0,
          "Music": 1,
          "Video": 2,
          "Game": 3
        }
      }
    }
  }
}
```

Schema Usage in Modules

```
-- Reading custom cluster attributes
local entertainment = state.PhilipsHueEntertainment or {}
local mode = entertainment.entertainmentMode  -- Schema maps name → ID

-- Invoking custom cluster commands
Matter.command("PhilipsHueEntertainment", "StartSync", {
    groupId = 1
})
-- Schema provides: cluster ID, command ID, TLV encoding for params
```

Custom Cluster Deep Dive: How Vendors Include "Code" Safely

A common question: "How do custom vendor clusters work if vendors can't include executable code?"

Clarification: Vendors CAN include code—Lua code. The distinction is that it's **sandboxed code** with strictly controlled I/O.

The Two-Part Solution

1. Schema (JSON) - Tells the platform HOW to encode/decode:

```
{
  "clusters": {
    "PhilipsHueEntertainment": {
      "id": 4294115000,
      "attributes": {
        "entertainmentMode": { "id": 0, "type": "enum8" },
```

```
        "syncActive": { "id": 1, "type": "boolean" }
      },
      "commands": {
        "StartSync": {
          "id": 0,
          "request": {
            "groupId": { "id": 0, "type": "uint16" },
            "mode": { "id": 1, "type": "enum8" }
          }
        }
      }
    }
  }
}
```

2. Lua Module (Sandboxed) - Uses custom cluster by name:

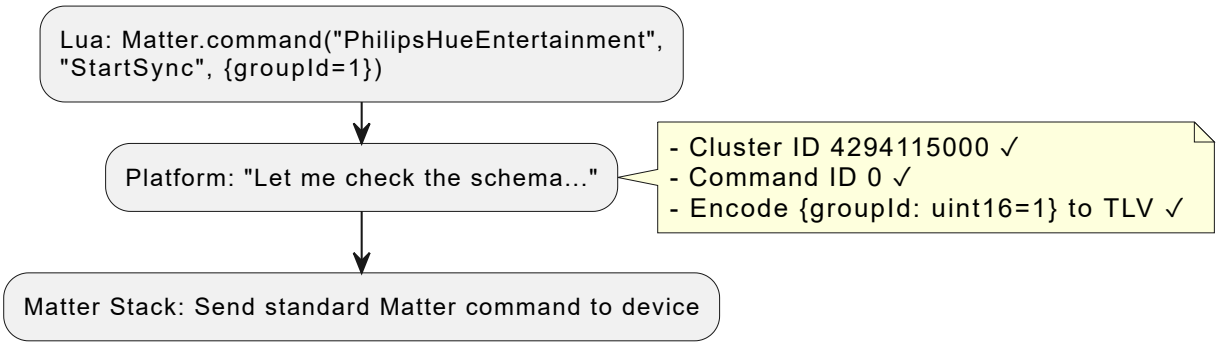
```
function Module.render(device, state)
    local entertainment = state.PhilipsHueEntertainment or {}
    local mode = entertainment.entertainmentMode or 0

    return UI.SegmentedControl {
        value = mode,
        options = {
            { label = "Off", value = 0 },
            { label = "Music", value = 1 },
            { label = "Video", value = 2 }
        },
        onChange = function(m)
            Matter.command("PhilipsHueEntertainment", "StartSync", {
                groupId = 1, mode = m
            })
        end
    }
end
```

What Makes This Safe

The Lua can only:

Allowed	Blocked
Return UI node tree	Access file system
Call `Matter.command()`	Make network requests
Read `state` (device attributes)	Access device sensors
Basic Lua logic (if/for/functions)	Import external libraries
Call `L()` for localization	Access other apps/processes

Platform as Gatekeeper



Every interaction with the outside world goes through platform-controlled APIs. The vendor can implement arbitrary UI logic but cannot escape the sandbox.

This is exactly how WidgetKit works: Widget code runs but can only return views and request timeline updates—no arbitrary network or file access.

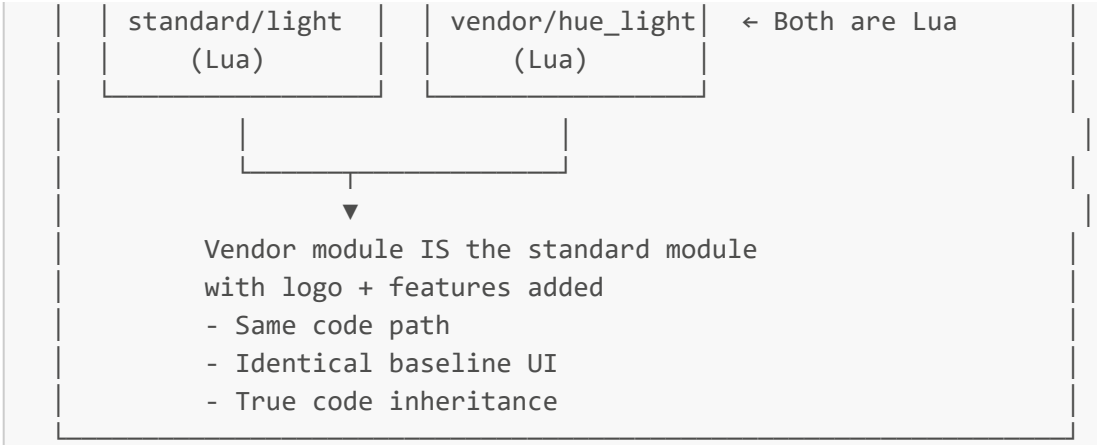
Standard Device Type Coverage

Why Standard Modules Must Be Lua (Not Native Platform Code)

A critical architectural decision: **platforms have no built-in device UIs**. Every device type—even basic ones like On/Off Light—renders through a Lua module downloaded from the registry.

Why this matters:

