Cross-Platform Mobile UI

"Compilers, Compilers Everywhere"

27 June 2023 – EOSS Prague

Andy Wingo

Igalia, S.L.

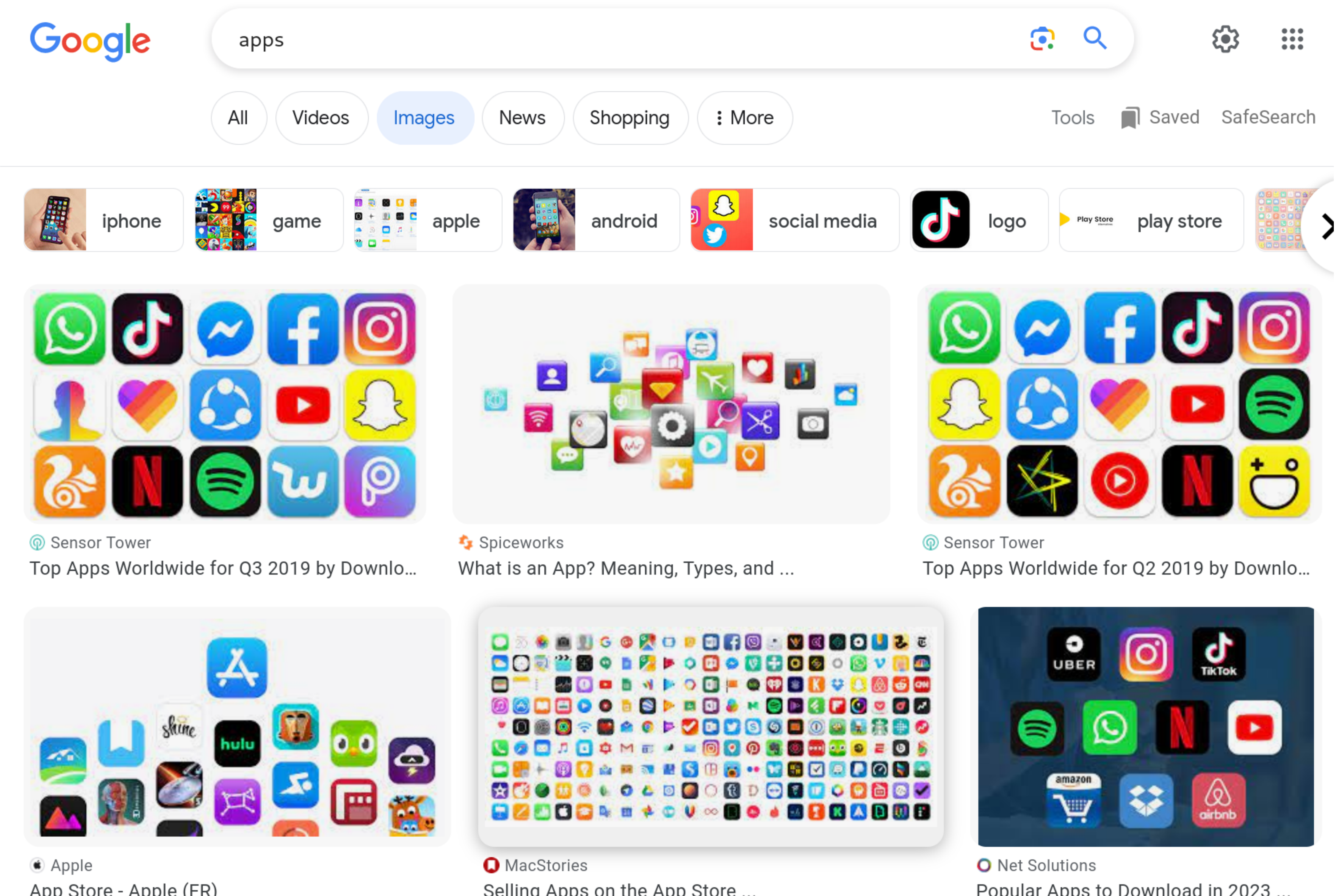
Apps apps apps

This is a talk about apps; good apps

And compilers; weird compilers

And open source, cross-platform app frameworks

And the unexpected end of the end of history


Google
apps
All
Videos
Images
News
Shopping
More
Tools
Saved
SafeSearch
iphone
game
apple
android
social media
logo
play store
Sensor Tower
Top Apps Worldwide for Q3 2019 by Downlo...
