FocusApp — Critical Issues & Solutions

Development Summary for Interview Preparation

1. App Building Process

CodeMirror Editor on iOS — WKWebView Integration

Issue: iOS can't use `NSTextView` (macOS-only). We needed a full-featured code editor with syntax highlighting, line numbers, and error diagnostics on iOS.

Solution: Built a custom `CodeMirrorEditorView` using `WKWebView` that loads a bundled HTML/JS/CSS CodeMirror editor. Communication between Swift and JavaScript uses `evaluateJavaScript()` for Swift-to-JS (updating code, setting language, pushing diagnostics) and `WKScriptMessageHandler` for JS-to-Swift (capturing user edits). Error diagnostics are serialized to JSON and pushed to the JS layer which renders red gutter markers and background highlights.

Data Journey Visualization — Scope-Aware Instrumentation

Issue: The `AutoInstrumenter` injects `Trace.step()` calls into user code for step-by-step visualization. Early versions captured all variables at every step, causing "use of local variable before its declaration" compiler errors.

Solution: Implemented scope-aware variable capture — `extractFunctionLevelDecls()` tracks which variables are declared at each line, and only captures variables declared *before* each insertion point. Loop binding variables (e.g., `count` from `for (num, count) in ...`) are scoped to their containing loop only, preventing cross-scope leaking.

Solution Editorial System — Lazy Loading for Performance

Issue: Loading all 200+ solutions from a single `Solutions.json` on startup caused lag.

Solution: Implemented `TopicSolutionStore` with 17 topic-partitioned JSON files plus an `index.json` manifest. Solutions load lazily per-topic on first access, with thread-safe `NSLock` protection. The store conforms to `SolutionProviding` protocol, making it a drop-in replacement.

WidgetKit Extension — Missing NSExtension Dictionary

Issue: Widget extension built fine but crashed on simulator installation with "extensionDictionary must be set in placeholder attributes." The auto-generated Info.plist was missing the `NSExtension` dictionary.

Solution: Created a manual `FocusWidget/Info.plist` with `NSExtensionPointIdentifier = "com.apple.widgetkit-extension"` and set `INFOPLIST_FILE` in the widget target's build settings. Xcode merges this manual plist with auto-generated `INFOPLIST_KEY_*` settings.

2. Data Verification, Synchronization & Data System

LeetCode Sync — Slug Matching Across Plans

Issue: The app's study plan uses problem URLs, but LeetCode's API returns problem slugs (e.g., "reverse-linked-list"). Matching these reliably was tricky since URLs and slugs have different formats.

Solution: Built `LeetCodeSlugExtractor` that normalizes both URL paths and API slugs to a common format, then `applySolvedSlugs()` does set intersection to find matches. The sync reports both `syncedCount` (newly found) and `totalMatched` (all matches).

Hidden Test Case Validation — orderMatters Flag

Issue: Some LeetCode problems say "return in any order", but our test comparison was doing exact string matching, causing false failures.

Solution: Added an `orderMatters: Bool` flag to `SolutionTestCase`. The AI prompt explicitly asks providers to set `orderMatters: false` for "return in any order" problems. `outputMatches()` tries exact match first, then falls back to sorted JSON array comparison when `orderMatters: false`. Uses `sortedJSONArray()` helper that only sorts flat arrays (no nested objects).

SwiftData Persistence — Date Format Pitfalls

Issue: Swift's `.iso8601` date decoding strategy doesn't handle microsecond precision (6 fractional digits), causing JSON decode failures.

Solution: Standardized all dates in JSON to use simple ISO 8601 format: `2026-02-06T08:40:26Z` or with milliseconds `2026-02-06T08:40:26.092Z`. Never use microsecond precision with Swift's built-in decoder.

Widget Data Sharing — App Group Container

Issue: The WidgetKit extension runs in a separate process and can't access the main app's SwiftData store directly.

Solution: Used an App Group (`group.com.dsafocus.focusapp`) with a shared JSON file. `WidgetDataWriter` computes a simplified `WidgetData` model from `AppData` and writes `widget-data.json` to the shared container on every `save()`. The widget reads via `WidgetDataReader.load()` and `WidgetCenter.shared.reloadAllTimelines()` triggers refresh.

