Engineering Notebook

VisionOS for Pre-Processed Systems

Update 1 (iOS Camera App Development)

Useful Sites:

- https://developer.apple.com/machine-learning/models/
- https://developer.apple.com/documentation/visionos/analyzing-the-performance-of-your-visionos-app
- https://developer.apple.com/swiftui/
- creating-your-first-visionos-app

Objective

Begin building an iOS camera application using Xcode and Swift. The goal for Day 1 was to set up the development environment, understand the core frameworks involved, establish a live camera preview on a real iPad, and lay the groundwork for real-time object detection.

1. Environment Setup

Tools & Requirements

- Xcode (downloaded from Mac App Store)
- Apple Developer account (free tier) created at developer.apple.com
- Swift selected as the programming language
- SwiftUI selected as the UI framework
- Physical iPad used as the test device (iOS Simulator does not support camera access)

Project Configuration

A new Xcode project was created with the App template. The entry point file CameraAppPrototypeApp.swift was left unmodified — it uses the @main attribute to mark the app entry point and launches ContentView inside a WindowGroup.

Camera permission was added to Info.plist:

```
Privacy - Camera Usage Description  ->  "This app uses the camera for live preview."
```

Without this key, iOS silently denies camera access and the app displays a black screen.

2. Project File Structure

The project contains four Swift source files, all located inside the inner CameraAppPrototype folder (the app target folder, not the project root):

`CameraAppPrototypeApp.swift`	App entry point. Auto-generated, not modified.
`CameraManager.swift`	Owns and runs the AVCaptureSession on a background thread.
`CameraPreview.swift`	Bridges AVCaptureVideoPreviewLayer into SwiftUI via UIViewRepresentable.
`ObjectDetector.swift`	Receives live camera frames and runs Vision object detection.
`ContentView.swift`	Root UI — displays camera feed and draws detection overlays.

3. Key Concepts Learned

AVFoundation Camera Pipeline

The camera stack in iOS flows from hardware through to the screen in distinct layers:

- AVCaptureDevice — represents the physical camera (front or back)
- AVCaptureDeviceInput — wraps the device so it can feed into a session
- AVCaptureSession — the central pipeline connecting inputs to outputs
- AVCaptureVideoPreviewLayer — renders live frames onto the screen
- AVCaptureVideoDataOutput — intercepts individual frames for processing

Threading Model

iOS enforces a strict threading rule: heavy work (camera, ML inference) runs on background threads, while all UI updates must happen on the main thread. Two key dispatch calls were used:

```
DispatchQueue.global(qos: .userInitiated).async { }  // jump OFF main thread
```

```
DispatchQueue.main.async { }                    // jump BACK onto main thread
```

UIViewRepresentable

SwiftUI cannot directly use UIKit views. The UIViewRepresentable protocol provides a bridge — makeUIView() creates the view once, and updateUIView() is called when SwiftUI state changes. This was used to embed AVCaptureVideoPreviewLayer inside SwiftUI.

Delegate Pattern

AVCaptureVideoDataOutputSampleBufferDelegate is a protocol that allows iOS to call captureOutput() automatically each time a new camera frame arrives, without the developer needing to poll. This is equivalent to an event listener in JavaScript.

Vision Coordinate System

Vision bounding boxes use a coordinate system where (0,0) is the bottom-left corner. SwiftUI's coordinate system has (0,0) at the top-left. A conversion function was written to flip the Y axis before drawing overlays:

```
y = (1 - box.maxY) * size.height
```

ObservableObject & @Published

Classes that conform to ObservableObject can publish changes to SwiftUI. Properties marked @Published automatically trigger UI re-renders when their value changes. This was used in ObjectDetector to push new bounding box data to ContentView in real time.

4. What Was Built

By end of Day 1, the app:

- Displays a full-screen live camera feed from the iPad's rear camera
- Requests camera permission from the user on first launch
- Intercepts individual video frames via AVCaptureVideoDataOutput
- Runs VNRecognizeAnimalsRequest (Vision framework) on each frame to detect cats and dogs
- Draws green bounding boxes and confidence percentage labels over detected objects in real time

Implementation of NSCameraUsage Custom String to bypass privacy errors on iPad and OS Systems

The object detection model was subsequently discussed for upgrade to YOLOv3 (Core ML) to support 80 common object categories beyond animals. The model file is to be downloaded from Apple's developer site and added to the project on Day 2.

5. Next Steps (Day 2)

- Download YOLOv3.mlmodel from developer.apple.com/machine-learning/models
- Add model file to Xcode project and update ObjectDetector.swift to use VNCoreMLRequest
- Test object detection on cups, chairs, phones, and other common items
- Implement user-facing annotation layer (text boxes pointing to detected objects)

Notes

The iOS Simulator cannot access the camera — all testing must be done on a physical device. Developer Mode must be enabled on the device under Settings > Privacy & Security > Developer Mode.

Update 2 (iOS Camera App Development)

Status: App running successfully on device from Day 1 build.

Problem Encountered

The green bounding box was flickering on and off instead of staying solid. The console was throwing this error:

"ForEach: the ID Object occurs multiple times within the collection, this will give undefined results"

The root cause was that every detected object was being labeled "Object", so all detections shared the same ID. SwiftUI couldn't distinguish between them, causing undefined rendering behavior and the visible flicker.

Changes Made

Three fixes were applied across two files:

`DetectedObject` was updated to conform to the `Identifiable` protocol with a unique `UUID` on each detection, so SwiftUI can track each box independently.

`ContentView` was updated to use `ForEach(detector.detectedObjects)` instead of `ForEach(..., id: \.label)`, relying on the new `Identifiable` conformance rather than the duplicate label string.

`ObjectDetector` was updated to hold onto the last known bounding boxes for 0.35 seconds when Vision returns no results between frames, preventing the boxes from disappearing and reappearing on every frame.

Result: Duplicate ID warning resolved, bounding boxes now render stably without flickering.

Known Limitation Identified

The current implementation uses Vision's rectangle detection rather than true object detection. It draws boxes around rectangular shapes in the frame but will not reliably identify what objects actually are. Reliable general object detection requires integrating the YOLOv3 Core ML model via VNCoreMLRequest.

Attempt 1 — Rectangle Tracking (Option A) **Fail**

First approach was to improve box stability using Vision's built-in rectangle detection combined with a tracking layer. All changes were made to ObjectDetector.swift.

- Detection set to run every 15 frames instead of every frame, with Vision tracking keeping boxes attached between detection passes.
- Each tracked box keeps the same UUID so SwiftUI renders it stably. Boxes dropped only when tracking confidence falls below 0.35.

Flickering persisted — every 15th frame the detector reran and briefly returned empty results. A second pass tightened the parameters:

- Detection interval raised from every 15 frames to every 45 frames.
- Tracking confidence threshold lowered from 0.35 to 0.15.
- Boxes held for up to 30 missed tracking frames before being cleared.

Fundamental limitation: Vision rectangle detection only reliably boxes rectangular-edged objects (books, screens, signs, papers). It cannot reliably track general objects like bottles, mugs, hands, or plants. Decided to move to Option B.

Attempt 2 — Core ML Object Detection (Option B) **Successful**

Switched to a real object detection Core ML model using VNCoreMLRequest instead of rectangle detection.

- YOLOv3 .mlmodel file downloaded and added to the Xcode project. The file appeared in the navigator as "Core ML Machine Learning Model.mlmodel" rather than YOLOv3.mlmodel.
- ObjectDetector.swift updated to try loading both bundle names at runtime (YOLOv3 first, then Core ML Machine Learning Model). Vision tracking layer kept on top of Core ML detection so boxes remain attached between passes.
- Additional fix required: .mlmodel file was in the Xcode navigator but not checked under Target Membership. Fixed by clicking the model file, opening File Inspector, and checking the CameraAppPrototype target under Target Membership.