Spiceworks
What is an App? Meaning, Types, and ...
Sensor Tower
Top Apps Worldwide for Q2 2019 by Downlo...
Apple
App Store - Apple (FR)
MacStories
Selling Apps on the App Store ...
Net Solutions
Popular Apps to Download in 2023 ...
UBER
TikTok
amazon
airbnb
hulu

https://signal.org

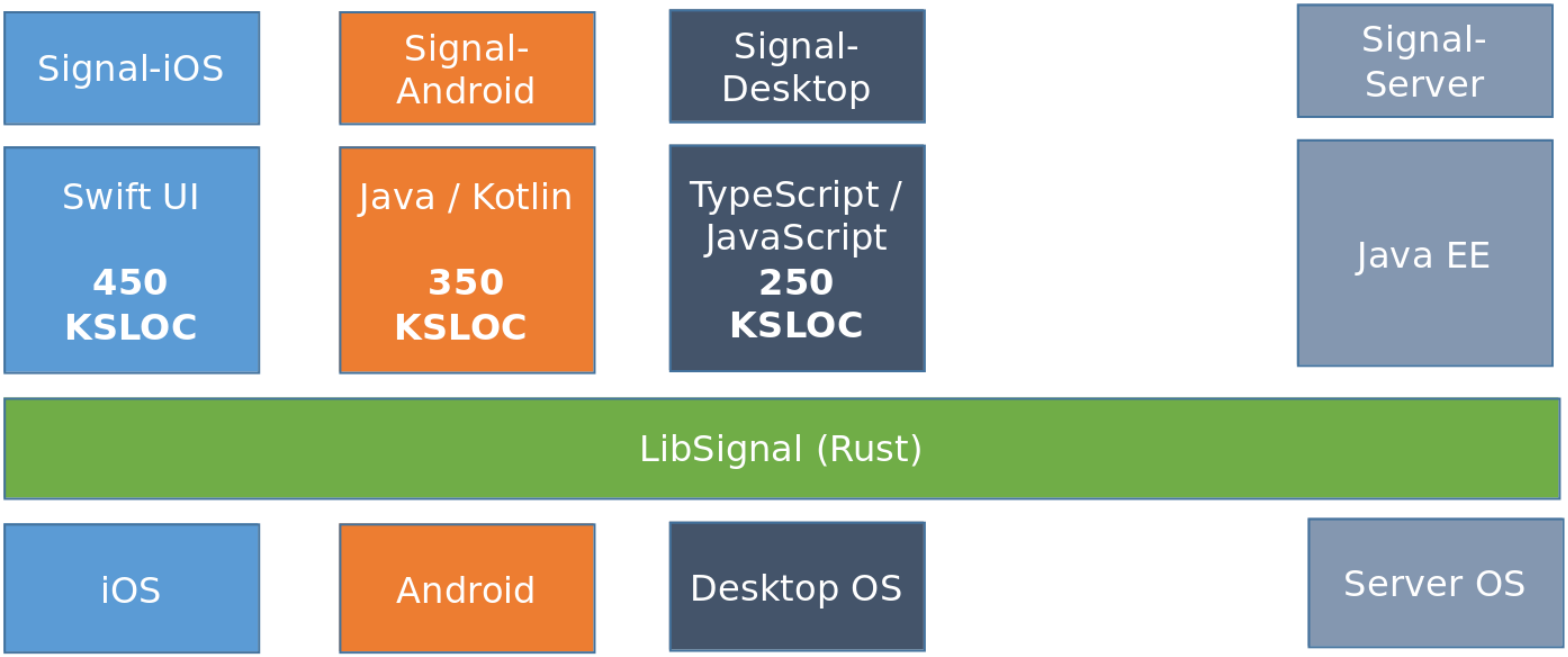
Search

Signal

Speak Freely

Say "hello" to a different messaging experience. An unexpected focus on privacy, combined with all of the features you expect.

