Teaching AI Models New APIs: Complete Strategy Guide

The Problem: Why AI Struggles with Fresh APIs

Knowledge Cutoff Reality

- **Claude's knowledge cutoff:** January 2025
- **visionOS 2.4 release:** December 2024 (barely in training data)
- **Android XR DP2:** May 2025 (completely absent)
- **Result:** AI models hallucinate plausible-sounding but non-existent APIs

Common AI Failures with New APIs

1. **Suggests deprecated patterns** from older versions
1. **Mixes APIs incorrectly** (iOS + visionOS confusion)
1. **Invents methods** that sound right but don't exist
1. **Misses breaking changes** between versions
1. **Doesn't know about new features** at all

Strategy 1: The Documentation Bundle Approach

Best For: Initial learning, architecture planning, comprehensive understanding

How It Works

Load the AI's context with official documentation in priority order:

```
Priority 1: Quick Start / Getting Started (shows idiomatic usage)
Priority 2: API Reference (complete method signatures)
Priority 3: Migration Guides (shows what changed from previous versions)
Priority 4: Sample Code (demonstrates real patterns)
Priority 5: Best Practices / Guidelines (architectural patterns)
```

Implementation Steps

Step 1: Gather Documentation

```bash
# Example for visionOS RealityKit
- Apple Developer Documentation (developer.apple.com)
```

- WWDC session transcripts
- Official sample projects
- Release notes (critical for "what's new")
```

**Step 2: Structure the Context**
Present documentation in a specific order that mirrors how you'd learn:

```markdown
1. "Here's what this API does (overview)"
2. "Here's how you initialize it (getting started)"
3. "Here are the main types/classes (API reference)"
4. "Here are common patterns (samples)"
5. "Here are things that changed (migration/release notes)"
```

Step 3: Be Explicit About Version
Always specify the exact version to avoid mixing old and new APIs:

```
"I'm using visionOS 2.4 (SDK version 26) and RealityKit 2.0.
Do not suggest APIs from visionOS 1.x or earlier RealityKit versions.
If you're uncertain whether an API exists in this version, say so."
```

Example in Practice

Bad approach:

```
"How do I create a 3D model in RealityKit?"
```

AI will suggest generic patterns that may be outdated.

Good approach:

```
I'm using visionOS 2.4 (SDK 26) and RealityKit 2.0.

Here's the relevant documentation:
[paste Quick Start guide]
[paste Entity API reference]
[paste ModelComponent documentation]
```
```

Based on THIS documentation only, show me how to:
1. Load a USDZ model
2. Add it to a RealityView
3. Enable gestures for interaction

If any API I'm asking about isn't in the documentation I provided,
tell me you don't see it rather than guessing.
```

---

## Strategy 2: The Iterative Refinement Method

### Best For: Debugging, fixing hallucinations, correcting mistakes

### How It Works

Start with AI-generated code, then refine it by feeding back compiler errors and documentation.

### Implementation Process

**Round 1: Initial Generation**

```
"Generate a basic implementation of [feature] using [API]"
```

Round 2: Error Correction

```
"The compiler says [error message]. Here's the actual API signature:
[paste from docs]

Please revise the code to match the real API."
```

Round 3: Validation

```
"Here's the working code. Explain why your initial suggestion was wrong
and what the correct pattern is, so you remember for this conversation."
```
```

Why This Works

- AI models are excellent at pattern matching once they see the correct pattern
- Compiler errors are unambiguous feedback
- Explaining corrections helps the AI build accurate mental models for the session

---

Strategy 3: The Example-Driven Learning Approach

Best For: Complex APIs, spatial computing concepts, unfamiliar paradigms

How It Works

Provide 2-3 complete, working examples from official sources, then ask for variations.