Result: Build succeeded. The app now correctly identifies and labels real objects in view including people, bottles, cups, and more.

UI Changes (ContentView.swift)

- The green bounding box header continues to show the model's detected label and confidence percentage.

- When a user labels an object (e.g. "max"), a yellow tag with that label appears beside the object's bounding box.
- Added labelPosition() helper function so the yellow label stays near its object and avoids going offscreen.

Label Persistence Improvement (ObjectDetector.swift)

Previously labels were fragile because each detection pass generated a new UUID for every object, causing saved labels to disappear on the next frame. Two strategies were combined to address this:

- New detections now try to reuse the previous object's UUID when bounding boxes overlap sufficiently between frames, keeping the label attached while the object stays in view.
- Labels are also saved by the model's detected category name (e.g. "person") as a fallback. If a labeled object leaves the frame and returns, the label is reapplied — but only when exactly one object of that category is visible, to avoid misidentification.

Known limitation: This is not true person recognition. To reliably identify a specific person among multiple people or after a long absence, face recognition or a dedicated person re-identification model would be required.

Result: Build succeeded. Labels now persist while objects remain in view and reattach on return for single-instance categories.

Zoom Feature Overview

Zoom is a hardware-level camera property controlled via AVFoundation, not a SwiftUI visual effect. The relevant property is videoZoomFactor on AVCaptureDevice, ranging from 1.0 (no zoom) up to the device's activeFormat.videoMaxZoomFactor. Any change to camera hardware properties requires locking the device for configuration first, making the change, then unlocking it.

Changes Made

- CameraManager.swift: added setZoom(_ factor: CGFloat) function. Locks the device, clamps the value between 1.0 and the device maximum using max(1.0, min(factor, maxZoom)), sets videoZoomFactor, then unlocks.
- ContentView.swift: added two @State variables — currentZoom (active zoom level) and lastZoom (zoom at the start of the last pinch gesture) — to correctly accumulate zoom across multiple pinch gestures.
- Pinch-to-zoom implemented via MagnificationGesture on the root ZStack. onChanged multiplies lastZoom by the current scale and calls camera.setZoom(). onEnded saves the final zoom level to lastZoom.

- Zoom buttons (+ and -) added at the bottom of the screen as a HStack, each stepping zoom by 0.5x. A zoom level indicator between the buttons displays the current zoom as e.g. "2.0x".

Result: Both pinch-to-zoom and button zoom work correctly. Zoom level display updates in real time. Build succeeded.

**Camera Freezing, Error Identified as: “Reading from public effective user settings.” **

Identified Risks (Causes): ContentView starts the session and then adds the detector output, which can race with session configuration, and the detector currently runs Vision work on every camera frame. I’m going to configure the detector output before starting the session and throttle Vision processing so the preview can keep moving.

Fixed Changes Made:

ObjectDetector: **guard** frameCount % processingInterval == 0 **else** { **return** }

private let processingInterval = 3

private let detectionInterval = 90

Camera Manager: **if** session.canSetSessionPreset(.hd1280x720) {

session.sessionPreset = .hd1280x720
}

Overview:

- ContentView.swift: attaches the detector output before starting the camera session. This avoids mutating the session while it is already starting.
- CameraManager.swift: uses hd1280x720 instead of .high, which lowers the frame size Vision/Core ML has to process.
- ObjectDetector.swift: throttles Vision work to every third frame.
- ObjectDetector.swift: spaces full Core ML detections farther apart, while tracking handles the in-between updates.

Error: Failed to send CA Event for app launch measurements for ca_event_type: 1 event_name: com.apple.app_launch_measurement.ExtendedLaunchMetrics

Fixed Changes Made:

- CameraManager.swift: capped max zoom at 2.7x, since your app/device starts freezing past that.
- ObjectDetector.swift: reduced Vision/Core ML workload:
    - processingInterval from 3 to 5
    - detectionInterval from 90 to 150

Note: Zoom Feature has been temporarily removed, since the camera on the iPad's end was freezing and stopped detecting various objects. Will revisit soon if needed.

Solution to fix the freezing (According to the XCode & SwiftUI built in Codex):

Changes in ObjectDetector.swift:
• processingInterval changed from 3 to 6, so Vision runs less often.
• detectionInterval changed from 90 to 180, so full Core ML detection runs less often.
• Added a camera-friendly pixel format for AVCaptureVideoDataOutput.
• Configured Core ML with .cpuAndNeuralEngine to reduce GPU contention with the camera preview.
• Switched tracking to .fast to reduce per-frame overhead.
Kept:
• 720p camera capture.
• No zoom feature.
• No 4-object cap.
• Detection confidence at 0.35.
• Newer yellow label/name-tag system.
Build succeeded. Test this version. The FigCaptureSourceRemote assertion is usually a symptom of the capture pipeline getting overloaded; this should reduce that without reverting to the version that boxed the corner.

2 issues occuring now, when I tap the green box it won't prompt me for a label; the second issue is that only one green box is appearing and it's the person one. No matter where I move the camera that box doesn't go away and it doesn't detect anything else either

The first set of changes to address these issues are:

- **ObjectDetector.swift**: removed the Vision object tracker from the live path. It was holding onto one stale person box, which is why the same box stayed around and didn't clear.
- **ObjectDetector.swift**: now runs fresh Core ML detection periodically and publishes the current results directly. If nothing is detected, boxes clear instead of being held forever.

- **ContentView.swift**: changed the overlay layout so the visible green box itself is the tappable area. The old layout used nested .position calls, so the tap target could be somewhere different from the box you saw.

Program was reverted back (Prior to new labeling system and zoom feature)

The Yellow Label System was re-implemented into the program, however once something is labeled it will stay that way.

Ex. If the camera recognizes a person, and you label them with the name “Max”, that label will apply to all things the camera recognizes as a person. It can be overridden by providing a new label, however that new label will then continue the streak of calling every person it recognizes with the new provided label.

Update 3 (visionOS Segmentation Development)

1. Project Overview

This notebook documents Days 3 and 4 of learning visionOS development using Xcode. The goal is to build a mixed reality app — RobotSelector — that uses ARKit scene reconstruction to allow a user to tap on a real-world industrial robot (KUKA arm) and highlight it with a colored overlay.

Objective

- Platform: Apple visionOS (Apple Vision Pro)
- Framework: SwiftUI + RealityKit + ARKit
- Use case: Industrial mixed reality — identifying and selecting machinery in a real environment
- Input method: Gaze + pinch gesture (Vision Pro spatial input)

2. Project Setup

2.1 Xcode Configuration

Created a new visionOS App project in Xcode with the following settings:

- Product Name: RobotSelector
- Interface: SwiftUI
- Initial Scene: Immersive Space
- Platform: visionOS 1.0+

2.2 Info.plist Permissions

Two privacy usage keys are required for ARKit world sensing and hand tracking. These strings appear in the system permission prompt shown to the user at first launch:

```
<key>NSWorldSensingUsageDescription</key>
 <string>Used to detect and select real-world objects like robots</string>

 <key>NSHandsTrackingUsageDescription</key>
 <string>Used to detect pinch gestures for object selection</string>
```

Note: Without these keys, the app will crash immediately when ARKit is initialized.

2.3 Issue Resolved — Duplicate @main Entry Point

When the project was created, Xcode auto-generated a file called AppleVisionPrototype.swift containing its own @main struct. This conflicted with RobotSelectorApp.swift, causing a build error:

```
error: 'main' attribute cannot be used on multiple types in a
module
```

Resolution: Deleted AppleVisionPrototype.swift. Used Xcode's Find Navigator (Cmd+Shift+F) to search for '@main' to locate the duplicate quickly.

