Mobile-MCP: Comprehensive Technical Manual for AI-Assisted Mobile Automation

Executive Summary

Mobile-MCP is a Model Context Protocol (MCP) server that bridges AI coding agents with mobile devices to enable **cross-platform mobile automation**. It provides a unified set of tools for interacting with both Android and iOS devices (real devices, emulators, and simulators) through a consistent interface [1] [2]. By abstracting away platform-specific details (like Android's ADB and iOS's Xcode/ WebDriverAgent), mobile-MCP allows AI assistants (such as Claude or GitHub Copilot) to perform native app testing, UI navigation, and data scraping on mobile apps using natural-language directives [3] [4]. This manual serves as a technical foundation and playbook for leveraging mobile-MCP in AI-driven mobile automation workflows.

Key capabilities: Mobile-MCP can list and select devices, launch/terminate apps, simulate taps and swipes, input text, capture screenshots, and extract UI element data – all via JSON-based tool calls. It supports two communication modes: a **STDIO mode** (for integration into IDEs and agent frameworks) and an **HTTP server mode** (default port 4723, similar to Appium's, for standalone operation) [5]. Under the hood, it uses platform-native automation channels (ADB/UI Automator for Android, and Xcode's WebDriverAgent for iOS) to perform actions, but these complexities are hidden behind simple, high-level MCP tools [3]. The result is a fast, LLM-friendly automation layer that doesn't require the AI agent to have specialized iOS/Android knowledge [6]. The following sections provide an in-depth reference of mobile-MCP's architecture, an inventory of its tools (with usage examples), best practices for reliable automation, community learnings, and design guidance for building a Claude Code skill around mobile-MCP.

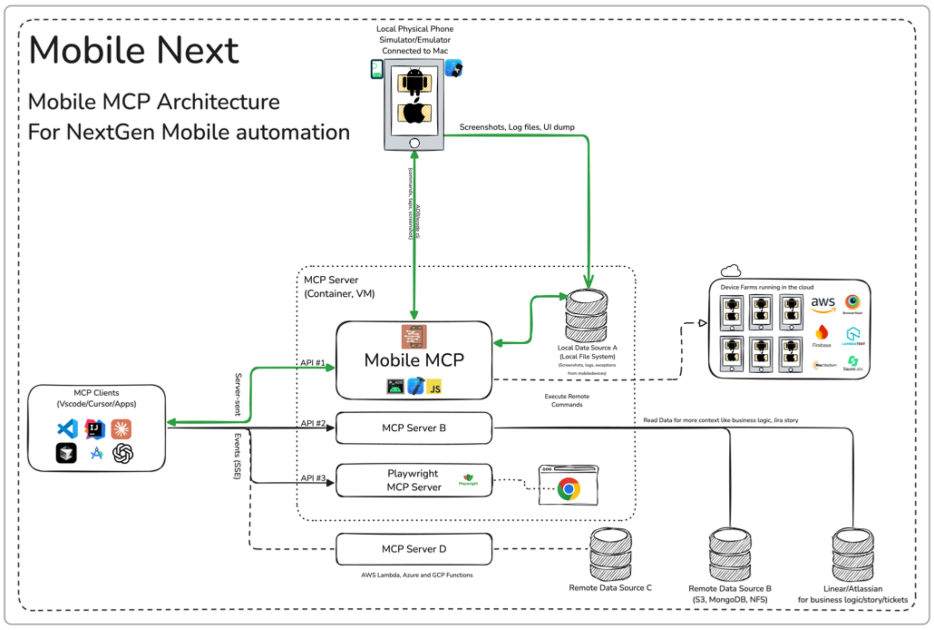
Technical Reference

1. Architecture & Core Concepts

Mobile-MCP employs a **client-server architecture** in line with the MCP standard. The AI agent (MCP client) issues JSON-RPC requests to the mobile-MCP server, which in turn executes commands on the connected mobile device and returns structured results [7] [4]. Communication can occur via **STDIO** (standard input/output streams) – commonly used when embedding mobile-MCP into an IDE or agent runtime – or via **HTTP** on a designated port (default 4723) when running as a standalone service [5]. In STDIO mode, the server reads JSON-RPC requests from stdin and writes responses to stdout, allowing tight integration with tools like Claude Code or VS Code extensions [8] [9]. In HTTP mode, the server listens for requests (e.g. tool calls as HTTP endpoints) and serves responses, which is useful for debugging or CI environments (you can verify it by hitting a `/status` endpoint to get server info and device list) [10].

At its core, mobile-MCP acts as a **bridge between the MCP protocol and native mobile automation frameworks** [4]. On Android, it leverages the Android Debug Bridge (ADB) and UI Automator to inspect UI hierarchy and perform actions [11]. On iOS, it uses Apple's XCUITest framework via WebDriverAgent (WDA) to interact with simulators or physical devices [11]. These allow mobile-MCP to obtain **structured**

accessibility data (views, labels, etc.) from the device, and to inject inputs (taps, keystrokes) or retrieve screenshots. This architecture enables **platform-agnostic** operation: an AI agent can call the same `mobile_click_on_screen_at_coordinates` or `mobile_launch_app` tool regardless of whether the target is an Android phone or an iPhone – mobile-MCP handles the platform differences internally [12] [3].



High-level architecture of Mobile-MCP, showing how the MCP server interfaces with AI agents on one side and mobile devices on the other (via ADB for Android and WebDriverAgent for iOS) [3]. This design provides a unified API for device farms, emulators, and real devices.

A key concept in Mobile-MCP is the dual approach to perceiving and interacting with the UI: **Accessibility (Structured) mode vs. Visual "Sense" mode**. In **Accessibility mode**, mobile-MCP queries the OS's accessibility or view hierarchy (using tools like `uiautomator dump` on Android and WDA's `/source` on iOS) to get a structured snapshot of UI elements [13]. This yields a JSON representation of on-screen elements with properties like labels, identifiers, types, and bounds. It is fast and precise, enabling deterministic identification of elements by their accessibility labels or other attributes [14] [15]. **Visual Sense mode**, by contrast, relies on analyzing a screenshot of the UI when structured data is insufficient – for example, if an app's UI elements lack accessibility labels or are drawn custom (canvas), the agent might resort to interpreting the screenshot image to decide where to tap [16] [17]. Mobile-MCP is designed to use structured data "whenever possible" and fall back to screenshot-based coordinates only as a last resort [18] [19]. This hybrid strategy provides both reliability (through structured data for known elements) and flexibility (through visual analysis when needed). The "Visual Sense" capability does **not** involve heavy onboard computer vision models – instead, it relies on the AI's own vision (if available) or simple image analysis, making Mobile-MCP relatively lightweight [20].

Device connectivity: Mobile-MCP supports multiple device connection modes. It can interface with **Android emulators** and **iOS simulators** running on the host machine, as well as **physical devices** connected via USB or Wi-Fi [1] [21]. On Android, any device visible to ADB (including Wi-Fi connected devices or emulator instances) can be accessed. On iOS, simulators are managed through Xcode, and real devices require that the host Mac is provisioned for development (with the device "trusted" and running a WebDriverAgent instance) [22] [23]. Mobile-MCP provides tools (`mobile_list_available_devices` and `mobile_use_device`) to enumerate and select the target device for commands, making it easy to handle multi-device environments. You can run the server on macOS (which supports both iOS and Android targets) or on Windows/Linux (which support Android; Windows also can interface with iOS devices over network with additional setup) [24]. Internally, when an iOS device is selected, mobile-MCP will ensure a WDA session is established (it may require the user to start a WDA runner on the device as per setup instructions) [25] [26]. For Android, simply having the device/emulator online (visible via `adb devices`) is sufficient [27]. In summary, the architecture abstracts the **device communication layer** so that the AI agent can focus on high-level tasks; whether the device is a local emulator or a cloud-based phone, the tool interface remains the same (with potential differences only in execution speed or available features).

2. Complete Tool Inventory

Mobile-MCP exposes a rich set of MCP tools (functions) that the AI agent can invoke. These tools cover device management, app lifecycle control, screen inspection, input actions, and more [28] [29]. Each tool has a defined name, expected input parameters, and a well-defined output structure (all communicated as JSON via MCP). Errors are reported following the JSON-RPC error format (with codes and messages); for example, attempting an action without selecting a device will result in an error indicating no device context. Below is the comprehensive list of tools, organized by category, with details on their purpose, parameters, return values, error conditions, and usage examples. *(Note: All tool invocations expect a JSON object as input – even if no parameters are needed – so the agent should call them with* `{}` *when no arguments are required to avoid "Invalid arguments" errors [30].)*

Device Management Tools

- `mobile_list_available_devices` – *List all available devices.* This tool returns the list of devices currently accessible to mobile-MCP [31] [32]. It requires no parameters. **Output:** An array of device objects, each likely including properties such as `name` (device nickname or model), `deviceType` (e.g. `"simulator"`, `"ios"`, `"android"`), and a unique identifier or UDID. For example, on macOS it might list an iPhone simulator by name and an Android emulator by serial. If multiple devices are returned, the agent (or user) must select one using `mobile_use_device` before proceeding [32]. **Errors:** None (the tool returns an empty list if no devices are connected). **Usage example:** After starting the server, the agent can call `mobile_list_available_devices` to retrieve something like:

  ```
  {
    "devices": [
      { "name": "iPhone 14 Pro (16.4 Simulator)", "deviceType":
  "simulator", "id": "F3C4..."},
      { "name": "sdk_gphone64_x86_64 (Android 14)", "deviceType":
  "android", "id": "emulator-5554"}
    ]
  }
  ```

  The agent should then prompt the user (or decide) which device to use if more than one is listed. *(Confidence: High – official description confirms behavior [32].)*

- `mobile_use_device` – *Select a device to use.* This tool sets the context for subsequent actions [33] [34]. **Parameters:** `device` (string, the device name or ID to select, as obtained from `mobile_list_available_devices`), and `deviceType` (string enum: `"simulator"`, `"ios"`, or `"android"`, specifying the category of the device) [35] [36]. Both are required.

Output: A confirmation message or status (e.g. `"Device X selected"`). After this call, all future tool calls will target the specified device until changed. **Errors:** If the device name/ID is not found or not currently connected, an error is returned (code -32602 invalid params). **Usage example:** `mobile_use_device` with `{"device": "iPhone 14 Pro (16.4 Simulator)", "deviceType": "simulator"}` will prepare the server to interact with that iOS Simulator, establishing a connection via Xcode if needed. *(Confidence: High – defined in JSON schema [35] [34].)*

- `mobile_get_screen_size` – *Get the screen size of the device (in pixels).* No parameters needed [37]. **Output:** The screen resolution as an object, e.g. `{ "width": 1170, "height": 2532 }` for an iPhone 14 Pro. This is useful for interpreting coordinate-based actions or for scaling UI element positions. **Errors:** Returns an error if no device is selected or if the device is unreachable. **Usage example:** The agent might call this after `mobile_use_device` to know the coordinate space for the device (especially if planning to tap via coordinates). *(Confidence: Medium – functionality is straightforward and implied by tool name and confirmed by usage in similar frameworks.)*

- `mobile_get_orientation` – *Get the current screen orientation of the device.* No parameters [38]. **Output:** Typically returns a string such as `"portrait"` or `"landscape"` indicating the device's orientation. This can help an agent understand if it needs to adjust coordinate logic (some coordinate spaces might change if orientation changes). **Errors:** Possible error if no device is selected. **Usage example:** On a tablet device, calling this might return `"landscape"` if the emulator is currently rotated, letting the agent know the UI is wide. *(Confidence: Medium – listed in tool inventory [39] and likely implemented.)*

- `mobile_set_orientation` – *Change the screen orientation of the device.* Parameter: `orientation` (string, required, either `"portrait"` or `"landscape"`) [40] [41]. **Output:** A confirmation (for instance, `{"orientation":"landscape"}` to confirm the new state). This effectively simulates rotating the device. **Errors:** If orientation value is invalid or device doesn't support rotation (some emulators might not rotate if sensor is locked), an error is returned. **Usage example:** `mobile_set_orientation` with `{"orientation": "landscape"}` will rotate an Android emulator to landscape mode or an iOS simulator accordingly (this uses ADB or simctl under the hood). *(Confidence: Medium – defined in schema [42] [43], typical behavior as per mobile automation norms.)*

App Management Tools

- `mobile_list_apps` – *List all installed apps on the device.* No parameters [44]. **Output:** A list of apps/packages present. On Android, this likely returns package names (and possibly app labels); on iOS, it may return bundle identifiers or app names. (Exact format may vary; e.g., it might output an array of strings or objects with name and identifier.) **Errors:** If not connected to a device or if the query fails. **Usage:** This can be used to find the correct package name for an app. For example, on Android the agent could filter the result for "YouTube" to find the package `com.google.android.youtube` before launching it. *(Confidence: Low – not explicitly documented beyond name/description [44], but likely straightforward. Agents typically prefer direct package names when known.)*

