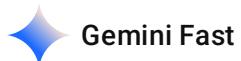


summarize this

<https://codingplainenglish.medium.com/why-uber-eats-gave-kotlin-multiplatform-a-second-chance-and-nailed-it-this-time-ccdec9f24db0>



The article explains how Uber Eats successfully integrated **Kotlin Multiplatform (KMP)** after an initial failed attempt. While Uber was an early adopter of Kotlin, their first foray into multiplatform code faced significant hurdles that they eventually overcame by refining their architecture and tooling.

Here is a summary of the key points:

### 1. The Initial Struggle (The First Chance)

Uber originally experimented with KMP a few years ago but encountered major "deal-breakers" that led them to pause the project:

- **Performance & Memory Issues:** Early versions of KMP (particularly the Kotlin/Native memory model) were difficult to work with on iOS, leading to leaks and performance lag.
- **Tooling Gaps:** The developer experience on iOS was subpar compared to native Swift development, making it hard for iOS engineers to buy into the ecosystem.
- **Binary Size:** Initial implementations added significant overhead to the application size.

### 2. Why the "Second Chance"?

Despite the first failure, Uber Eats revisited KMP because the "two-platform" problem remained. Maintaining identical business logic (like delivery state tracking, pricing calculations, and cart management) in both Swift and Kotlin led to:

- **Logic Drift:** Features would behave slightly differently on iOS vs. Android.
- **Double Effort:** Engineers had to write, test, and bug-fix the same logic twice.
- **Maintenance Burden:** Updating a single business rule required synchronized releases across two codebases.

### 3. What Changed (Why They "Nailed It")

The success of the second attempt was driven by both the evolution of KMP and Uber's internal strategy:

- **The New Memory Model:** JetBrains' update to the Kotlin/Native memory model solved the primary "freezing" and concurrency issues that plagued the first attempt, making it safe and performant for iOS.
- **Infrastructure Investment:** Uber built custom internal tooling to bridge the gap between KMP and their existing monorepo. This included better **Swift interop**, allowing iOS developers to use shared Kotlin code as if it were a native Swift library.
- **Focus on Business Logic, Not UI:** Uber avoided the "write once, run everywhere" trap for UI. They kept the UI strictly native (SwiftUI/UIKit on iOS and Jetpack Compose on Android) but shared the **Domain and Data layers** (API calls, data parsing, and complex state machines).
- **Cultural Buy-in:** Instead of forcing it, they worked to ensure iOS engineers felt empowered by the shared code rather than hindered by it, focusing on how it reduced their workload for "boring" logic.

#### 4. Key Results

- **Code Parity:** Business logic is now 100% consistent across platforms.
- **Faster Iteration:** New features or rule changes can be implemented once in the shared module and deployed to both apps.
- **Reduced Bugs:** Fixing a bug in the shared core automatically fixes it for both Android and iOS users.

**Bottom Line:** Uber Eats succeeded the second time by treating KMP not as a replacement for native development, but as a specialized tool for **sharing the "brains" of the app** while keeping the "face" of the app native.

You Asked 

Make a bullet point list answering those questions

what is Kotlin Multiplatform

are teh tools developed by uber eats open source?

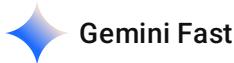
confirm that only the business logic was moved to a shared repo,  
UI remains native code

when you say " This included better Swift interop, allowing iOS

developers to use shared

Kotlin code as if it were a native Swift library", it suggests that on Android it is easier to access the shared code as a library, but Swift is picky?

Why "write once, run everywhere" is a trap?



Based on the article and the current state of the technology, here are the answers to your questions:

- **What is Kotlin Multiplatform (KMP)?**
  - It is a framework developed by JetBrains that allows developers to write code in Kotlin and share it across multiple platforms (iOS, Android, Web, Desktop, and Server).
  - Unlike other cross-platform tools, it does not force you to use a specific UI framework; it focuses on sharing the "logic" while allowing you to keep the native performance and APIs of each platform.
- **Are the tools developed by Uber Eats open source?**
  - **Mostly no.** While Uber is a major contributor to the open-source community (creating tools like Ribs or Needle), the specific internal infrastructure and "glue code" they built to integrate KMP into their massive monorepo for this transition are proprietary.
  - However, Uber frequently publishes engineering blogs and talks that influence how the community builds similar open-source tools (such as SKIE or KMP-NativeCoroutines, which solve the same problems Uber faced).
- **Confirmation: Is UI still native?**
  - **Yes.** Uber Eats explicitly kept the UI native (SwiftUI/UIKit on iOS and Jetpack Compose on Android).
  - They only moved the **Domain Layer** (business rules, data processing) and the **Data Layer** (API networking, database management) to the shared KMP repository.
- **Is Swift "pickier" than Android regarding shared code?**
  - **Yes.** Android uses the JVM (Java Virtual Machine) and Kotlin is its primary language, so accessing a KMP library on Android is "native to native"—it works seamlessly without any translation.
  - **Swift is "pickier"** because Kotlin code must be compiled into an Objective-C framework to be read by iOS. This process historically loses "flavor":

