

Swift Beyond iOS

Sharing Product Logic with Android
Without Sharing UI



Raul Menezes

Lead Design System & iOS Engineer @ Bumble

Questions I heard today

“Why?”

“I didn’t know
it was a thing.”

“Can it do UI?”

“Is it like KMP?”

01

THE PROBLEM

Why share Swift with Android?

Existing Swift logic. A native Android app.

01 The problem

02 Swift and Android

03 Sharing product logic

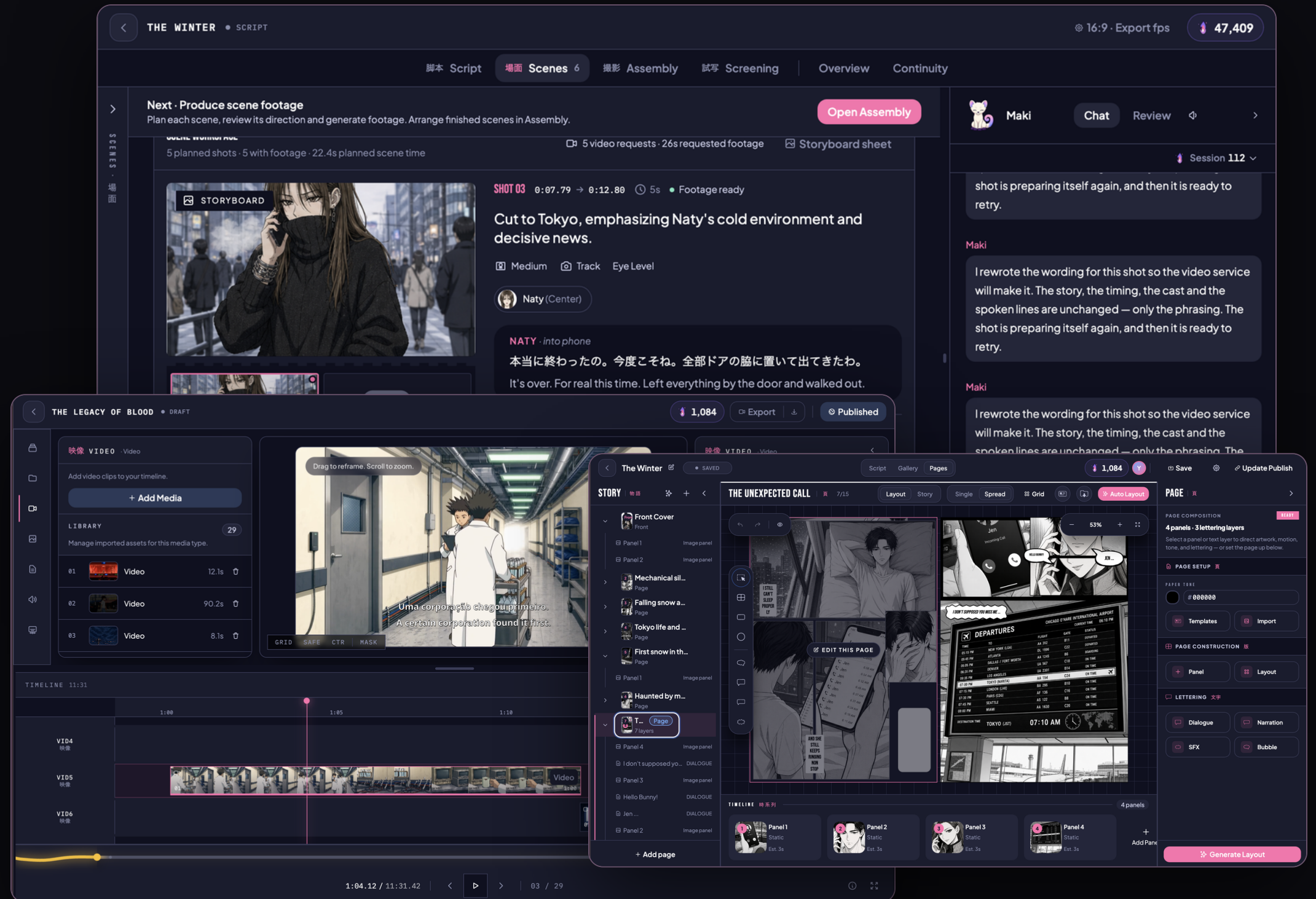
04 The demo

05 Was it worth it?

AniCanon

The creative platform

Scene direction, episode assembly
and manga creation.