Implementation

Step 1: Provide Working Examples

```
Here are 3 working examples from Apple's official samples:

Example 1: Basic RealityView
[paste complete, compilable code]

Example 2: Adding gestures
[paste complete, compilable code]

Example 3: Spatial anchoring
[paste complete, compilable code]

All examples are from visionOS 2.4 SDK.
```

Step 2: Request Variations

```
"Based on ONLY the patterns in these examples, help me:
- Combine gesture handling with spatial anchors
- Add a Model3D with the same gesture system from Example 2
- Explain what each @State property is doing"
```

```

**Step 3: Validate Against Examples**

```
"Does your suggested code follow the same patterns as the examples?
Point out where it matches and where it diverges."
```

### Why This Works

- Working code is ground truth (can't hallucinate)
- AI excels at pattern interpolation
- Reduces risk of mixing incompatible API versions

---

## Strategy 4: The Constraint-Based Prompting Method

### Best For: Preventing hallucinations, ensuring accuracy

### How It Works

Explicitly constrain the AI to only use information you've provided.

### Effective Constraints

**Version Lock:**

```
"ONLY use APIs available in visionOS 2.4. If you're unsure whether
something exists in this version, say 'I don't see this in visionOS 2.4'
rather than guessing."
```

Documentation Lock:

```
"Base your response ONLY on the documentation I've provided below.
Do not use knowledge from your training data, as it may be outdated.
[documentation]"
```

Uncertainty Expression:
```

```
"If you're not 100% certain an API exists or a pattern is correct,
say so explicitly. I prefer 'I don't know' to a wrong answer that
wastes debugging time."
```

Hallucination Prevention:

```
"Do not invent method names, parameters, or types. If the documentation
doesn't show how to do something, say 'This isn't covered in the
provided documentation' rather than improvising."
```

Strategy 5: The Architecture-First Approach

Best For: Planning large features, system design, avoiding API lock-in

How It Works

Have AI design architecture independent of specific APIs, then map to real APIs later.

Process

Step 1: Abstract Design

```
"Design the architecture for a spatial note-taking app in XR.
Focus on:
- State management approach
- Data flow patterns
- Component hierarchy
- Interaction model

Don't worry about specific APIs yet—focus on concepts."
```

Step 2: API Mapping

```
```

"Now, here's the RealityKit documentation:
[documentation]

Map the abstract architecture to RealityKit:
- Which components become RealityKit Entities?
- Where does state live (@State, @StateObject, etc.)?
- How do gestures trigger state changes?"
```

**Step 3: Gap Analysis**

```
"Are there parts of the architecture that RealityKit doesn't
directly support? What workarounds or additional libraries
would we need?"
```

Why This Works

- Separates architectural thinking (AI is good at this) from API specifics (AI is bad with new APIs)
- Creates a conceptual model first, API bindings second
- Reduces risk of API-driven design (letting API limitations drive architecture)

Strategy 6: The Differential Learning Method

Best For: Learning API changes, migrating between versions

How It Works

Explicitly teach the AI what changed between versions.

Implementation

Step 1: Show Old vs. New

```
"Here's how you created a spatial scene in visionOS 1.0:
[old code]

Here's how you do it in visionOS 2.4:
[new code]

What changed and why?"
```

Step 2: Pattern Extraction

```
"Based on these changes, what's the new pattern for:
- Window management
- Spatial anchoring
- Gesture handling"
```

Step 3: Deprecation Awareness

```
"These APIs were deprecated in 2.4:
[list deprecated APIs]

If I ask for something that would use these, suggest the 2.4 alternative."
```

Strategy 7: The Multi-Turn Deep Dive

Best For: Complex features, learning as you build

How It Works

Break learning into focused, single-topic conversations.

Session Structure

Session 1: Core Concepts

```
"Let's focus only on RealityKit Entity hierarchy today.
[paste Entity documentation]

Help me understand:
- When to use Entity vs. ModelEntity
- Parent-child relationships
- Component pattern"
```
```

```

**Session 2: Building on Concepts**

```
"Yesterday we covered Entity hierarchy. Today: gestures.
Here's what I learned before: [summary]
Here's the gesture documentation: [docs]

How do gestures interact with the Entity hierarchy we discussed?"
```

**Session 3: Integration**

```
"Now I want to combine Entity hierarchy + gestures + spatial anchoring.
Here's what we've learned:
[summary from previous sessions]

Design a system that integrates all three."
```

Why This Works

- Each session builds a focused mental model
- AI maintains context within a conversation
- Complexity is introduced gradually
- You can correct misunderstandings before they compound

Best Practices: Do's and Don'ts

✅ DO:

1. **Always specify exact versions**
- "visionOS 2.4 SDK 26" not just "visionOS"
1. **Provide official documentation**
- Apple Developer docs, Android developer docs, official samples
1. **Ask AI to explain its reasoning**
- "Why did you choose this approach over X?"
1. **Validate generated code immediately**
- Compile it, run it, feed errors back
1. **Use AI for patterns, not specifics**
```