Get Signal



With thanks to Yong He from Futurewei

Do we know how to make apps?

- SwiftUI
- React Native
- Java and Android views
- Jetpack Compose
- OpenGL / Vulkan
- AppKit
- Capacitor
- NativeScript
- Flutter
- UIKit

Step back

I.M.H.O.—H before the O

Observe and learn: look for meaning and motivation

Come back with lessons, then apply them to now

Lessons

1.
2.
3.
4.
5.

Lesson 1

The old thing: stateful widget trees

```
var count = 0
let stack = new VStack
let text = new Text("Count: \(count)")
stack.add_child(text)
let button = new Button("Increment")
button.set_onclick(||
  count += 1
  text.set_text("Count: \(count)")
)
stack.add_child(button)
```

https://raphlinus.github.io/ui/druid/2019/11/22/reactive-ui.html

Lesson 1: Declarative UI won on the web

The newer thing: Declarative UI

2013: React

```
function Hello({ name }) {
  return (
    <p>Hello from React, {name}!</p>
  );
}
```

UI is a function: translate state to immutable tree of elements

Nowadays many derivatives of this paradigm

Lesson 1: Declarative UI won on Android

2019: Jetpack Compose

```
@Composable
fun MessageCard(name: String) {
  Text(text = "Hello from " +
        "Jetpack Compose, $name!");
}
```

Sometimes UI tree implicitly collected instead of returned

Lesson 1: Declarative UI won on iOS

2019: SwiftUI

```
struct ContentView: view {
  var name: String
  var body: some View {
    Text("Hello from SwiftUI, \(name)!")
      .padding()
  }
}
```

Particularly lovely ergonomics

Lesson 1: Declarative UI won, cross-platform

2017: Flutter

```
class Hello extends StatelessWidget {
  const Hello({required this.name,
              super.key});
  final String name;

  @override
  Widget build(BuildContext context) {
    return Text('Hello from Flutter,'
                + '$name!');
  }
}
```

Even when people abstract away from platform, they go declarative

Lesson 1: Declarative UI won

But why? 3 reasons

- Managers like it: Decompose UI into org chart (Conway's law)
- Comprehensively avoid view/model state mismatch
- Developers seem to like it too

Lessons

1. Declarative UI won
2.
3.
4.
5.

Lesson 2

The rise of the framework

- Developer declares UI
- Framework translates to imperative operations on e.g. GPU
- Framework determines when UI needs recomputation

Observation: UI tree computation O(n) in UI size

How to avoid performance disaster?

Lesson 2: Frameworks limit performance

Division of labor: app developers say what, framework developers say how

Risky bargain

4 main techniques

- ❧ Managed state
- ❧ Incremental render
- ❧ Concurrent render
- ❧ Concurrent GC

Lesson 2: Frameworks limit performance

Technique 1: Managed state

Framework re-renders only when needed

```
function Hello({ name }) {
  const [count, setCount] = useState(0);
  function inc() {
    setCount(x => x+1);
  }
  return (
    <div>
      <p>Count: {count}</p>
      <button onclick={inc}>+1</button>
    </div>
  );
}
```

Lesson 2: Frameworks limit performance

Technique 2: Incremental render

Functional on top, but always imperative underneath

- Web: DOM
- React Native: UIKit / Android view tree
- Flutter: GPU pipeline objects

Don't recreate whole DOM on each frame: just apply changes

```
Pixels[N+1] := Pixels[N] +
               Diff(UI[N+1], UI[N])
```

Lesson 2: Frameworks limit performance

Technique 3: Concurrent render

Basic: Build UI on one thread, render to GPU/DOM on another

Hard on the web, easier on mobile

Limited gains for per-frame concurrency

Build and display frames in 16ms

Since there are two separate threads for building and rendering, you have 16ms for building, and 16ms for rendering on a 60Hz display. If latency is a concern, build and display a frame in 16ms *or less*. Note that means built in 8ms or less, and rendered in 8ms or less, for a total of 16ms or less.