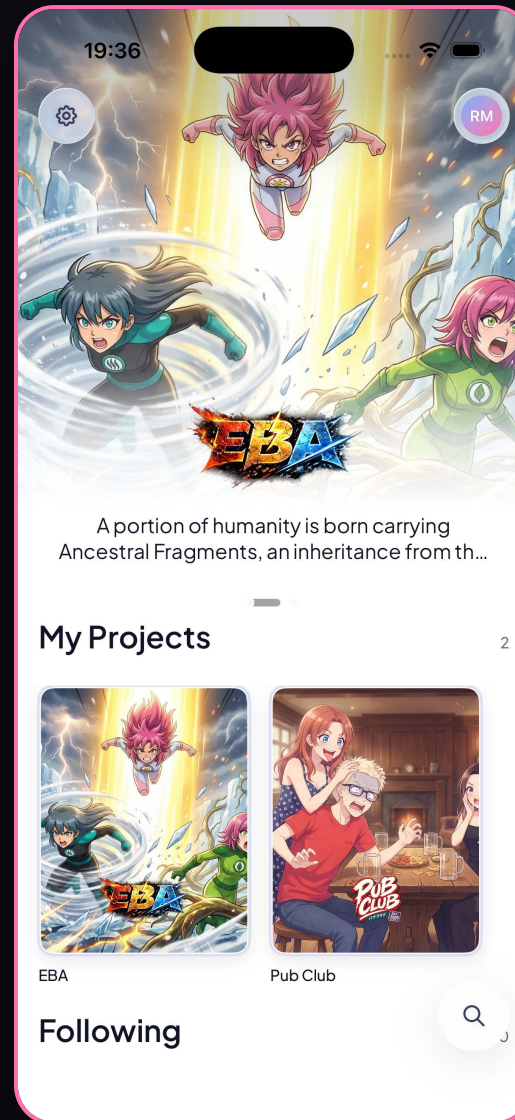
EXAMPLE PROJECT

AniCanon

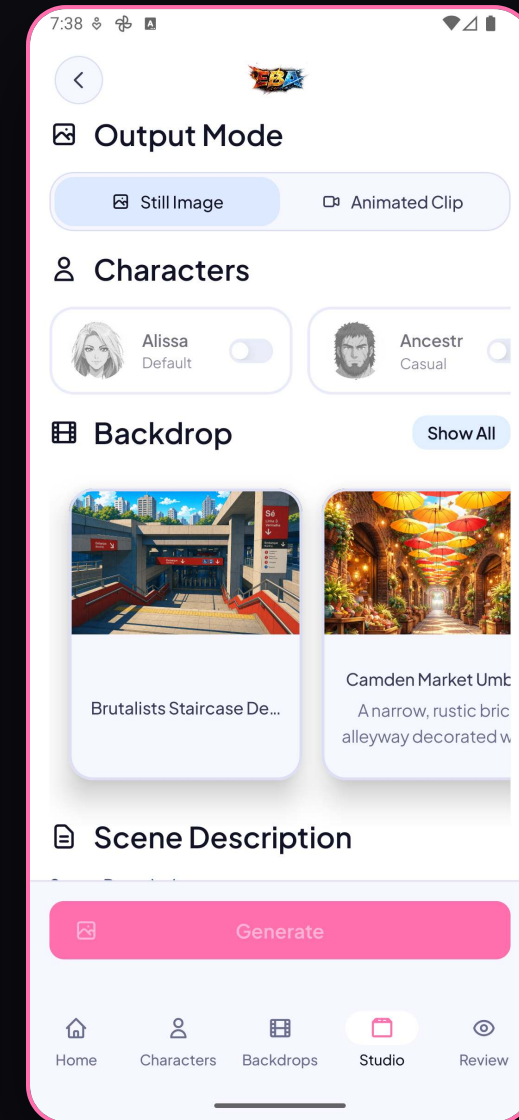
A project I'm building
for creating and reading stories.