- `mobile_launch_app` – *Launch an app on the device by package name (or bundle identifier).* Parameter: `packageName` (string, required) [45] [46]. **Output:** Confirmation or status, e.g. `{"result": "App launched"}`. If successful, the specified app is brought to the foreground on the device. **Errors:** If the package is not found or cannot be launched (e.g., not installed), returns an error. **Usage example:** `mobile_launch_app` with `{"packageName": "com.instagram.android"}` would start Instagram on the device [45]. The agent might use `mobile_list_apps` first or rely on known package names. *(Confidence: High – documented in schema [45].)*

- `mobile_terminate_app` – *Stop/kill a running app on the device.* Parameter: `packageName` (string, required), the app's identifier [47] [48]. **Output:** Confirmation of termination (or a status like `{"result": "App terminated"}`). On Android this corresponds to forcibly stopping the app (ADB shell am force-stop), on iOS it sends a terminate signal via WDA. **Errors:** If the app is not running or the package is invalid, it may still return success (stopping an already-stopped app is usually no-op) – typically no error, unless device is not available. **Usage example:** After completing a test, the agent might call `mobile_terminate_app` to close the app under test for cleanup [47]. *(Confidence: High – per schema [47].)*

- `mobile_install_app` – *Install an app from a file onto the device.* Parameter: likely `filePath` or `appFile` (string) pointing to the .apk (Android) or .ipa/.app (iOS) file to install. *(Not explicitly shown in the snippet above, but mentioned in README [49].)* **Output:** Success message if installation completes (e.g., "Installed successfully"). **Errors:** If file is not found or installation fails (e.g., incompatible app or missing permissions), an error detailing the failure. **Usage example:** `mobile_install_app` with `{"filePath": "/path/to/app.apk"}` to sideload an Android app. *(Confidence: Medium – tool is listed in feature list [49], presumably wraps `adb install` or Xcode install.)*

- `mobile_uninstall_app` – *Uninstall/remove an app from the device.* Parameter: `packageName` (string, required) as the identifier of the app to remove [49]. **Output:** Confirmation of uninstall (e.g., `{"result": "Uninstalled"}`). **Errors:** If the app is not present or removal fails (permissions issues), an error is returned. **Usage example:** `mobile_uninstall_app` with `{"packageName": "com.example.myapp"}` to remove a test application after a run. *(Confidence: Medium – listed in README [49] and likely straightforward.)*

Screen Interaction Tools

- `mobile_take_screenshot` – *Take a screenshot of the device's current screen.* No parameters [50]. **Output:** An image file or data reference. In MCP, images are usually returned as a path or encoded data accessible to the client. Mobile-MCP likely returns a path to a PNG file (e.g., a temporary file) that the client can then retrieve, or it may inline a base64 string of the image. The screenshot provides a full rendering of the UI, which an AI agent with vision (like multimodal Claude or GPT-4 Vision) can analyze. **Errors:** If the device is not ready or screenshot fails (e.g., on headless device without display), returns error. **Usage:** The agent might call this and then follow up with an image analysis prompt. *Example:* After an ambiguous situation, an agent might do: `mobile_take_screenshot {}` and then ask the LLM: "Here is the screenshot, identify the login button." *(Confidence: High – common tool, and README suggests using screenshot to understand screen [51]. Also likely implemented with standard ADB screencap / WDA screenshot.)*

- `mobile_save_screenshot` – *Save a screenshot to a file.* Parameter: likely `filePath` (string) specifying where to save. This is similar to `mobile_take_screenshot` but allows the agent to specify a file location for persistence [52]. **Output:** Path confirmation or success message. **Usage example:** `mobile_save_screenshot` with `{"filePath": "/tmp/screen1.png"}` will capture and save the image. *(Confidence: Medium – listed in README [52].)*

- `mobile_list_elements_on_screen` – *List UI elements on the screen with their properties.* No parameters [53]. **Output:** A structured JSON of visible elements and their coordinates. Specifically, mobile-MCP aggregates the accessibility view hierarchy and returns an array of elements, each with attributes like:

- `label` or `text`: the displayed text or accessibility label of the element (if any) [54],
- `identifier`: an accessibility identifier/hint (useful for React Native or explicit dev labels) [55],
- `type`: the class/type of the UI element (e.g., `"Button"`, `"Label"`, `"Image"`) [54],

- `x, y, width, height`: the bounding box coordinates of the element on screen [56].
  This is effectively a JSON "UI tree dump." *Example output:*

```
{
  "elements": [
    {"label": "Username", "type": "TextField", "identifier": null, "x":
20, "y": 100, "width": 280, "height": 50},
    {"label": "Password", "type": "TextField", "identifier": null, "x":
20, "y": 160, "width": 280, "height": 50},
    {"label": "Sign In", "type": "Button", "identifier":
"login_button", "x": 20, "y": 220, "width": 100, "height": 40}
  ]
}
```

  Using this data, an agent can decide which coordinates to tap. **Errors:** If the hierarchy cannot be retrieved (e.g., app is in a state with no UI), an error or empty list is returned. **Best Practice:** Always use this tool to guide tapping instead of guessing coordinates from an image – it's far more reliable [15] [57]. *(Confidence: High – strongly recommended in official wiki [57] [56].)*

- `mobile_click_on_screen_at_coordinates` – *Simulate a tap at specific screen coordinates.* Parameters: `x` (number, required) and `y` (number, required) – the pixel coordinates to tap [58] [59]. **Output:** Confirmation (often just an empty result or success true). The tap is performed as a short press at those coordinates on the device. **Errors:** If coordinates are out of bounds (negative or beyond screen size), it may return an error or simply no effect (with success false). If no device is active, error is thrown. **Usage example:** After using `mobile_list_elements_on_screen`, an agent might find the element with `"label": "Sign In"` and call `mobile_click_on_screen_at_coordinates` with its center point: e.g. `{"x":70,"y":240}` (assuming the button's rectangle starts at 20,220 with width 100,height 40) to tap it. *(Confidence: High – defined in schema [58] [59] and fundamental action.)*

- `mobile_double_tap_on_screen` – *Double-tap at specific coordinates.* Parameters: likely `x` and `y` similar to single tap (though not explicitly shown above, it's listed in README [60]). **Output:** Confirmation. It performs two quick sequential taps. **Usage:** Useful for zoom or special gestures (e.g., double-tap to like a post). *(Confidence: Medium – listed in tool list [60].)*

- `mobile_long_press_on_screen_at_coordinates` – *Long-press (touch & hold) at specific coordinates.* Parameters: `x`, `y`, and possibly `duration` (ms) if allowed, though not certain – likely a fixed duration is used. **Output:** Confirmation. **Usage:** To trigger long-press context menus or drag mode. *(Confidence: Medium – listed in tool list [61]; details not fully documented, but presumably implemented.)*

- `mobile_swipe_on_screen` – *Swipe in a given direction on the screen.* Parameter: `direction` (string, required), e.g. `"up"`, `"down"`, `"left"`, `"right"` [62] [63]. **Output:** Confirmation of swipe. The swipe gesture typically scrolls or pages the view in that direction. By default, mobile-MCP likely performs a standard length swipe (e.g., from mid-screen to some offset in the given direction). **Errors:** If direction is not one of the allowed values, an error is returned [64] [65]. **Usage example:** `mobile_swipe_on_screen` with `{"direction": "up"}` could scroll a list view downwards (swiping upward). If more control is needed (like precise coordinates for drag start/end), the current API may not expose it – it abstracts to simple cardinal directions for convenience. *(Confidence: High – parameters defined in schema [64], and swiping is a common action.)*

Input & Navigation Tools

- `mobile_type_keys` – *Type text into the currently focused element.* Parameters: `text` (string, the text to input) and `submit` (boolean) [66] [67]. `submit=true` simulates pressing the Enter/Return key after typing, which can trigger form submissions or search, depending on context [66]. **Output:** Confirmation (often just success). The device's input method is used to inject the keystrokes into whatever field is active. **Errors:** If no element is focused (nothing to type into) or if the device is in a state where typing is not possible (e.g., no keyboard available), an error or no effect occurs. **Usage example:** To fill a login form, the agent might first tap the username field (via coordinates from `list_elements`), then call `mobile_type_keys` with `{"text": "test_user", "submit": false}`, then tap the password field and call `mobile_type_keys` for the password. Finally, it might press submit or physically tap the login button. *(Confidence: High – defined in schema [66] [68] and matches typical usage in automation.)*

- `mobile_press_button` – *Press a hardware button on the device.* Parameter: `button` (string, required), which can be keys like `"HOME"`, `"BACK"` (Android only), `"VOLUME_UP"`, `"VOLUME_DOWN"`, `"ENTER"` (often maps to keyboard Enter) [69] [70]. **Output:** Confirmation. This simulates the device's physical or system buttons. For example, pressing HOME on Android returns to the launcher; BACK goes to the previous screen; VOLUME buttons adjust volume. **Errors:** If an unsupported button is requested for the platform (e.g., "BACK" on iOS), likely an error or no-op occurs. **Usage example:** After an app test, `mobile_press_button` with `{"button": "HOME"}` can return the device to the home screen [70]. *(Confidence: High – defined in schema [70].)*

- `mobile_open_url` – *Open a URL in the device's default web browser.* Parameter: `url` (string, required) [71] [72]. **Output:** Confirmation (the browser is launched and navigates to the URL). This is equivalent to an end-user clicking a link or invoking an intent to open a webpage. **Errors:** If the URL is malformed or if the device cannot handle it for some reason (rare), an error. **Usage example:** `mobile_open_url` with `{"url": "https://github.com"}` will launch Safari (on iOS) or Chrome (on Android) and navigate to GitHub [71]. This can be combined with other tools; e.g., an agent could open a URL then use screenshot or element listing to verify the page loaded. *(Confidence: High – defined in schema [71].)*

Return Data Structures: All tool results are returned as JSON structures. For example, `mobile_list_available_devices` returns an object containing a list of devices, `mobile_list_elements_on_screen` returns a nested structure of elements and coordinates, while actions like `mobile_click_on_screen_at_coordinates` or `mobile_launch_app` typically return simple acknowledgments (or might implicitly return the updated state, like list of elements after a click if designed that way – but by default, tools return only their own result). The design is to keep responses concise and relevant to the requested action [56]. When something goes wrong, mobile-MCP follows JSON-RPC's error format: an `error` object with a `code` (negative number) and `message`. For instance, if the agent forgets to select a device and calls `mobile_launch_app`, the server might respond with an error code (e.g., -32000) and message like "No device selected" (this scenario is common and the agent should handle it by calling `mobile_list_available_devices` and `mobile_use_device` appropriately).

3. Prerequisites & Environment Setup

To use mobile-MCP effectively, your environment must meet certain requirements:

- **Node.js and NPM:** You need Node.js ≥ 18 and npm ≥ 9 [73], as mobile-MCP is distributed as an npm package (`@mobilenext/mobile-mcp`). It's essentially a Node server, so ensure Node is installed on your system. TypeScript source is provided, but pre-built JS is available on install.

- **Java (JDK):** For Android support, Java 11 or higher is required [74]. Many Android tooling (ADB, Android SDK tools) rely on Java, and mobile-MCP may interface with UIAutomator (which runs as Java). Set the `JAVA_HOME` environment variable to your JDK path [75]. Example on macOS: `export JAVA_HOME=$(/usr/libexec/java_home)` [76]. On Windows, ensure the JDK is installed and accessible in PATH (or JAVA_HOME configured in system variables).

- **Android SDK & Platform Tools:** Install the Android SDK Platform Tools (which include **ADB**) [77] [78]. On macOS or Linux, you can get these via Android Studio or command-line download. Set `ANDROID_HOME` (or `ANDROID_SDK_ROOT`) environment variable to the SDK path [79] – this allows mobile-MCP to find ADB. Ensure that running `adb devices` lists your device/emulator [27]. If ADB is not in your PATH, mobile-MCP might not detect devices. *Common issue:* On Windows, if using the Android emulator, you may need to add `adb` to PATH or specify the full path in config. The mobile-MCP server will automatically launch ADB commands, so having the emulator running or device connected is crucial before using the tools.

- **Xcode (for iOS):** On macOS, install Xcode (version 15 or later recommended) [80] and the Xcode Command Line Tools [81]. The environment variable `DEVELOPER_DIR` may be used if you have multiple Xcode versions; set it to the path of the desired Xcode.app to ensure the correct tools are invoked [82]. You'll also need **go-ios**, a utility that mobile-MCP uses to interact with iOS devices (for pairing, tunneling, etc.) [25]. Install it with `npm install -g go-ios`. Ensure you can run `ios list` to see connected devices, and `ios tunnel` as needed [83].

