Contents

Swift Deployer App Reference

App Architecture

Overview

The Swift Deployer is a native macOS application for deploying and managing RADIANT infrastructure to AWS.

Requirements: - macOS 13.0 (Ventura) or later - Swift 5.9+ - Xcode 15+

File Structure (36 Files)

```
apps/swift-deployer/
  Package.swift
  Sources/RadiantDeployer/
      RadiantDeployerApp.swift          # App entry point
      AppState.swift                    # Global state management

      Config/
          RadiantConfig.swift           # App configuration

      Models/ (6 files)
          Configuration.swift           # Deployment configuration
          Credentials.swift             # AWS credentials model
          Deployment.swift              # Deployment state model
          DomainConfiguration.swift     # Domain settings
          InstallationParameters.swift  # Install params
```

```
    ManagedApp.swift              # Managed app model

Services/ (21 files)
    AIAssistantService.swift      # AI deployment assistant
    AIRegistryService.swift       # AI model registry
    APIService.swift              # API communication
    AWSService.swift              # AWS SDK wrapper
    AuditLogger.swift             # Audit logging
    CDKService.swift              # CDK deployment
    CredentialService.swift       # Credential management
    DNSService.swift              # DNS configuration
    DatabaseService.swift         # Local SQLite/SQLCipher
    DeploymentLockService.swift   # Deployment locking
    DeploymentService.swift       # Main deployment logic
    GitHubPackageRegistry.swift   # Package downloads
    HealthCheckService.swift      # Health monitoring
    LocalStorageManager.swift     # Encrypted local storage
    MultiRegionService.swift      # Multi-region deployment
    OnePasswordService.swift      # 1Password integration
    PackageService.swift          # Package management
    SeedDataService.swift         # Database seeding
    SnapshotService.swift         # State snapshots
    TimeoutService.swift          # Timeout handling
    VoiceInputService.swift       # Voice commands

Views/ (8+ files)
    ABTestingView.swift           # A/B testing config
    ContentView.swift             # Main content
    DeploymentView.swift          # Deployment UI
    SettingsView.swift            # App settings
    ... (other views)

Components/ (4 files)
    MacOSComponents.swift         # Design tokens & components
    AppCommands.swift             # Menu bar commands
    DataTableComponents.swift     # Table components
    DetailViewComponents.swift    # Detail view patterns
```

Models

Configuration.swift

Deployment configuration model.

```swift
struct DeploymentConfiguration: Codable, Sendable {
    var appId: String
    var environment: Environment
```

```
    var tier: Int
    var region: AWSRegion
    var domain: String?
    var enabledStacks: Set<StackName>
    var customParameters: [String: String]
}

enum Environment: String, Codable, CaseIterable, Sendable {
    case development = "dev"
    case staging = "staging"
    case production = "prod"
}

enum StackName: String, Codable, CaseIterable, Sendable {
    case networking
    case foundation
    case data
    case storage
    case auth
    case ai
    case api
    case admin
    case batch
    case collaboration
    case monitoring
    case security
    case webhooks
    case scheduledTasks
    case multiRegion
}
```

Credentials.swift

AWS credentials model.

```
struct AWSCredentials: Codable, Sendable {
    let accessKeyId: String
    let secretAccessKey: String
    let sessionToken: String?
    let region: String
    let profile: String?
    let expiresAt: Date?

    var isExpired: Bool {
        guard let expiresAt else { return false }
        return Date() > expiresAt
    }
}
```

```swift
struct CredentialProfile: Codable, Identifiable, Sendable {
    let id: UUID
    var name: String
    var credentials: AWSCredentials
    var isDefault: Bool
    var lastUsed: Date?
}
```

Deployment.swift

Deployment state tracking.

```swift
struct Deployment: Codable, Identifiable, Sendable {
    let id: UUID
    var configuration: DeploymentConfiguration
    var status: DeploymentStatus
    var stackStatuses: [StackName: StackStatus]
    var startedAt: Date
    var completedAt: Date?
    var errorMessage: String?
    var outputs: [String: String]
}

enum DeploymentStatus: String, Codable, Sendable {
    case pending
    case preparing
    case deploying
    case verifying
    case completed
    case failed
    case rollingBack
    case cancelled
}

enum StackStatus: String, Codable, Sendable {
    case pending
    case creating
    case updating
    case complete
    case failed
    case rollbackInProgress
    case rollbackComplete
    case deleted
}
```