If your frames are rendering in well under 16ms total in profile mode, you likely don't have to worry about performance even if some performance pitfalls apply, but you should still aim to build and render a frame as fast as possible. Why?

- Lowering the frame render time below 16ms might not make a visual difference, but it **improves battery life** and thermal issues.
- It might run fine on your device, but consider performance for the lowest device you are targeting.
- As 120fps devices become more widely available, you'll want to render frames in under 8ms (total) in order to provide the smoothest experience.

Lesson 2: Frameworks limit performance

Technique 4: Concurrent GC

Concurrent: Runs while program runs. Move O(n) trace off main thread

Without concurrent GC:

```
// UI thread
frame . frame . frame . pause: trace+finish . frame
```

With concurrent GC:

```
// UI thread
frame . frame . frame . pause: finish . frame
// GC thread
          trace..................
```

Reduce long-pole GC pause

Lesson 2: Frameworks limit performance

Providing good performance is a framework concern

Frameworks nudge app developers into good performance patterns

Frameworks limit performance too

Lessons

1. Declarative UI won
2. Frameworks limit performance
3.
4.
5.

Lesson 3

Lesson 3: Compilers are back

Example 1: Front-end

```
@Composable
fun MessageCard(name: String) {
  Text(text = "Hello from " +
        "Jetpack Compose, $name!");
}
```

`@Composable` decorator: make it look like you declare a tree, but compile to imperative operations

Lesson 3: Compilers are back

Not just Jetpack Compose

- SwiftUI ResultBuilder
- HarmonyOS ArkUI with “eTS”
- React JSX

Compilers are a core part of the modern UI story

To work in this space: control the means of production

Lesson 3: Compilers are back

Example 2: Deployment

Problem: Minimize startup latency, maximize runtime predictability

Trend: Move to ahead-of-time compilation

- React Native Hermes
- Panda ArkTS (without `eval`!)
- Dart AOT

Predictability over performance

Lesson 3: Compilers are back

Example 3: Graphics

Shader compilation jank

Contents

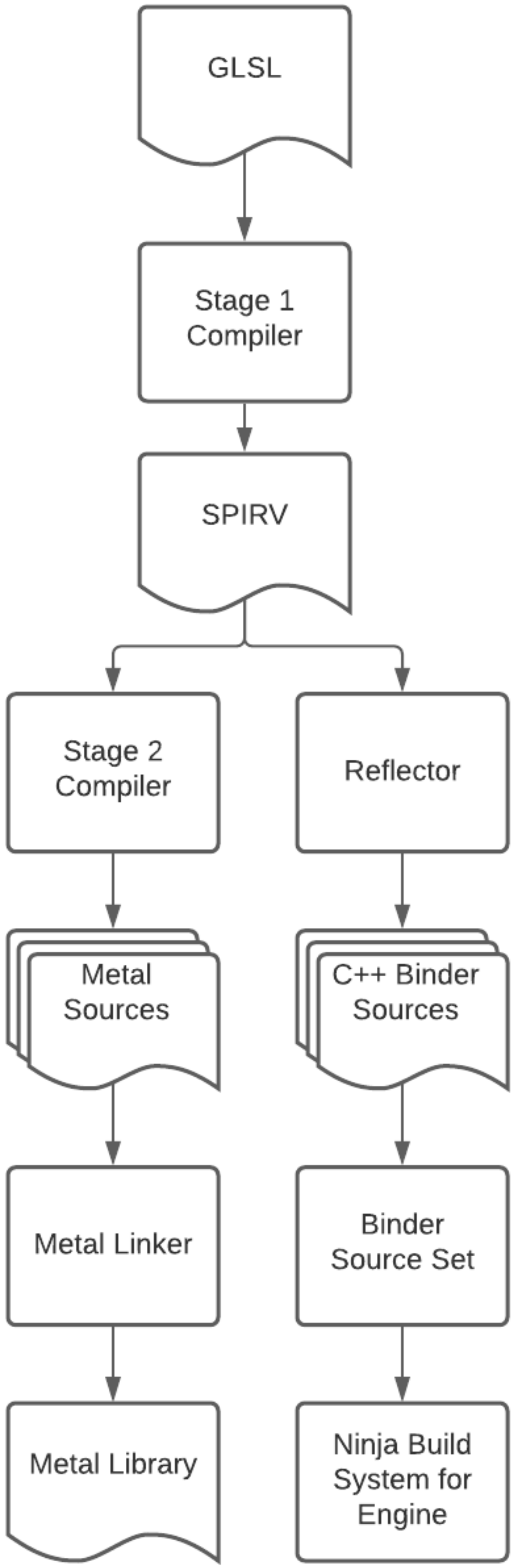
Note: To learn how to use the **Performance View** (part of Flutter DevTools) for debugging performance issues, see Using the Performance view.