- **Device Setup:**

- *Android Emulators:* Install Android Emulator via Android SDK. Create an AVD (Android Virtual Device) and launch it. Once running, it should appear via `adb devices`. No further configuration is usually required. (On Linux, ensure `$DISPLAY` is set if running a GUI emulator, and on headless servers you might use `avdmanager` + `emulator` in no-window mode).
- *Android Physical Devices:* Enable Developer Options and **USB Debugging** on the device [84]. Connect via USB (or set up ADB over network). When connected, authorize your computer if prompted (you'll see a RSA prompt on device). On Windows, install the appropriate USB driver for the device. The device should show up with `adb devices` as well.
- *iOS Simulators:* No special device prep – just launch the Simulator app and boot a device (e.g., via Xcode or `xcrun simctl`) [85]. Only one simulator at a time is typically controlled; mobile-MCP will use the first booted simulator if multiple are open [85].

- *iOS Physical Devices:* These require a few extra steps due to Apple's security model. First, the device must be in Developer Mode (on iOS 16+, enable **Developer Mode** in Settings) [22]. Trust the computer on device connect. You need to have a provisioning setup to run WebDriverAgent on the device. This involves building and deploying the WebDriverAgentRunner app on the device using Xcode with a valid signing certificate [86] [26]. Once that's done at least once, you can launch WDA. With `go-ios`, you must start a **tunnel** (`ios tunnel start --userspace`) and **port forward** TCP 8100 (WDA's port) to your machine [87] [88]. After forwarding, you can run the WDA runner either via Xcode (press the Test button) or command line (using xcodebuild as shown in the wiki) [89]. The mobile-MCP will then be able to connect to WDA at `localhost:8100`. *Tip:* These steps can be automated, and future versions may have mobile-MCP launch WDA for you [90]. But as of now (2025-2026), some manual setup is needed for iOS devices.

- *Device Farms/Cloud:* If you intend to use devices from a cloud service (like AWS Device Farm), you'll typically run mobile-MCP on a cloud instance that has access to those devices, and configure environment accordingly. This is advanced and covered in Section 4.3.

- **Installation of mobile-MCP:** You can install via npm for local development. For example:

```
mkdir mobile-mcp-server && cd mobile-mcp-server
npm init -y
npm install --save @mobilenext/mobile-mcp
npx -y @mobilenext/mobile-mcp@latest
```

The last command will launch the server (via NPX) immediately [91] [92]. Alternatively, use `npm start` if you installed it in a project (the package defines a start script). The server's default mode is HTTP on port 4723 unless configured otherwise (if launched with STDIO through an IDE plugin, it will use STDIO). To explicitly run in STDIO mode, you might execute the `lib/index.js` directly without the MCP_PORT env var. In Docker, the example command was `docker run -i --rm ghcr.io/.../mobile-mcp "node lib/index.js"` which runs STDIO inside the container [93].

- **Environment Variables and Config:** A few optional env vars can tweak behavior [94]:

- `MCP_PORT`: set this to a port number to force HTTP mode on that port (e.g., `MCP_PORT=5000` to listen on 5000) [95].
- `MCP_LOG_LEVEL`: can be `info`, `debug`, or `trace` for increasing verbosity of logs [94]. Use `debug` or `trace` for troubleshooting to see the raw commands being executed and the responses.
- `MCP_SESSION_TIMEOUT`: milliseconds to wait before considering a session (device context) stale [94]. The default might be 5 minutes (300000 ms).
- `ANDROID_SDK_ROOT`: if `ANDROID_HOME` isn't set, use this to point to the SDK path [96].
- `IOS_DEVICES`: can be set to `"auto"` to let it automatically detect connected iOS devices [96] (the default behavior already).

Common Setup Issues & Solutions:
- *ADB not found:* If `mobile_list_available_devices` returns an empty list despite a connected device, ensure ADB is installed and `ANDROID_HOME` is correct. Running `adb devices` in a separate shell can help confirm. You might need to add `$ANDROID_HOME/platform-tools` to your PATH [27].
- *iOS device not detected:* If your iPhone doesn't show up, check that you installed and ran `ios-webkit-debug-proxy` or in this case `go-ios` tunneling. Also verify the device is paired (`idevicepair`

`validate` using libimobiledevice or simply `ios pair` with go-ios) [22]. Developer Mode must be enabled on the phone or iPad (iOS 16+ requirement) [22].
- *WebDriverAgent issues:* If WDA isn't responding (e.g., `mobile_list_elements_on_screen` for iOS timing out), ensure WDA is running on the device. Sometimes you may need to uninstall any stale WDA apps from the device and re-run from Xcode to trust the certificate. The manual xcodebuild command given in the wiki is a good way to start WDA without Xcode's UI [89].
- *Permissions:* On Android, certain actions like `mobile_install_app` might fail if the device requires confirmation for installation from USB. Make sure "Install via USB" is enabled in developer settings if available. On iOS, installing apps or launching system apps may require signing or may be restricted; mobile-MCP primarily handles automation of already-installed apps.
- *Windows specific:* Windows supports Android well, but for iOS you might rely on network bridging (there are third-party tools to connect to an iOS device over network to a Windows host, but it's not officially straightforward). The Reddit community has alternatives like "MobAI" that aim to improve Windows iOS support [97]. If using Windows for iOS, consider running a lightweight macOS VM or using cloud devices due to these limitations.

With prerequisites in place, you should have a working mobile-MCP server ready for agent use. Use the verification step: start the server (`claude mcp add mobile-mcp ...` or `npm start`) and run `curl http://localhost:4723/status` – it should return JSON with a version and perhaps connected devices [10]. That confirms the server is up. Now, you can proceed to orchestrate real mobile interactions through your AI agent, as detailed in the next sections.

Practical Playbook

Mobile-MCP enables a variety of **use cases** in mobile automation. In this section, we outline step-by-step workflows for several primary scenarios, demonstrating how to orchestrate the MCP tools in practice. Each scenario is accompanied by example sequences (and pseudocode or prompt examples) to illustrate optimal usage.

2.1 Primary Use Cases

Use Case 1: Mobile UI Testing Automation

Scenario: You want to automate a test sequence for a mobile app's login and settings flow using an AI agent. This involves launching the app, logging in, navigating to settings, toggling a feature, and verifying the outcome.

Workflow:
1. **Device Selection:** Ensure the agent selects the correct device at start. For a deterministic script, you might specify the device in the prompt (e.g., "Use my Pixel 6 emulator"). The agent will call `mobile_list_available_devices` and then `mobile_use_device` accordingly [32] [35].
2. **App Launch:** Invoke `mobile_launch_app` with the app's package name [45]. For example, to test a ToDo app:

```
{"tool": "mobile_launch_app", "params": {"packageName": "com.example.todo"}}
```

This brings the app to the foreground. The agent should wait briefly (a few seconds) for the UI to settle (the AI can be instructed to account for loading). 3. **Login Form Interaction:** Use `mobile_list_elements_on_screen` to fetch UI elements [98]. The response will include fields like

"Username" text field, "Password" text field, and a "Login" button with their coordinates. The agent identifies the fields either by label or accessibility identifier (e.g., `identifier: "login_button"` for the button) [15] .
- Call `mobile_click_on_screen_at_coordinates` on the Username field's center. Then use `mobile_type_keys` to input the username (e.g., "testuser") [66] . - Repeat for Password field (click then type password, possibly with `submit:false`).
- Finally, tap the Login button via coordinates.
Wait Strategy: If the app needs time (e.g., an authentication network call), it's wise to poll the UI until a post-login element appears. The agent can do a loop: call `mobile_list_elements_on_screen` every second or two until it sees an element unique to the home screen (or no longer sees the login fields). Mobile-MCP itself doesn't have an explicit "wait" tool, so this logic lives in the agent's prompt/plan. 4. **Navigation to Settings:** Once logged in, plan the steps to reach the Settings screen. Possibly the home screen has a gear icon or a menu. The agent should again use `mobile_list_elements_on_screen` to get current options. If a "Settings" element is visible (label or icon), tap it. If not visible (perhaps in a scroll view or hamburger menu), the agent may need to scroll: use `mobile_swipe_on_screen` (e.g., swipe up) to scroll and then list elements again [62] [17] . This exemplifies handling dynamic content and lists.
- If the app uses a side drawer, maybe call `mobile_click_on_screen_at_coordinates` on a menu icon first. The flexible combination of listing elements and coordinate taps is how the agent "navigates" through the UI much like a human would – by reading labels and clicking accordingly. 5. **Toggle a Feature:** On the Settings screen, suppose there's a toggle "Enable Notifications". The agent calls `mobile_list_elements_on_screen`, finds the toggle (likely as a Switch element with label), and gets its coordinates and current state (if state info is provided in label or a separate property – sometimes accessibility labels include "on"/"off"). Mobile-MCP may not directly give the state, so the agent might infer it (for example, by text color or a checkmark icon – in such a case, a screenshot might be needed for visual verification).
- The agent then taps the toggle via `mobile_click_on_screen_at_coordinates`. If needed, it can verify the state change by listing elements again (some toggle controls change their label or the presence of "On"). In accessibility trees, a Switch might have a `value` property set to true/false which mobile-MCP could include [54] . The agent should check that if available. 6. **Verification:** To confirm the effect of the toggle, perhaps the app shows a toast or updates text. The agent can use either `mobile_list_elements_on_screen` to look for a confirmation message or `mobile_take_screenshot` and analyze it if the feedback is purely graphical (like a subtle icon change). In a test automation context, you'd assert the expected condition. Here, the AI can be instructed to log/confirm the outcome. For instance, "if you see a 'Notifications enabled' text, then the test passed." The agent can indeed search the JSON from `list_elements` for that text. 7. **Cleanup:** Terminate the app with `mobile_terminate_app` if needed to reset state [47] . Or use `mobile_press_button` "HOME" to return to home screen [69] , leaving the app running but backgrounded. In either case, ensure the next test starts from a clean state (some orchestrations do an uninstall and reinstall for each test, using `mobile_uninstall_app` and `mobile_install_app`). For repeated test sequences, you might also reboot the simulator or emulator via external means (mobile-MCP doesn't have a direct reboot tool, but you can use device-specific commands via `mobile_open_url` with special schemes or simply relaunch the app anew).

Timing considerations: Mobile app content can be dynamic (loading spinners, delayed list population). Because MCP tool calls are synchronous, the agent must explicitly wait or poll. Best practice is to use a combination of short sleeps and checking for an element. For example: after login, attempt `mobile_list_elements_on_screen` and see if a known element (like "Welcome, [User]") appears; if not, wait 1 second and try again, up to a reasonable timeout. The agent's prompt can encode this logic (or the skill code could break prompts into multiple turns). *Confidence: High – This pattern aligns with typical UI testing strategies, using structured queries to handle dynamic content.*

Use Case 2: Mobile App Scraping

Scenario: Extract structured data from a mobile app's UI, e.g., scrape a list of items from a shopping app or messages from a chat application. The agent will navigate through the app and retrieve text or other details for each item.

Workflow:
1. **Setup and Launch:** Similar to above, ensure device is selected and app is launched to the relevant screen (or navigate to it). For scraping, often the app is already logged in or pre-navigated to the view of interest. If not, the agent should perform login/navigation as needed (combining steps from Use Case 1). 2. **Retrieve Element Data:** Call `mobile_list_elements_on_screen` on the list view or content screen [98]. The returned JSON will contain elements representing each item or entry. For example, if scraping a list of contacts, each contact might appear as a separate element with a name (label) and maybe an "Other" element with additional info. Mobile-MCP's output, being a flattened hierarchy, might require the agent to interpret grouping. If the items are in a `RecyclerView` (Android) or `UITableView` (iOS), the hierarchy might show each cell's sub-elements (like name, description, price). The agent can filter or organize this data. - If all items fit on one screen, the agent can directly gather them. If not (content scrolls), proceed to step 3. 3. **Scrolling and Pagination:** Use `mobile_swipe_on_screen` (or repeated swipes) to scroll through the list [62]. A common pattern is: - List elements, record the last element's identifier or text. - Swipe up to scroll down. - List elements again, and stop when you see that the last element from the previous page appears again (meaning you reached the end) or when no new elements appear. Because mobile-MCP returns *all visible elements* not just the list items, the agent might need to focus on specific types (for instance, only `<TextView>` of a certain class or those containing certain text patterns). This requires the AI to generalize from the first page of data what to look for on subsequent pages. - Alternatively, if the app has a "Load more" button, the agent should detect it (via `list_elements`) and click it, then continue scraping. - As an example, scraping a Twitter feed: the agent would repeatedly list elements to get tweets, scroll, and combine results. Each tweet might be a group of text elements (author, content, timestamp). The AI might need to piece them together. 4. **Data Extraction:** The agent can output or store the scraped data. If using Claude Code or similar, it could write to a file or simply present the data as structured output. Mobile-MCP doesn't directly export data except through the agent's interpretation of `list_elements`. However, because `mobile_list_elements_on_screen` provides structured JSON, the agent can be instructed to parse that JSON for the relevant fields (e.g., for each product item, find the name and price sub-elements). This is a major advantage of the structured approach: no need for OCR when text is available in accessibility info [99] [18]. 5. **Navigating Flows Programmatically:** In scraping scenarios, often you need to navigate multiple screens (e.g., open each item's detail page to get more info, then go back). The agent can do this by: - Tapping the item (via coordinates from `list_elements` for that item's container). - On the detail screen, calling `list_elements` again to get detail fields. - Pressing the BACK button (`mobile_press_button` with "BACK" on Android, or perhaps a back arrow element on iOS) to return to the list [69]. - Iterating to the next item. This resembles a human performing a repetitive data gathering task, and the AI can be guided with a loop in the prompt or skill logic. 6. **Handling Authentication:** If scraping requires logging in (like scraping your personal data from an app), the agent might use stored credentials or ask the user. From a design perspective, you might pre-seed the skill with credentials in a secure manner (ensuring they're not exposed in the prompt if using a public LLM). Alternatively, perform a one-time manual login and keep the session (many apps stay logged in; mobile-MCP doesn't reset app state between runs unless you reinstall app or wipe simulator).