3. App File Structure

The final project consists of four Swift source files plus Info.plist:

File	Type	Purpose
RobotSelectorApp.swift	App Entry	Defines WindowGroup and ImmersiveSpace scenes; injects AppModel
AppModel.swift	State Object	ObservableObject holding shared state: selection status, highlight entity
ContentView.swift	SwiftUI View	2D UI with Enter AR / Exit AR button and selection feedback label
ImmersiveView.swift	AR View	ARKit session, scene mesh processing, pinch-to-select logic
Info.plist	Config	Privacy permission strings for world sensing and hand tracking

4. Key Code Sections

4.1 App Entry Point — RobotSelectorApp.swift

Defines both the flat 2D window and the full immersive AR space. The .mixed immersion style enables passthrough — the user sees the real world with 3D overlays.

```
@main
 struct RobotSelectorApp: App {
     @StateObject var appModel = AppModel()

     var body: some Scene {
         WindowGroup {
             ContentView()
                 .environment(\.colorScheme, .dark)
```

```
                .environmentObject(appModel)
        }
        ImmersiveSpace(id: "RobotSpace") {
            ImmersiveView()
                .environmentObject(appModel)
        }
        .immersionStyle(selection: .constant(.mixed), in: .mixed)
    }
}
```

4.2 Shared State — AppModel.swift

A simple ObservableObject that tracks app-wide state. Marked @MainActor to ensure UI updates happen on the main thread.

```
import SwiftUI
import RealityKit
import Combine

@MainActor
class AppModel: ObservableObject {
    @Published var immersiveSpaceIsShown = false
    @Published var objectIsSelected = false
    @Published var highlightEntity: ModelEntity? = nil
}
```

Note: import Combine is required explicitly — relying on SwiftUI to re-export it caused compile errors in this environment.

4.3 ARKit Scene Mesh — ImmersiveView.swift

The core AR logic. ARKit's SceneReconstructionProvider continuously builds a 3D mesh of the environment. Each mesh chunk is turned into a RealityKit ModelEntity with collision enabled so tap gestures can register hits.

```
let sceneRecon = SceneReconstructionProvider(modes:
[.classification])

func addOrUpdateMesh(anchor: MeshAnchor) async {
    guard let meshResource = try? await MeshResource(from:
anchor) else { return }

    let entity = ModelEntity(mesh: meshResource, materials:
[OcclusionMaterial()])
    entity.transform = Transform(matrix:
anchor.originFromAnchorTransform)
    entity.generateCollisionShapes(recursive: false)
    entity.components.set(InputTargetComponent())
```

```
        meshEntities[anchor.id] = entity
        rootEntity.addChild(entity)
    }
```

4.4 Tap-to-Select — handleTap()

On pinch, the tapped mesh entity is highlighted orange. All other meshes are reset to invisible (OcclusionMaterial). Uses the ModelComponent API — the correct modern RealityKit pattern.

```
func handleTap(value: EntityTargetValue<SpatialTapGesture.Value>)
{
        // Reset all meshes
        for entity in meshEntities.values {
            if var mc = entity.components[ModelComponent.self] {
                mc.materials = [OcclusionMaterial()]
                entity.components.set(mc)
            }
        }
        // Highlight tapped entity
        var mat = UnlitMaterial()
        mat.color = .init(tint: UIColor(red: 1.0, green: 0.4, blue:
0.0, alpha: 0.5))
        let tapped = value.entity
        if var mc = tapped.components[ModelComponent.self] {
            mc.materials = [mat]
            tapped.components.set(mc)
        }
        appModel.objectIsSelected = true
    }
```

5. Bugs Encountered & Resolutions

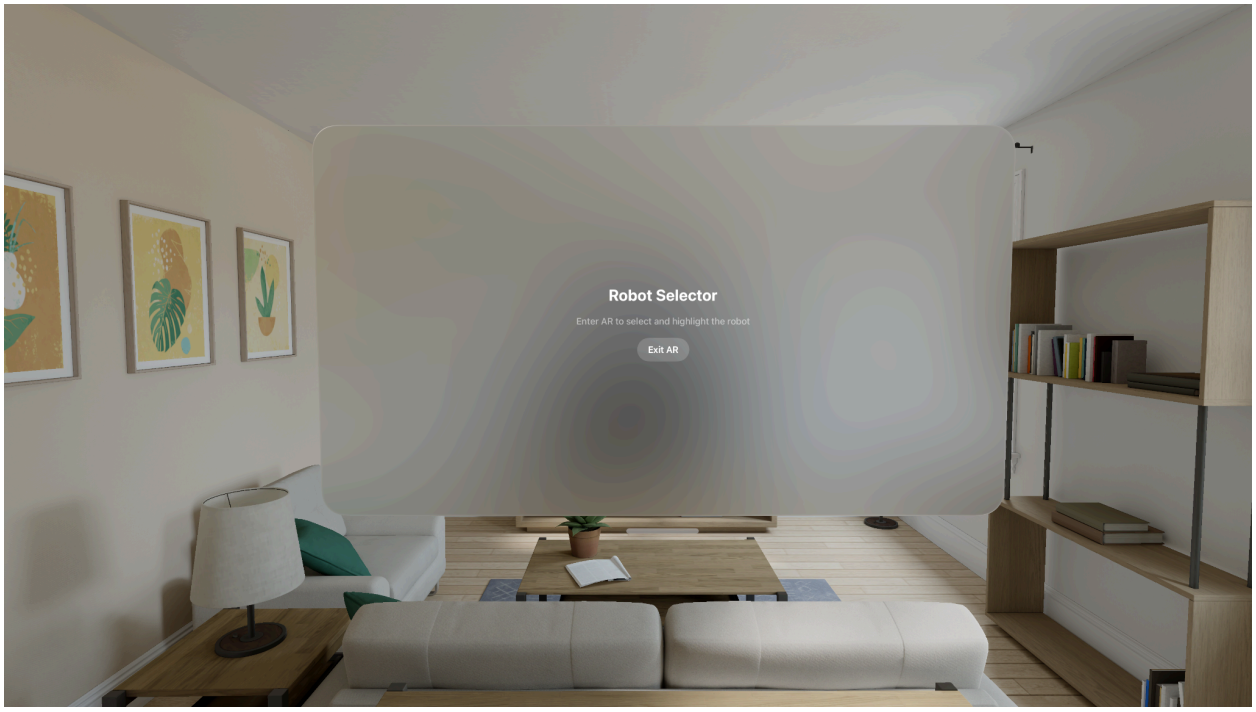
Error	Cause	Fix
'main' attribute on multiple types	Xcode auto-generated AppleVisionPrototype.swift with its own @main	Deleted the auto-generated file; kept RobotSelectorApp.swift
Cannot find 'rootEntity' in scope	Missing @State property declaration in ImmersiveView	Added @State private var rootEntity = Entity()
'Entity' has no member 'model'	Plain Entity has no .model property; only ModelEntity does	Switched to components[ModelComponent.self] pattern

6. Simulator Testing Notes

The app was successfully built and launched in the visionOS Simulator. The following console messages appeared and are all expected behavior on the simulator:

- `AddInstanceForFactory: No factory registered...` → Harmless internal Apple system log — always appears
- `nw_socket_copy_info getsockopt TCP_INFO failed` → Networking socket info the simulator cannot replicate — harmless
- `Authorization requests cannot be performed on Simulator` → ARKit cannot request permissions on simulator — expected
- `World sensing not authorized` → Our own guard print() firing correctly — expected on simulator

Scene mesh reconstruction and pinch-to-select require real Vision Pro hardware and cannot be tested in the simulator.



7. Concepts Learned

ImmersiveSpace A full-environment scene type unique to visionOS. With .mixed immersion, passthrough is enabled and the user sees the real world.

