CodeKata Production Troubleshooting Guide

Based on research and real-world iOS development experience, here are the critical troubleshooting areas when scaling from MVP to production:

🚨 Phase 1: Data Foundation Issues (Weeks 1-4)

SwiftData Integration Problems

Issue: "No such module 'SwiftData'"

Symptoms:

```swift
import SwiftData // ❌ Compilation error
```

Root Causes:

- iOS deployment target < 17.0
- Using iOS Simulator < 17.0
- Xcode version < 15.0

Solutions:

```swift
```

```
// ✅ Check deployment target in project settings
// Target → General → Minimum Deployments: iOS 17.0

// ✅ Fallback for older iOS versions
#if canImport(SwiftData)
import SwiftData
#else
import CoreData
#endif
```

Issue: Model Container Crashes

Symptoms:

```
Fatal error: Could not create ModelContainer
```

Root Causes:

- Invalid model relationships
- Schema migration failures
- CloudKit configuration conflicts

Solutions:

```swift
```

```
// ✅ Robust container setup with error handling
var sharedModelContainer: ModelContainer = {
    let schema = Schema([
        Challenge.self,
        UserProgress.self
    ])

    let modelConfiguration = ModelConfiguration(
        schema: schema,
        isStoredInMemoryOnly: false // Set to true for testing
    )

    do {
        let container = try ModelContainer(for: schema, configurations: [modelConfiguration])
        print("✅ ModelContainer created successfully")
        return container
    } catch {
        print("❌ ModelContainer error: \(error)")
        // Fallback to in-memory container
        let fallbackConfig = ModelConfiguration(
            schema: schema,
            isStoredInMemoryOnly: true
        )
        return try! ModelContainer(for: schema, configurations: [fallbackConfig])
    }
}()
```

Issue: Data Not Persisting

Symptoms:

- Challenges disappear on app restart

- User progress resets
- Sample data keeps re-adding

Root Causes:

- Missing `modelContext.save()`
- Context not injected properly
- In-memory configuration accidentally enabled

Solutions:

```swift
```

```
// ✅ Proper data insertion pattern
func addChallenge(_ challenge: Challenge) {
    modelContext.insert(challenge)

    do {
        try modelContext.save()
        print("✅ Challenge saved successfully")
    } catch {
        print("❌ Save failed: \(error)")
        // Handle error - show user feedback
    }
}

// ✅ Check if data already exists before adding samples
private func addSampleDataIfNeeded() {
    let descriptor = FetchDescriptor<Challenge>()
    let existingChallenges = (try? modelContext.fetch(descriptor)) ?? []

    if existingChallenges.isEmpty {
        Challenge.sampleData.forEach { modelContext.insert($0) }
        try? modelContext.save()
    }
}
```

Memory Management Issues

Issue: Memory Leaks with @StateObject

Symptoms:

- App memory usage grows continuously
- Slow performance after extended use

- Crash reports showing memory pressure

Root Causes:

- Retain cycles in closures
- Missing `weak self` references
- ObservableObject not properly released

Solutions:

```swift
```

```
// ❌ Creates retain cycle
class ChallengeViewModel: ObservableObject {
    func loadData() {
        networkService.fetch { result in
            self.challenges = result // Retain cycle
        }
    }
}

// ✅ Proper memory management
class ChallengeViewModel: ObservableObject {
    func loadData() {
        networkService.fetch { [weak self] result in
            guard let self = self else { return }
            DispatchQueue.main.async {
                self.challenges = result
            }
        }
    }

    deinit {
        print("✅ ChallengeViewModel deallocated")
    }
}
```

🔥 Phase 2: Performance & Scaling Issues (Weeks 5-8)

SwiftUI Performance Problems

Issue: List Performance Degradation

Symptoms:

- Scrolling becomes choppy with 50+ challenges
- UI freezes during data loading
- High CPU usage in Instruments

Root Causes:

- Complex view hierarchies in list rows
- Missing `LazyVStack` for large datasets
- Expensive operations on main thread

Solutions:

```swift
```

```swift
// ❌ Performance killer
List(challenges) { challenge in
    VStack {
        // Complex nested views
        ForEach(challenge.tags, id: \.self) { tag in
            // Expensive operations
            ComplexTagView(tag: tag)
        }
        // More complex views...
    }
}

// ✅ Optimized list performance
LazyVStack(spacing: 8) {
    ForEach(challenges) { challenge in
        ChallengeRowView(challenge: challenge)
            .id(challenge.id) // Stable identity
    }
}
.onAppear {
    // Load only visible items
}

// ✅ Optimized row view
struct ChallengeRowView: View {
    let challenge: Challenge

    var body: some View {
        HStack {
            // Simple, flat hierarchy
            Text(challenge.title)
                .lineLimit(1)
            Spacer()
            Text(challenge.difficulty.emoji)
        }
        .padding(.vertical, 4)
    }
}
```