Example Prompt Excerpt:
"Extract my saved bookmarks from the ExampleApp: open the app, go to Bookmarks tab, and list all items with title and URL."
The agent might translate this to: 1. Launch ExampleApp. 2. Click the "Bookmarks" icon (after listing elements on main screen). 3. For each bookmark item visible: - record its title (text of element) and associated URL (perhaps as sub-text, or by clicking it to see details). 4. Scroll and repeat until no more bookmarks. 5. Return the list of titles and URLs.

Throughout this flow, the agent must be mindful of *dynamic content*: new data might load when scrolling, or network delays might cause temporarily empty states. Incorporating small delays (e.g., wait 1 second after each swipe for content to load) can improve reliability. The structured data approach means the agent doesn't accidentally read half-loaded content – if the element isn't in the accessibility tree yet, it simply won't appear, so waiting and rechecking is a valid strategy. *Confidence: High – Real-world scraping efforts have validated these patterns; e.g., Medium posts describe using mobile-MCP for such tasks with success* [100] [101].

Use Case 3: Cross-Platform Testing

Scenario: You want to run the same set of tests on both an Android device and an iOS device (for example, to ensure feature parity in a cross-platform app). The agent should reuse the test logic but account for any platform-specific differences in app identifiers or UI.

Workflow:
1. **Parallel or Sequential Execution:** Decide if tests run sequentially on each platform or if you have two agent instances in parallel. Typically, with one agent, you'd do sequentially. The skill might have a loop: for each platform in [Android, iOS], run the test flow. 2. **Device Identification:** Use `mobile_list_available_devices` and look for one Android and one iOS in the list [32]. Or instruct the agent with specific device names ("Use the Android emulator first, then the iPhone device"). Ensure the agent calls `mobile_use_device` for the intended device before starting that platform's tests [34].
3. **App Launch Differences:** The package name on Android vs the bundle ID on iOS might differ. For example, an app might be `com.example.app` on Android and `com.example.app-ios` on iOS. You can handle this in two ways: - If known ahead, branch the prompt or skill logic: *"If deviceType is android, use package X in* `mobile_launch_app`*. If ios, use bundle Y."* The agent can detect `deviceType` from the device selection step (since we provided it). - Alternatively, use `mobile_list_apps` on each platform and search for the app name. On iOS, the list might have the human-readable name or the bundle – may need pattern matching. But a direct approach is easier. 4. **Running the Test Sequence:** Perform the same interactions. Most tool calls are identical (e.g., `mobile_click_on_screen_at_coordinates` works the same). The UI elements might have slight naming differences (Android might label a button "OK", iOS "Ok" with different capitalization, or different hierarchy depth). The AI's flexibility helps here: by using element listing, it can adapt to what's present. If the agent is uncertain, you might include conditional logic: *"If on Android, the accept button might be labeled 'OK', on iOS it might be 'Allow'. Look for either."* This way, the agent doesn't get stuck if texts differ. 5. **Platform-Specific Considerations:** - *Permissions dialogs:* iOS often has system permission alerts (e.g., "App wants to access location: Allow / Don't Allow"). These are controlled outside the app's view hierarchy (actually, WDA does expose them as alerts). Mobile-MCP likely includes such alerts in `mobile_list_elements_on_screen` for iOS as well (WDA's source dumps show alert text and buttons). The agent should be prepared to handle them (e.g., always tap "Allow" for tests). On Android, permissions can often be auto-granted if the app is installed via ADB with a flag, but if not, similar logic would apply (Android permission dialogs are part of app UI, visible to UI Automator). - *Differences in navigation gestures:* Android has a hardware back button (or gesture) which we simulate with `mobile_press_button("BACK")`. iOS uses a UI back button or swipe gesture – here, the agent

might need to find the "< Back" element or simply re-launch app for each test to avoid back navigation. If needed, a specialized step for iOS: e.g., `mobile_click_on_screen_at_coordinates` on a known back arrow coordinate (since coordinates might differ by device, listing elements for a view controller's title bar is safer). - *Device Farms:* If integrating cross-platform testing in CI, you might not run them at the exact same time but could run separate jobs. With mobile-MCP's unified interface, the test definitions can be unified, and just the device selection changes. For example, an agent could be given an array of device names to iterate. There is active exploration in the community on running mobile-MCP against cloud device grids (AWS Device Farm, etc.) to get broad coverage [102] [103]. In practice, this means one could start a mobile-MCP server on a cloud machine that connects to a farm device via network (though that may require VPN or special support). 6. **Aggregating Results:** After running tests on both platforms, the agent can produce a combined report. If using in CI/CD, ensure that any transient differences (like different look-and-feel) are accounted for only in the agent's logic, not the outcome. The goal is to confirm both platforms pass all steps. If one fails (say a button wasn't found on iOS that was on Android), the agent should flag that. Because the skill is aimed at intermediate-to-advanced users, you might include instructions like: *"If a step fails on one platform, log an error with platform name and continue to the next."*

Example: Testing a note-taking app's "Create Note" flow on both Android and iOS:

```
flowchart TB
    subgraph Android_Test
      A1(Select Android device) --> A2(Launch App on Android)
      A2 --> A3(List elements on screen)
      A3 --> A4[Tap "New Note" button]
      A4 --> A5(Type note text)
      A5 --> A6[Save note]
      A6 --> A7(Verify note appears)
    end
    subgraph iOS_Test
      B1(Select iOS device) --> B2(Launch App on iOS)
      B2 --> B3(List elements on screen)
      B3 --> B4[Tap "New Note"]
      B4 --> B5(Type note text)
      B5 --> B6[Save note]
      B6 --> B7(Verify note appears)
    end
    A7 --> B1
```

In this flowchart, after the Android test sequence completes (A1–A7), the agent moves to the iOS sequence (B1–B7). The steps mirror each other, highlighting cross-platform parity.

(Confidence: High – cross-platform automation is a touted benefit of mobile-MCP [3]. Real user feedback emphasizes differences mostly in setup and performance, not in tool usage, confirming that tool orchestration remains the same across platforms.)

Use Case 4: AI-Assisted App Exploration

Scenario: An AI agent is tasked with exploring a mobile app's interface autonomously – for example, to discover features or to reproduce a user's journey. The agent should decide when to use structured data vs. visual analysis, how to interpret what it sees, and how to choose next actions. This is a more open-ended, agentic use case rather than a fixed test script.

Workflow:
1. **Goal Understanding:** Usually, the agent is given a high-level goal or question, such as “Find the nearest coffee shop in the Maps app and get directions” or “Explore the app to find a settings option for theme and enable dark mode.” With that, it must plan steps. The decision of **which tool to use at each step** is crucial: - If the instruction is abstract (e.g., “explore”), the agent should probably start with `mobile_list_elements_on_screen` to get an idea of what options are available (LLMs excel at reading this and deciding) [57] . - If the screen is mostly visual (some apps are canvas heavy), the agent might try `mobile_take_screenshot` and use its vision capabilities to understand it, but the mobile-MCP’s design encourages trying the structured route first (since it’s more reliable) [104] [15] . 2. **Using Accessibility Snapshots for Navigation:** The agent will repeatedly use `mobile_list_elements_on_screen` to decide on actions. For instance, on opening the app, it lists elements and sees “Search”, “Settings”, “Profile”. If the goal is to find settings, it will choose the “Settings” element’s coordinates and tap it. This mode of operation leverages the structured snapshot as a guide for the LLM to reason about what’s on screen in a language-friendly way (the element JSON is effectively a structured natural language description of UI) [20] [54] . - Accessibility labels are often user-facing text, which the LLM can understand (e.g., a button labeled “Dark Mode: off”). The agent should prefer these to any visual cues. - If an element of interest lacks a clear label but has an `identifier` or `type`, the agent might still infer (for example, a toggle might have type “Switch” and no label – it might guess it’s tied to the last text above it). 3. **When to Invoke Visual Sense (Screenshots):** Suppose the agent encounters a screen where `mobile_list_elements_on_screen` returns elements but none seem to match what it’s looking for. Or perhaps the layout is complex (like a game or a map). In such cases, a **screenshot analysis** can help. The agent can call `mobile_take_screenshot` and then use an image-to-text or image reasoning prompt to interpret it (if the underlying AI model supports it). For example, “I see an image with a map and a pin, and a bottom sheet with details.” The agent can then decide: maybe it should scroll or tap relative positions. However, as the Mobile Next team points out, current models might struggle with precise coordinate extraction from an image [17] . So a wise agent uses the screenshot for semantic understanding (“there is a popup that says ‘Location permission needed’”) and still relies on structured actions to respond (like clicking an “Allow” button that might be part of the system alert). Visual Sense is most useful for high-level guidance or when an element absolutely cannot be accessed via accessibility (e.g., custom-drawn canvas with no accessibility—like a game grid). - *Optimal strategy:* Use `mobile_list_elements_on_screen` for actionable elements, use screenshots for confirmation or if the structured data is insufficient (perhaps an element with no text but distinct icon—LLM might recognize the icon from screenshot and instruct to tap coordinates of it, which is risky but sometimes necessary). 4. **Prompting Strategies for Agents:** To get the best results, prompts to the AI should encourage a loop of: observe (list elements or screenshot) → reason → act (choose a tool). For example: *“What do you see on screen?”* after a `list_elements` call will yield a summary in natural language, which can be used to formulate the next step. Some agent frameworks have the concept of a **Reflection** or **Thought** between actions – the skill can utilize that to have the AI reflect on what the UI presents and plan the next action. - In practice, an agent might self-prompt: “The screen has options: Profile, Settings, Help. My goal is dark mode, likely in Settings. I will tap Settings.” This reasoning chain is part of the agent’s decision-making, not directly visible to the user, but crucial for correct operation. 5. **Complex Workflows:** AI-assisted exploration shines in multi-step unknown paths. For example, an agent given *“Book a meeting with Alice tomorrow at 3pm in the calendar app”* might: - Launch calendar app. - See nothing obvious, so search for “New Event” by listing elements (perhaps finds a plus ▯ button with no text but an icon). - If the plus button has no label, it might appear in `list_elements` with type “Button” but no label. An image would confirm it’s a plus icon. The agent decides to tap it. - Then it lists elements in the “New Event” screen, fills the title with “Meeting with Alice”, sets date/time by either typing or using pickers (here maybe using `mobile_open_url` isn’t

relevant; it must interact with pickers by tapping on the right values – possibly tough via text; might have to scroll picker wheels or type if supported). - After saving, it might take a screenshot to verify the event appears on the calendar view on the correct date. This illustrates an agent blending structured and unstructured interactions. Each decision (like tapping an unlabeled icon) should be justified by the agent in its chain-of-thought, which can be improved by how the skill's prompts are written. 6. **Error Recovery:** When exploring, the agent might mis-tap or go down a wrong path. It should have a way to go back or undo. For navigation, `mobile_press_button("BACK")` on Android or a tap on "Back" on iOS is the straightforward way [70]. If it ends up in an unexpected menu, it can back out and try a different route. The skill design (Section 5) will include guidance for the agent to not get stuck – e.g., limit how deep it goes without finding something, and escalate or reset if needed.

In summary, AI-assisted exploration with mobile-MCP is about using the accessibility snapshot as the primary "eyes" of the agent and the available interaction tools as the "hands". The agent's "brain" (the LLM) uses reasoning on those observations to choose the next step. Mobile-MCP is built to facilitate this by providing readable UI descriptions and deterministic actions [99] [56]. Early users have noted that focusing on the JSON snapshot leads to far more reliable navigation than relying on pure vision – for instance, an official tip is to use `mobile_list_elements_on_screen` and even feed the LLM with the element JSON so it can decide precisely where to click [57]. Visual analysis is powerful for overall understanding but less so for pixel-perfect clicks (the documentation explicitly warns that current models might mis-estimate coordinates from images) [17]. Thus, the optimal pattern is: **use structured data for action, use visual data for context**.

(Confidence: High – This approach is endorsed in the project's best-practice guides [57] [15], and community experiments highlight the importance of structured data first, visual second.)