```
Alternative A: Platforms have native UIs (REJECTED)
┌─────────────────────────────────────────────────────────────┐
│  Platform App                                               │
│  ┌──────────────────┐  ┌──────────────────┐                 │
│  │ Native Light UI  │  │ Native Thermo UI │  ← Built into app │
│  └──────────────────┘  └──────────────────┘                 │
│           │                    │                            │
│           ▼                    ▼                            │
│  Vendor must REIMPLEMENT everything in Lua                  │
│  - Different code path                                      │
│  - Inconsistent look/behavior                               │
│  - More work, more bugs                                     │
└─────────────────────────────────────────────────────────────┘

Alternative B: ALL UI is Lua modules (CHOSEN)
┌─────────────────────────────────────────────────────────────┐
│  Platform App                                               │
│  ┌──────────────────────────────────────────────────────┐   │
│  │              Lua Runtime + UINode Renderer           │   │
│  │                    (ONLY this)                       │   │
│  └──────────────────────────────────────────────────────┘   │
│           │                                                 │
│           ▼                                                 │
│  ┌──────────────────┐  ┌──────────────────┐                 │
```



CSA Standard Modules: The Vendor Starting Point

CSA provides Lua modules for ALL standard device types. These are not "reference implementations"—they are **the actual production code** that:

1. **Platforms use directly** for devices without vendor-specific types
2. **Vendors fork and extend** for branded experiences

When a vendor wants a branded UI:

1. Download the CSA standard module for their device type
2. Copy the Lua code
3. Add vendor device type to metadata
4. Add logo, branding, extended features
5. Submit to registry

The baseline UI code is literally copied, not reimplemented.

Example: Philips Forking the Standard Module

Step 1: CSA standard color light module (`standard/color_light`)

```lua
-- CSA Standard Extended Color Light Module
-- Device type: 0x010D
local Module = {
    name = "Color Light",
    deviceType = 0x010D,
    version = "1.0.0"
}

function Module.render(device, state)
    local onOff = state.OnOff or {}
    local level = state.LevelControl or {}
    local color = state.ColorControl or {}

    return UI.Column {
        spacing = 16,
        children = {
```

```
            -- Power toggle
            UI.Toggle {
                value = onOff.onOff or false,
                label = "Power",
                onChange = function(v) Matter.setOnOff(v) end
            },

            -- Brightness slider
            UI.Slider {
                value = level.currentLevel or 0,
                min = 1, max = 254,
                label = "Brightness",
                onChange = function(v) Matter.setLevel(v) end
            },

            -- Color wheel
            UI.ColorWheel {
                hue = color.currentHue or 0,
                saturation = color.currentSaturation or 0,
                onChange = function(h, s) Matter.setColor({hue=h, saturation=s})
end
            }
        }
    }
end

return Module
```

Step 2: Philips copies this code, adds branding (vendor/philips_hue_light)

```
-- Philips Hue Color Light Module
-- FORKED FROM: standard/color_light v1.0.0
local Module = {
    name = "Philips Hue Color Light",
    deviceType = 0xC000_1234,      -- ← Changed: vendor device type
    baseDeviceType = 0x010D,       -- ← Added: documents what this extends
    version = "1.0.0",

    branding = {                   -- ← Added: vendor branding
        logo = "https://cdn.philips-hue.com/logo.png",
        accentColor = "#0066FF"
    }
}

function Module.render(device, state)
    local onOff = state.OnOff or {}
    local level = state.LevelControl or {}
    local color = state.ColorControl or {}

    return UI.Column {
        spacing = 16,
        children = {
```

```
            -- ↓↓↓ ADDED: Vendor logo ↓↓↓
            UI.Image { src = Module.branding.logo, height = 24 },

            -- Power toggle (UNCHANGED from standard)
            UI.Toggle {
                value = onOff.onOff or false,
                label = "Power",
                onChange = function(v) Matter.setOnOff(v) end
            },

            -- Brightness slider (UNCHANGED from standard)
            UI.Slider {
                value = level.currentLevel or 0,
                min = 1, max = 254,
                label = "Brightness",
                onChange = function(v) Matter.setLevel(v) end
            },

            -- Color wheel (UNCHANGED from standard)
            UI.ColorWheel {
                hue = color.currentHue or 0,
                saturation = color.currentSaturation or 0,
                onChange = function(h, s) Matter.setColor({hue=h, saturation=s})
end
            },

