Dual Vibe Coding

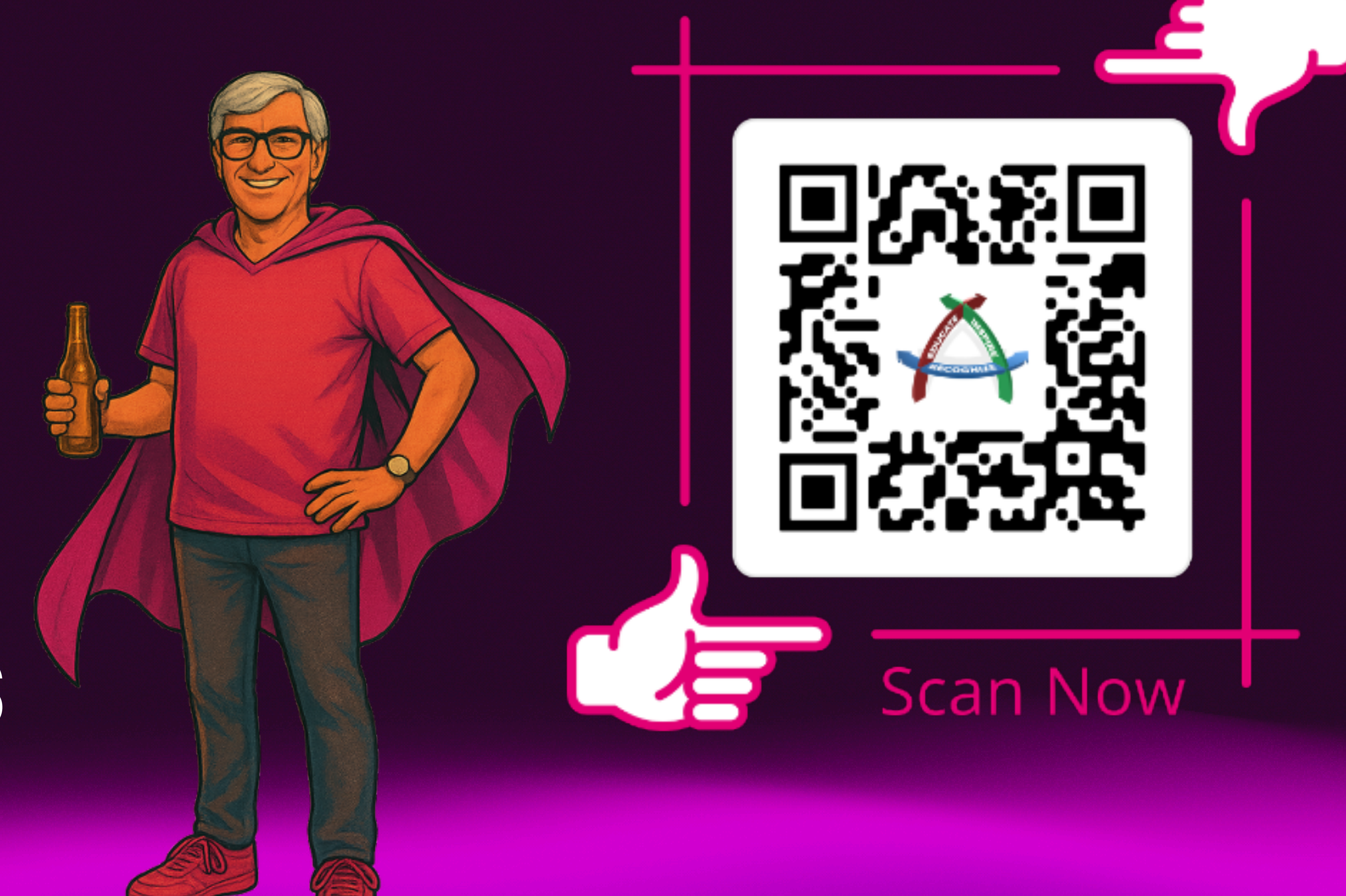
How to design immersive native XR experiences once
- and fluidly port them between the most advanced spatial platforms.

One codebase to rule them all (we hope)

Terry xR. Schussler, Senior Director of Next Generation Devices / Deutsche Telekom

But first... some vibes

Bring me some delicious Belgian beer!