Lesson 3: Compilers are back

Flutter Impeller: Compile shaders ahead-of-time, not at run-time

Requires different rendering backend: tesselate into many primitive triangles instead of generating specialized shaders

Write all shaders in GLSL, compile to Metal / Vulkan


GLSL
Stage 1 Compiler
SPIRV
Stage 2 Compiler
Reflector
Metal Sources
C++ Binder Sources
Metal Linker
Binder Source Set
Metal Library
Ninja Build System for Engine

Lessons

1. Declarative UI won
2. Frameworks limit performance
3. Compilers are back
4.
5.

Lesson 4: Programming languages are back?!

End of end of history (viz Java)

Declarative UI: Functional reactive programming (FRP)

Declarative syntax requiring *language* work

Are all languages the same?

Dart, Swift

Lesson 4: Programming languages are back?!

Types! Swift, Dart, TypeScript

Co-design of language with plaform

- Dart went sound and null-safe for better AOT performance and binary size

Shift from run-times to compilers

Lessons

1. Declarative UI won
2. Frameworks limit performance
3. Compilers are back
4. Programming languages are back?!
5.

Lesson 5: There’s no winner yet

Marginal cross-platform app development, viz Signal

Even relative winners have “new architecture”

- React Native: Fabric
- Flutter: Impeller

Froth in JavaScript space: new winner every other year

Flutter bundles the kitchen sink

Lesson 5: There's no winner yet

Lots of awkward choices

- Jetpack Compose
- ArkTS eTS
- React Native / Hermes needing transpilers

"Do we know how to build apps?"

Lessons

1. Declarative UI won
2. Frameworks limit performance
3. Compilers are back
4. Programming languages are back?!
5. No winner yet

There is space for something else

What is to be done?

What is to be done?