            -- ↓↓↓ ADDED: Vendor-specific feature ↓↓↓
            UI.Button {
                label = "Hue Scenes",
                onClick = function()
                    Navigation.push("hue_scenes")
                end
            }
        }
    }
end

return Module
```

What Philips actually changed:

- 3 lines of metadata
- 1 line for logo
- 5 lines for Scenes button
- **Everything else is identical to the standard module**

Benefits of True Code Inheritance

Benefit	Explanation
Vendors get 90% of code for free	Literally copy-paste the standard module

Benefit	Explanation
Guaranteed consistent baseline	Same toggle, slider, color wheel code = same behavior
Updates flow downstream	When CSA improves standard module, vendors can merge changes
Lower barrier to entry	Small vendors ship branded UI in hours, not months
Platform simplicity	Platforms only implement Lua runtime + UINode renderer
No special cases	Standard and vendor modules use identical code path

Reference Modules by Category

Lighting (6 types)

Device Type	ID	Clusters	Module
On/Off Light	0x0100	OnOff	`standard/onoff_light`
Dimmable Light	0x0101	OnOff, LevelControl	`standard/dimmable_light`
Color Temperature Light	0x010C	OnOff, LevelControl, ColorControl	`standard/color_temp_light`
Extended Color Light	0x010D	OnOff, LevelControl, ColorControl	`standard/color_light`
On/Off Plug-in Unit	0x010A	OnOff	`standard/onoff_plug`
Dimmable Plug-in Unit	0x010B	OnOff, LevelControl	`standard/dimmable_plug`

Sensors (9 types)

Device Type	ID	Clusters	Module
Temperature Sensor	0x0302	TemperatureMeasurement	`standard/temperature_sensor`
Humidity Sensor	0x0307	RelativeHumidityMeasurement	`standard/humidity_sensor`
Occupancy Sensor	0x0107	OccupancySensing	`standard/occupancy_sensor`
Contact Sensor	0x0015	BooleanState	`standard/contact_sensor`
Light Sensor	0x0106	IlluminanceMeasurement	`standard/light_sensor`
Pressure Sensor	0x0305	PressureMeasurement	`standard/pressure_sensor`
Flow Sensor	0x0306	FlowMeasurement	`standard/flow_sensor`
Water Leak Detector	0x0043	BooleanState	`standard/water_leak`
Smoke CO Alarm	0x0076	SmokeCoAlarm	`standard/smoke_alarm`

HVAC (2 types)

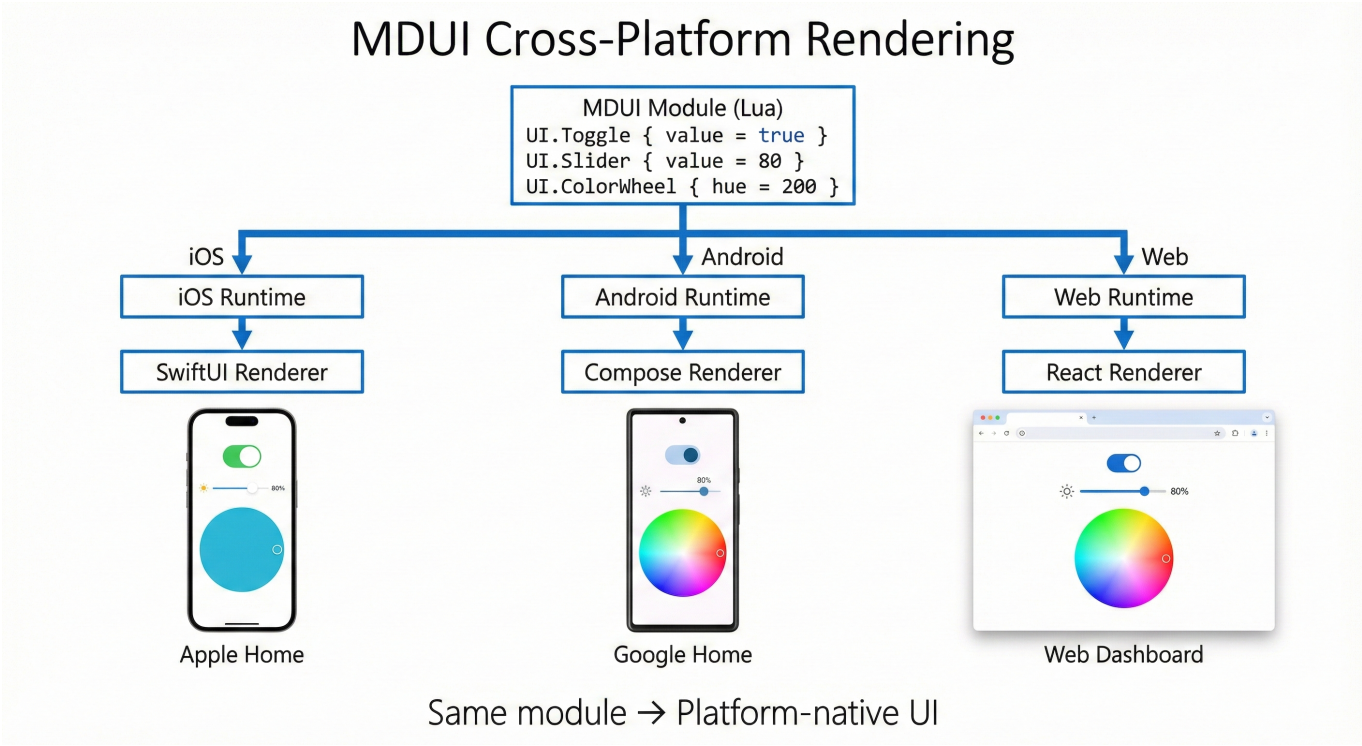
Device Type	ID	Clusters	Module

Device Type	ID	Clusters	Module
Thermostat	0x0304	Thermostat	`standard/thermostat`
Fan	0x002B	FanControl	`standard/fan`

Closures (2 types)

Device Type	ID	Clusters	Module
Door Lock	0x000A	DoorLock	`standard/door_lock`
Window Covering	0x0202	WindowCovering	`standard/window_covering`

Part 3: Security Model



Threat Model

MDUI modules are code that executes on user devices within platform apps (Apple Home, Google Home). The security model must address:

1. **Malicious vendors** - Intentionally harmful modules
2. **Compromised vendors** - Legitimate vendors whose signing keys are stolen
3. **Supply chain attacks** - Tampering during distribution
4. **Resource exhaustion** - Modules that consume excessive CPU/memory
5. **Data exfiltration** - Modules that leak user data
6. **Privilege escalation** - Modules that access resources beyond their scope

Security Architecture

1. Sandboxed Execution

Modules execute in a sandboxed Lua VM with:

No access to:

- File system
- Network (beyond Matter commands)
- Device sensors (camera, microphone, GPS)
- Other apps or processes
- System APIs

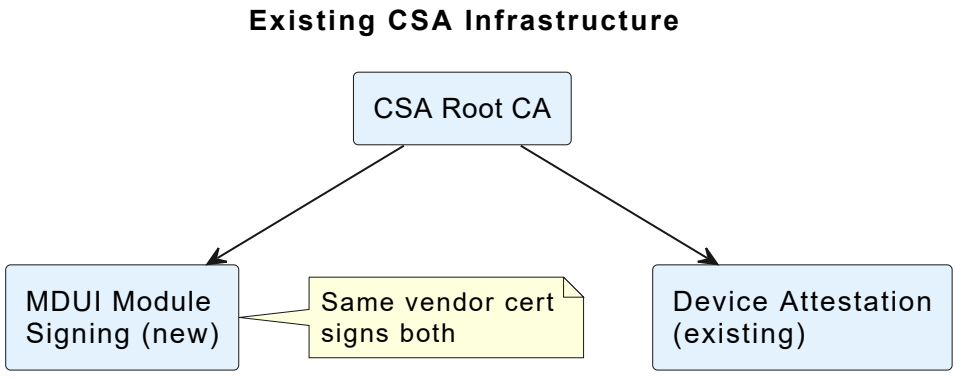
Access only to:

- Device state (attributes from Matter subscription)
- Matter command API (send commands to the device)
- UI primitive API (describe UI structure)
- Localization strings (bundled with module)