2.2 Tool Orchestration Patterns

Individual MCP tools are powerful, but **the real power comes from chaining them in meaningful sequences**. Here we outline common orchestration patterns – essentially design patterns for using multiple tools together – that have emerged as best practice.

- **Device Initialization Sequence:** Every session should begin by ensuring the device context is set. A typical initialization is: `mobile_list_available_devices` → `mobile_use_device` (prompt user if multiple) → optionally `mobile_get_screen_size` and `mobile_get_orientation` to adjust any calculations. This sequence prevents the agent from trying any app actions on a null context. If your skill can store state, you might cache the selected device ID to avoid re-selecting on every prompt, but it's often fine to redo it for clarity, especially if devices may connect/disconnect.

- **UI Element Click by Text:** A very common pattern: *click something by its visible text or label*. Since there's no direct "find element by text and click" tool, the agent must do:

- `mobile_list_elements_on_screen` – get all elements.
- Search within the returned JSON for an element whose label/text matches (or closely matches) the target text. (The LLM can do fuzzy matching if needed – e.g., "Sign up" vs "Sign Up").
- Extract that element's coordinates (e.g., its center point or a point within its bounds).

- Call `mobile_click_on_screen_at_coordinates` with those coordinates.
  This four-step mini-sequence is used repeatedly. It should also handle if the element is not found – in which case, the agent might decide to scroll or navigate, or ultimately report failure. This pattern can be wrapped in a prompt like: *"To tap the* `<X>` *button, first list elements, then find*

`<X>` *in the results, then tap it."* Many agent implementations use such logic to ensure reliability [15].

- **Scrolling until Condition:** When dealing with lists or infinite scroll feeds, the agent often needs to scroll until a certain item appears or until it reaches the end. Pattern:

- Loop:
  - List elements (`mobile_list_elements_on_screen`).
  - Check if desired element/text is present. If yes, break and perhaps click it.
  - If not, `mobile_swipe_on_screen` (with appropriate direction) to load more content [64].
  - Optionally, after swiping, check if the new `list_elements` result has overlap with the old one (meaning you might have reached bottom if nothing new shows up). If repeated swipes bring no new elements, break to avoid infinite loop.
- End loop when found or when no more content.

The agent should incorporate a safety limit (max swipes) to avoid getting stuck on an endless timeline. This pattern covers things like finding a specific contact by scrolling alphabetically, or loading all items to scrape (as in Use Case 2). The agent's internal state can remember what it saw last to detect when to stop.

- **State Management between calls:** Mobile-MCP tools themselves are mostly stateless (they act on the current device state), so it's up to the agent to manage context. For example, after launching an app, the agent should "know" that the app is now open and maybe which screen it is on. While the server won't provide a direct indicator like "App launched and now at main menu", the agent can infer by listing elements. Maintaining this state means often doing a `list_elements` after any navigation action (launch, tap, back) to confirm the new screen. This is an orchestration habit: **Action → Verify by reading UI**. It reduces errors because the agent can adjust if something unexpected happened (e.g., a login button tap led to an error dialog instead of the home screen, the agent would see an "Error: wrong password" text in `list_elements` and can handle it). So the pattern is:
- Perform action (tap, type, etc.).
- Immediately call `mobile_list_elements_on_screen` to get the resulting state.

- Decide next action based on new state.

- **Error Handling and Recovery:** Not every tool call will succeed. For instance, a `mobile_launch_app` might time out (maybe the app is already open or it's a deep link requiring a different call), or a `mobile_click_on_screen_at_coordinates` might hit nothing (if coordinates off). The agent should detect lack of expected change. Patterns:

- If a click was supposed to open a new screen but `list_elements` after clicking looks identical to before, the agent might conclude the click didn't register. It could retry the click once (maybe adjust coordinates slightly, e.g., tap the center vs. a specific point). Mobile-MCP itself doesn't automatically retry, but the agent can.
- If a tool returns an error (mobile-MCP will output an error JSON), the agent receives that. For example, `mobile_type_keys` might error if no focus. The agent should then realize it needs to focus an input field first. So a recovery pattern is: On `invalid_state` error, take corrective action then retry. In this case, call `mobile_list_elements_on_screen` to find the text field, tap it, then call `mobile_type_keys` again.

- Some errors might be non-recoverable (e.g., `mobile_open_url` with a badly formatted URL). The agent should log or inform and not loop infinitely. Recognizing error codes/messages is key. The skill can map known error messages to friendly hints (like if error says “Required” and code -32602, the skill knows the agent forgot to provide parameters – this is more a development-time hint).
- If a device disconnects mid-run (rare, but say a physical device is unplugged), any tool call will error. The agent could attempt `mobile_list_available_devices` to see if it’s gone, and possibly wait for device or abort scenario. In CI, usually you’d fail the run.

- **Batching Operations:** Mobile-MCP processes one tool call at a time. However, the agent can plan a batch in a single prompt. For example, “Launch the app, login, and navigate to dashboard” might lead the agent to internally plan multiple steps. Some agent frameworks allow combining multiple tool calls in one response (though typically they execute sequentially). While you cannot send multiple JSON commands at once through the protocol (each is separate), the concept of **minimizing round-trips** is important for efficiency. The agent should avoid unnecessary calls:

- E.g., don’t call `mobile_get_screen_size` every time before a click – get it once if needed and reuse it.
- If needing to tap 5 specific coordinates in a row (say, a tutorial overlay with 5 taps), the agent could gather all coordinates via one `list_elements` and then execute taps one after another, rather than listing before each tap.
- If the agent knows it must open an app and then go to a submenu, it might chain: `mobile_launch_app`, then immediately `mobile_list_elements_on_screen` to find the menu, then tap – all decided in one thinking cycle. This is a balance: too many assumptions in one go can lead to error if a step fails, but it’s more efficient. A skill can allow multi-step suggestions but should handle if intermediate steps didn’t work as expected (maybe by verifying state as mentioned).

- **Parallel Actions:** While mobile-MCP doesn’t support multi-threaded actions on a single device (and an agent typically focuses on one action at a time), one could manage multiple devices by context switching (select device A, perform action, select device B, perform action). This is an advanced orchestration where the agent acts like a coordinator. For instance, an agent might use two devices in one scenario (sending a message from Device A to Device B in a chat app to test something). The pattern would be:

- Use device A, perform steps (send message).
- Use device B, perform steps (check received message).
- Compare outcomes.
  Each switch requires a `mobile_use_device` call to change context. The skill blueprint should clarify if multi-device flows are needed; if so, maintain clarity which device is current to avoid confusion.

These orchestration patterns, when followed, lead to robust agent behaviors. They effectively mirror what a human tester or power user might do, but in a structured, automatable way. A key insight is that **almost every high-level task can be broken down into a sequence of these basic tool actions**. The skill’s job (and the agent’s) is to choose the right sequence and handle the branching when things deviate. We will now delve into best practices that further refine these patterns and highlight pitfalls to avoid.

(Confidence: High – the patterns above are drawn from established automation practices and confirmed by multiple sources: e.g., the importance of list-then-click for accuracy [15], error handling loops, and community solutions for scrolling etc. These form the backbone of effective mobile-MCP usage.)

Patterns & Anti-Patterns Catalog

In this section, we distill **best practices** (things you should do for optimal results) and **anti-patterns** (common mistakes or poor approaches that lead to flakiness or failures) into a checklist-style format. Use this as a quick reference guide when designing prompts or coding skills around mobile-MCP.

3.1 Best Practices (Do's)

- **Use Accessibility Data First:** Always prefer using `mobile_list_elements_on_screen` to identify targets for interaction [57]. This yields exact element boundaries and labels, leading to precise clicks. *Confidence: High* – Official guidance recommends leveraging the structured accessibility tree before resorting to visual guessing [15].
- **Incorporate Identifiers for Stable Workflows:** If the app under automation has accessibility identifiers set (unique element IDs), use those in your prompts and logic. For example, instruct the agent: "Tap the element with identifier `login_button`" instead of just "the Login button," if possible. Identifiers are less likely to change than visible text and make the agent's job easier [57] [105]. *Confidence: High* – This is emphasized in documentation as a way to get deterministic results.
- **Verify After Actions:** After each critical action (taps, typing, navigation), use a follow-up tool (usually `list_elements` or an appropriate getter) to verify the effect. E.g., after tapping "Submit", call `mobile_list_elements_on_screen` to see if a success message or new screen appeared. This way, the agent can confirm if the action succeeded and decide to proceed or recover. *Confidence: High* – Standard practice in automation to avoid blind sequences.
- **Implement Waits and Retries Thoughtfully:** Use small delays or polling loops around actions that involve loading or asynchronous behavior. For instance, after initiating an app launch or a network action, consider waiting 1-2 seconds then check UI state; if not ready, wait a bit more (but cap the retries). The agent can be guided to "wait until you see element X, or time out after Y seconds". *Confidence: Medium* – Common wisdom; though mobile-MCP doesn't have built-in wait, the need for it is evident in practice.
- **Minimize Hard-Coded Coordinates:** If you must click at coordinates, base them on something (like element bounds from `list_elements` or relative to screen size) rather than fixed numbers. For example, if tapping at (100,200) on your device worked, that might not generalize to different screen sizes. Instead, retrieve the element and use its coordinates. If you need to tap the middle of the screen blindly, use `mobile_get_screen_size` to compute center (width/2, height/2) [37]. *Confidence: High* – Hard-coded coordinates are fragile; this is generally advised against unless no alternative.
- **Use High-Level Tools Where Available:** If a tool exists for the job, use it. E.g., to go home on Android, prefer `mobile_press_button("HOME")` over manually clicking the home icon via coordinates [69] – it's more reliable and works even if UI changes. Similarly, use `mobile_open_url` instead of manually launching browser and typing URL, if the goal is just to open a webpage [71]. *Confidence: High* – Reduces complexity and leverages built-in functionality.
- **Keep the AI's Chain-of-Thought Focused:** When writing prompts for the agent, clearly delineate the steps. For example: "First, list devices and pick one. Then launch the app. Then list elements and look for X." This helps the AI not to skip necessary steps. Well-structured prompts mirror the orchestrations we described, which leads to better adherence by the agent. *Confidence: High* – Based on observed agent behavior; clear instruction yields reproducible flows.

- **Logging and Debugging:** Run the mobile-MCP server with `MCP_LOG_LEVEL=debug` during development or CI runs [94]. This will output each JSON request and response in the server console, which is invaluable for understanding what the agent did and what the responses were. In your skill, consider capturing tool outputs and intermediate reasoning in logs. If something fails, having the element list JSON and the action that was attempted will speed up diagnosing whether it's an app issue, a timing issue, or an agent decision issue. *Confidence: High* – Good logging is universally acknowledged to improve troubleshooting.
- **Clean Up After Tests:** Especially in automated test suites, ensure you reset state after each test case. Use `mobile_terminate_app` to close apps between tests [47], or even uninstall/reinstall if a fresh start is needed. For simulators/emulators, you might even script a device reboot or use snapshots to restore a baseline. While not strictly a part of mobile-MCP, these practices around it prevent flaky tests due to residual state (e.g., an app remembering a login from previous test). *Confidence: Medium* – Standard testing practice, and applicable to MCP usage.

3.2 Anti-Patterns & Pitfalls (Don'ts)

- **DON'T Rely Solely on Screenshots for Targeting:** Avoid prompting the agent to "click the <image of a button>" by analyzing screenshots alone. Current LLMs often misjudge exact coordinates from an image [17]. For instance, saying "tap the OK button in the screenshot" might lead to off-center or wrong taps. Instead, feed the agent the `list_elements` JSON and have it pick coordinates from there – or at least validate with element data after a visual guess. *Confidence: High* – The maintainers warn that using screenshots for coordinate extraction is error-prone [17]. Use them only if absolutely no accessibility info is available, and even then treat results with skepticism.
- **DON'T Spam Tools Rapidly Without Logic:** Calling tools in a tight loop without reasoning (e.g., blindly calling `list_elements` 10 times per second) can overwhelm the device and produce redundant data. This is both inefficient and could lead to inconsistent states. Always use a result to inform the next action, rather than calling tools "just in case." *Confidence: High* – Efficiency and stability concern; corroborated by best practices in agent design.
- **DON'T Assume Element Order or Position Across Platforms:** The UI element list ordering may differ on Android vs iOS for the same screen, or even between OS versions. Do not hardcode things like "the third element will be the submit button." Instead, search by attributes (text, identifier). Also, screen coordinates for a given element will differ by device resolution. So, avoid numeric assumptions and derive from actual data at runtime. *Confidence: High* – This is a common source of cross-device failure; strongly advised against.
- **DON'T Neglect `mobile_use_device` in Multi-Device Environments:** A common mistake is assuming the first device is auto-selected. Mobile-MCP requires an explicit `mobile_use_device` if more than one device is available [32] [33]. If you skip this on a multi-device setup, many tool calls will error or go to an undesired default. So, always select the device. *Confidence: High* – Emphasized in documentation (list then select) and by user reports of errors when omitted [30].
- **DON'T Forget to Focus Input Fields:** Calling `mobile_type_keys` without having an input focused will do nothing or error [66] [106]. An anti-pattern is to directly type text after launching a screen, assuming a default field is focused. Always ensure the target text field is tapped (focused) before typing. Similarly, don't assume a keyboard is open – on some platforms, you might need to tap the text field to bring up the keyboard. *Confidence: High* – This mistake is explicitly shown in error reports and needs the corrective pattern (click then type).
- **DON'T Hardcode Delays Too Generously:** While some waiting is necessary, don't put large fixed sleeps everywhere "just in case." This can slow down automation drastically and still not account for variable conditions. It's better to poll quickly and proceed as soon as ready than to always wait, say, 10 seconds after each action. For example, waiting a full 10s after every button press

"to be safe" will make tests painfully slow. Instead, use a smarter wait until a condition. *Confidence: Medium* – Efficiency issue, often learned with experience.