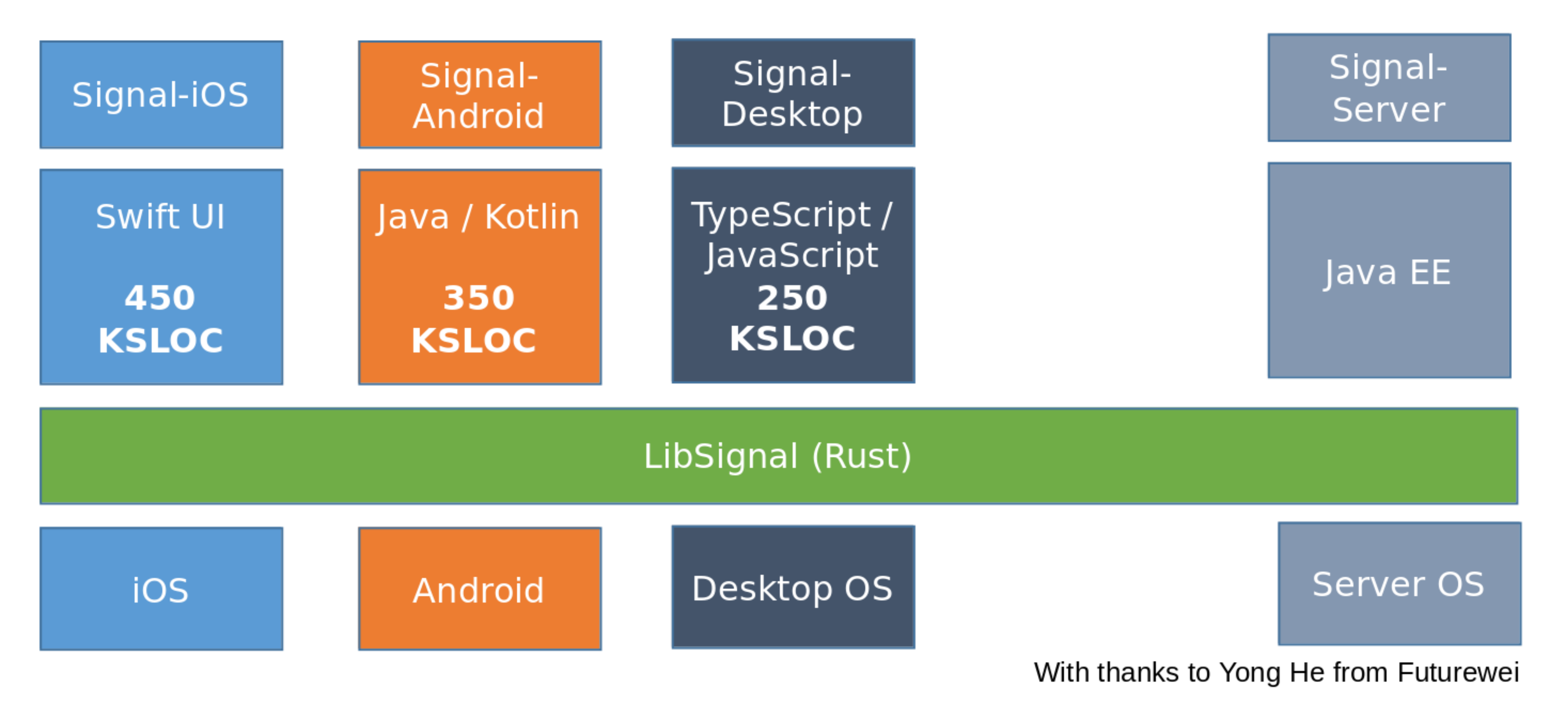
Use Flutter

What is to be done?