Issue: State Management Chaos

Symptoms:

- Views not updating when data changes
- Multiple sources of truth conflicts
- Random UI glitches

Root Causes:

- Mixed @State, @StateObject, @ObservedObject usage
- Passing state down too many levels
- Environment values not properly set

Solutions:

swift

```
// ❌ State management confusion
struct ContentView: View {
    @State private var challenges: [Challenge] = []
    @StateObject private var viewModel = ChallengeViewModel()
    @ObservedObject var userProgress: UserProgress // Wrong usage

    var body: some View {
        // Multiple competing state sources
    }
}

// ✅ Clear state hierarchy
struct ContentView: View {
    @StateObject private var challengeStore = ChallengeStore()

    var body: some View {
        NavigationView {
            ChallengeListView()
        }
        .environmentObject(challengeStore) // Single source of truth
    }
}

struct ChallengeListView: View {
    @EnvironmentObject var challengeStore: ChallengeStore

    var body: some View {
        List(challengeStore.challenges) { challenge in
            // Clean dependency injection
        }
    }
}
```

Data Consistency Issues

Issue: Challenge Completion State Desync

Symptoms:

- Challenges show as incomplete after solving
- Progress not updating across views
- Duplicate completion records

Root Causes:

- Multiple model contexts
- Race conditions in async operations
- Missing data validation

Solutions:

```
swift
```

```swift
// ✅ Atomic completion operation
@MainActor
func completeChallenge(_ challenge: Challenge, score: Double) async {
    // Use single transaction
    let context = modelContext

    // Check if already completed
    guard !challenge.isCompleted else {
        print("⚠️ Challenge already completed")
        return
    }

    // Update challenge
    challenge.isCompleted = true
    challenge.completionDate = Date()

    // Update user progress
    if let userProgress = getCurrentUserProgress() {
        userProgress.totalScore += Int(score)
        userProgress.completedChallenges += 1
        userProgress.updateStreak()
    }

    // Save everything atomically
    do {
        try context.save()
        print("✅ Challenge completion saved")

        // Notify other parts of app
        NotificationCenter.default.post(
            name: .challengeCompleted,
            object: challenge
        )
    } catch {
        print("❌ Failed to save completion: \(error)")
        // Rollback changes
        context.rollback()
    }
}
```

⚡ Phase 3: Production & Scale Issues (Weeks 9-12)

Network & Server Integration Problems

Issue: Code Execution Timeouts

Symptoms:

- Solutions never return results
- App hangs on submission
- Random execution failures

Root Causes:

- Server overload
- Network connectivity issues
- Infinite loops in user code
- Missing timeout handling

Solutions:

```
swift
```

```swift
// ✅ Robust execution service with timeouts
class CodeExecutionService: ObservableObject {
    private let session = URLSession.shared
    private let timeout: TimeInterval = 10.0

    func executeCode(_ code: String, testCases: [TestCase]) async -> ExecutionResult {
        do {
            let result = try await withTimeout(seconds: timeout) {
                try await self.performExecution(code, testCases: testCases)
            }
            return result
        } catch is TimeoutError {
            return ExecutionResult.timeout
        } catch {
            return ExecutionResult.networkError(error)
        }
    }

    private func performExecution(_ code: String, testCases: [TestCase]) async throws -> ExecutionResult {
        let request = ExecutionRequest(code: code, testCases: testCases)
        let (data, response) = try await session.data(for: request.urlRequest)

        guard let httpResponse = response as? HTTPURLResponse,
              httpResponse.statusCode == 200 else {
            throw ExecutionError.serverError
        }

        return try JSONDecoder().decode(ExecutionResult.self, from: data)
    }
}

// ✅ Timeout utility
func withTimeout<T>(seconds: TimeInterval, operation: @escaping () async throws -> T) async throws -> T
```

```
  try await withThrowingTaskGroup(of: T.self) { group in
    group.addTask {
      try await operation()
    }

    group.addTask {
      try await Task.sleep(nanoseconds: UInt64(seconds * 1_000_000_000))
      throw TimeoutError()
    }

    guard let result = try await group.next() else {
      throw TimeoutError()
    }

    group.cancelAll()
    return result
  }
}
```