- Kotlin's **Coroutines** (async code) don't naturally look like Swift's **Async/Await**.
  - Kotlin's **Flows** (reactive streams) don't naturally look like Swift's **Combine**.
  - Generic types often get "lost in translation."
- Uber's "better interop" refers to building tools that wrap the Kotlin code so it looks and feels like idiomatic Swift, hiding the fact that it was originally written in Kotlin.
- Why is "Write Once, Run Everywhere" (WORE) considered a trap?
  - **Uncanny Valley UI:** When you try to share UI (like in Flutter or React Native), the app often feels "slightly off" to users—scrolling physics, text selection, or navigation transitions might not match the system's native behavior.
  - **Least Common Denominator:** WORE often limits you to features that exist on *both* platforms, making it difficult to use the latest, platform-specific innovations (like a new iOS-only dynamic island feature) without complex workarounds.
  - **Abstraction Leaks:** When a bug occurs in a cross-platform UI layer, you often have to dig through layers of abstraction to find if the bug is in the framework, the bridge, or the native code.
  - **The Uber Philosophy:** By only sharing the logic, Uber avoids the UI "trap" entirely—users get a 100% native experience, while developers get the efficiency of writing the complex math and rules only once.

You Asked 

how hard would it be for a single developer to wrap some very basic business logic (less than 1000 lines) so that Swift can access it as a library?

Are there some ways of writing the Kotlin code (e.g. avoid certain language features) that would make this easier?



For a single developer with basic business logic (under 1,000 lines), wrapping Kotlin code for Swift is **not very hard**, but it does require setting up a specific "bridge" environment.

If you were to do this today, you would spend about **2–4 hours on initial setup** and then very little time thereafter.

Here is the breakdown of the difficulty and how to write the code to make it easier.

## 1. How hard is it?

The difficulty is **low to medium**, depending on your familiarity with Gradle (the build tool Kotlin uses).

- **The "Easy" Way:** You don't need Uber-level tools. You can use the **Kotlin Multiplatform Mobile (KMM)** plugin in Android Studio. It generates a `.framework` file that you simply drag and drop into your Xcode project.
  - **The "Hard" Part:** The struggle isn't the logic; it's the **environment**. You need a Mac with both Android Studio and Xcode installed, and you need to ensure your "CocoaPods" or "Framework" build scripts are correctly linked.
- 

## 2. How to write Kotlin to make Swift's life easier

Swift and Kotlin are "cousins," but they handle certain things very differently. To make the shared code feel like a native Swift library, follow these rules:

### A. Avoid "Value Classes" and "Inline Classes"

Kotlin has `value class` for performance, but Swift doesn't understand these. They will often be "boxed" or obscured, making them hard to use in Xcode. Stick to standard `data class`.

### B. Minimize the use of `Flow` and `Coroutines`

This is the biggest hurdle.

- **The Problem:** Swift doesn't natively understand a Kotlin `Flow` (a stream of data). If you export a `Flow`, it will look like a messy, unreadable object in Swift.
- **The Strategy:** Wrap your Kotlin functions. Instead of exposing a `Flow`, create a "Wrapper" function in Kotlin that uses a simple **Callback/Closure** to return data. Swift understands closures perfectly.

### C. Avoid "Top-Level" Functions

In Kotlin, you can just write a function in a file without a class. When exported to iOS, these get tucked into a generated class like `FilenameKt.function()`.

- **The Strategy:** Wrap your logic inside a `class` or `object`. This ensures that in Swift, you can call it clearly, like `MyLibrary.calculateResult()`.

### D. Be Careful with Generics and Enums

- **Generics:** Kotlin's advanced generics (like `out` and `in` variances) can get "lost" and turn into `Any?` (Swift's version of `id` or `Optional<Any>`) when they cross over to Swift. Keep generics simple.

- **Enums:** Kotlin Enums work well, but **Sealed Classes** (a favorite Kotlin feature) become very clunky in Swift. You will have to write a `switch` statement that looks quite ugly in Swift unless you use a helper library.

## E. Use "SKIE" (The Secret Weapon)

There is an open-source tool called **SKIE** (by Touchlab).

- For a single developer, this is a game-changer. It automatically bridges Kotlin Sealed Classes and Coroutines into native Swift `Enums` and `async/await`.
- If you use SKIE, the "difficulty" of your project drops significantly because it handles the "pickiness" of Swift for you.

## Summary Checklist for Success:

1. **Use Android Studio** with the KMP Plugin to create the project.
2. **Keep the logic simple:** Stick to `String`, `Int`, `Boolean`, `List`, and `Map`.
3. **Use Data Classes** for your models.
4. **Avoid complex Concurrency** in the public API; use simple callbacks or the SKIE tool.
5. **Expose a single entry point:** Create one "Manager" or "Service" class in Kotlin that Swift interacts with.