While that is being delivered, I ask you to:

- Move up close to the front - let’s be friends!
- Scan this QRCode which will lead you to my entire presentation and more
- Put your phone down, relax and listen. No need to take photos of my slides.

The most important tool you have is your imagination.

Hackers
→ Prompt Engineers
→ Vibe Coders

Creative problem solvers

XR is cool but developing it is hard

Vibe Coding to the rescue! Because manually coding is so 2023...

- Vibe coding = **intent-first** development (Tell the AI your dreams, let it handle the nightmares)
- **Natural language** → code, scenes, tests
- **IDEs with agents:** Cursor, Antigravity, Lovable
- **You focus on vision;** AI handles scaffolding & refactors
- **Programmers get superpowers** (more on that later) but everyone can use vibe coding.

We Interrupt This Message for... a Reality Check!

HYPE!	REALITY
You can build an entire app in one prompt	It takes lots of iteration to get things right
AI does all the coding for you	AI code isn’t perfect - you’ll still need to do work
Anyone can ship finished apps	Many steps need programmer expertise
You can build anything	Best for modern frameworks / small-to-medium apps

Natural language-driven AI coding frameworks

Lovable

- Web- and SaaS-focused (React, Next.js, Node.js, Supabase)
- Best for: **WebXR, Three.js/Babylon.js prototyping**

`Travel is for Sharing Demo`

Cursor

- Deep, repo-aware IDE for many languages including C# and Swift
- Best for: **Single-project deep dives, visionOS + Unity refactoring**
- Works within Xcode, Android Studio, Visual Studio Code workflows

Google Antigravity

- An agent-first environment where multiple agents can collaborate to plan, code, and test
- Best for: **Cross-platform synchronization across visionOS, Android XR, and WebXR**

Vibe Coding for Cross-Platform XR

How AI/agentic tools streamline XR app refactoring and multi-IDE workflows

- XR projects span many tools and stacks: Xcode (visionOS), Unity (C#), Android Studio (Android XR)
- Agentic AI keeps a “big picture” view of your repo across languages and folders
- Automatically finds cross-cutting code: scenes, input handlers, interaction systems, shaders
- Can identify all RealityKit interactions in Swift and map them to Unity XR Interaction Toolkit or Android XR APIs
- Generates multi-file refactors instead of you hand-editing dozens of scripts
- Translates concepts (anchors, gestures, gaze, teleports) between SDKs, not just syntax
- Turns tedious discovery and mapping work into an automated, repeatable workflow

Planning & LLM Selection

Strategic Planning with LLMs

- **Start with Claude** (best reasoning and planning)
- Define **modularization strategy**: core logic, platform adapters, assets
- Identify **platform-specific constraints** early (input models, rendering pipelines)
- Create a **phased breakdown**: MVP → Platform A → Platform B → Optimization
- Document assumptions and unknowns
- Generate **initial architecture diagrams** and component separation

```
(SIMPLIFIED) CLAUDE PROMPT:

"I'm building an XR experience for Apple
Vision Pro and Samsung Galaxy XR.
Help me design a modular architecture
where:
1. Core interaction logic is platform-
agnostic
2. Rendering is abstracted (RealityKit vs
Unity OpenXR)
3. Input is mapped to common verbs
(select, grab, teleport)

Create a phased plan: Phase 1 (visionOS
MVP), Phase 2 (port to Android XR),
Phase 3 (WebXR fallback). For each phase,
list what code is new vs reused."
```

Planning & LLM Selection

Choosing Your LLM: **Claude vs ChatGPT vs Gemini** (Spoiler: you'll need them all anyway)

- **Claude Sonnet** → Planning, architecture, complex reasoning, large context windows
- **ChatGPT** → Fast iteration, web knowledge, general reliability
- **Gemini 3 Pro** → Deep integration with Google tools (Antigravity), fast reasoning
- **Smaller models** (Claude 3 Haiku, GPT-4o mini) → Fast, cheap, good for follow-ups and refinement
- **When to switch:** Start with Claude for design, switch to ChatGPT/Gemini for coding

Planning & LLM Selection

Vibe Coding Workflow with Prompting Strategy (Your roadmap just got an AI copilot)

Phase 0 (**Planning**): Claude → architecture & phased roadmap

Phase 1 (**Scaffolding**): Cursor + ChatGPT → generate initial codebase

Phase 2 (**Iteration**): Antigravity + prompt refinement → cross-platform refactors

Phase 3 (**Polish**): Profiling feedback → Cursor/Claude for optimization prompts

Phase 4 (**Maintenance**): Koog agents + periodic Claude reviews → keep platforms in sync

You can even build tools to build out these phases, especially phases 0 and 1.
Let's take a look at my Vibe Foundation project.