SceneReconstructionProvider ARKit provider that continuously generates a 3D mesh of the surrounding environment, including furniture and machinery.

MeshAnchor Represents one chunk of the scene mesh. Updated in real time as the user moves through space.

OcclusionMaterial A RealityKit material that is invisible but blocks rendering of objects behind it — used to give scene meshes physical presence without being visible.

InputTargetComponent Must be added to any entity you want to receive gesture input. Without it, SpatialTapGesture will not fire on the entity.

ModelComponent API The correct way to read and write materials on a RealityKit entity. Access via entity.components[ModelComponent.self], mutate, then set back.

@MainActor Swift concurrency annotation ensuring a class or function runs on the main thread — required for any code that updates UI or RealityKit entities.

8. Next Steps

- Test on physical Apple Vision Pro hardware to verify scene mesh and pinch selection
- Add floating text label that appears above the robot when selected
- Explore proximity-based auto-highlighting using ARKit transforms
- Investigate CoreML integration for automatic robot detection by color/shape (orange segmentation)
- Add spatial audio feedback on selection

Update 4 (iPad OS Support)

Note: Currently I don’t have access to apple’s developer mode, so code cannot be correctly pushed onto the apple vision pro

1. iPad Version — Motivation

Since the visionOS app cannot be tested without a physical Apple Vision Pro, an iPad version of RobotSelector was developed to allow immediate real-world testing of the core concept — tapping on the KUKA robot to highlight it in AR.

The iPad version uses the same underlying frameworks (ARKit and RealityKit) but adapts the interaction model from gaze+pinch to screen tap, and replaces visionOS-specific APIs with their iPadOS equivalents.

2. Platform Comparison

Concept	visionOS Version	iPad Version
Entry Point	ImmersiveSpace + WindowGroup	WindowGroup only
AR View	RealityView	ARView via UIViewRepresentable
Gestures	SpatialTapGesture	UITapGestureRecognizer
Mesh Access	SceneReconstructionProvider	ARWorldTrackingConfiguration
Hit Detection	Collision on mesh entities	arView.raycast()
Highlight	Material on mesh entity	Sphere entity at hit point

3. iPad File Structure

File	Type	Purpose

RobotSelectorARApp.swift	App Entry	Minimal entry point — no ImmersiveSpace needed on iPad
ARViewModel.swift	State + Logic	Owns ARView instance, starts ARKit session, handles raycasting and highlight placement
ARViewContainer.swift	UIViewRepresentable	Bridges ARView (UIKit) into SwiftUI; sets up tap gesture recognizer via Coordinator pattern
ContentView.swift	SwiftUI View	Full-screen AR view with overlay UI — status message, selection indicator, clear button

4. Key iPad Code Sections

12.1 ARKit Session Setup — ARViewModel.swift

The session checks for LiDAR availability at runtime. LiDAR iPads (iPad Pro) get a dense scene mesh; other iPads fall back to plane detection only.

```
func startSession() {
     let config = ARWorldTrackingConfiguration()

     if
ARWorldTrackingConfiguration.supportsSceneReconstruction(.mesh) {
         config.sceneReconstruction = .mesh
         statusMessage = "LiDAR detected — high quality mesh
active"
     } else {
         statusMessage = "Point camera at the robot and tap it"
     }

     config.environmentTexturing = .automatic
     config.planeDetection = [.horizontal, .vertical]
     arView.session.run(config)
 }
```

12.2 Raycast Hit Detection

Unlike visionOS which uses collision shapes on mesh entities, on iPad a raycast is fired from the 2D tap point into the 3D world. The first hit result gives a world-space transform where the highlight is placed.

```
func handleTap(at point: CGPoint) {
     let results = arView.raycast(
         from: point,
         allowing: .estimatedPlane,
         alignment: .any
     )
     guard let hit = results.first else {
         statusMessage = "Missed — try tapping directly on the
robot"
         return
     }
     placeHighlight(at: hit.worldTransform)
     objectIsSelected = true
     statusMessage = "Robot selected!"
 }
```

12.3 Highlight Placement

An orange UnlitMaterial sphere is placed at the world-space hit point. Previous highlights are removed by name before placing a new one.

```
func placeHighlight(at transform: simd_float4x4) {
     // Remove previous highlight
     arView.scene.anchors.forEach { anchor in
         if anchor.name == "highlight" {
             arView.scene.removeAnchor(anchor)
         }
     }
     // Place orange sphere
     let mesh = MeshResource.generateSphere(radius: 0.05)
     var material = UnlitMaterial()
     material.color = .init(tint: UIColor(red: 1.0, green: 0.4,
blue: 0.0, alpha: 0.8))
     let sphere = ModelEntity(mesh: mesh, materials: [material])
     let anchor = AnchorEntity(world: transform)
     anchor.name = "highlight"
     anchor.addChild(sphere)
     arView.scene.addAnchor(anchor)
 }
```

12.4 UIViewRepresentable Bridge — ARViewContainer.swift

ARView is a UIKit component. The Coordinator pattern is used to bridge UIKit gesture recognizers into the SwiftUI/ViewModel world.

```
struct ARViewContainer: UIViewRepresentable {
     @ObservedObject var viewModel: ARViewModel

     func makeUIView(context: Context) -> ARView {
         let tapGesture = UITapGestureRecognizer(
             target: context.coordinator,
             action: #selector(Coordinator.handleTap(_:))
         )
         viewModel.arView.addGestureRecognizer(tapGesture)
         viewModel.startSession()
         return viewModel.arView
     }

     func makeCoordinator() -> Coordinator {
         Coordinator(viewModel: viewModel)
     }

     class Coordinator: NSObject {
         let viewModel: ARViewModel
         init(viewModel: ARViewModel) { self.viewModel = viewModel
}

         @objc func handleTap(_ gesture: UITapGestureRecognizer) {
             let point = gesture.location(in: viewModel.arView)
             viewModel.handleTap(at: point)
         }
     }
 }
```

5. New Concepts Learned — iPad

UIViewRepresentable A SwiftUI protocol that wraps a UIKit view so it can be used inside a SwiftUI layout. Required because ARView is a UIKit component.

Coordinator Pattern A nested class inside UIViewRepresentable that acts as the delegate/target for UIKit callbacks, bridging them into Swift/SwiftUI code.

ARWorldTrackingConfiguration The main ARKit configuration for iPad. Tracks the device's position and orientation in 3D space and optionally builds a scene mesh.

arView.raycast() Fires a ray from a 2D screen point into the 3D AR world and returns hit results — the iPad equivalent of visionOS collision detection.

AnchorEntity(world:) Places a RealityKit entity at an absolute world-space position, independent of any detected surface or plane.

LiDAR vs Non-LiDAR iPad Pro models with LiDAR get dense scene reconstruction via sceneReconstruction = .mesh. Other iPads rely on plane detection, which is less precise but still functional.

6. Updated Next Steps

- Build and run RobotSelectorAR on physical iPad to test raycast selection
- Compare mesh quality between LiDAR and non-LiDAR devices
- Add floating text annotation above the selected robot
- Test on physical Apple Vision Pro once available
- Investigate CoreML for automatic orange color segmentation
- Explore sharing code between visionOS and iPadOS targets in a single Xcode project
- Ask for developer ID
- Push and test code on the Apple Vision Pro by using a kruppa robot as a base model (testing segmentation)

Update 5 (InfluxDB Integration)

RobotSelectorAR — iPadOS Port

InfluxDB

9. iPad Version — Motivation

Since the visionOS app cannot be tested without a physical Apple Vision Pro, an iPad version of RobotSelector was developed to allow immediate real-world testing of the core concept — tapping on the KUKA robot to highlight it in AR.