SwiftUI on iOS.

Kotlin and Compose on Android.



 **SwiftUI**



 **Kotlin + Compose**

Why keep the UI native?

The reader's page turn.



Different ways to share



Swift core

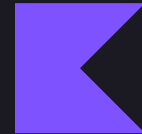
Extend an existing Swift domain.

PRESENTATION

Separate native UIs

TEAM INVESTMENT

Swift expertise + interop ownership



Kotlin Multiplatform

Share a Kotlin domain across platforms.

PRESENTATION

Native UI or shared Compose UI

TEAM INVESTMENT

Kotlin expertise + platform integration

.NET

.NET MAUI

Share C# application code and UI.

PRESENTATION

Shared UI with native integration

TEAM INVESTMENT

C# / XAML + platform customisation

Our starting point: a Swift product core worth keeping.

02

SWIFT AND ANDROID

How do Swift and Android work together?

Native code, generated bindings and asynchronous calls.

- 01 The problem
- 02 Swift and Android
- 03 Sharing product logic
- 04 The demo
- 05 Was it worth it?

Swift 6.4

Android builds

Android SDK, NDK 30 and Swift Build support

Java interop

More async, throwing and callback support

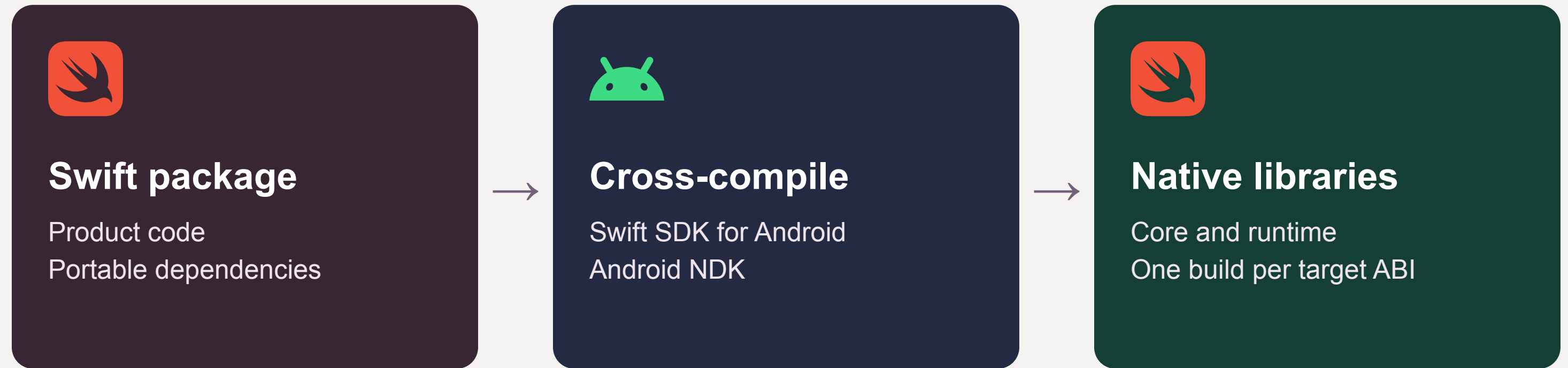
Application boundary

Still yours to design and test

What actually runs on Android?

Native Swift code inside your Android app.

Cross-compiling the Swift package



ANDROID PACKAGE

Kotlin application + native Swift libraries

Inside the Android app

ONE ANDROID APPLICATION PROCESS



Kotlin + Compose

Managed application code

Android Runtime

ART / managed objects

Generated bindings

JNI



In-process calls



Swift product core

Native machine code

Swift runtime

ARC / standard library / Foundation

Android system libraries

Linux kernel

Kotlin and Swift in one process



Managed

`useCase.start()`

Kotlin / Java arguments

JNI



Generated
call adapters



Native

Swift implementation

Product models and behavior

One process. **Two object models.**

Generating the Java API



Swift declarations

Public types and functions



swift-java jextract

Generates Java wrappers and Swift JNI entry points



Generated Java API

Ready for Kotlin to call

Swift code, called from Kotlin

Build time: swift-java generates the bindings. Shortened example.