Services

CDKService.swift

AWS CDK deployment service.

```
actor CDKService {
    private let shell: ShellService
    private let logger: AuditLogger

    // Deploy a single stack
    func deployStack(
        _ stack: StackName,
        config: DeploymentConfiguration,
        credentials: AWSCredentials
    ) async throws -> StackOutput {
        let command = buildCDKCommand(
            action: "deploy",
            stack: stack,
            config: config
        )

        return try await shell.execute(
            command,
            environment: credentials.asEnvironment()
        )
    }

    // Deploy all stacks in dependency order
    func deployAllStacks(
        config: DeploymentConfiguration,
        credentials: AWSCredentials,
        progressHandler: @Sendable (StackName, StackStatus) -> Void
    ) async throws -> DeploymentResult {
        let orderedStacks = topologicalSort(config.enabledStacks)
        var outputs: [StackName: StackOutput] = [:]

        for stack in orderedStacks {
            progressHandler(stack, .creating)
            do {
                outputs[stack] = try await deployStack(stack, config: config, credentials: cred
                progressHandler(stack, .complete)
            } catch {
                progressHandler(stack, .failed)
                throw DeploymentError.stackFailed(stack, error)
            }
        }

        return DeploymentResult(outputs: outputs)
```

```
    }

    // Stack dependency order
    private func topologicalSort(_ stacks: Set<StackName>) -> [StackName] {
        // networking → foundation → data/storage/auth → ai → api/admin → rest
        let order: [StackName] = [
            .networking, .foundation, .data, .storage, .auth,
            .ai, .api, .admin, .batch, .collaboration,
            .monitoring, .security, .webhooks, .scheduledTasks, .multiRegion
        ]
        return order.filter { stacks.contains($0) }
    }
}
```

DeploymentService.swift

Main deployment orchestration.

```
@MainActor
class DeploymentService: ObservableObject {
    @Published var currentDeployment: Deployment?
    @Published var deploymentHistory: [Deployment] = []
    @Published var isDeploying = false

    private let cdkService: CDKService
    private let healthService: HealthCheckService
    private let auditLogger: AuditLogger
    private let localStorage: LocalStorageManager

    func startDeployment(config: DeploymentConfiguration) async throws {
        guard !isDeploying else {
            throw DeploymentError.alreadyInProgress
        }

        isDeploying = true
        let deployment = Deployment(
            id: UUID(),
            configuration: config,
            status: .preparing,
            stackStatuses: [:],
            startedAt: Date()
        )
        currentDeployment = deployment

        do {
            // 1. Validate configuration
            try await validateConfiguration(config)
```

```
            // 2. Acquire deployment lock
            try await acquireLock(config.appId)

            // 3. Deploy stacks
            currentDeployment?.status = .deploying
            let result = try await cdkService.deployAllStacks(
                config: config,
                credentials: try await getCredentials(),
                progressHandler: { [weak self] stack, status in
                    Task { @MainActor in
                        self?.currentDeployment?.stackStatuses[stack] = status
                    }
                }
            )

            // 4. Verify deployment
            currentDeployment?.status = .verifying
            try await healthService.verifyDeployment(result)

            // 5. Complete
            currentDeployment?.status = .completed
            currentDeployment?.completedAt = Date()
            currentDeployment?.outputs = result.flatOutputs

        } catch {
            currentDeployment?.status = .failed
            currentDeployment?.errorMessage = error.localizedDescription
            throw error
        } finally {
            isDeploying = false
            if let deployment = currentDeployment {
                deploymentHistory.append(deployment)
                try? await localStorage.saveDeployment(deployment)
            }
        }
    }

    func rollback(deploymentId: UUID) async throws {
        // Rollback implementation
    }
}
```