```
-- These are the ONLY globals available to modules:
UI = { Column, Row, Text, Toggle, Slider, ... }  -- UI primitives
Matter = { setOnOff, setLevel, command, ... }    -- Matter commands
L = function(key) ... end                         -- Localization
device = { id, name, deviceTypes, ... }          -- Device metadata
state = { OnOff = {...}, ... }                    -- Current state
```

2. Module Signing (Leveraging Existing CSA PKI)

Key insight: Vendors already have signing key pairs at CSA for Matter device certification. MDUI piggybacks on this existing infrastructure—no new PKI required.



All modules MUST be signed with the vendor's existing CSA certificate:

```
Module Package
├── module.lua      (the code)
├── schema.json     (cluster definitions)
├── strings/        (localization)
├── manifest.json   (metadata: deviceType, version, etc.)
└── signature.sig   (signed with existing CSA vendor key)
```

Signature covers:

- SHA-256 hash of all files
- Vendor's existing CSA certificate
- Timestamp

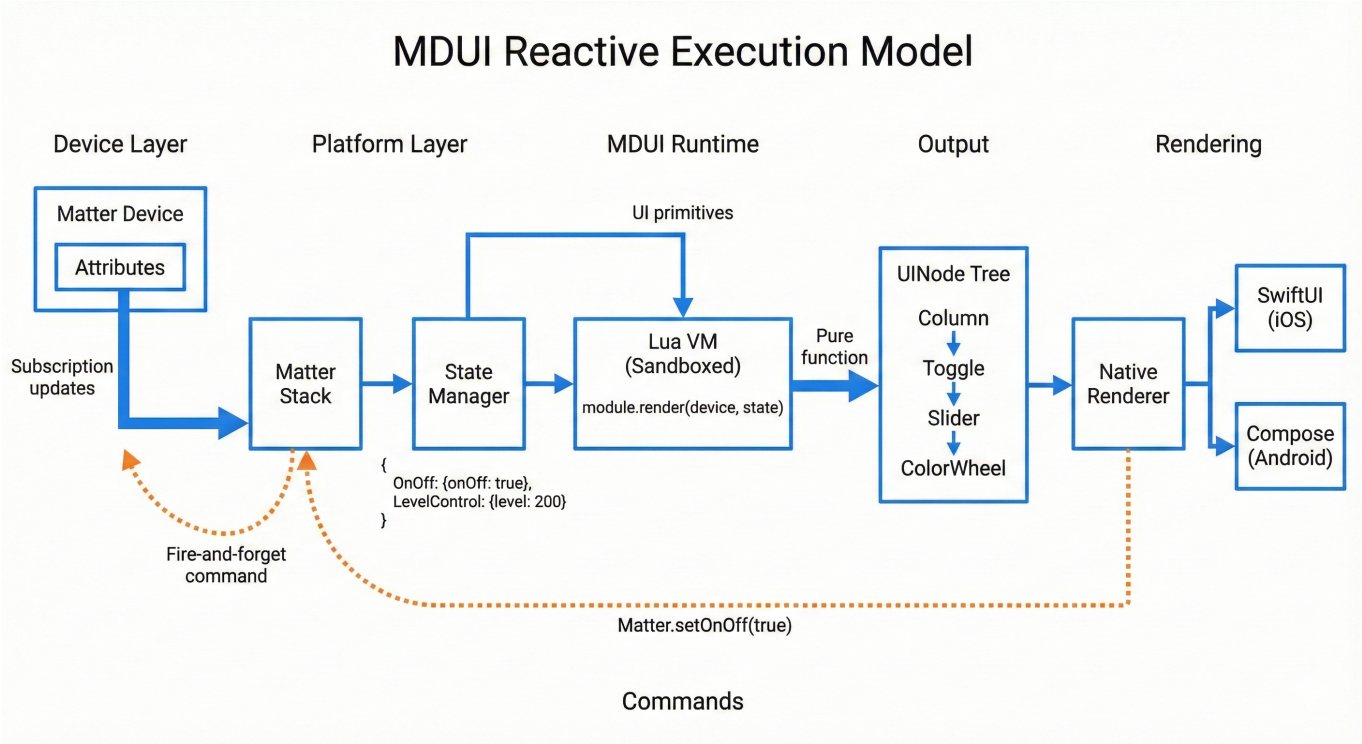
Benefits of reusing existing PKI:

- No new key ceremony for vendors
- Device type ID range (0xC000-0xFFFE) identifies vendor
- Revocation already handled by CSA
- Vendors already understand the process

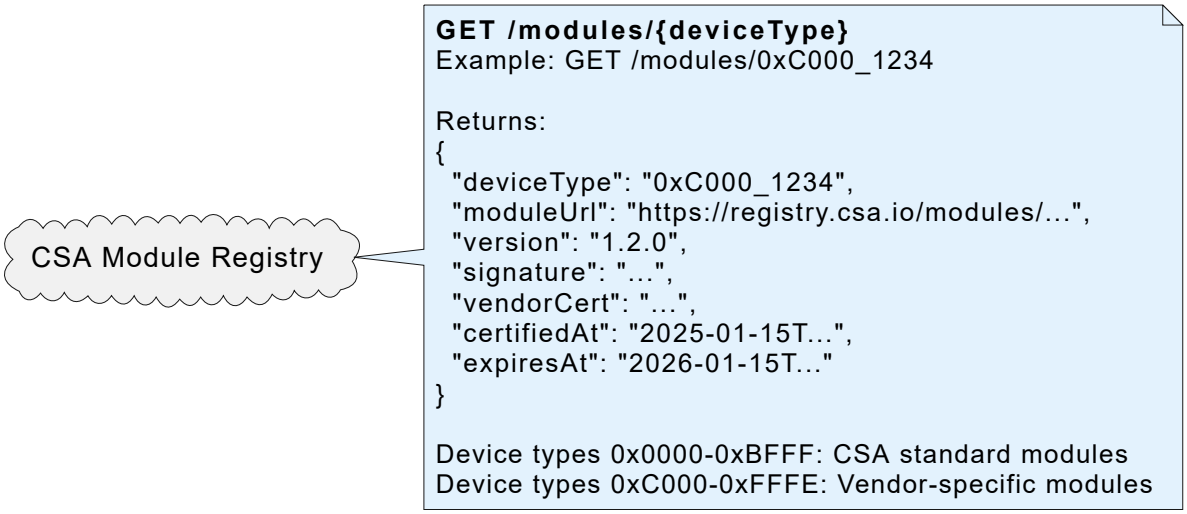
Verification:

1. Platform downloads module from registry
2. Verifies signature against vendor's CSA public key
3. Verifies device type ID is in vendor's allocated range (for vendor modules)
4. Verifies certificate is valid and not revoked (existing CSA CRL)
5. Only then loads module into VM

3. CSA Module Registry



Central registry for certified modules, indexed by device type:



Caching Strategy:

Event	Action
First device commission	Fetch module, cache locally with version
App launch	Use cached module immediately
Background (daily)	Check registry for updates, download if newer
Module update available	Update cache, re-render on next view
Device offline	Cached module still works (no network needed)
Cache miss + offline	Fall back to platform generic UI

Modules are small (typically <50KB), so caching is cheap and updates are fast.

4. Certification Process

Before a module is listed in the registry:

1. **Automated analysis**
    - Static code analysis for suspicious patterns
    - Resource usage profiling
    - API usage verification (no forbidden calls)
2. **Runtime testing**
    - Execute in test environment
    - Verify renders correctly
    - Check resource consumption
    - Test error handling
3. **Human review** (for new vendors or significant changes)
    - Code review
    - Behavior verification

- Vendor identity verification

4. **Certification issuance**
    - Module signed with CSA registry key
    - Listed in registry
    - Distributed to platforms

5. Resource Limits

