Developer Guide

This guide covers development, building, and extending the AI Orchestrator Manager macOS app.

Architecture Overview

SwiftUI App Structure

The app follows the MVVM (Model-View-ViewModel) pattern with observable state management:

```
┌────────────────────────────────────────────────────────────┐
│                    AIOrchestatorManagerApp                 │
│                    (Main Entry Point)                      │
└────────────────────────────────────────────────────────────┘
                               │
        ┌─────────────────────┼─────────────────────┐
        │                     │                     │
        ▼                     ▼                     ▼
┌──────────────┐  ┌──────────────┐  ┌──────────────┐
│ WindowGroup  │  │ MenuBarExtra │  │   Settings   │
│(Main Window) │  │  (Status)    │  │  (Prefs)     │
└──────────────┘  └──────────────┘  └──────────────┘
        │
        ▼
┌────────────────────────────────────────────────┐
│              ContentView                       │
│  ┌────────────────────────────────────────────┐  │
│  │  NavigationSplitView                       │  │
│  │  ┌────────────┐  ┌────────────────────────┐ │  │
│  │  │  Sidebar   │  │    Detail View          │ │  │
│  │  │            │  │    - DashboardView      │ │  │
│  │  │ - Dashboard│  │    - ConfigurationView  │ │  │
│  │  │ - Config   │  │    - ServerMgmtView     │ │  │
│  │  │ - Server   │  │    - LogsViewerView     │ │  │
│  │  │ - Logs     │  │                         │ │  │
│  │  └────────────┘  └────────────────────────┘ │  │
│  └────────────────────────────────────────────┘  │
└────────────────────────────────────────────────┘
```

State Management

```
// Shared state objects
AppState.shared            // UI state, alerts, loading
ServerManager.shared       // Server process management
ConfigurationManager.shared // API keys, settings
InstallationManager.shared  // Installation process
```

Data Flow

1. **User Action** → View triggers action
2. **Manager** → Processes the action asynchronously
3. **@Published Property** → Updates observable state
4. **SwiftUI** → Automatically re-renders affected views

Key Components

InstallationManager

Handles the complete installation process:

```
class InstallationManager {
    func checkSystemRequirements(completion:)
    func installOrchestrator(at path:)
    func cancelInstallation()

    // Steps:
    // 1. Create directory
    // 2. Clone repository (or setup from template)
    // 3. Create Python venv
    // 4. Install pip dependencies
    // 5. Configure environment
    // 6. Setup server
}
```

ServerManager

Manages the MCP server process:

```
class ServerManager {
    func startServer()    // Launch server process
    func stopServer()     // Terminate gracefully
    func restartServer()  // Stop then start

    // Features:
    // - Process output capture
    // - Health monitoring
    // - Auto-restart on crash
    // - Launch agent for auto-start
}
```

ConfigurationManager

Securely manages configuration:

```
class ConfigurationManager {
    func loadAPIConfiguration() -> APIConfiguration
    func saveAPIConfiguration(_, completion:)
    func testAPIConnection(provider:, key:, completion:)

    // Security:
    // - API keys stored in macOS Keychain
    // - Keychain access via Security framework
    // - .env file generation for Python
}
```

Adding New Features

1. Add a New View

```swift
// Sources/Views/MyNewView.swift
import SwiftUI

struct MyNewView: View {
    @EnvironmentObject var appState: AppState

    var body: some View {
        VStack {
            Text("My New Feature")
        }
    }
}
```

2. Add to Navigation

```swift
// In AppState.swift, add to AppTab enum:
enum AppTab {
    // ...
    case myFeature = "My Feature"
}

// In ContentView.swift, add to switch:
case .myFeature:
    MyNewView()

// In SidebarView.swift, add to List:
Label("My Feature", systemImage: "star")
    .tag(AppTab.myFeature)
```

3. Add New Manager

```swift
// Sources/Managers/MyManager.swift
class MyManager: ObservableObject {
    static let shared = MyManager()

    @Published var someState = false

    private init() {}

    func doSomething() {
        DispatchQueue.global(qos: .userInitiated).async {
            // Background work
            DispatchQueue.main.async {
                self.someState = true
            }
        }
    }
}
```

4. Register in App

```
// In AIOrchestratorManagerApp.swift:
@StateObject private var myManager = MyManager.shared

// Pass to views:
.environmentObject(myManager)
```

Error Handling

Use the centralized error handler:

```
// For custom errors:
ErrorHandler.shared.handle(
    AppError.serverError("Connection refused")
)

// For caught errors:
do {
    try someOperation()
} catch {
    ErrorHandler.shared.handle(error, context: "Operation name")
}

// With Result:
result.handleError(context: "Operation name")
```

Testing

Unit Tests

```
import XCTest
@testable import AI_Orchestrator_Manager

class ServerManagerTests: XCTestCase {
    func testServerStart() {
        let manager = ServerManager.shared
        manager.startServer()
        // Wait for startup
        let expectation = expectation(description: "Server started")
        DispatchQueue.main.asyncAfter(deadline: .now() + 5) {
            XCTAssertTrue(manager.isRunning)
            expectation.fulfill()
        }
        wait(for: [expectation], timeout: 10)
    }
}
```

UI Testing

```swift
import XCTest

class UITests: XCTestCase {
    func testNavigateToConfiguration() {
        let app = XCUIApplication()
        app.launch()

        app.outlineRows["Configuration"].click()
        XCTAssert(app.staticTexts["API Configuration"].exists)
    }
}
```

Debugging

View Hierarchy

```swift
// Add to any view for debugging:
.border(.red) // Visual debugging
.onAppear { print("View appeared") }
```

State Inspection

```swift
// In a view:
let _ = print("Current state: \(appState.selectedTab)")
```

Server Logs

Logs are available at:
- In-app: Logs tab
- File: `~/Library/Logs/AIOrchestratorManager.log`
- Server: `~/ai_orchestrator/mcp_server/server.log`

Performance

Best Practices

1. **Use background queues** for heavy operations:
   ```
   swift
     DispatchQueue.global(qos: .userInitiated).async {
         // Heavy work
         DispatchQueue.main.async {
             // Update UI
         }
     }
   ```

2. **Limit log entries** to prevent memory issues:
   ```
   swift
     if logs.count > 1000 {
         logs.removeFirst(logs.count - 1000)
     }
   ```

3. **Use @StateObject for managers**, not @ObservedObject

4. **Avoid expensive computations in body**:
   ```
   swift
      // Good - cached
      var computedValue: String {
          // Called only when dependencies change
      }
   ```

Release Checklist

- [ ] Update version in Info.plist
- [ ] Update CHANGELOG.md
- [ ] Run all tests
- [ ] Build release: `./Scripts/build.sh`
- [ ] Create DMG: `./Scripts/create_dmg.sh`
- [ ] Code sign: `./Scripts/codesign.sh`
- [ ] Notarize: `./Scripts/notarize.sh`
- [ ] Test on clean macOS installation
- [ ] Update documentation