Keeping Reasoning Models Current

Your training data called. It's already obsolete

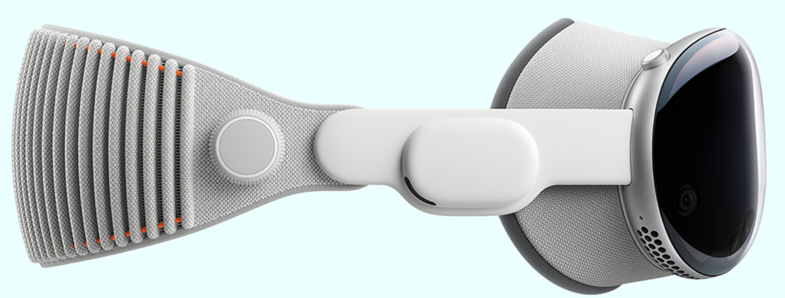
- **Create structured reference artifacts** - Document exact class names, methods, and code templates; make it harder for the model to guess than to admit what it doesn't know
- **Organize knowledge hierarchically** - Layer from "What should I build?" → "How do I build it?" so the model follows a path instead of filling gaps
- **Mark confidence levels explicitly** - Label claims as proven (✓), inferred (~), or unknown (?); the model can't hallucinate what you've marked as uncertain
- **Index for quick lookups** - Structure so models find "How do I [task]?" and "What's the class for [feature]?" instantly; less time searching = less time inventing

Keeping Reasoning Models Current (Bonus)

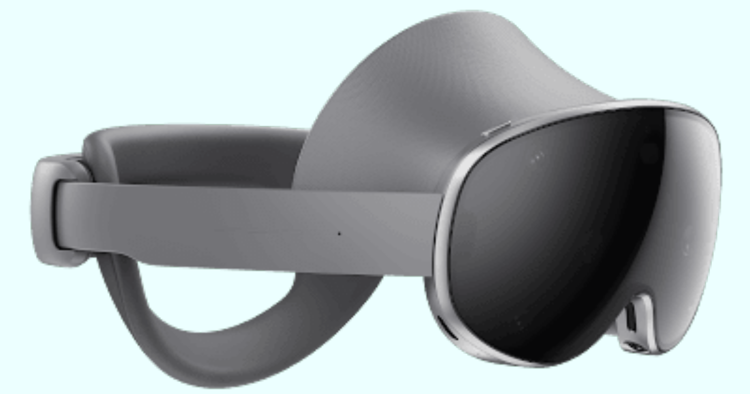
Teach your LLM to ask for what it needs, when it needs it.

- **Automated context injection beats manual updates** - Generate live API schemas, endpoint documentation, and deprecation notices programmatically at runtime; let your codebase introspect itself rather than hand-maintaining stale docs
- **Reasoning models + tool use = API-agnostic future-proofing** - Claude, o1, and Gemini can now function-call custom tools; give them your latest SDK specs in real-time and let them reason through new APIs without retraining
- **Hybrid knowledge layers for XR velocity** - Combine model training data (stable architectural patterns) with live API documentation (latest endpoints) and agent-generated examples (newest best practices) to ship against APIs released after the model's knowledge cutoff
- **Real-time search integration (Gemini advantage)** - Gemini 2.5 can fetch live docs during reasoning; pair this with Antigravity for cross-platform sync so your agentic workflow stays ahead of the SDK release cycle

Platform Overview



- **Apple Vision Pro (visionOS)**
  - Native dev stack: Swift + RealityKit + Reality Composer + Unity possibilities
  - Language: **Swift**
  - Frameworks: RealityKit, SwiftUI, Reality Composer Pro
  - Input: eyes, hands, voice