Issue: CloudKit Sync Conflicts

Symptoms:

- Progress lost between devices
- Duplicate challenges appearing
- Sync failures with cryptic errors

Root Causes:

- Record ID conflicts
- Schema mismatches

- Network connectivity during sync
- User account changes

Solutions:

```swift
```

```swift
// ✅ Robust CloudKit sync handling
class CloudKitSyncManager: ObservableObject {
    @Published var syncStatus: SyncStatus = .idle

    private let container = CKContainer.default()
    private let database: CKDatabase

    init() {
        self.database = container.privateCloudDatabase
    }

    @MainActor
    func syncUserProgress() async {
        syncStatus = .syncing

        do {
            // Check account status first
            let accountStatus = try await container.accountStatus()
            guard accountStatus == .available else {
                syncStatus = .accountError
                return
            }

            // Perform sync with conflict resolution
            try await performSyncWithConflictResolution()

            syncStatus = .completed
        } catch {
            print("❌ Sync failed: \(error)")
            syncStatus = .error(error)

            // Implement exponential backoff retry
            await scheduleRetry()
        }
    }

    private func performSyncWithConflictResolution() async throws {
        // Fetch remote changes
        let remoteRecords = try await fetchRemoteChanges()

        // Merge with local data
        for record in remoteRecords {
            try await mergeRecord(record)
        }

        // Push local changes
        try await pushLocalChanges()
    }

    private func mergeRecord(_ record: CKRecord) async throws {
        // Implement last-writer-wins or custom merge logic
        let localRecord = await findLocalRecord(id: record.recordID)

        if let local = localRecord {
            // Conflict resolution
            let merged = resolveConflict(local: local, remote: record)
            await saveLocalRecord(merged)
        } else {
            // New record from remote
            await createLocalRecord(from: record)
        }
    }
}

enum SyncStatus {
    case idle
    case syncing
    case completed
    case accountError
    case error(Error)

    var displayMessage: String {
        switch self {
        case .idle: return "Ready to sync"
        case .syncing: return "Syncing progress..."
        case .completed: return "Sync completed"
        case .accountError: return "Please sign in to iCloud"
        case .error(let error): return "Sync failed: \(error.localizedDescription)"
        }
    }
}
```

Security & Validation Issues

Issue: Code Injection Attacks

Symptoms:

- Server crashes from malicious code
- Unauthorized system access attempts
- Resource exhaustion attacks

Root Causes:

- Insufficient input sanitization
- Missing execution sandboxing
- No resource limits

Solutions:

```
swift
```

```swift
// ✅ Code sanitization before execution
class CodeValidator {
    private let dangerousPatterns = [
        "import os",
        "subprocess",
        "eval(",
        "exec(",
        "__import__",
        "open(",
        "file(",
        "input(",
        "raw_input"
    ]

    private let maxCodeLength = 10000
    private let maxLines = 500

    func validateCode(_ code: String) throws {
        // Length checks
        guard code.count <= maxCodeLength else {
            throw ValidationError.codeTooLong
        }

        let lines = code.components(separatedBy: .newlines)
        guard lines.count <= maxLines else {
            throw ValidationError.tooManyLines
        }

        // Pattern matching for dangerous code
        for pattern in dangerousPatterns {
            if code.lowercased().contains(pattern.lowercased()) {
                throw ValidationError.dangerousPattern(pattern)
            }
        }

        // Basic syntax validation
        try validateSyntax(code)
    }

    private func validateSyntax(_ code: String) throws {
        // Language-specific syntax validation
        // This is a simplified version
        let balancedBraces = isBalanced(code, pairs: [("{", "}"), ("(", ")"), ("[", "]")])
        guard balancedBraces else {
            throw ValidationError.unbalancedBraces
        }
    }
}

enum ValidationError: Error, LocalizedError {
    case codeTooLong
    case tooManyLines
    case dangerousPattern(String)
    case unbalancedBraces

    var errorDescription: String? {
        switch self {
        case .codeTooLong:
            return "Code exceeds maximum length limit"
        case .tooManyLines:
            return "Code exceeds maximum line limit"
        case .dangerousPattern(let pattern):
            return "Code contains dangerous pattern: \(pattern)"
        case .unbalancedBraces:
            return "Code has unbalanced braces or parentheses"
        }
```

```
  }
}
```