The iPad version uses the same underlying frameworks (ARKit and RealityKit) but adapts the interaction model from gaze+pinch to screen tap, and replaces visionOS-specific APIs with their iPadOS equivalents.

10. Platform Comparison

Concept	visionOS Version	iPad Version
Entry Point	ImmersiveSpace + WindowGroup	WindowGroup only
AR View	RealityView	ARView via UIViewRepresentable
Gestures	SpatialTapGesture	UITapGestureRecognizer
Mesh Access	SceneReconstructionProvider	ARWorldTrackingConfiguration
Hit Detection	Collision on mesh entities	arView.raycast()
Highlight	Material on mesh entity	Sphere entity at hit point

11. iPad File Structure

File	Type	Purpose
RobotSelectorARApp.swift	App Entry	Minimal entry point — no ImmersiveSpace needed on iPad
ARViewModel.swift	State + Logic	Owns ARView instance, starts ARKit session, handles raycasting and highlight placement
ARViewContainer.swift	UIViewRepresentable	Bridges ARView (UIKit) into SwiftUI; sets up tap gesture recognizer via Coordinator pattern
ContentView.swift	SwiftUI View	Full-screen AR view with overlay UI — status message, selection indicator, clear button

12. Key iPad Code Sections

12.1 ARKit Session Setup — ARViewModel.swift

The session checks for LiDAR availability at runtime. LiDAR iPads (iPad Pro) get a dense scene mesh; other iPads fall back to plane detection only.

```
func startSession() {
     let config = ARWorldTrackingConfiguration()

     if
ARWorldTrackingConfiguration.supportsSceneReconstruction(.mesh) {
         config.sceneReconstruction = .mesh
         statusMessage = "LiDAR detected — high quality mesh
active"
     } else {
         statusMessage = "Point camera at the robot and tap it"
     }

     config.environmentTexturing = .automatic
     config.planeDetection = [.horizontal, .vertical]
     arView.session.run(config)
 }
```

12.2 Raycast Hit Detection

Unlike visionOS which uses collision shapes on mesh entities, on iPad a raycast is fired from the 2D tap point into the 3D world. The first hit result gives a world-space transform where the highlight is placed.

```
func handleTap(at point: CGPoint) {
     let results = arView.raycast(
         from: point,
         allowing: .estimatedPlane,
         alignment: .any
     )
     guard let hit = results.first else {
         statusMessage = "Missed — try tapping directly on the
robot"
         return
     }
     placeHighlight(at: hit.worldTransform)
     objectIsSelected = true
     statusMessage = "Robot selected!"
 }
```

12.3 Highlight Placement

An orange UnlitMaterial sphere is placed at the world-space hit point. Previous highlights are removed by name before placing a new one.

```
func placeHighlight(at transform: simd_float4x4) {
     // Remove previous highlight
     arView.scene.anchors.forEach { anchor in
         if anchor.name == "highlight" {
             arView.scene.removeAnchor(anchor)
         }
     }
     // Place orange sphere
     let mesh = MeshResource.generateSphere(radius: 0.05)
     var material = UnlitMaterial()
     material.color = .init(tint: UIColor(red: 1.0, green: 0.4,
blue: 0.0, alpha: 0.8))
     let sphere = ModelEntity(mesh: mesh, materials: [material])
     let anchor = AnchorEntity(world: transform)
     anchor.name = "highlight"
     anchor.addChild(sphere)
     arView.scene.addAnchor(anchor)
 }
```

12.4 UIViewRepresentable Bridge — ARViewContainer.swift

ARView is a UIKit component. The Coordinator pattern is used to bridge UIKit gesture recognizers into the SwiftUI/ViewModel world.


```
struct ARViewContainer: UIViewRepresentable {
     @ObservedObject var viewModel: ARViewModel
```

```
    func makeUIView(context: Context) -> ARView {
        let tapGesture = UITapGestureRecogniz
```
Update 7 - VisionOS Upload

---

27. visionOS Full Implementation — Overview

The complete visionOS version of RobotSelector combines everything built so far: ARKit scene mesh selection, Vision framework foreground segmentation, and InfluxDB live telemetry, all rendered through RealityKit as a floating AR overlay above the selected robot.

28. Complete visionOS File Structure

File	Purpose
RobotSelectorApp.swift	App entry point — defines WindowGroup and ImmersiveSpace
AppModel.swift	Shared state — selection status, highlight entity
ContentView.swift	2D entry window with Enter AR button
ImmersiveView.swift	Core AR view — ARKit session, mesh, tap selection, segmentation trigger, InfluxDB polling, overlay attachment
SegmentationHelper.swift	Runs Apple Vision foreground segmentation on camera frames
InfluxDBClient.swift	HTTP client — writes and queries InfluxDB using line protocol and Flux
SensorDataModel.swift	Holds live sensor values, manages 2-second polling timer

RobotDataOverlay.swift	Floating SwiftUI panel showing live temperature, speed, load

29. Runtime Flow

1. User puts on Vision Pro and launches the app — 2D window appears with Enter AR button
2. ImmersiveView opens in .mixed immersion style — passthrough AR begins
3. ARKitSession starts world sensing and camera access; SceneReconstructionProvider silently builds a 3D mesh of the room
4. User gazes at the robot and pinches — SpatialTapGesture fires and raycasts into the mesh
5. handleTap() highlights the tapped mesh chunk orange via ModelComponent materials
6. runSegmentation() captures the latest camera frame and runs Vision's foreground segmentation mask
7. applySegmentationMask() renders the mask as a semi-transparent orange overlay plane in the scene
8. attachDataOverlay() places a floating RobotDataOverlay panel 0.5m above the tapped entity
9. SensorDataModel starts a 2-second polling timer that queries InfluxDB via InfluxDBClient
10. RobotDataOverlay displays live temperature, speed, and load with color-coded thresholds, refreshing every 2 seconds

30. Complete Code Listings

30.1 RobotSelectorApp.swift

swift

```swift
@main
struct RobotSelectorApp: App {
    @StateObject var appModel = AppModel()

    var body: some Scene {
        WindowGroup {
            ContentView()
```

```
            .environment(\.colorScheme, .dark)
            .environmentObject(appModel)
        }
        ImmersiveSpace(id: "RobotSpace") {
            ImmersiveView()
            .environmentObject(appModel)
        }
        .immersionStyle(selection: .constant(.mixed), in: .mixed)
    }
}
```

30.2 SegmentationHelper.swift

Wraps Apple's Vision framework foreground instance mask request. Runs on a camera frame and returns a UIImage mask where white pixels represent foreground objects.

swift

```
class SegmentationHelper {
    func segment(pixelBuffer: CVPixelBuffer) async -> UIImage? {
        return await withCheckedContinuation { continuation in
            let request = VNGenerateForegroundInstanceMaskRequest()
            let handler = VNImageRequestHandler(cvPixelBuffer: pixelBuffer, options: [:])
            do {
                try handler.perform([request])
                guard let result = request.results?.first else {
                    continuation.resume(returning: nil)
                    return
                }
```

```
            let mask = try result.generateScaledMaskForImage(
                forInstances: result.allInstances,
                from: handler
            )
            let ciImage = CIImage(cvPixelBuffer: mask)
            let context = CIContext()
            guard let cgImage = context.createCGImage(ciImage, from: ciImage.extent) else {
                continuation.resume(returning: nil)
                return
            }
            continuation.resume(returning: UIImage(cgImage: cgImage))
        } catch {
            continuation.resume(returning: nil)
        }
    }
  }
}
```

Note: this is generic foreground segmentation, not a KUKA-specific detector — see Section 23.3 for the Codex recommendation to replace this with a trained CoreML model.