3. Network Issues & Robustness

iOS Code Execution — LeetCode API with Retry Logic

Issue: iOS can't use `Process()` for local code execution. We needed a reliable remote execution path, but LeetCode's API can be flaky (rate limits, timeouts, transient failures).

Solution: Built `LeetCodeExecutionService` conforming to the shared `CodeExecuting` protocol. It uses 3 retries with exponential backoff (1s, 2s, 4s delays) and a 20-second timeout per request. The two-phase flow: POST to `/problems/{slug}/interpret_solution/` then poll `/submissions/detail/{id}/check/` with `finished` field detection.

LeetCode Submission Polling — Completion Detection

Issue: The `LeetCodeSubmissionService` initially checked `state == "SUCCESS"` for completion, but LeetCode sometimes returns other state strings for completed submissions.

Solution: Changed to check the `finished` field first (`finished == true`), with fallback to `state.uppercased() == "SUCCESS"`. The `isComplete` property now uses a two-tier check for robustness.

Username Validation — Network-First Verification

Issue: Users could enter invalid LeetCode usernames, causing all subsequent syncs to fail silently.

Solution: Before saving a username, `validateUsername()` makes a real API call to LeetCode to verify the profile exists. Visual feedback: green border + "Valid" on success, red border + "User not found" on failure. Validation resets when the user starts typing again.

---

4. Application Design

Clean Architecture / VIPER Pattern

Issue: Early code had business logic mixed into views, making testing difficult and creating tight coupling.

Solution: Adopted a Clean Architecture pattern: View -> Presenter (observable state, `@MainActor`) -> Interactor (business logic) -> AppStateStore (centralized data). Each feature has its own Presenter and Interactor. `AppContainer` handles dependency injection. This makes every layer independently testable (600+ tests).

Cross-Platform Code Sharing — Feature-Folder Structure

Issue: macOS and iOS have different UI frameworks (`NSTextView` vs `WKWebView`, `NSPanel` vs `WidgetKit`), but share 80% of business logic. Initially, iOS views were in a flat folder, disconnected from their macOS counterparts.

Solution: Feature-folder structure where each feature (Today, Plan, Stats, CodingEnvironment, etc.) has shared Presenter/Interactor at the root, with `macOS/` and `iOS/` subfolders

for platform-specific views. All files are in both targets with `#if os()` guards. iOS types follow `ViewNameiOS` naming convention.

Dependency Injection — Platform-Specific Services

Issue: The same `CodeExecuting` protocol needed completely different implementations on each platform (local `Process` on macOS, LeetCode API on iOS).

Solution: `AppContainer.swift` uses `#if os(iOS)` guards to wire `LeetCodeExecutionService` on iOS and `CodeExecutionService` on macOS. Both conform to the same `CodeExecuting` protocol, so presenters and interactors are platform-agnostic.

---

5. Optimization

Code Wrapping Bypass on iOS

Issue: `wrappedCodeForExecution()` adds a local execution harness (imports, `Trace` struct, runner function) that's necessary for macOS local execution but breaks LeetCode's API since it has its own harness.

Solution: Added `#if os(iOS) return source #else ... #endif` in `wrappedCodeForExecution()` — iOS returns raw source code, macOS adds the full wrapper. Clean and zero-cost.

Lazy Solution Loading — Topic Partitioning

Issue: Loading 200+ solutions from a single JSON file was slow and used unnecessary memory.

Solution: `partition_solutions.swift` script splits `Solutions.json` into 17 topic files. `TopicSolutionStore` loads only the topic file needed, using `NSLock` for thread safety. Solutions are cached after first load.

Hidden Test Gate — Relaxed Success Criteria

Issue: `ExecutionResult.isSuccess` requires `error.isEmpty`, which catches harmless stderr warnings (Swift deprecation notices, Python warnings) as failures.

Solution: For hidden test execution, use the relaxed check `exitCode == 0 && !timedOut && !wasCancelled` instead of `isSuccess`. This prevents false failures from stderr warnings while still catching real errors.