- **Samsung Galaxy XR, XReal Project Aura**
  - Android XR basics, Kotlin + OpenXR + Unity possibilities
  - Language: **Kotlin** (native), **C#** (Unity)
  - Frameworks: Android XR SDK, OpenXR, Unity XR Interaction Toolkit, AR Foundation
  - Input: eyes, hands, controllers, voice, trackpad

Platform Overview

Key differences and common XR development challenges

- Different stacks & SDKs per platform
- Different interaction paradigms & UI idioms
- **Shared issues:**
  - Performance goals (90–120 fps)
  - Comfortable interaction
  - Asset & scene complexity
  - Constantly evolving SDKs and device features

Platform Overview

Why cross-platform modularization matters for future spatial computing

- **More XR devices are coming,** not fewer
- **Avoid lock-in** to one vendor's stack
- **Modular** code & assets = easier ports
- AI tools work best on **well-structured projects**

Refactoring for Samsung Galaxy XR

Porting visionOS scene logic & assets to Android XR (Kotlin + OpenXR or Unity)

- Re-use your models, textures, and sources
- Maintain core interaction logic but re-implement rendering and input using Android XR / OpenXR packages or native Android XR APIs.
- AI tools can help map specific behaviors to the right Android XR or Unity primitives

```
PROMPT:

I have a Swift RealityKit system that
[describes behavior].
Translate this to Kotlin + Android XR
OpenXR, preserving:
- The interaction flow
- The state management approach
- Performance characteristics
Flag any API differences and suggest
workarounds.
```

Refactoring for Samsung Galaxy XR

Managing platform-specific inputs and XR ecosystem APIs while retaining core experience

Input is one of the biggest differences between headsets

Design an abstraction for input and interaction with consistent “verbs”

- Select
- Grab
- Teleport
- Open Menu

Each platforms’s implementation maps its own combo of hand trading, controller buttons or gaze events

```
PROMPT:

Design an input abstraction for XR that works
across:
- Apple Vision Pro (eyes + hands, no controllers)
- Samsung Galaxy XR (eyes + hands + controllers)

Create:
1. An InputInterface defining action verbs
   (Select, Grab, Teleport, Menu, Point)
2. A VisionProInputHandler
3. An AndroidXRInputHandler
4. A configuration system to remap inputs per
device

Provide full code with comments.
```

Prompting Best Practices for XR

DO’s and DON’Ts

- ✅ Specify the platform: "This is for Android XR with OpenXR"
- ✅ Include constraints: "Keep frame time under 11ms on Snapdragon XR2"
- ✅ Reference specific APIs and versions: "Use RealityKit's PhysicsComponent in visionOS 26.2 RC”
- ✅ Describe the user experience goal: "User should feel instant tactile feedback"
- ✅ Request code comments: "Explain where platform differences appear”

- ❌ Vague: "Make XR interaction better"
- ❌ Disconnected: "Write code" (without context)
- ❌ Assume capabilities: Don't assume AI knows your codebase structure
- ❌ Skip iteration: If code doesn't work, show the error and re-prompt
- ❌ Ignore testing: Always ask for unit tests alongside code

Vibe Coding WebXR

Integrating WebXR as a fallback/extension pathway for broader reach

WebXR democratizes spatial experiences

- Any browser on any device (mobile, desktop, headset) becomes an XR canvas **without app store gatekeeping**

Vibe coding accelerates WebXR prototyping

- Scaffolding **Three.js** or **Babylon.js** scenes from natural language, turning concept-to-demo into hours instead of days

Cross-platform asset reuse saves weeks

- Your glTF models, interaction logic, and state management from native projects port directly, avoiding the "rewrite from scratch" tax

Future Outlook

Programming and creative skills still matter. You are being augmented, not replaced!

- AI is already transforming how teams prototype, iterate, and debug XR apps
- AI tools are **not magic app generators**; they are powerful collaborators
  - Use them to avoid repetitive development tasks and spend more time on interaction design, storytelling, and overall experience
- **Idea to Outcome:** Focus on rapid prototyping and code scaffolding
- **Be fearless:** Use any API quickly. Experiment often. Try new things
- **Benefit from acceleration:** Avoid "dirty work" and build 10x or more faster