30.3 ImmersiveView.swift — Key Sections

Properties added for segmentation and InfluxDB, alongside the existing ARKit/mesh properties:

swift

```
let cameraFrameProvider = CameraFrameProvider()
let segmentationHelper = SegmentationHelper()
@State private var segmentationOverlay: ModelEntity? = nil
```

```
@StateObject private var sensorData = SensorDataModel()

private let influxClient = InfluxDBClient()
```

handleTap() now triggers three things after the highlight: segmentation, the data overlay, and InfluxDB polling:

swift

```
appModel.objectIsSelected = true

Task { await runSegmentation() }

attachDataOverlay(to: tappedEntity)

sensorData.startPolling(client: influxClient)
```

Segmentation pipeline — captures a camera frame, runs the Vision mask, and renders it as a textured plane in front of the user:

swift

```
func runSegmentation() async {

    guard let cameraFrame = await getLatestCameraFrame() else { return }

    guard let mask = await segmentationHelper.segment(pixelBuffer: cameraFrame) else { return
}

    await applySegmentationMask(mask)

}


func applySegmentationMask(_ mask: UIImage) async {

    segmentationOverlay?.removeFromParent()

    guard let cgImage = mask.cgImage,

        let texture = try? await TextureResource(image: cgImage, options: .init(semantic: .color))

    else { return }


    var material = UnlitMaterial()
```

```
    material.color = .init(
        tint: UIColor(red: 1.0, green: 0.4, blue: 0.0, alpha: 0.4),
        texture: .init(texture)
    )
    let plane = ModelEntity(mesh: .generatePlane(width: 1.0, height: 1.0), materials: [material])
    plane.position = [0, 0, -1.5]
    plane.name = "segmentationOverlay"
    segmentationOverlay = plane
    rootEntity.addChild(plane)
}
```

Data overlay attachment — positions the SwiftUI panel 0.5m above the selected entity using ViewAttachmentComponent:

swift

```
func attachDataOverlay(to entity: Entity) {
    rootEntity.children.filter { $0.name == "dataOverlay" }.forEach { $0.removeFromParent() }
    let overlayEntity = ModelEntity()
    overlayEntity.name = "dataOverlay"
    var transform = entity.transform
    transform.translation.y += 0.5
    overlayEntity.transform = transform
    overlayEntity.components.set(
        ViewAttachmentComponent(swiftUIView: AnyView(RobotDataOverlay(sensorData: sensorData)))
    )
    rootEntity.addChild(overlayEntity)
```

```
}
```

30.4 InfluxDBClient.swift and SensorDataModel.swift

Identical implementation to the iPad version documented in Section 17 — write() uses InfluxDB line protocol, queryLatest() uses a Flux query, and SensorDataModel manages the 2-second Combine polling timer. The same file works unchanged across both platforms.

30.5 RobotDataOverlay.swift

A SwiftUI panel showing temperature, speed, and load with color-coded thresholds (green/orange/red), attached to a RealityKit entity rather than shown in a 2D window. Identical thresholds to the iPad version documented in Section 19.

31. Understanding RealityKit

RealityKit is Apple's framework for rendering and simulating 3D content in AR, providing photo-realistic rendering, camera effects, animations, and physics. It works alongside ARKit but handles a different layer of the pipeline.

31.1 RealityKit vs ARKit

ARKit provides the fundamental tracking and scene understanding — device position and orientation, plane detection, lighting estimation, and in this app's case, scene mesh reconstruction. RealityKit builds on top of that data to handle rendering and animation, turning raw ARKit data into visible, interactive 3D content.

31.2 Core Concepts

Scene graph — A hierarchical structure representing relationships between objects in the 3D scene.

Entities — The basic building blocks of a RealityKit scene — every mesh chunk, highlight, and overlay panel in this app is an Entity (specifically a ModelEntity).

Components — Define properties and behavior of entities. This app uses ModelComponent (mesh + materials), InputTargetComponent (enables gesture hits), and ViewAttachmentComponent (embeds SwiftUI views in 3D space).

Materials — Define visual appearance — color, texture, reflectivity. This app uses OcclusionMaterial (invisible but blocks rendering, used for raw mesh) and UnlitMaterial (flat color, used for the orange highlight and segmentation overlay).

31.3 RealityKit's Role in This App

- Converts raw ARKit MeshAnchor data into tappable ModelEntity objects
- Applies materials for the orange selection highlight and segmentation overlay
- Enables collision detection so pinch gestures can hit the room mesh
- Hosts the SwiftUI RobotDataOverlay panel in 3D space via ViewAttachmentComponent

31.4 Cross-Platform Note

RealityKit works across visionOS, iOS, iPadOS, macOS, and tvOS, allowing the same RealityKit code (entities, components, materials) to be written once and used on both the iPad and Vision Pro versions of this app, even though the underlying AR session setup (ARWorldTrackingConfiguration vs ARKitSession + SceneReconstructionProvider) differs between platforms.

32. Updated Next Steps

- Fill in InfluxDB credentials in InfluxDBClient.swift for both iPad and visionOS targets
- Test segmentation overlay rendering performance on visionOS — camera frame capture rate vs Vision processing time
- Replace Vision foreground segmentation with a trained CoreML model for KUKA-specific detection
- Solve multi-robot identification using spatial position mapping before CoreML instance segmentation is ready
- Test full visionOS build on physical Apple Vision Pro hardware once available
- Consider extracting shared RealityKit material/component logic into a common Swift package for both platforms

Update 6 (ML/Core Learning)

Codex:

1. Robot Data Overlay & Live AR Integration

• Your app overlays live KUKA robot sensor data (temperature, speed, load) in a card UI, only when a robot is detected in the AR view.
• We improved your RobotDataOverlay with a .live(...) helper to automatically start/stop data polling according to the overlay's presence, making UI management easier.

———

2. Sensor Data Model Safeguards

• Fixed the SensorDataModel so it cannot start duplicate polling timers, improving stability.
• Removed a problematic stopPolling() call from deinit (it's a @MainActor method, which cannot be called from a Swift class de-initializer).

———

3. Robot Detection Logic

• Explained that your app is designed to use ML (a YOLO detector) to recognize the robot from the camera feed.
• The loop that ran inference on each frame was re-activated, but we identified that ARKitSession on visionOS doesn't provide the currentFrame directly.
• You now need to use an ARKit delegate or data provider to supply camera frames for ML detection—this is a to-do for robust robot detection in the visionOS environment.

———

4. Custom YOLO Training Guide

• Explained that robust detection requires gathering and labeling photos of your actual robot.
• Provided a step-by-step guide for training YOLOv5/YOLOv8 (recommended) and converting the model for use on Apple Vision Pro.
• For YOLOv3, clarified that collecting/labelling images and exporting in YOLO format is key; optionally, cloud tools like Roboflow can be used to manage this workflow.

———

5. InfluxDB Client Review

• Walk through your InfluxDBClient for reading/writing robot sensor data.
• Reminded you to fill in real InfluxDB credentials and ensure your database and permissions are correct.
• Note that to use real data in your app, you must provide a SensorDataSource implementation that wraps InfluxDBClient and wire it up in your view model/overlay.

———

6. General App Architecture

• Advised on where a state like objectIsSelected should be set/reset for correct overlay presentation.
• Emphasized the importance of the camera ML pipeline, sensor data polling, and error handling for a robust end-to-end experience.

Links to KUPPA Videos:

- **FRONT/SIDE**
  - https://drive.google.com/file/d/13VGl5DiAXWZNjs2KiiAJw0ayfxP-JdpL/edit
- **BACK/BOTTOM**
  - https://drive.google.com/file/d/15jsBXkvVhRtOAxRrO-3uEwV4nGhx9zLa/edit

Update 7 (RoboFlow)

****NOTE - Confidence Threshold must be between 20% - 88% for the model to correctly recognize a KUKA Robot**

This update adds live machine learning inference to the visionOS RobotSelector app using a custom-trained Roboflow instance segmentation model. Rather than relying on Apple Vision's generic foreground segmentation, the app now sends camera frames to the Roboflow Serverless API every 1.5 seconds and receives back segmentation predictions specific to the KUKA robots.