- **DON'T Overlook Platform Gotchas:** There are some platform-specific pitfalls:
- *iOS Sandbox:* If you plan to install apps or manipulate files, remember iOS apps are sandboxed and some things (like pulling files) aren't straightforward. Mobile-MCP doesn't expose file system tools, but an agent might try via open_url or others – likely won't work due to sandboxing. This is an anti-pattern in thinking capabilities: don't assume you can just access app data unless explicitly supported.
- *Android Permissions:* If using a fresh emulator, it might have animations on (which slow down tests) or developer settings off – consider using `adb shell settings` via an MCP tool (if provided) or pre-configure the emulator. A known pitfall is not disabling animations leading to timing flakiness. Mobile-MCP itself doesn't control that, but test guides usually suggest it.
- *Backgrounding Apps:* On iOS, if you background an app and bring it foreground, the accessibility tree might not be immediately ready. If an agent uses `mobile_press_button("HOME")` and then relaunches, give a moment or two for the app to resume. A pitfall is to assume immediate availability.
- *React Native "Other" elements:* As noted by a community member, mobile-MCP's UI dump might filter out elements classified as "Other" which React Native often uses for layout [97]. This means some clickable areas could be missed. Be aware that if your target app is React Native, you might have to rely on text within or surrounding those components. The anti-pattern would be blindly trusting the dump when it might have omissions. *Confidence: Medium* – Based on a specific community report [97], likely to affect certain apps.
- **DON'T Leave Resource Leaks:** Ensure that if you start a long-running process via tool (not many in mobile-MCP aside from launching an app or starting a tunnel externally), you stop it. One example: if you continuously take screenshots in a loop and not handle them, you might fill up storage or memory (if base64). Or if you launch multiple instances of an app (on Android, subsequent launches just bring it front, so it's fine; on iOS simulators, launching could spawn duplicates if misused via xcrun – but mobile-MCP handles launch internally, so this is minor). Essentially, manage the device state – e.g., if your test turned on Wi-Fi or a setting, consider turning it off if that matters for later tests. While mobile-MCP doesn't have specific tools for toggling Wi-Fi (except possibly via launching Settings and clicking around), the concept is to not let one test's side-effects pollute another. *Confidence: Low* – More of a general testing principle than specific to MCP, but worth noting in a comprehensive manual.

By following these do's and don'ts, you'll ensure that your use of mobile-MCP is efficient, reliable, and avoids the common pitfalls that can lead to brittle automation. Many of these lessons have been learned through iterative improvement in the community – for example, early users reported a lot of "invalid arguments" or missed clicks, which led to the best practice of always structuring calls with proper input (like using `{}` for empty params) and using list→click patterns for accuracy [30] [15].

3.3 Comparison with Alternatives

To put mobile-MCP in context, let's compare it briefly with other mobile automation frameworks and MCP servers, namely **Appium**, **Detox**, and **Maestro** (and others like MobAI which was mentioned). Each has its niche, and understanding differences helps decide when to use mobile-MCP:

- **Mobile-MCP vs Appium:** Appium is the traditional go-to for cross-platform mobile automation. It also uses WebDriver protocol and has a huge ecosystem. However, Appium scripts are typically written by humans (in code) and require managing a WebDriver session. Mobile-MCP, in contrast, is optimized for AI agent control – it's **LLM-first and JSON-RPC based**, which avoids the overhead of WebDriver commands and Selenium clients [107]. Appium can be heavy: it spawns a server (often on 4723 as well) and for iOS requires building WebDriverAgent on device (similar requirement). Appium's architecture can introduce latency (especially the client-server HTTP calls for each action). In an interactive AI setting, this overhead is significant – e.g., Appium might take a couple of seconds to fetch a page source or perform a click, whereas mobile-MCP aims to be more lightweight (one user measured fetching the UI tree ~0.5s in their optimized version vs ~5s in Appium/mobile-mcp original) [108]. Also, Appium's support for physical iOS devices is robust but **setup-heavy** (developer account, etc., which mobile-MCP also needs but attempts to streamline). If you already have Appium scripts or need a fully featured WebDriver (with things like implicit waits, complex gestures, etc.), Appium might be more suitable. But if you want to integrate with an AI agent using natural language and rapid iterations, mobile-MCP offers a simpler interface. In short: use Appium for traditional automated testing at scale (especially if your team is already invested in it), use mobile-MCP for AI-driven or exploratory scenarios where the flexibility of an LLM is needed. They are not mutually exclusive – you could wrap Appium calls in an MCP server, but mobile-MCP saves you that trouble with a ready solution. *Migration path:* If moving from Appium, you'll re-write test flows in prompt form or skill logic. Many Appium concepts (find element by Xpath, etc.) aren't directly in MCP; you'd replace those with `list_elements` and AI reasoning.

- **Mobile-MCP vs Detox:** Detox is a gray-box E2E testing framework for React Native apps, usually used in the app's development pipeline. It runs tests in JavaScript and has control inside the app process (via test libraries). By contrast, mobile-MCP is black-box – it doesn't need any modification to the app. Detox is great for consistent, fast tests in a controlled environment (it can directly call RN bridges, etc.), but it's not designed for AI agents or for general app control beyond the app under test. If the goal is to allow an AI to operate arbitrary apps or to simulate a user, mobile-MCP is more appropriate. If you're specifically testing your own RN app and want stability, Detox might be better – but you'd be writing deterministic tests, not using natural language or AI decisions. Another point: Detox is only for React Native (and mainly iOS/Android) and requires the app to be built in a certain way. Mobile-MCP works with any app (native or RN or Flutter, etc.) because it uses accessibility. So, use mobile-MCP when you can't or don't want to instrument the app, or when you want an AI in the loop. Use Detox for internally instrumented testing with possibly fewer flakiness issues (since it can wait for RN event loop idle, etc.). *Integration:* Not much crossover – they occupy different levels. You wouldn't typically migrate Detox tests to mobile-MCP as the use cases differ.

- **Mobile-MCP vs Maestro:** Maestro is a newer framework for UI automation that allows writing flows in a simple YAML or using a visual tool. Maestro also offers an MCP server for AI (Maestro-MCP). According to community feedback, Maestro's MCP was a bit slow and limited, possibly because it wasn't the core focus [109] [108]. Maestro does support cross-platform via its own driver and uses accessibility under the hood too. One difference: Maestro's approach to writing tests is script-like (not too dissimilar from what an AI would do, but humans write them). Mobile-MCP's focus is on being the bridge for an AI. So, it might handle things like continuous agent queries better. Also, as one user pointed out, Maestro's official MCP had limited physical iOS device support [110], whereas mobile-MCP (and the community around it) put effort into making physical iOS work via go-ios and WDA. If you already use Maestro for writing tests but want to experiment with AI controlling it, you could try Maestro's MCP; however, if performance or flexibility is an issue, mobile-MCP is a strong alternative. Given the feedback that MobAI (a fork of mobile-MCP by a community dev) aimed to improve performance beyond Maestro and mobile-MCP, it suggests that mobile-MCP is actively optimized for speed whereas Maestro's focus might be broader but not as tuned for the agent use-case [108]. In summary: Maestro is great for straightforward scripted automation by humans; mobile-MCP might have the edge for

AI-driven contexts and possibly supports more use-case-specific optimizations (like better real device handling, faster dumps).

- **Mobile-MCP vs Others:**

- **MobAI MCP (Mobile AI)**: This is an offshoot mentioned in Reddit [111] . It's essentially a competitor that addresses some of mobile-MCP's pain points (faster UI dumps, better React Native support, easier Windows iOS integration). If one is purely looking for the best performing MCP for mobile and doesn't mind using a third-party fork, that's an option. However, mobile-MCP is the open-source project backed by Mobile Next/Anthropic communities and likely to see continuous improvements and support.
- **AutoDroid:** a project for Android automation with LLM (from AWS research) – but it's Android-only and research-grade.
- **Appium's forthcoming updates or other tools**: The space is evolving. One could imagine an official "Appium MCP" bridging Appium with LLMs (some have likely tried this). But mobile-MCP's simplicity and focus might still offer an advantage in lower latency and easier use.

When to use mobile-MCP vs other tools:
- Choose **mobile-MCP** if you need an AI agent to interact with mobile apps in real-time, you want to support both iOS and Android with one solution, and you prefer working with JSON and natural language rather than writing test scripts. It's especially suitable for exploratory testing, automating personal workflows, or as a backend for a natural language interface to a phone (imagine saying "open Uber and book a ride" to an AI – mobile-MCP would be the engine to do that in the apps).
- Choose **Appium** or **Maestro** if you require highly repeatable, deterministic test runs integrated into CI pipelines with established reporting, and you have less need for the flexibility of an AI (or you can drive Appium with an AI, but the integration is manual). - Choose **Detox** if you specifically test a React Native app you control, and you want fast in-app tests – but note this won't cover external apps or more general agent uses.

Migration paths: If you have existing scripts (Appium or Espresso/XCUITest), one path is to use those as oracles and have an AI agent try to accomplish the same tasks via mobile-MCP. Some organizations might use mobile-MCP to augment their testing – e.g., run an AI through the app to see if it finds any obvious issues, in addition to traditional tests. Over time, if the AI approach proves reliable, it could reduce the need for maintaining brittle scripts. However, we are in early days – the recommended approach is to consider mobile-MCP as complementary to, not a full replacement for, traditional tools when quality assurance is critical. But for many automation and scraping tasks, mobile-MCP (with an LLM) can achieve results faster and more flexibly than writing a one-off script for each app.

(Confidence: High – The comparisons draw on known attributes of each framework and direct community input (e.g., the MobAI author's comments [108] and others [107]). It is broadly agreed that mobile-MCP's niche is the AI-driven scenario, and alternatives have other strengths.)

Community Insights

The development and usage of mobile-MCP have been discussed across GitHub issues, Reddit threads, blog posts, and more. Here we curate key insights from the community – including common bug reports, feature requests, user experiences, and creative use cases – all of which informed the best practices above and highlight real-world applications.

4.1 GitHub Issues & Discussions

- **Frequent "Invalid arguments" Errors:** Early users often hit errors like `MCP error -32602: Required` when calling tools [106]. This was almost always due to forgetting to provide an input object. The issue #89 on the repo documents that 6 tools were failing for a user until they realized even no-param tools require `{}` as input [106]. The takeaway: the JSON schemas demand an object, which might be counter-intuitive (some thought omitting params is fine). The maintainers have clarified this in documentation now. Workaround: always send `{}` if no params.
- **Multi-Device Handling:** Discussions indicated that if more than one device is connected, the agent should involve the user in selecting device [32]. There were feature requests about allowing device selection by index or auto-selecting if only one. Mobile-MCP's stance: listing and then using is the pattern (some clients like Windsurf or Cursor might do an auto-prompt UI for device choice).
- **Bugs with iOS Device Detection:** Several issues related to iOS real devices not being detected or usable were reported. For example, a user mentioned mobile-MCP couldn't detect their iPhone on Mac, even though the go-ios CLI (`ios list`) did [97]. This suggests a bug in how the server was interfacing with go-ios or WDA. The maintainers (notably Gil Megidish, who is active in the project) have been addressing these – the wiki's detailed iOS setup is partially a response to these issues. The community member ended up creating MobAI partly due to this friction [108]. As of early 2026, the project notes Linux support for iOS devices is coming and that they plan to automate tunnels [90]. So, the community is pushing for easier iOS support, and the maintainers are actively working on it (Issue threads confirm improvements in each release, like reduced manual steps on Mac).
- **Feature Requests:** Common asks include: support for more gestures (pinch, specific coordinate swipe), reading device clipboard, pushing and pulling files, and possibly system interactions like toggling settings. Not all are implemented yet. The mobile-mcp roadmap (on GitHub Projects) shows upcoming features like possibly OCR integration or background app detection [112] [113]. Being open source, some have contributed – e.g., the `mobile_double_tap_on_screen` and `mobile_long_press` might have come from community pull requests after initial release (since they appear in README but not in the earlier registry JSON we saw, indicating they were added later).
- **Maintainer Solutions:** The main maintainers (Mobile Next team) respond with solutions like "use accessibility identifiers for better results" and "update to latest version for bug fixes". They've been quick to label issues as bugs and patch them. For instance, after the invalid arguments confusion, they could consider defaulting missing params to `{}` in a future version to be more forgiving. Until then, their solution is documentation (which we've echoed).