1. Write Swift

```
public class Counter {  
    public var count: Int64  
  
    public init() {  
        count = 0  
    }  
}
```

2. Generated Java

```
public long getCount() {  
    return $getCount(  
        $memoryAddress()  
    );  
}  
  
private static native long  
    $getCount(long self);
```

3. Call from Kotlin

```
val count =  
    counter.getCount()
```

Existing Swift-backed counter.
The getter returns a number.

Runtime: Kotlin → Java wrapper → JNI → Swift getter

Async/await across the boundary

One call returns a Swift-backed model, asynchronously.

Swift · returning a model

```
func fetch() async throws  
    -> Product
```

Generated Java · future + lifetime

```
CompletableFuture<Product>  
    fetch(SwiftArena arena);
```

Kotlin · calling the generated Java API

```
val future: CompletableFuture<Product> = shared.fetch(arena)
```

Future

When the result becomes available.

Arena

Manages the returned model's native lifetime.

Streaming across the boundary

AsyncStream delivers values over time.

Swift · consuming an AsyncStream

```
let updates: AsyncStream<Update>
    = source.updates()

for await update in updates {
    render(update)
}
```

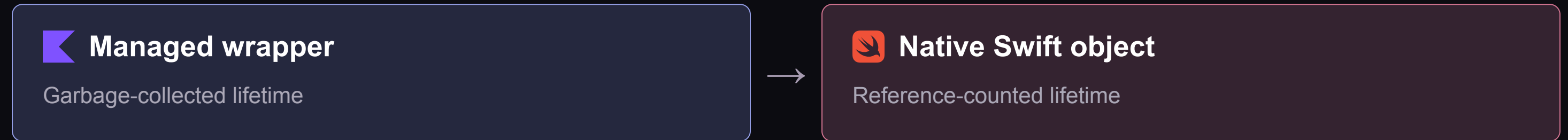
Java / Kotlin · without our tooling

No Java binding for this method

The generator used here skips methods returning raw AsyncStream. It reports an unresolved type.

Android needs an adapter to consume these updates.

Native memory and SwiftArena



SwiftArena coordinates native cleanup

Memory cleanup and operation cancellation are separate.