AIAssistantService.swift

AI-powered deployment assistant.

```
actor AIAssistantService {
    private let apiService: APIService
```

```
struct AssistantResponse: Sendable {
    let message: String
    let suggestions: [Suggestion]
    let actions: [SuggestedAction]
}

enum SuggestedAction: Sendable {
    case deployStack(StackName)
    case checkHealth
    case viewLogs(String)
    case runDiagnostics
    case contactSupport
}

// Get deployment guidance
func getDeploymentGuidance(
    for config: DeploymentConfiguration,
    currentStatus: DeploymentStatus?
) async throws -> AssistantResponse {
    let prompt = buildGuidancePrompt(config: config, status: currentStatus)
    let response = try await apiService.chat(
        messages: [.init(role: .user, content: prompt)],
        model: "anthropic/claude-3-haiku"
    )
    return parseAssistantResponse(response)
}

// Diagnose deployment error
func diagnoseError(
    error: Error,
    stackName: StackName,
    logs: String
) async throws -> AssistantResponse {
    let prompt = """
    Analyze this AWS CDK deployment error and provide:
    1. Root cause analysis
    2. Specific fix steps
    3. Prevention recommendations

    Stack: \(stackName.rawValue)
    Error: \(error.localizedDescription)
    Logs:
    \(logs.prefix(2000))
    """

    let response = try await apiService.chat(
        messages: [.init(role: .user, content: prompt)],
```

```
            model: "anthropic/claude-3-5-sonnet"
        )
        return parseAssistantResponse(response)
    }
}
```

LocalStorageManager.swift

Encrypted local storage using SQLCipher.

```
actor LocalStorageManager {
    private let db: Connection
    private let encryptionKey: String

    init() throws {
        let path = LocalStorageManager.databasePath
        db = try Connection(path)
        encryptionKey = try KeychainService.getOrCreateDatabaseKey()
        try db.key(encryptionKey)
        try createTablesIfNeeded()
    }

    // Tables
    private func createTablesIfNeeded() throws {
        try db.execute("""
            CREATE TABLE IF NOT EXISTS deployments (
                id TEXT PRIMARY KEY,
                data BLOB NOT NULL,
                created_at INTEGER NOT NULL
            );

            CREATE TABLE IF NOT EXISTS credentials (
                id TEXT PRIMARY KEY,
                profile_name TEXT NOT NULL,
                encrypted_data BLOB NOT NULL,
                is_default INTEGER DEFAULT 0,
                last_used INTEGER
            );

            CREATE TABLE IF NOT EXISTS settings (
                key TEXT PRIMARY KEY,
                value TEXT NOT NULL
            );
        """)
    }

    // Save deployment
    func saveDeployment(_ deployment: Deployment) throws {
```

```
        let data = try JSONEncoder().encode(deployment)
        try db.run("""
            INSERT OR REPLACE INTO deployments (id, data, created_at)
            VALUES (?, ?, ?)
        """, deployment.id.uuidString, data, Date().timeIntervalSince1970)
    }

    // Get deployment history
    func getDeploymentHistory() throws -> [Deployment] {
        let rows = try db.prepare("""
            SELECT data FROM deployments ORDER BY created_at DESC LIMIT 100
        """)
        return try rows.compactMap { row in
            guard let data = row[0] as? Data else { return nil }
            return try JSONDecoder().decode(Deployment.self, from: data)
        }
    }

    // Credential management
    func saveCredentials(_ credentials: CredentialProfile) throws {
        let encrypted = try encrypt(credentials)
        try db.run("""
            INSERT OR REPLACE INTO credentials (id, profile_name, encrypted_data, is_default, l
            VALUES (?, ?, ?, ?, ?)
        """, credentials.id.uuidString, credentials.name, encrypted,
             credentials.isDefault ? 1 : 0, credentials.lastUsed?.timeIntervalSince1970)
    }
}
```

HealthCheckService.swift

Deployment health verification.

```
actor HealthCheckService {
    struct HealthCheckResult: Sendable {
        let service: String
        let status: HealthStatus
        let latencyMs: Int?
        let message: String?
    }

    enum HealthStatus: Sendable {
        case healthy
        case degraded
        case unhealthy
        case unknown
    }
```

```swift
    func verifyDeployment(_ result: DeploymentResult) async throws {
        var checks: [HealthCheckResult] = []

        // Check API Gateway
        if let apiUrl = result.outputs["ApiUrl"] {
            checks.append(await checkEndpoint(apiUrl + "/health", service: "API Gateway"))
        }

        // Check LiteLLM
        if let litellmUrl = result.outputs["LiteLLMUrl"] {
            checks.append(await checkEndpoint(litellmUrl + "/health", service: "LiteLLM"))
        }

        // Check Database
        checks.append(await checkDatabase(result.outputs["DatabaseEndpoint"]))

        // Evaluate results
        let unhealthy = checks.filter { $0.status == .unhealthy }
        if !unhealthy.isEmpty {
            throw HealthCheckError.servicesUnhealthy(unhealthy)
        }
    }

    private func checkEndpoint(_ url: String, service: String) async -> HealthCheckResult {
        let start = Date()
        do {
            let (_, response) = try await URLSession.shared.data(from: URL(string: url)!)
            let httpResponse = response as! HTTPURLResponse
            let latency = Int(Date().timeIntervalSince(start) * 1000)

            return HealthCheckResult(
                service: service,
                status: httpResponse.statusCode == 200 ? .healthy : .degraded,
                latencyMs: latency,
                message: nil
            )
        } catch {
            return HealthCheckResult(
                service: service,
                status: .unhealthy,
                latencyMs: nil,
                message: error.localizedDescription
            )
        }
    }
}
```