---

6. Miscellaneous but Critical

macOS Entitlements vs iOS Entitlements

Issue: Adding App Group entitlement to `FocusApp.entitlements` broke the macOS build because App Groups require a provisioning profile on macOS, but the app uses no code signing for personal use.

Solution: Created separate entitlements files — `FocusApp.entitlements` (macOS, network client only) and `FocusAppiOS.entitlements` (iOS, network client + App Group). Updated the iOS target's build settings to reference the iOS-specific file.

Xcode Bundle Resources — Flattened Paths

Issue: Using `Bundle.url(forResource:withExtension:subdirectory:)` with a subdirectory parameter failed because Xcode's "Copy Bundle Resources" flattens files into `Contents/Resources/` root.

Solution: Never use `subdirectory:` parameter for resources added via "Copy Bundle Resources". Use `subdirectory: nil` (default) unless using blue folder references.

NSRegularExpression Escaping in Swift

Issue: Regex patterns for signature parsing had double-escaping bugs — `"func\\\\s+"` in Swift compiles to literal `func\\s+` instead of the intended `func\s+`.

Solution: Use single backslash escaping in NSRegularExpression patterns: `"func\\s+"` becomes the regex `func\s+`. Also added `NSNotFound` guards for optional capture groups.

SwiftLint Type Body Length — Extension Splitting

Issue: Large types (presenters, views) hit the 500-line SwiftLint error threshold.

Solution: Split logic into extensions in separate files (e.g., `CodingEnvironmentPresenter+Execution.swift`). Each extension file stays under limits. Important: `@Environment` properties must be `internal` (not `private`) when accessed from extensions in separate files.

---

7. Guardrails for Scalability, Verification & Validation

Protocol-Driven Architecture — Swap Without Breaking

Approach: Every major service conforms to a protocol (`CodeExecuting`, `SolutionProviding`, `TestCaseAIProviding`, `RequestExecuting`). This means we can swap implementations without touching consumers. For example, `CodeExecuting` has three implementations: `CodeExecutionService` (macOS local), `LeetCodeExecutionService` (iOS API), and `NoOpCodeExecutionService` (fallback/testing). Adding a fourth would just mean writing a new conformance.

Why it matters: When LeetCode's API changes or we want to add a new language executor, we change one file, not twenty. The protocol boundary acts as a contract that prevents silent breakage.

600+ Unit Tests Across Every Layer

Approach: Tests are organized by feature folder mirroring the source structure. Every Presenter, Interactor, and business logic module has dedicated tests. We use

`InMemoryAppStorage` (a test-only `AppStorageProtocol` conformance) so tests never touch disk or network.

Guardrail: After every workstream or bug fix, we run the full test suite. The rule is: "if tests pass, ship it." Pre-existing expected failures are marked so they don't block real regressions.

Multi-Target Build Verification

Approach: Every change is verified against three targets: macOS, iOS, and the WidgetKit extension. A change that builds on macOS but breaks iOS gets caught immediately.

Example: This caught the entitlements conflict (App Group breaking macOS) and the code wrapping issue (local harness breaking LeetCode API).

SwiftLint as Automated Code Quality Gate

Approach: `.swiftlint.yml` enforces hard limits — 200 chars line length, 100 lines function body, 500 lines type body, cyclomatic complexity of 25. Custom rules block `print()` statements, hardcoded colors, and missing `@MainActor` on Presenters.

Why it scales: When a Presenter grows past 500 lines, SwiftLint forces you to split into extensions before the code becomes unmanageable. It's architectural enforcement, not just style.

Hidden Test Gate — Pre-Submission Validation

Approach: Before any code hits LeetCode's official submit endpoint, it must pass up to 50 AI-generated hidden test cases locally. This catches edge cases (empty arrays, single elements, maximum constraints) that the user's manual test cases might miss.

Multi-layer defense: User-visible tests -> Hidden AI tests -> LeetCode official submission. Code never reaches LeetCode unless it passes everything locally first.

Type-Safe Data Contracts Between Processes

Approach: The widget extension and main app share data via `WidgetData` — a `Codable` struct with explicit fields and default values. The writer and reader use the same model, so schema mismatches cause compile-time errors, not runtime crashes.