03

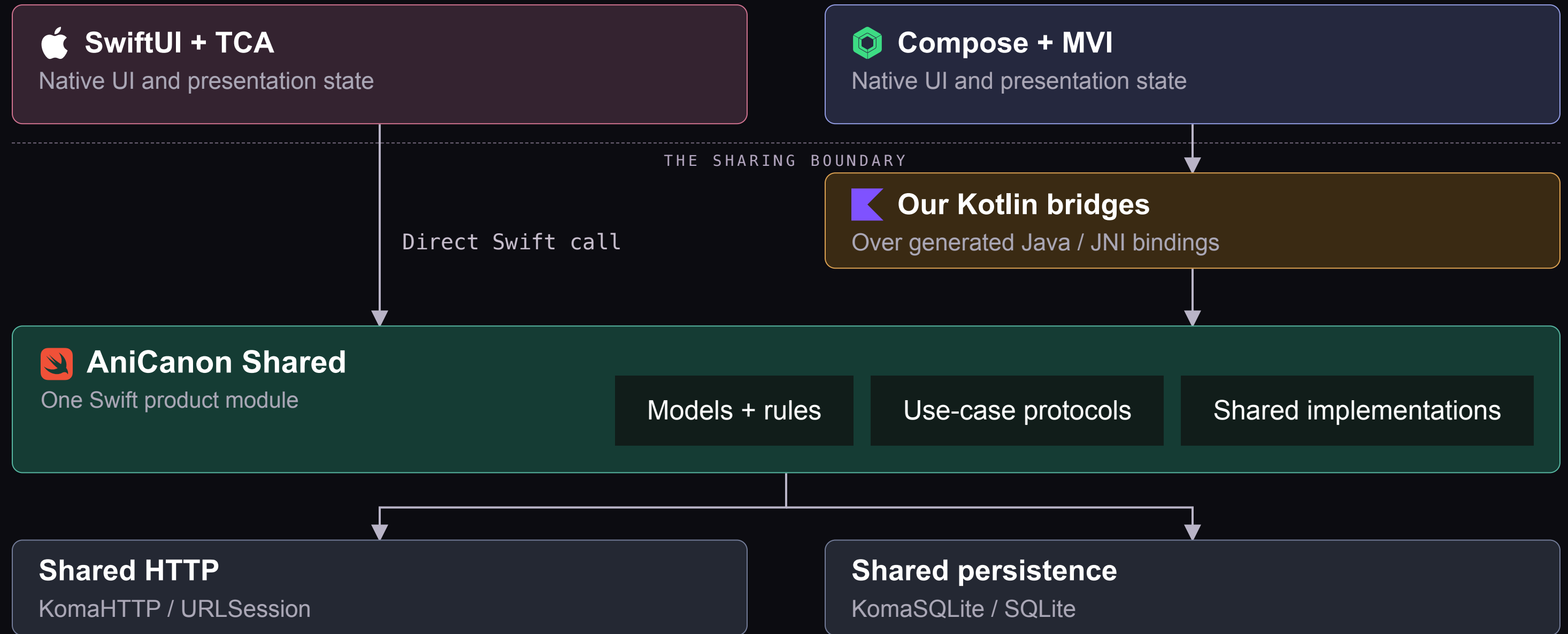
SHARING PRODUCT LOGIC

How do we share product logic?

AniCanon's architecture, adapters and trade-offs.

- 01 The problem
- 02 Swift and Android
- 03** Sharing product logic
- 04 The demo
- 05 Was it worth it?

The shared Swift layer



The Kotlin API we wanted

Android features use Kotlin. Our tooling handles the interop.

Android feature

Properties
`model.property`

suspend
`useCase.start(...)`

Flow
`updates.collect { ... }`

Our accessors + bridges

Kotlin-facing models, asynchronous calls and stream adapters

Generated Java / JNI

The language boundary

AniCanon Shared

The product logic stays in Swift

Where our tooling fits

Build and integration pipeline



Swift declarations

Models and use cases + our generated Swift stream observations



swift-java

Generated Java and native JNI bindings



Kotlin accessors + generated bridges

Our tooling: properties, suspend functions and Flow adapters



Android features

Kotlin + Compose consume the API

Our Kotlin accessors and bridges

The same Java future, wrapped once for our Android features.

Our generated bridge

```
suspend fun start(...):  
    OutfitGenerationStartResult =  
        shared.start(..., arena).await()
```

Android feature · inside a coroutine

```
val result = startUseCase.start(...)  
  
handleResult(result)
```

The bridge awaits the future. The feature works with the result.

Accessors expose model properties and collections through a Kotlin-facing API.

Adapting streams for Kotlin

The AsyncStream stays in Swift. We generate an API around it.

1 · Our Swift generator

```
AsyncStream<Update>  
  
next() async -> Update?  
cancel() async
```

Creates a concrete observation before jextract runs.

2 · swift-java

```
next(arena)  
    → Future<Optional<Update>>  
  
cancel() → Future<Void>
```

Exports those supported methods as Java futures.

3 · Our Kotlin bridge

```
observationFlow(  
    open = { ... },  
    next = { ... },  
    cancel = { ... }  
)
```

Awaits each value and emits it into a Kotlin Flow.

Android feature

```
progressUseCase.generationProgress(...).collect(::applyProgress)
```

Packaging the Swift core for Android

The package we edit in Xcode becomes part of the Android Studio app.