Components

MacOSComponents.swift

Design tokens and reusable components.

```swift
// Design Tokens
enum RadiantSpacing {
    static let xxs: CGFloat = 2
    static let xs: CGFloat = 4
    static let sm: CGFloat = 8
    static let md: CGFloat = 12
    static let lg: CGFloat = 16
    static let xl: CGFloat = 24
}

enum RadiantRadius {
    static let sm: CGFloat = 4
    static let md: CGFloat = 8
    static let lg: CGFloat = 12
    static let xl: CGFloat = 16
}

// Status Badge Component
struct StatusBadge: View {
    let status: String
    let color: Color

    var body: some View {
        Text(status)
            .font(.caption)
            .fontWeight(.medium)
            .padding(.horizontal, RadiantSpacing.sm)
            .padding(.vertical, RadiantSpacing.xxs)
            .background(color.opacity(0.15))
            .foregroundColor(color)
            .cornerRadius(RadiantRadius.sm)
    }
}

// Progress Indicator
struct DeploymentProgressView: View {
    let stacks: [StackName]
    let statuses: [StackName: StackStatus]

    var body: some View {
        VStack(alignment: .leading, spacing: RadiantSpacing.sm) {
            ForEach(stacks, id: \.self) { stack in
                HStack {
```

```
                    statusIcon(for: statuses[stack] ?? .pending)
                    Text(stack.rawValue)
                        .font(.system(.body, design: .monospaced))
                    Spacer()
                    StatusBadge(
                        status: (statuses[stack] ?? .pending).rawValue,
                        color: statusColor(statuses[stack] ?? .pending)
                    )
                }
            }
        }
    }
}
```

AppCommands.swift

Menu bar commands with keyboard shortcuts.

```
struct AppCommands: Commands {
    @ObservedObject var appState: AppState

    var body: some Commands {
        CommandGroup(replacing: .newItem) {
            Button("New Deployment") {
                appState.showNewDeploymentSheet = true
            }
            .keyboardShortcut("n", modifiers: .command)

            Button("Import Configuration...") {
                appState.importConfiguration()
            }
            .keyboardShortcut("i", modifiers: [.command, .shift])
        }

        CommandMenu("Deployment") {
            Button("Deploy All Stacks") {
                Task { await appState.deployAll() }
            }
            .keyboardShortcut("d", modifiers: [.command, .shift])
            .disabled(appState.isDeploying)

            Button("Stop Deployment") {
                appState.stopDeployment()
            }
            .keyboardShortcut(".", modifiers: .command)
            .disabled(!appState.isDeploying)

            Divider()
```

```
            Button("View Logs") {
                appState.showLogs = true
            }
            .keyboardShortcut("l", modifiers: [.command, .option])

            Button("Run Health Check") {
                Task { await appState.runHealthCheck() }
            }
            .keyboardShortcut("h", modifiers: [.command, .shift])
        }

        CommandMenu("AWS") {
            Button("Switch Profile...") {
                appState.showProfileSwitcher = true
            }
            .keyboardShortcut("p", modifiers: [.command, .option])

            Button("Refresh Credentials") {
                Task { await appState.refreshCredentials() }
            }
            .keyboardShortcut("r", modifiers: [.command, .shift])
        }
    }
}
```

UI Patterns (10 macOS Patterns)

The Swift Deployer follows these macOS design patterns:

1. **NavigationSplitView** - Sidebar + Content + Inspector
2. **Liquid Glass** - On navigation/controls only
3. **Toolbar-as-Command-Center** - Grouped actions + overflow
4. **Scroll Edge Effects** - Floating UI legibility
5. **Master List → Detail** - 3-level navigation
6. **Search as First-Class** - Toolbar trailing position
7. **Tables for Data** - Lists for collections
8. **Multi-Select + Context Menus** - Drag & drop support
9. **Full Menu Bar** - Keyboard shortcuts
10. **Settings Window + Inspectors** - macOS-native patterns