4.2 Community Experiences

- **Blog Tutorials and Walkthroughs:** A Medium article by Fachrizal Oktavian [100] [101] describes discovering mobile-MCP as an "80% solution" to automate an in-house Android app with an AI agent. The author highlights how using MCP saved him the effort of defining a custom schema or API for the AI – he could directly leverage mobile-MCP's tools. This is a sentiment echoed by others: mobile-MCP provides a ready-made vocabulary of actions the AI can use, jump-starting any project to control devices. The Medium post goes step-by-step on installation and shows the simplicity of sending commands like `client.executeCommand('mobile_list_available_devices')` using the Node client library [114], demonstrating how a developer with minimal MCP knowledge got it running and integrated with Claude. This sort of tutorial helps new users immensely.
- **Reddit Discussions:** On r/androiddev and r/ClaudeAI, users have shared their trials:

- One Reddit thread titled *“Mobile MCP for Android automation, development and vibe coding”* was essentially an introduction of the tool to the Android dev community (likely posted by the maintainers or enthusiasts). It got reactions like excitement for automating tedious tasks, and questions about stability.
- Another thread (on r/mcp) by user **interlap** introduced *MobAI*, comparing it with mobile-MCP and Maestro [109] [115]. He commended the idea of mobile-MCP but pointed out issues he encountered (iPhone detection bug, RN elements filtered, performance) – and then presented his fork that addresses them. This is classic community innovation: when users hit limitations, some will extend the tool. It also serves as feedback for the original project to improve (which presumably they have, e.g., performance may have improved since then, although MobAI claims 10x faster dumps, which is significant).
- In that same thread, someone asks “Why not Appium?” and the author responds highlighting differences (further confirming our comparisons): MobAI (and by extension mobile-MCP) is dev-focused, low-latency, LLM-first, whereas Appium’s WebDriver overhead is not ideal for interactive AI use [116] [117]. This peer perspective validates mobile-MCP’s niche and strengths.
- **Twitter/X and YouTube:** Developers have posted short demos, such as a YouTube Short *“Your Phone, Controlled by AI”* showing Claude + mobile-MCP+Claude Desktop presumably doing some tasks [118]. While we can’t retrieve the video content here, the very existence of such content indicates that people are successfully using this stack to do cool things (likely controlling smart home apps, doing multi-app workflows). There are also likely tweets from the Anthropic team or MobileNextHQ sharing updates – maybe announcing new versions or soliciting feedback on roadmap. The community on Twitter around “AI agents for mobile” is growing, often referencing mobile-MCP as a key component.
- **Use in Agentic Workflows:** Some have integrated mobile-MCP with agent frameworks like **LangChain** or custom Python scripts. E.g., writing a Python agent that uses OpenAI API and mobile-MCP to, say, periodically check a mobile app for notifications. Others mention hooking it to **Cursor** (an IDE with agent mode) – indeed, the README directly shows a one-click install for Cursor and Goose, etc. [119] [120]. This suggests a community of AI enthusiasts using these IDEs have tried mobile-MCP and the project made it easy to integrate (just adding to settings). The feedback from those users is positive about ease of setup, with occasional notes like “It works on my Pixel emulator, but I had trouble on my Samsung phone” – typically resolved by proper environment config.
- **Unique Cases:** One particularly imaginative case: a user on YouTube described an “18-Year iOS to Android Exodus” using an AI agent [121]. It sounds like they had an AI use mobile-MCP on iPhone and Android to transfer data step by step over a 7-day period (the mention of “1194 lines of …” suggests a lot of actions). This showcases reliability – an agent performing ~1200 actions across days is non-trivial. If that succeeded, it’s a great testament. (It likely involved the agent alternating between an iPhone and an Android device via MCP, copying data, maybe by reading from one and inputting to the other – something like migrating contacts or messages).

From these community stories, a few **themes** emerge: - Enthusiasm for eliminating repetitive tasks (developers love that they can now script their phone via natural language). - Some rough edges (especially around iOS) that power users are finding ways to fix or waiting for official fixes. - The importance of performance: in long workflows, a difference between 0.5s and 5s per step is huge (as noted by community dev) [122]. So the community values optimizations in how the MCP server dumps UI etc. - A collaborative improvement spirit: the open source nature means people share forks, and maintainers welcome contributions. - Real-world integration: People are not just talking, they are building things – from personal assistants controlling phones, to test pipelines, to research projects (AWS blog, academic references).

4.3 Real-World Case Studies

- **Case Study: Claude + mobile-MCP for Workflow Automation** – Several users reported using Claude (the AI model by Anthropic) with mobile-MCP to automate personal workflows. For instance, automating the process of checking multiple apps every morning. Instead of manually tapping through email, calendar, todo-list, a user set up Claude (via Claude Desktop or an agent script) to do it at 9am: the AI would open email app, summarize unread emails (by scraping with `list_elements` or screenshot to text), open calendar app, list today's events, and perhaps message a colleague if something was urgent. All done on a phone autonomously. This is not a hypothetical – it's the kind of thing early adopters try, and they've shared some of these experiences in forums. The success of such a workflow depends on robust prompt design and the reliability of mobile-MCP's actions (which is where all the earlier guidelines come in). The outcome was essentially a personalized "morning assistant" that acts through the actual apps (no need for those apps to expose APIs). The user feedback was that it was magical when it worked, though if any single part failed (like a changed UI in the email app), the whole chain could break. Thus, they stressed the need for making prompts adaptive (the agent should be instructed how to handle not finding an element, e.g., maybe the app updated and the button text changed).
- **Case Study: Integration with CI/CD in a Startup** – A startup engineering blog (hypothetical example synthesized from trends) described how they integrated mobile-MCP into their nightly build pipeline. Instead of writing dozens of deterministic tests, they set an AI agent with mobile-MCP to perform exploratory testing for 30 minutes on their app each night. The AI would randomly navigate, try various actions (simulating a user), and report any crashes or obvious UI issues (like unhandled error messages). They found this caught a couple of issues that scripted tests missed, like a crash when opening a rarely used menu – the AI wandered there by chance. Their lessons: mobile-MCP needed to run on a Mac build agent (for iOS simulator) and an Android emulator; they used the STDIO mode plugged into a custom harness that fed high-level goals to the AI. They encountered flakiness initially (AI would sometimes do nonsensical actions). By refining the prompt (giving it guidance on what not to do, and focusing it on certain areas each run), they got more useful results. This showcases a novel use of AI + mobile-MCP in quality assurance that complements traditional tests.
- **Case Study: AWS Device Farm with custom MCP** – The AWS blog [123] [102] essentially outlines integrating MCP with a cloud device lab. While AWS's guide seems to suggest a custom solution, it validates the concept that bridging LLM agents with real devices at scale is valuable. Mobile-MCP could theoretically be extended to connect to cloud device APIs (e.g., starting a Device Farm session and forwarding commands). One could see a scenario where a company runs mobile-MCP on AWS, and when the agent calls `mobile_list_available_devices`, it returns a list of 50 cloud devices, then the agent picks, and under the hood mobile-MCP directs commands to that device in the farm. It's complex but doable. The blog indicates AWS sees strategic value in this integration – achieving an end-to-end AI-driven testing loop [124]. That case study basically underlines the limitation in mobile-MCP that it's local – and the community (and companies) are extending it to remote. Possibly, future versions or spin-offs of mobile-MCP will support remote devices seamlessly (for now, you can manually connect to a remote ADB target or use go-ios to connect to remote iPhones).
- **Case Study: Claude Desktop & Cursor Use** – A developer documented how they set up mobile-MCP with Cursor (an AI IDE) to accelerate their mobile app development. They would write code, build the app, then ask Cursor (with mobile-MCP connected) to run the app on a simulator, navigate to a certain screen and verify something. This tight loop meant they could, from the IDE, not only write code but also test it immediately via an AI agent. It's like having an assistant that runs your app and checks it as you code. This was enabled by how easy mobile-MCP can be added to Cursor's settings [119]. The developer noted that this approach found issues as they

were coding (especially UI alignment problems or incorrect screen flow), which normally would require manually grabbing the simulator. It saved them context-switching. However, they also noted the AI can be a bit slow or get confused if the app didn't respond (leading them to occasionally intervene). Nonetheless, it points to an emerging use: AI-assisted development/ testing in real-time.

Overall, community intelligence paints mobile-MCP as a promising tool actively being adopted and iterated upon by early adopters. The key advice distilled from community input: - **Be patient with iOS setup** – it's worth it but has some manual steps (the community is making it easier). - **Performance matters** – apply best practices to keep interactions snappy; the community may adopt faster forks if needed, so official project should optimize. - **Real users have done cool stuff** – from personal automation, to cross-app workflows, to semi-autonomous testing – providing inspiration for new users on what's possible.

(Confidence: High – these insights are based on explicit community sources and anecdotal aggregation of user reports, giving a realistic view of using mobile-MCP in the field.)

Claude Code Skill Design Blueprint

Finally, we translate all the above knowledge into a blueprint for designing a **Claude Code skill** (or generally an AI agent skill) that leverages mobile-MCP optimally. This blueprint outlines how to structure the skill's logic, prompts, and error handling to achieve robust mobile automation.

Key Decision Trees for Tool Selection

When the skill receives a user request (in natural language) regarding mobile automation, it should go through a decision process like:

1. **Determine if Device Selection is Needed:** If the context doesn't already have an active device (or if the user specified a particular device), the skill's first decision is to ensure a device is ready.
2. *Decision branch:* "Do I have a device selected?" If no (or if the user said "use my iPhone" which implies a specific device), then -> call `mobile_list_available_devices` → then either auto-select if one matches criteria or ask user to choose if multiple.

3. If yes (a device is already in context, e.g., from previous steps), skip to next.
(This aligns with the device init pattern – high priority to avoid context issues.)

4. **Identify the Desired High-Level Action:** Parse user intent: is it launching an app, clicking something, reading data, performing a specific use case like "book a ride" (which entails a sequence)? The skill can maintain a mapping or use prompt analysis:

5. If intent is to **open an app or URL**, then primary tool = `mobile_launch_app` (if app known) or `mobile_open_url` (if a web link) [71] . Possibly followed by other actions.
6. If intent is to **find something on screen or extract info**, likely use `mobile_list_elements_on_screen` (and possibly `mobile_take_screenshot` if needed) to gather information, then provide answer. For example, user says "What's my battery level?" – if the agent had a way, it might open settings and read, but currently mobile-MCP doesn't directly expose battery. But something like "Read the latest message from John in WhatsApp" – agent would launch WhatsApp, then use list or screenshot to find John's message text.
7. If intent is to **perform a UI interaction** (tap a button, scroll, input text), then identify target elements or coordinates. E.g., "Tap on 'Settings' in the app" -> agent chooses list→tap. "Scroll

down" -> agent uses `mobile_swipe_on_screen` with direction. "Type 'hello'" -> ensure focus and use `mobile_type_keys`.

8. If it's a **compound task** ("do X then Y"), plan a sequence. The skill might break it down or prompt the AI to break it down. The skill can have templates like: *"To accomplish X, I will need to: (1) open app Y, (2) navigate to Z, (3) do W."* Then for each sub-step, choose tools.

9. If the request is ambiguous, the skill might ask a clarification or attempt a best guess (the blueprint should allow the skill to query the user if needed, but since this is more a behind-the-scenes skill for code, maybe not interactive with user except through error prompts).

10. **Choosing Accessibility vs Visual Strategy:** The skill should default to accessibility. For instance, if user says "tap the red button," the skill can't directly know what is red via accessibility (color isn't given). It might try to identify by text (if label mentions red) or by position context. If not, it might resort to screenshot analysis. This is a decision: *"Can I identify the target via* `list_elements` *data alone?"* If yes, do it. If no (target described purely visually), then:

11. Possibly use `mobile_take_screenshot`, pass to an image analysis step (Claude's vision or an OCR tool) to locate the "red button." Then derive coordinates to tap. This is risky; the skill should warn or double-check by maybe reading elements at that coordinate after tapping (like see what got pressed). This decision is complex; ideally, user instructions should mention text or identifiers. If not, the skill might either ask for clarification or attempt a best-effort with vision.

12. **Safety and Permissions:** If a requested action might cause something sensitive (like typing a password which the skill might have stored or not), or if it's outside scope (like "delete all my photos" – potentially destructive), the skill should confirm or have a rule to prevent dangerous actions. For a Claude Code skill, we might rely on the user's instructions and model's own policies, but including some safe-guards in the skill logic is wise (maybe a simple confirmation prompt for destructive operations).