Platforms enforce resource limits per module:

Resource	Limit	Action on Exceed
CPU time per render	50ms	Terminate, show error
Memory	10MB	Terminate, show error
Render frequency	60/sec max	Throttle
UI node count	500 nodes	Truncate
Network calls	0 (blocked)	Block
File access	0 (blocked)	Block

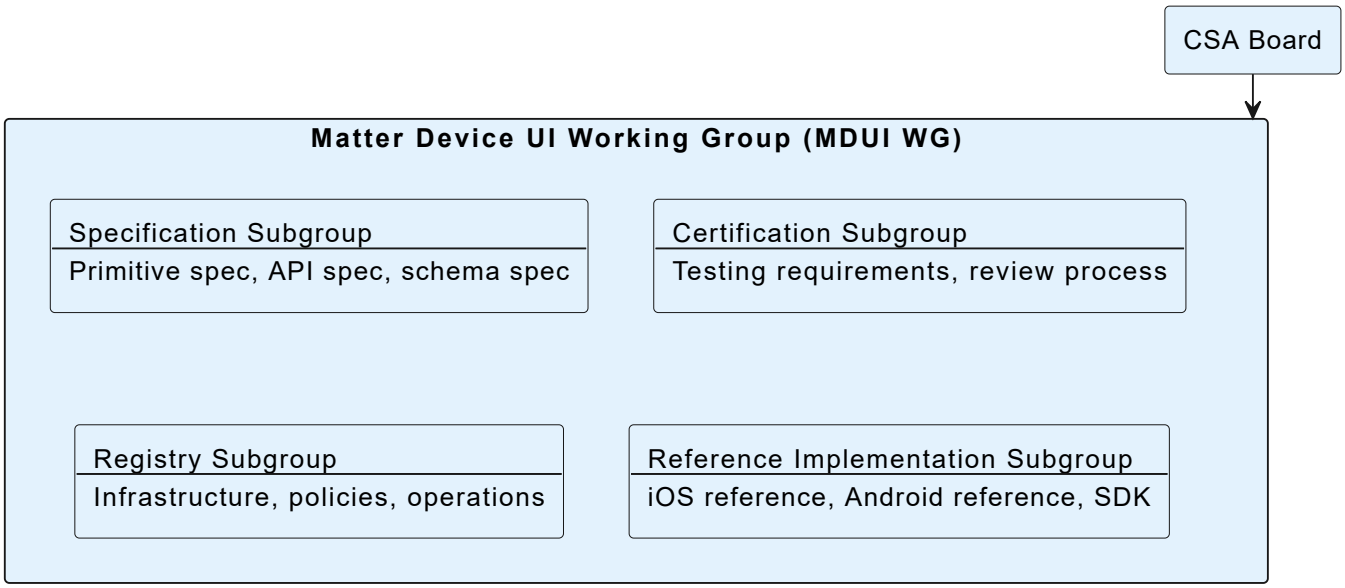
6. Revocation

If a module is found to be malicious:

1. **Registry revocation** - Module removed from registry
2. **Certificate revocation** - Vendor cert added to CRL
3. **Platform notification** - Platforms notified of revocation
4. **Client enforcement** - Platforms stop loading revoked modules
5. **Cache invalidation** - Cached copies deleted

Part 4: Governance Model

CSA Working Group Structure



Specification Governance

Primitive Additions

New UI primitives require:

1. Proposal with use cases
2. Review by MDUI WG
3. Platform vendor feedback (can they implement?)
4. Specification update
5. Reference implementation
6. Certification test updates

API Changes

Changes to Matter API or module format:

1. Backward compatibility analysis
2. Migration path for existing modules
3. Platform update requirements
4. Version negotiation specification

Vendor Participation

Becoming an MDUI Vendor

Since vendors already have CSA credentials for Matter certification, MDUI participation is simple:

1. ☑ Already a CSA member with Matter certification (existing)
2. ☑ Already have signing key pair (existing)
3. Register for MDUI program (new - administrative only)
4. Accept MDUI terms of service (new)
5. Begin submitting modules

No new keys, no new verification, no new fees - just an opt-in to the MDUI program.

Module Submission

1. Develop module following specification
2. Test with reference implementation
3. Submit to certification pipeline
4. Address any certification feedback
5. Module listed in registry upon approval

Ongoing Obligations

- Maintain modules for supported products
- Respond to security issues within 72 hours
- Update modules when specification changes
- Renew vendor certificate annually

Part 5: Migration Path

Platform Adoption Phases

Phase 1: Custom Cluster Support

Platforms implement MDUI runtime to support devices with vendor-specific clusters.

Trigger: Device has clusters not in platform's native UI library **Benefit:** Immediate support for devices that currently require vendor apps

Phase 2: Vendor Branding

Platforms allow MDUI modules for standard device types, enabling vendor branding.

Trigger: Vendor provides module for standard device type **Benefit:** Vendors can differentiate within platform apps

Phase 3: Full MDUI

Platforms transition all device UIs to MDUI rendering.

Trigger: Platform strategic decision **Benefit:** Consistent architecture, easier maintenance

Vendor Adoption

For Vendors with Custom Clusters