Gradle coordinates the Android build



Swift sources

Shared package + observations



Generated API

Java / JNI + Kotlin adapters



Swift + JNI glue

Native libraries for each ABI



Kotlin + Java

Compile and dex



Android package

App bytecode + native libraries

Shared logic and native integration



Shared Swift

Product models and rules

Use cases and progress policy

HTTP, persistence and server events

Notification destination parsing



Native clients

Screens and presentation state

Navigation and platform services

Collection lifecycle

Push UI and live notifications

Swift describes an update. The native app decides how to display it.

Shared infrastructure

REQUEST PIPELINE

HTTP

KomaHTTP



URLSession transport



Remote API

Common request and response handling

LOCAL DATA

Persistence

KomaSQLite



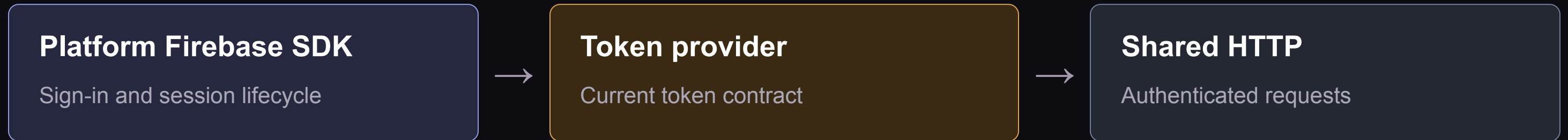
SQLite C interface



On-device database

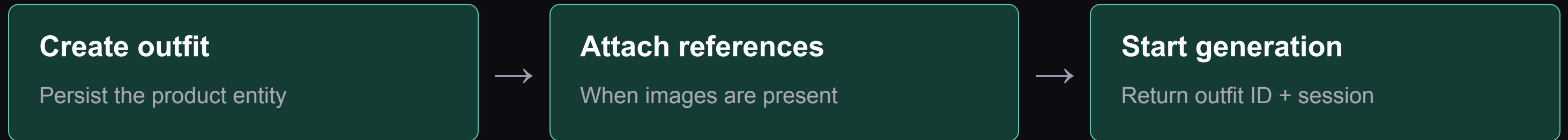
A separate database on each installation

Authentication stays native



Native service, shared contract

The shared generation use case

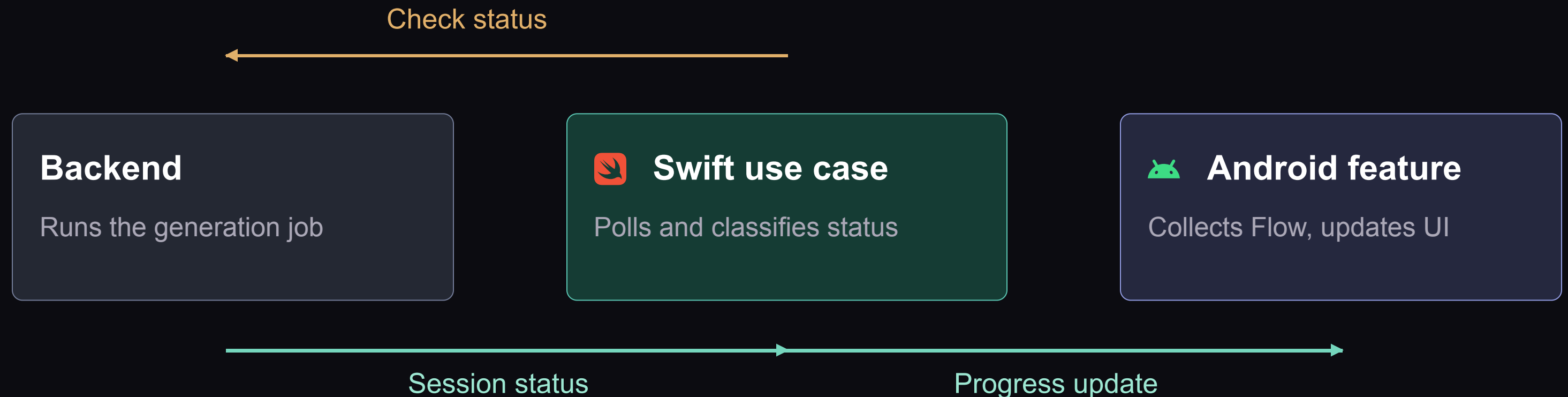


One shared Swift use case

Both clients call the same product sequence.