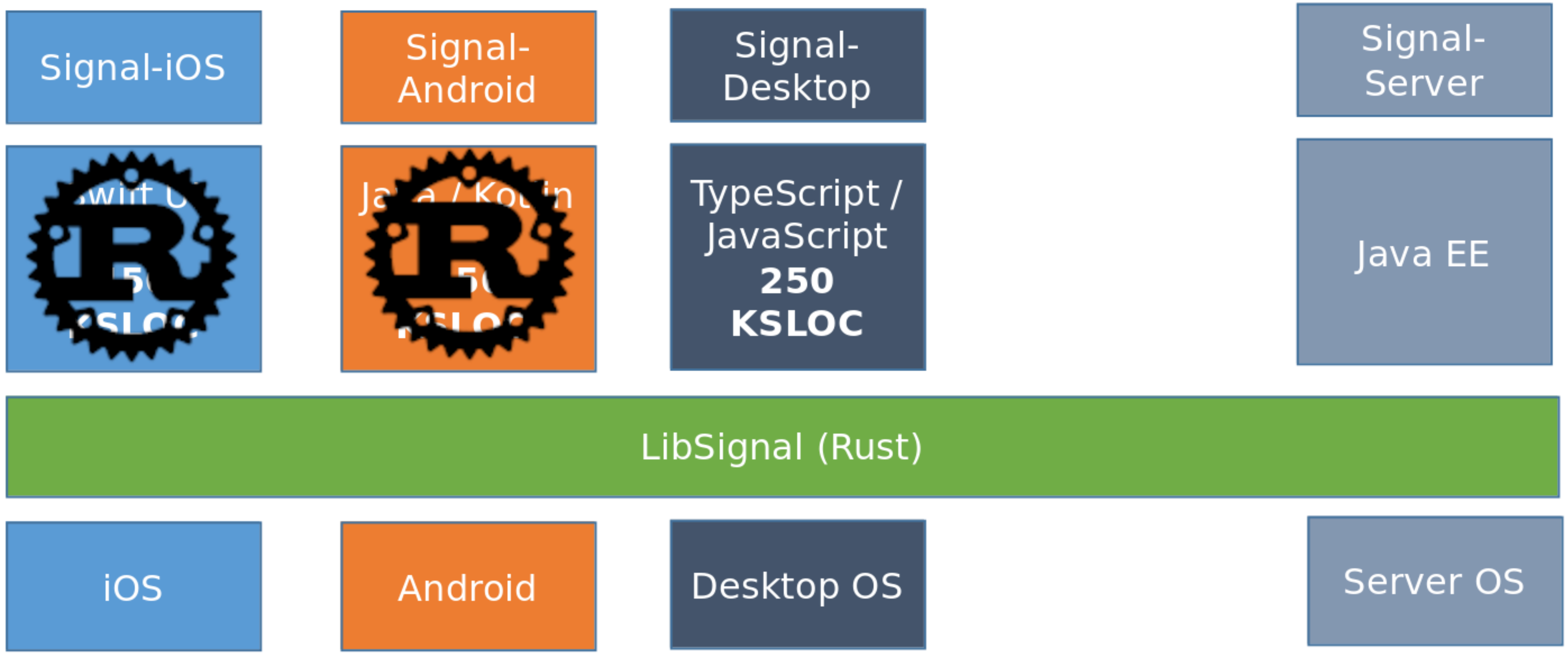
Use Flutter

Caveats: Text, Impeller, Google

What is to be done?



Rust?!?



With thanks to Yong He from Futurewei

Future 1: Rust

Declarative: Dioxus, `dioxuslabs.com`

```
fn app(cx: Scope) -> Element {
  cx.render(rsx!{
    div {
      "Hello, world!"
    }
  })
}
```

Experimental WebGPU backend

Other options out there

Future 1: Rust

State story limited (same as React)

Compilers? Yes! Predictable AOT

Language: lightweight experimentation via macros; `rsx!`

Flutter on Rust is *great* pitch

Future 2

Future 2: JS

Ride wave of JavaScript popularity

Lots of activity: NativeScript, Capacitor, React Native, …

Native widgets: NativeScript, React Native

AOT compilation: ~NativeScript, React Native

Still room for new frameworks

Future 2: JS

Risk: you sail in the wake of a big ship

Flutter's choice to abandon JS understandable though also risky

Far-sighted option: sound typing for TypeScript

Future 3

What does Flutter need from a platform? Build that

Future 3: Wasm and WebGPU

…and WebHID and ARIA and WebBluetooth and…

Pitch: commoditize platforms by providing same binary ABI

User apps are Wasm modules that import WebGPU et al capabilities

Efficient interoperation facilitated by GC in WebAssembly 2.0

Summary

Apps at the end of the end of history

- ❧ Declarative
- ❧ Platform / language codesign

Strong cross-platform contenders: React Native and Flutter

There is room for more

Crystal ball: in 2y, Flutter in Rust; in 5y, sound TypeScript AOT

To read more: `wingolog.org`