1. Define schema for custom clusters
2. Create MDUI module
3. Submit for certification
4. Devices immediately work fully in all platforms

For Vendors with Standard Devices

1. Create MDUI module with branding
2. Submit for certification
3. Devices show vendor branding in platforms

For Vendors Evaluating Matter

MDUI becomes a reason to choose Matter:

- "Build one module, reach all platforms"
- "No mobile app development required"
- "Full feature support in Apple/Google/Amazon"

Part 6: Reference Implementation

Deliverables

CSA will provide:

1. Platform Runtime Libraries

iOS (Swift)

- Luau VM integration
- UINode → SwiftUI renderer
- Matter bridge
- Module loader with caching
- Security sandbox

Android (Kotlin)

- Luau VM integration (JNI)
- UINode → Compose renderer
- Matter bridge
- Module loader with caching
- Security sandbox

2. Module Development SDK

- Module template generator
- Local testing harness
- Schema validator
- Certification pre-check
- Simulator for all platforms

3. Registry Infrastructure

- Module hosting
- Certification pipeline
- API for platform queries

- Vendor portal
- Analytics dashboard

Timeline

Phase	Duration	Deliverable
Specification Draft	3 months	Complete MDUI spec
Reference Implementation	6 months	iOS + Android runtimes
Pilot Program	3 months	10 vendors, 2 platforms
Public Launch	-	Registry opens, spec finalized

Part 7: Known Complexities (Deferred)

The following topics require detailed specification work but are intentionally deferred from this initial proposal. They do not block the core architecture.

Topic	Complexity	Notes
Groups	How does a module render UI for a group of devices?	May require group-aware render function or platform-level aggregation
Scenes	Scene creation/editing UI, scene activation	Platform may handle scene management; modules just render individual devices
Multi-endpoint devices	Devices with multiple endpoints (e.g., power strip with 4 outlets)	Module receives endpoint list; renders sub-UIs per endpoint
Bridged devices	Devices behind a Matter bridge	Transparent to modules—bridge exposes standard endpoints
Offline behavior	What to show when device is unreachable	Platform responsibility; module receives `device.online` flag
Commissioning UI	Device-specific setup flows during commissioning	Separate module type or platform handles generically
Firmware update UI	OTA update progress, notifications	Platform responsibility; standard OTA cluster
Accessibility	Screen readers, voice control, high contrast	Platform applies accessibility to rendered primitives

These will be addressed in subsequent specification drafts as the working group refines the standard.

Appendix A: Sample Modules

A.1 Thermostat Module (Standard)

This is a CSA standard module, registered under the standard Thermostat device type:

```
local Module = {
    name = "Thermostat",
    deviceType = 0x0301  -- Standard Thermostat device type
}

function Module.render(device, state)
    local thermo = state.Thermostat or {}

    local currentTemp = thermo.localTemperature
    local displayTemp = currentTemp and string.format("%.1f°", currentTemp / 100)
or "--"

    local mode = thermo.systemMode or 0
    local heatSetpoint = (thermo.occupiedHeatingSetpoint or 2000) / 100
    local coolSetpoint = (thermo.occupiedCoolingSetpoint or 2600) / 100

    local isHeating = (thermo.thermostatRunningState or 0) & 0x01 ~= 0
    local isCooling = (thermo.thermostatRunningState or 0) & 0x02 ~= 0

    return UI.Column {
        spacing = 16,
        children = {
            -- Current temperature
            UI.Card {
                children = {
                    UI.Column {
                        alignment = "center",
                        children = {
                            UI.Text { text = displayTemp, size = 64, weight =
"bold" },
                            UI.Text {
                                text = isHeating and "Heating" or (isCooling and
"Cooling" or "Idle"),
                                color = "secondary"
                            }
                        }
                    }
                }
            },

            -- Mode selector
            UI.SegmentedControl {
                value = mode,
                options = {
                    { label = "Off", value = 0 },
                    { label = "Heat", value = 4 },
                    { label = "Cool", value = 3 },
                    { label = "Auto", value = 1 }
```

```
                },
                onChange = function(m)
                    Matter.writeAttribute("Thermostat", "systemMode", m)
                end
            },

            -- Heat setpoint
            (mode == 4 or mode == 1) and UI.Slider {
                value = heatSetpoint,
                min = 7, max = 30,
                step = 0.5,
                label = "Heat to",
                onChange = function(v)
                    Matter.writeAttribute("Thermostat", "occupiedHeatingSetpoint",
v * 100)
                end
            },

            -- Cool setpoint
            (mode == 3 or mode == 1) and UI.Slider {
                value = coolSetpoint,
                min = 16, max = 32,
                step = 0.5,
                label = "Cool to",
                onChange = function(v)
                    Matter.writeAttribute("Thermostat", "occupiedCoolingSetpoint",
v * 100)
                end
            }
        }
    }
end

return Module
```

A.2 Door Lock Module (Standard)

This is a CSA standard module, registered under the standard Door Lock device type:

```
local Module = {
    name = "Door Lock",
    deviceType = 0x000A  -- Standard Door Lock device type
}

function Module.render(device, state)
    local lock = state.DoorLock or {}

    local lockState = lock.lockState
    local isLocked = lockState == 1
    local isUnlocked = lockState == 2
```

```
    return UI.Column {
        spacing = 16,
        children = {
            -- Lock state indicator
            UI.Card {
                children = {
                    UI.Column {
                        alignment = "center",
                        children = {
                            UI.Icon {
                                name = isLocked and "lock" or "lock_open",
                                size = 64,
                                color = isLocked and "primary" or "warning"
                            },
                            UI.Text {
                                text = isLocked and "Locked" or "Unlocked",
                                size = 24,
                                weight = "bold"
                            }
                        }
                    }
                }
            },

            -- Control buttons
            UI.Row {
                spacing = 16,
                children = {
                    UI.Button {
                        label = "Lock",
                        icon = "lock",
                        variant = isLocked and "filled" or "outlined",
                        flex = 1,
                        onClick = function()
                            Matter.command("DoorLock", "Lock", {})
                        end
                    },
                    UI.Button {
                        label = "Unlock",
                        icon = "lock_open",
                        variant = isUnlocked and "filled" or "outlined",
                        flex = 1,
                        onClick = function()
                            Matter.command("DoorLock", "Unlock", {})
                        end
                    }
                }
            }
        }
    }
end

return Module
```

Appendix B: Platform Rendering Reference

iOS (SwiftUI)

```
struct UINodeView: View {
    let node: UINode

    var body: some View {
        switch node {
        case .column(let children, let spacing, let alignment):
            VStack(alignment: alignment.toSwiftUI(), spacing: spacing) {
                ForEach(children.indices, id: \.self) { i in
                    UINodeView(node: children[i])
                }
            }

        case .toggle(let value, let label, let onChange):
            Toggle(label, isOn: Binding(
                get: { value },
                set: { onChange($0) }
            ))

        case .slider(let value, let min, let max, let label, let onChange):
            VStack(alignment: .leading) {
                if let label = label {
                    Text(label)
                }
                Slider(
                    value: Binding(get: { value }, set: { onChange($0) }),
                    in: min...max
                )
            }

        case .button(let label, let variant, let icon, let onClick):
            Button(action: onClick) {
                Label(label, systemImage: icon ?? "")
            }
            .buttonStyle(variant == "filled" ? .borderedProminent : .bordered)

        // ... additional cases
        }
    }
}
```

Android (Compose)

```
@Composable
fun UINodeRenderer(node: UINode) {
    when (node) {
        is UINode.Column -> {
            Column(
                verticalArrangement = Arrangement.spacedBy(node.spacing.dp),
                horizontalAlignment = node.alignment.toCompose()
            ) {
                node.children.forEach { UINodeRenderer(it) }
            }
        }

        is UINode.Toggle -> {
            Row(verticalAlignment = Alignment.CenterVertically) {
                Text(node.label)
                Spacer(Modifier.weight(1f))
                Switch(
                    checked = node.value,
                    onCheckedChange = node.onChange
                )
            }
        }

        is UINode.Slider -> {
            Column {
                node.label?.let { Text(it) }
                Slider(
                    value = node.value,
                    onValueChange = node.onChange,
                    valueRange = node.min..node.max
                )
            }
        }

        is UINode.Button -> {
            when (node.variant) {
                "filled" -> FilledTonalButton(onClick = node.onClick) {
                    node.icon?.let { Icon(it.toImageVector(), null) }
                    Text(node.label)
                }
                else -> OutlinedButton(onClick = node.onClick) {
                    node.icon?.let { Icon(it.toImageVector(), null) }
                    Text(node.label)
                }
            }
        }

        // ... additional cases
    }
}
```

Appendix C: Glossary

Term	Definition
MDUI	Matter Device UI - this proposed standard
Module	A package containing UI definition, schema, and assets for a device type
Device Type	A 16-bit or 32-bit ID identifying a class of device. Standard types (0x0000-0xBFFF) defined by CSA; vendor-specific types (0xC000-0xFFFE) allocated to manufacturers
Vendor Device Type	A device type in the 0xC000-0xFFFE range, allocated to a specific vendor. Used to trigger vendor-specific UI modules
deviceTypeList	Matter descriptor cluster attribute listing all device types an endpoint supports. Modules are discovered by scanning this list
Primitive	A basic UI component (Toggle, Slider, etc.) that platforms render natively
UINode	A node in the declarative UI tree returned by module.render()
Schema	Definition of custom clusters, attributes, and commands for TLV encoding
Registry	CSA-operated service for discovering and downloading certified modules, indexed by device type ID
Vendor	Company that manufactures Matter devices
Platform	App that controls Matter devices (Apple Home, Google Home, etc.)

Document History:

- v0.2 (2025-01) - Changed module discovery from vendorId+deviceType to vendor-specific device types
- v0.1 (2025-01) - Initial draft for review