Roboflow project: https://app.roboflow.com/yuvrajs-workspace-kuppa/kuka-robot-detection/1

What Was Built

RoboflowDetectionService.swift

A new Swift service that handles all communication with the Roboflow Serverless REST API:

- Accepts UIImage or CGImage as input
- Converts the image to JPEG and base64-encodes it for the HTTP POST request
- Sends the request to the Roboflow inference endpoint for the KUKA robot detection model
- Parses the JSON response — extracting predictions including class labels, confidence scores, and segmentation polygon coordinates
- Returns an array of RobotDetection objects filtered to 'KUKA robot' class with confidence >= 0.5
- Triggers telemetry logic when a qualifying detection is found
- Prints all detection results to the Xcode console for debugging

API Key Security

To keep the Roboflow API key out of source control, the following pattern was adopted:

- API key stored in an .xcconfig file that is excluded from git via .gitignore
- The .xcconfig value is referenced in Info.plist using $(ROBOFLOW_API_KEY) substitution
- The Swift code reads the key at runtime from Bundle.main.infoDictionary rather than hardcoding it

```
// Reading key at runtime — never hardcode in source
 let apiKey = Bundle.main.infoDictionary?["ROBOFLOW_API_KEY"] as?
String ?? ""
```

ImmersiveView Pipeline Integration

ImmersiveView.swift was updated with three new capabilities:

State Management

- latestPixelBuffer — stores the most recent camera frame as CVPixelBuffer
- roboflowResults — stores the latest array of RobotDetection objects
- A Cancellable property to manage the inference timer lifecycle

Live Camera Frame Acquisition

- ARKit session updated to run both SceneReconstructionProvider and WorldTrackingProvider simultaneously
- A .task block streams camera frames from WorldTrackingProvider as CVPixelBuffer
- Each incoming frame updates latestPixelBuffer so the inference timer always has the freshest image available

Live Inference Timer

- A second .task block fires every 1.5 seconds
- Converts the latest CVPixelBuffer to CGImage
- Passes it to RoboflowDetectionService for inference
- Prints detection results to the Xcode console

The 1.5 second interval was chosen to balance detection responsiveness against Roboflow API rate limits.

Complete Detection Pipeline

The full pipeline from camera to telemetry now works as follows:

1. Vision Pro camera streams frames via WorldTrackingProvider as CVPixelBuffer
2. ImmersiveView stores each frame in latestPixelBuffer
3. Inference timer fires every 1.5 seconds — converts frame to CGImage
4. RoboflowDetectionService encodes image as JPEG and sends HTTP POST to Roboflow API
5. Roboflow returns JSON with class labels, confidence scores, and segmentation polygon coordinates
6. Service filters results for KUKA robot class with confidence >= 0.5
7. Qualified detections trigger telemetry logic — InfluxDB polling starts for the detected machine ID
8. RobotDataOverlay displays live temperature, speed, and load above the robot

Updated visionOS File Structure

File	Status / Purpose
RobotSelectorApp.swift	✅ No change
AppModel.swift	✅ No change
ContentView.swift	✅ No change
ImmersiveView.swift	🔄 Updated — WorldTrackingProvider added; latestPixelBuffer state; 1.5s inference timer task
SegmentationHelper.swift	⚠️ Superseded — generic Vision segmentation replaced by Roboflow inference
RoboflowDetectionService.swift	🆕 New — Roboflow Serverless API client; JPEG encoding; JSON parsing; confidence filtering
InfluxDBClient.swift	✅ No change
SensorDataModel.swift	✅ No change
RobotDataOverlay.swift	✅ No change

New Concepts Learned

Roboflow Serverless API A REST endpoint provided by Roboflow that runs inference on a hosted model. No local GPU required — the model runs in Roboflow's cloud and returns predictions as JSON.

WorldTrackingProvider An ARKit provider on visionOS that tracks device position and orientation. Also provides access to camera frames as CVPixelBuffer, which is what the inference pipeline uses as input.

Instance Segmentation vs Detection Object detection returns bounding boxes. Instance segmentation additionally returns polygon coordinates tracing the exact outline of each detected object — enabling a precise visual mask, not just a rectangle.

Confidence Threshold A minimum confidence score (set to 0.5 here) below which predictions are discarded. Prevents spurious detections from triggering telemetry.

.xcconfig API Key Pattern A standard iOS/visionOS pattern for keeping secrets out of source code. Values defined in an .xcconfig file are injected into Info.plist at build time and read at runtime via Bundle.main.infoDictionary.

CVPixelBuffer → CGImage The conversion chain needed to get from ARKit's raw camera frame format (CVPixelBuffer) to a format that can be JPEG-encoded and sent to a REST API. Uses CIImage and CIContext as intermediaries.

Current State of the App

- Live, automatic ML inference running on the real Vision Pro camera feed every 1.5 seconds **- Apple doesn't allow for third-party ML/AI into the current model of visionOS *finding solutions now***
- KUKA-specific detection via custom-trained Roboflow model — not generic foreground segmentation
- API key stored securely using .xcconfig pattern — not in source control
- Detection results available for AR overlay, InfluxDB telemetry routing, and console debugging
- Modern visionOS architecture using WorldTrackingProvider + SceneReconstructionProvider together
- Foundation in place for visual AR overlays, multi-robot identity routing, and advanced robotics workflows

Next Steps

- Wire detected robot identity (kuka_1 / kuka_2) into InfluxDB query filter for correct telemetry routing
- Add visual AR overlay drawn from Roboflow segmentation polygon coordinates
- Continue labeling images in Roboflow to improve model accuracy and recall
- Test detection performance at different distances and lighting conditions in the facility
- Fill in InfluxDB credentials once instance is configured
- Consider caching the last good detection so the overlay persists between 1.5s inference cycles

Update 8 (Live Camera Feed + Testing Fixtures)

Platform limitation discovered — camera access restrictions on visionOS

Discovery — visionOS Camera Access Restriction

During implementation of the live Roboflow inference pipeline, a fundamental platform limitation was identified: as of visionOS 1.0–1.2, Apple does not provide a documented public API to stream live camera frames (CVPixelBuffer) directly to an app for ML inference. This is an intentional privacy and security restriction, not a bug or oversight.

> ⚠️ Important: Raw camera feed access for general ML inference is not available on visionOS as of v1.0–1.2. Apps cannot obtain CVPixelBuffer frames from the camera in the same way iOS apps can via AVFoundation.

Why Apple Restricts This

The Vision Pro is a mixed reality headset that sees everything in the user's environment — their home, workplace, and surroundings. Granting apps unrestricted access to the live camera feed would represent a significant privacy risk. Apple's approach is to process visual data on-device at the system level (for features like eye tracking, hand tracking, and scene reconstruction) and surface only specific, permissioned abstractions to developers rather than the raw pixel stream.

This is fundamentally different from iOS, where AVFoundation gives apps direct access to camera frames for ML inference, AR, and video recording.