Defensive coding: `WidgetDataReader.load()` returns sensible defaults if the JSON file is missing, corrupt, or from an older schema version. The widget never crashes.

Retry + Timeout Boundaries on All Network Calls

Approach: Every network call has explicit timeout (20s) and retry logic (3 retries with exponential backoff). Network errors are surfaced with user-friendly messages, not raw HTTP errors.

Why this matters: If LeetCode adds rate limiting or changes response times, our retry logic absorbs it gracefully.

---

8. Where AI Assistance Hit Its Limits

AI Could Not Always Get project.pbxproj Right

Reality: Xcode's project file is a deeply nested, ID-referenced format where one wrong character breaks the entire project. AI-generated pbxproj edits frequently had incorrect UUIDs, missing section references, or build phase entries pointing to non-existent files.

What we did: Carefully hand-verified every pbxproj edit. When the widget target was first created, it compiled fine but failed at install time because the NSExtension plist entry was missing — something the AI didn't catch because the build succeeded.

Lesson: For pbxproj changes, always verify by building AND installing/running. A successful build doesn't mean the binary is correctly configured.

AI Generated Code That Compiled But Had Subtle Runtime Bugs

Reality: The AI would sometimes generate code that compiled perfectly but had logic issues. For example, the initial `wrappedCodeForExecution()` applied the macOS local harness on iOS too — it compiled because the harness code was valid Swift, but at runtime, LeetCode's API rejected it.

What we did: Added platform-specific `#if os()` guards and tested the actual execution flow end-to-end, not just compilation.

Lesson: AI is great at generating syntactically correct code, but you need to understand the runtime context. Always ask: "What will the other side (API, OS, framework) actually do with this?"

AI Struggled with Cross-Cutting Concerns Across Many Files

Reality: When restructuring 78+ files into feature folders, the AI sometimes moved files but forgot to update all the corresponding pbxproj paths, or moved a file into `iOS/` but didn't add the `#if os(iOS)` guard.

What we did: Adopted a systematic approach — do all renames in one pass, build immediately after each batch of changes, fix all errors before proceeding.

Lesson: For large refactors, break the work into small, verifiable chunks. Don't try to do 78 file moves in one shot. Move 10, build, fix, repeat.

AI Couldn't Predict Platform-Specific Xcode Behaviors

Reality: Several issues were specific to how Xcode handles things behind the scenes: - App Group entitlement requiring provisioning profiles on macOS but not on iOS - `GENERATE_INFOPLIST_FILE` not generating the NSExtension dictionary without an explicit `INFOPLIST_FILE` pointer - Bundle resources being flattened (no subdirectories) in "Copy

Bundle Resources" - `@Environment` properties needing `internal` access when extensions are in separate files

What we did: Each of these was discovered through build/run failures, then we researched the specific Xcode behavior, documented it in `MEMORY.md` for future reference.

Lesson: AI works from training data, but Xcode's build system has undocumented behaviors and version-specific quirks. When you hit a wall, read Apple's actual build logs and inspect the generated artifacts. Document every gotcha.

AI Test Generation Had Quality Variance

Reality: The AI providers generating hidden test cases sometimes produced incorrect expected outputs, especially for complex graph/tree problems.

What we did: Added the `orderMatters` flag to handle ambiguous output ordering. Failed hidden tests go to the test panel for inspection, so users can see what failed and judge if it's a real bug or a bad test.

Lesson: Never blindly trust AI-generated test data. Always give users visibility into what's being tested. The hidden test gate is a helper, not a gatekeeper.

AI Sometimes Over-Engineered or Under-Engineered Solutions

Reality: Sometimes the AI would propose adding entire new frameworks for problems that had simple solutions. Other times, it would under-engineer — like using simple string comparison when order-insensitive comparison was needed.

What we did: Applied the "minimum viable approach" principle — always start with the simplest solution that works, then add complexity only when a specific test case or real-world scenario demands it.

Lesson: AI suggestions are starting points, not finished designs. Review every proposal against the actual requirements. Ask: "Is there a simpler way?" and "What's the minimal change that fixes this specific problem?"