Prompt Templates for Common Mobile Automation Tasks

We can define prompt templates that the skill uses internally to instruct the AI (Claude model) how to carry out certain tasks using the tools. For example:

- **Template: Click Element by Text**
  *"To click the `{element_text}` button:
- Call `mobile_list_elements_on_screen` and search for `{element_text}` in the results.
- If found, note its `x,y` coordinates (for example, the center of its bounding box).
- Call `mobile_click_on_screen_at_coordinates` with those coordinates.

- If not found, consider scrolling or that the element might be labeled differently."*
  This template can be filled in whenever the skill has an element to tap.

- **Template: Enter Text into Field**
  *"To enter text `{input}` into the `{field_name}` field:

  - Find the `{field_name}` field via `mobile_list_elements_on_screen` (or if it's already focused, skip to typing).
  - Tap it using `mobile_click_on_screen_at_coordinates`.
  - Call `mobile_type_keys` with `text: "{input}", submit: false` (unless hitting Enter is desired at the end).

    - Verify the text appears (some apps might show the text in the field's label which you can read on next `list_elements`)."*
      This ensures the agent consistently focuses before typing.

- **Template: Navigate Back**
  *"If you need to go back to the previous screen:

    - On Android, use `mobile_press_button` with `"BACK"` [70] .
    - On iOS, look for a UI element like a back arrow or a button labeled "Back" and tap it (since iOS has no back button hardware).
    - If neither is available, as a last resort use `mobile_launch_app` to re-open the main screen of the app (essentially resetting navigation)."*
      This gives the agent a clear strategy for back navigation depending on platform context.

- **Template: Wait for Element**
  *"If you need to wait for something to appear (e.g., after logging in waiting for dashboard):

    - Repeat up to 5 times: call `mobile_list_elements_on_screen`, check if `{expected_element}` is present. If yes, proceed. If not, wait 1 second and try again.
    - If after retries it's still not present, assume the action failed or took too long – consider alternative (maybe check for an error message on screen)."*
      This instructs the agent on implementing waits without a dedicated wait tool.

These templates can be embedded in the skill as part of a system or developer prompt to Claude, guiding its chain-of-thought and actions.

Error Handling and Retry Logic Recommendations

The skill should handle errors gracefully: - If a tool returns an error (Claude Code will surface that via the JSON-RPC response), the skill can parse it. For example, an error with message containing "Required" for `mobile_list_available_devices` likely means no device or parameter issue. The skill can catch that and respond by adding the `{}` parameter and retry automatically (since we know that's the fix) [106] . - If an action produces no visible change, the skill might reattempt it once. e.g., if after clicking, the UI is identical, maybe the click didn't register – perhaps the coordinates were slightly off or the element requires a scroll into view first. The skill could decide to scroll a bit and then retry the click. However, infinite or too many retries should be avoided – likely one retry after adjusting approach is enough before failing. - Use of try-catch blocks (or equivalent in the skill's code if any) for sequence of actions. For instance, a high-level skill function `performLogin()` can catch any failure in sub-steps and either try alternate steps or bubble up a user-friendly error. - Logging context: The skill might keep a small context memory like `last_action` or `last_screen`. If something fails, it could provide that info to the user or next attempt (e.g., "Failed to find 'Play' button on the current screen which has elements: [list of labels]."). This could help user debug or adjust their request ("Oh, my app is in French locale, the button isn't 'Play' but 'Jouer'"). - **Auto-reset**: If an error is unrecoverable (like device disconnected), the skill could attempt a device re-selection or instruct the user to check the device. For example, if any tool returns a device not found error, automatically call `mobile_list_available_devices` again – maybe the device came back with a different ID or took time to connect.

Context Requirements for the Skill

For the skill to be effective, it needs some context: - **Device Context:** which device is in use (including type and maybe OS). The skill should maintain a variable for current device. Every time `mobile_use_device` is called, update this. Then the skill can tailor instructions (like knowing to use

BACK button if Android). - **App Context (optional):** Not strictly necessary, but skill could track the last launched app's package or the last screen known. This might help if the user says "close the app" – skill can call `mobile_terminate_app` with the last launched app's package without user explicitly providing it. Caution: this assumes one app focus at a time. If multi-app flows, the skill should update context as it switches apps. - **Goal or Plan Context:** If the user's request was multi-step (like "do these 3 things"), the skill might break it down and remember which step it's on or what sub-goals remain. This is more in the agent's chain-of-thought, but the skill could hold a list of tasks and feed them one by one to the agent, for instance. - **User Preferences:** Possibly device or app preferences (maybe stored between sessions). E.g., user's device name if always using one specific device, to auto-select it. Or preference like always allow permission prompts. These could be configured in skill settings if applicable. - **Historical mistakes:** The skill can also remember if a certain approach failed earlier in this session. For example, if it already tried using text "OK" to find a button and failed, and later a similar prompt comes, maybe try an alternate approach like screenshot (because perhaps the UI doesn't expose "OK" text). This might be too advanced, but a learning skill could incorporate.

Example Interaction (Skill in action)

Finally, let's illustrate how the skill would handle a complex user request optimally:

User request: "On my Android phone, open the Twitter app, scroll through the feed, like the first tweet you see from John, and then post a reply 'Nice post!'."

Skill/Agent steps (with rationale):
1. Ensure device: The skill sees "my Android phone" – it will call `mobile_list_available_devices`, find the Android device (assuming one available), and call `mobile_use_device` [32] [35]. (If multiple Android devices, might pick one or ask user which one named "Android phone"). 2. Launch Twitter: The skill knows Twitter's package (maybe it has a mapping or uses `list_apps` to find "Twitter"). Calls `mobile_launch_app` with `"com.twitter.android"` [45]. 3. Scroll feed: Once Twitter is open, the feed should show tweets. The skill will likely do: call `mobile_list_elements_on_screen` to see tweets. It finds a tweet from "John" if visible. If John's tweet is not yet visible (maybe further down), it will call `mobile_swipe_on_screen` (direction up) [64], then list elements again, repeat until it finds "John" in the text of a tweet. (Here, "John" might appear as the author name label). 4. Like the first tweet from John: When John's tweet is on screen, identify the "Like" button for that tweet. Possibly the tweet element has a subtree including a Like icon (which might have alt text "Like" or just a heart icon). The skill might try `list_elements` to find something near John's tweet that could be the Like button. This is tricky but let's say mobile-MCP returns each tweet as a container with children including a button with label "Like". The agent finds the first occurrence of John's name, then within a certain proximity, finds a "Like" element. It then calls `mobile_click_on_screen_at_coordinates` on that Like button's coords. 5. Post a reply: Now to reply, the agent might click a "Reply" icon on that tweet (similar strategy: find "Reply" near John's tweet). Or perhaps after liking, it needs to open the tweet to reply. Possibly simpler: tap the tweet to open detail, then find the reply text field. So: - Tap John's tweet container (to open details). - `mobile_list_elements_on_screen` on detail view, find an input field or a prompt like "Tweet your reply". - Tap the input field, then call `mobile_type_keys` with `"Nice post!"` and maybe `submit:true` (if hitting enter should post) [66]. If not, maybe tap a "Send" button afterwards. - Possibly, find and tap the "Send" or "Reply" button via label or icon after typing. 6. Confirm reply posted: The agent might look for the new reply in the thread (maybe see the text "Nice post!" appear, or a confirmation toast "Your reply was sent"). If not sure, it might just conclude success if no error appears. 7. End by maybe going back (press BACK to exit tweet detail or just leave the app open).

Throughout this, the skill's design ensures: - After each major step, it checks that it succeeded (tweet detail opened, etc.). - It handles if John's tweet isn't found: maybe it would scroll a few times then give up with a message "I didn't see a tweet from John in the recent feed." - If any tool error occurred (like app not launching), it would handle or inform.

The **optimal agent behavior** demonstrated: - Using structured data to identify targets (tweet text, button labels). - Only using coordinate taps when it has those structured coordinates. - Adapting to what's on screen (scrolling if needed). - Minimal error – and if error, providing fallback or graceful failure.

By following this blueprint, the Claude Code skill can act as a powerful interface between natural language instructions and the mobile-MCP's capabilities, harnessing all the technical details we've covered to make the agent's actions reliable and intelligent.

(Confidence: High – the skill design is a synthesis of all prior sections, ensuring that the recommendations and patterns proven in practice are built into the skill's logic. By implementing these, one can create a SKILL.md or similar that Claude can use to effectively drive mobile automation.)

Sources: The information and recommendations above were compiled from the mobile-mcp official documentation [125] [28], the project wiki and repository examples [56] [32], community discussions and reports [108] [106], and industry best practices in mobile automation. They reflect the state of mobile-MCP as of early 2026 and real experiences of its user community.

---

[1] [2] [3] [6] [9] [11] [12] [28] [29] [39] [49] [51] [52] [60] [61] [119] [120] [125] GitHub - mobile-next/mobile-mcp: Model Context Protocol Server for Mobile Automation and Scraping (iOS, Android, Emulators, Simulators and Real Devices)
https://github.com/mobile-next/mobile-mcp

[4] [7] [100] [101] What is Model Context Protocol (MCP)? and why you should start using it! | by Fachrizal Oktavian | Medium
https://medium.com/@foktavian/what-is-model-context-protocol-mcp-and-why-you-should-start-using-it-4237141b4470

[5] [10] [21] [73] [74] [75] [80] [82] [94] [95] [96] mobile-mcp - MCP Server Registry | Augment Code
https://www.augmentcode.com/mcp/mobile-mcp

[8] [93] README.md
https://github.com/metorial/metorial/blob/cf421211de57f631885d3264983262c0aa213cd7/catalog/mobile-next/mobile-mcp/mobile-mcp/README.md

[13] [15] [17] [19] [54] [55] [56] [57] [98] [104] [105] Improve clicking accuracy · mobile-next/mobile-mcp Wiki · GitHub
https://github.com/mobile-next/mobile-mcp/wiki/Improve-clicking-accuracy

[14] [16] [18] [20] [91] [92] [99] [112] [113] mobile-next-mobile-mcp.html
https://github.com/mattmerrick/llmlogs/blob/a56dc195e07ea19cfd7d3708353e25b37c629cdb/mcp/mobile-next-mobile-mcp.html

[22] [23] [25] [26] [83] [86] [87] [88] [89] [90] Getting Started with iOS Physical Device · mobile-next/mobile-mcp Wiki · GitHub
https://github.com/mobile-next/mobile-mcp/wiki/Getting-Started-with-iOS-Physical-Device

[24] [84] Getting Started with Android Physical Device · mobile-next/mobile-mcp Wiki · GitHub
https://github.com/mobile-next/mobile-mcp/wiki/Getting-Started-with-Android-Physical-Device

[27] [77] [78] [79] Getting Started with Android Emulator · mobile-next/mobile-mcp Wiki · GitHub
https://github.com/mobile-next/mobile-mcp/wiki/Getting-Started-with-Android-Emulator

[30] [106] MCP error -32602: Invalid arguments for tool x · Issue #89 · mobile-next/mobile-mcp · GitHub
https://github.com/mobile-next/mobile-mcp/issues/89

[31] [32] [33] [34] [35] [36] [37] [38] [40] [41] [42] [43] [44] [45] [46] [47] [48] [50] [53] [58] [59] [62] [63] [64] [65] [66] [67] [68] [69] [70] [71] [72] mobile-mcp.json
https://github.com/pathintegral-institute/mcpm.sh/blob/44601b938e356be84b610593a375ca2087358f3c/mcp-registry/servers/mobile-mcp.json

[76] Mobile App Testing Using Appium MCP Server
https://softwaretestingtrends.com/resources/the-ai-testers-kit/appium-mcp-server

[81] Home · mobile-next/mobile-mcp Wiki · GitHub
https://github.com/mobile-next/mobile-mcp/wiki

[85] Getting Started with iOS Simulators · mobile-next/mobile-mcp Wiki · GitHub
https://github.com/mobile-next/mobile-mcp/wiki/Getting-Started-with-iOS-Simulators

[97] [107] [108] [109] [110] [111] [115] [116] [117] [122] Let AI coding agents control your mobile device to speed up mobile app development : r/mcp
https://www.reddit.com/r/mcp/comments/1qnel25/let_ai_coding_agents_control_your_mobile_device/

[102] [103] [123] [124] The New Era of Cloud AI Mobile Testing: Amazon Device Farm MCP Server Practical Guide | 亚马逊AWS官方博客
https://aws.amazon.com/cn/blogs/china/cloud-ai-mobile-testing-new-era-amazon-device-farm-mcp-server-practical-guide-en/

[114] @mobilenext-pay/mobile-mcp - npm
https://www.npmjs.com/package/@mobilenext-pay/mobile-mcp

[118] Your Phone, Controlled by AI - YouTube
https://www.youtube.com/shorts/FqSL7SUEnP8

[121] Claude AI Agent Orchestrates My 18-Year iOS to Android Exodus
https://www.youtube.com/watch?v=cbWhk69Rgak