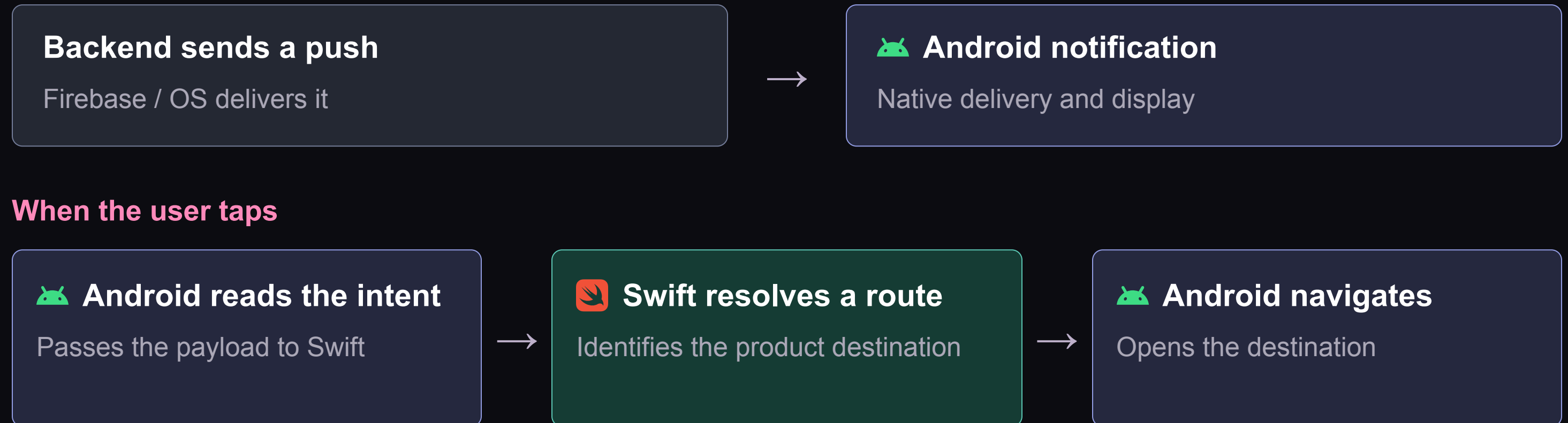
Following generation progress

While the screen is open, Swift checks the job and Android collects the updates.



Swift checks the backend and emits progress. Android updates its UI.

Returning through a notification



Android receives the push. After a tap, Swift resolves the route and Android navigates.

04

THE DEMO

What does it look like in the app?

The outfit creation experience.

01 The problem

02 Swift and Android

03 Sharing product logic

04 The demo

05 Was it worth it?

The same Swift use case

iOS and Android keep their own interfaces.



Separate recordings. Waiting time shortened.

05

WAS IT WORTH IT?

Would this work for your team?

The benefits, the cost and the decision.

01 The problem

02 Swift and Android

03 Sharing product logic

04 The demo

05 Was it worth it?

It depends.

What do you gain by sharing?

What will it take to make it work?

The adoption cost

Build tooling	SDK, NDK, ABIs and packaging
Generated API	Bindings, Kotlin ergonomics and version compatibility
Runtime contracts	Errors, cancellation and native ownership
Engineering time	Documentation, debugging and cross-platform tests

Should your team adopt it?

Existing investment	Is there a Swift product core worth keeping?
Clear boundary	Can you name what each native app still owns?
Portable dependencies	Does a real vertical slice work on Android?
Ownership	Can the team maintain the tooling and integration?

What you would need to own

A portable Swift package.

A Kotlin API your features can use.

A build and runtime boundary you can maintain.

Questions?

rmenezes.me

Swift Beyond iOS