🛠️ General Debugging Strategies

Logging & Monitoring Setup

```
swift
```

```swift
// ✅ Comprehensive logging system
import os.log

extension Logger {
    private static var subsystem = Bundle.main.bundleIdentifier!

    static let challenges = Logger(subsystem: subsystem, category: "challenges")
    static let network = Logger(subsystem: subsystem, category: "network")
    static let data = Logger(subsystem: subsystem, category: "data")
    static let performance = Logger(subsystem: subsystem, category: "performance")
}

// Usage throughout app
func loadChallenges() async {
    Logger.challenges.info("Loading challenges started")

    let startTime = CFAbsoluteTimeGetCurrent()
    defer {
        let timeElapsed = CFAbsoluteTimeGetCurrent() - startTime
        Logger.performance.info("Challenge loading took \(timeElapsed) seconds")
    }

    do {
        let challenges = try await challengeService.fetchChallenges()
        Logger.challenges.info("Successfully loaded \(challenges.count) challenges")
    } catch {
        Logger.challenges.error("Failed to load challenges: \(error.localizedDescription)")
    }
}
```

Production Monitoring

swift

```swift
// ✅ Crash reporting and analytics setup
class AnalyticsManager {
    static let shared = AnalyticsManager()

    func trackChallengeStarted(_ challenge: Challenge) {
        // Track user engagement
        let properties = [
            "challenge_id": challenge.id,
            "difficulty": challenge.difficulty.rawValue,
            "category": challenge.category
        ]

        // Send to analytics service
        Analytics.track("challenge_started", properties: properties)
    }

    func trackError(_ error: Error, context: String) {
        let errorInfo = [
            "error": error.localizedDescription,
            "context": context,
            "timestamp": ISO8601DateFormatter().string(from: Date())
        ]

        // Send to crash reporting service
        CrashReporting.recordError(error, userInfo: errorInfo)
    }
}
```

🔍 Testing Strategy for Each Phase

Phase 1: Data Foundation Testing

swift

```swift
// ✅ SwiftData integration tests
class DataPersistenceTests: XCTestCase {
  var container: ModelContainer!
  var context: ModelContext!

  override func setUp() {
    let schema = Schema([Challenge.self])
    let config = ModelConfiguration(isStoredInMemoryOnly: true)
    container = try! ModelContainer(for: schema, configurations: [config])
    context = ModelContext(container)
  }

  func testChallengePeristence() {
    // Given
    let challenge = Challenge(title: "Test", description: "Test challenge")

    // When
    context.insert(challenge)
    try! context.save()

    // Then
    let descriptor = FetchDescriptor<Challenge>()
    let challenges = try! context.fetch(descriptor)
    XCTAssertEqual(challenges.count, 1)
    XCTAssertEqual(challenges.first?.title, "Test")
  }
}
```

Phase 2: Performance Testing

```swift
// ✅ UI performance tests
class PerformanceTests: XCTestCase {
  func testChallengeListScrollPerformance() {
    measure {
      // Test scrolling through 100+ challenges
      let challenges = (1...100).map { Challenge.sample(id: $0) }
      // Measure rendering time
    }
  }
}
```

Phase 3: Integration Testing

```swift
// ✅ Network integration tests
class NetworkIntegrationTests: XCTestCase {
  func testCodeExecutionEndToEnd() async {
    let expectation = XCTestExpectation(description: "Code execution")

    let service = CodeExecutionService()
    let result = await service.executeCode("print('hello')", testCases: [])

    XCTAssertNotNil(result)
    expectation.fulfill()

    await fulfillment(of: [expectation], timeout: 15.0)
  }
}
```

📋 Production Readiness Checklist

Pre-Launch Essentials

- [ ] **Crash reporting** integrated (Crashlytics/Sentry)
- [ ] **Analytics tracking** for key user actions
- [ ] **Error handling** for all network operations
- [ ] **Offline mode** graceful degradation
- [ ] **Performance profiling** completed
- [ ] **Memory leak testing** passed
- [ ] **Device compatibility** tested (iPhone/iPad)
- [ ] **iOS version compatibility** verified
- [ ] **App Store guidelines** compliance checked
- [ ] **Privacy policy** and data handling documented

Monitoring & Alerts

- [ ] **Server uptime monitoring**
- [ ] **API response time alerts**
- [ ] **Crash rate monitoring** (< 1%)
- [ ] **User retention metrics** tracking
- [ ] **App Store review monitoring**

This troubleshooting guide should help you anticipate and solve the most common issues when scaling CodeKata from MVP to production. Each phase has its own typical problems, and being prepared for them will save significant development time.