What IS Available on visionOS

While raw camera frames are not accessible, the following higher-level APIs are available and were used in this project:

API / Provider	What It Gives You
SceneReconstructionProvider	3D mesh of the environment — geometry only, no color/texture pixel data
WorldTrackingProvider	Device position and orientation in 3D space; no raw camera frames
ARKit hand tracking	Joint positions for both hands — no camera pixels
VNGenerateForegroundInstanceMaskRequest	Generic foreground segmentation run by the system — result only, not the underlying frame

What to Watch For — Future API Access

Apple may introduce a supported camera frame provider in a future visionOS release. When this happens, the expected pattern would look similar to:

```
// Hypothetical future API — not yet available
 // Watch WWDC sessions and Apple developer documentation for:

 // Option A — a dedicated frame provider
 for await frame in cameraFrameProvider.frames {
     let pixelBuffer = frame.pixelBuffer
     await runInference(on: pixelBuffer)
 }

 // Option B — a FrameStreamProvider or CameraFeedProvider
 let provider = FrameStreamProvider()
 for await sample in provider.cameraFrameUpdates(for: .left) {
     let pixelBuffer = sample.pixelBuffer
     await runInference(on: pixelBuffer)
 }
```

Resources to monitor for updates:

- developer.apple.com/documentation/visionos — official visionOS API documentation
- WWDC sessions each June — Apple typically announces new privacy-gated APIs here first
- Apple Developer Forums — visionOS category for early reports from other developers
- ARKit release notes — camera-related APIs often appear here first

Impact on This Project

This limitation directly affects the Roboflow live inference pipeline designed in Update 8. The inference timer loop was built assuming camera frames would be available via WorldTrackingProvider, but this is not currently supported. The pipeline code is structured and ready — it simply needs a valid CVPixelBuffer source to function end-to-end.

Current workarounds under consideration:

Use ARKit mesh + tap selection only

The mesh-based pinch-to-select system (already working on Vision Pro) does not require camera frames. Robot selection works via geometry, not pixels. This is the current fallback.
Tradeoff: Loses pixel-level segmentation — acceptable for now

Trigger inference on captured still frames

Instead of a live stream, capture a single frame at the moment of tap using an available system mechanism, run inference once, then display the result. Less real-time but avoids the continuous stream requirement.
Tradeoff: Depends on whether any single-frame capture API is available

Wait for Apple to expose the API

The cleanest long-term solution. The code is already structured to drop in a CVPixelBuffer source — when Apple supports it, minimal changes are needed.

Tradeoff: Timeline unknown — monitor WWDC

Key Concepts Learned

Privacy-First Platform Design Apple intentionally restricts raw sensor access on Vision Pro to protect user privacy. Developers receive processed, permissioned abstractions rather than raw camera pixels.

Platform Limitation vs Bug Not all missing functionality is a bug. Some limitations are deliberate platform decisions that may be revisited in future OS versions. Recognizing the difference informs how to structure code to be forward-compatible.

Forward-Compatible Architecture By separating the inference logic (RoboflowDetectionService) from the frame acquisition logic (ImmersiveView), the app is structured so that adding a real CVPixelBuffer source in the future requires minimal code changes.

Development strategies while live camera access remains unavailable

Summary of Restriction

⚠️ There is currently no way for third-party developers to access live camera frames for ML in visionOS. Private API hacks are not supported, are not App Store-compliant, and are not safe. The only correct path is to wait for Apple to officially expose this capability.

This update documents the recommended development strategies to continue building and testing the Roboflow detection pipeline while the live camera restriction remains in place. The key principle: the app code is already structured correctly — it just needs a valid CVPixelBuffer source, which Apple will eventually provide.

What to Watch For in Future visionOS Releases

When monitoring Apple developer documentation, WWDC sessions, and SDK release notes, look specifically for new APIs that mention any of the following:

- Camera feed or camera frame access
- Frame streaming or FrameStreamProvider
- ML or computer vision support in ARKit providers
- CVPixelBuffer or ARFrame access from a provider
- A .frames property on any ARKit provider
- CameraFeedProvider or any similar named type

When such an API appears, the integration into this app is straightforward:

```
// When Apple exposes a frame stream — drop this in to
ImmersiveView
 // Your Roboflow detection loop will then automatically process
real-time frames

 for await frame in someAppleProvider.frames {
     self.latestPixelBuffer = frame.pixelBuffer
```

```
    // The 1.5s inference timer picks this up automatically
}
```

Current Development Workarounds

Four strategies are available to continue development and testing without a live camera feed:

Workaround 1 — Static Test Images (Primary)

The most practical approach for immediate development. Add KUKA robot images to the Xcode asset catalog and use them as stand-in frames for the Roboflow detection pipeline.

Name the images with the exact pattern shown below and add them to your asset catalog:

```
// Asset catalog image names
"robot-test-1"   // front view of KUKA arm
"robot-test-2"   // side view
"robot-test-3"   // different angle or distance
// Add more as needed
```

The inference timer loads the next image every 1.5 seconds, cycles through the list, converts it to CGImage, and passes it to RoboflowDetectionService — simulating a video feed for development purposes. Output is printed to the Xcode console and overlays render normally.

> 📌 Make sure images are added with these exact names in the asset catalog, or the fallback mode will silently skip them.

Workaround 2 — Simulated Frame Playback

A more advanced version of Workaround 1. Load a larger sequence of pre-recorded images from the KUKA facility and cycle through them to simulate a live feed. While this won't reflect real-time robot state, it makes the UI feel more dynamic during demos and presentations, since the detection overlay will visibly change as different images cycle through.

- Add 10-20 photos of the KUKA robots at different poses and angles to the asset catalog
- Configure the playback interval to match realistic observation timing
- Useful for stakeholder demos where the live camera limitation is not immediately visible

Workaround 3 — visionOS Simulator with Static Assets

The visionOS Simulator does not support camera streaming, but all UI, overlay rendering, and detection pipeline logic can be tested using static assets. This is the correct environment for testing ContentView layout, RobotDataOverlay appearance, and the full InfluxDB integration without needing real hardware.

- Test all SwiftUI layout and UI state changes in the simulator
- Verify InfluxDB polling and sensor data display logic
- Test ImmersiveSpace transitions and Enter/Exit AR flow
- Reserve real Vision Pro testing for ARKit mesh and gesture functionality

Workaround 4 — Enterprise / Internal Build Note

> 🚫 Private APIs and entitlement hacks to bypass camera restrictions are not supported. These approaches are not safe, will not pass App Store review, and may break across OS updates. Even for internal or enterprise-only builds, this is not a viable path.

Immediate Action Plan

The recommended development sequence while waiting for Apple to expose live camera access:

1. Add static KUKA robot images to the Xcode asset catalog named robot-test-1, robot-test-2, etc.
2. Verify the 1.5-second inference timer fires and sends images to Roboflow — confirm console output
3. Test all UI overlays and RobotDataOverlay rendering using static detection results
4. Fill in InfluxDB credentials and verify the full telemetry pipeline works end to end
5. Continue adding labeled images in Roboflow to improve model accuracy in preparation for live frames
6. When Apple exposes a frame stream API, set self.latestPixelBuffer = frame.pixelBuffer — everything else is already wired up

App Readiness for Live Frames

The RobotSelector visionOS app is architecturally ready for live camera frames. The separation of concerns between frame acquisition (ImmersiveView) and inference (RoboflowDetectionService) means that connecting a real CVPixelBuffer source requires a single assignment:

```
// This one line is all that changes when Apple exposes camera frames
 self.latestPixelBuffer = frame.pixelBuffer

 // Everything downstream — inference, detection, overlay, InfluxDB — already works
```

No changes are required to RoboflowDetectionService, RobotDataOverlay, InfluxDBClient, SensorDataModel, or the ARKit mesh selection system. The app will transition from static image testing to live inference without restructuring.

Next Steps

- Add robot-test-1, robot-test-2, robot-test-3 images to the Xcode asset catalog
- Confirm Roboflow inference fires on static images and prints results to console
- Test the complete UI pipeline — selection, overlay, sensor data — using static detection output
- Monitor WWDC and Apple developer documentation for live camera frame access
- Continue labeling KUKA robot images in Roboflow to improve model performance
- Configure InfluxDB credentials so telemetry is live and tested independently of camera access