- Good: “What’s the idiomatic way to handle state?”
- Bad: “What’s the exact method signature for X?”
1. **Create artifacts/documents within conversations**
- Build up a knowledge base the AI can reference
1. **Request uncertainty signals**
- “Tell me when you’re guessing”

❌ DON’T:

1. **Assume AI knows the latest APIs**
- It doesn’t. Always provide docs.
1. **Accept the first answer without validation**
- Hallucinations are common with new APIs
1. **Mix versions in one conversation**
- “visionOS 1.x vs 2.x” → confusion
1. **Ask for complete apps without examples**
- Too much room for hallucination
1. **Ignore compiler errors as “AI mistakes”**
- They’re teaching opportunities—feed them back
1. **Use AI-generated code without understanding it**
- You need to debug it when it breaks
1. **Expect AI to know undocumented APIs**
- Partner programs, beta features → AI has no data

Tool-Specific Tips

Claude (200K context window)

- **Strengths:** Can hold entire API reference, excellent at architectural reasoning
- **Strategy:** Load full documentation in one prompt, use artifacts to build living documentation
- **Best for:** Architecture planning, complex integrations

GitHub Copilot

- **Strengths:** Inline completions, IDE integration
- **Strategy:** Use comments to provide context: `// Using visionOS 2.4 RealityKit...`
- **Best for:** Boilerplate, repetitive patterns once you’ve established them

ChatGPT (with GPT-4)

- **Strengths:** Iterative refinement, explaining concepts
- **Strategy:** Multi-turn deep dives, use Code Interpreter for validation
- **Best for:** Learning concepts, debugging specific issues

Measuring Success: Are You Teaching Effectively?

Good Signs:

✅ AI suggests code that compiles on first try
✅ AI says “I don’t see that in the docs” when appropriate
✅ AI patterns match official examples
✅ AI can explain *why* it chose an approach

Warning Signs:

⚠️ Repeated compiler errors for same API
⚠️ AI invents method names that “sound right”
⚠️ AI mixes patterns from different SDK versions
⚠️ AI overconfident about APIs it shouldn’t know

Real-World Example: Teaching Claude About Android XR Jetpack SceneCore

The Challenge

Android XR’s Jetpack SceneCore was released in May 2025—completely outside Claude’s training data.

Effective Approach

Prompt:

```
I'm working with Android XR Developer Preview 2 (May 2025) and Jetpack SceneCore.
This is a brand new API you won't have training data on.

Here's the official Getting Started guide:
[paste from developer.android.com]
```

Here's a complete working example from Google:
[paste official sample]

Based ONLY on this documentation, help me understand:
1. The core architecture (Session, Scene, Entity pattern)
2. How this differs from ARCore (which you do know)
3. The lifecycle management approach

If I ask about features not covered in these docs, tell me you don't have that information rather than guessing based on ARCore patterns, as SceneCore is a different API.
```

**Why This Works:**

- Explicitly states API is new (sets expectations)
- Provides ground truth (official docs + working code)
- Constrains AI to provided info only
- Acknowledges related knowledge (ARCore) but warns against mixing

---

## Advanced Technique: The “Living Documentation” Artifact

### Concept

Create a structured artifact within Claude that becomes the authoritative reference for the conversation.

### Implementation

**Step 1: Initialize the Artifact**

```
"Create a markdown artifact titled 'visionOS 2.4 RealityKit Reference'.

Structure it as:
# Core Concepts
# Key APIs
# Common Patterns
# Gotchas & Breaking Changes

I'll provide documentation iteratively, and you'll update this artifact.

This becomes our source of truth for the conversation."
```

Step 2: Build It Incrementally

```
"Add to the artifact:

## Entity System
[paste Entity docs]

Summarize the key points developers need to know."
```

Step 3: Reference It

```
"Based on our living documentation artifact, show me how to:
- Create an Entity hierarchy
- Add components
- Handle parent-child relationships"
```

Benefits

- Creates persistent, conversation-scoped knowledge base
- AI can reference and update it
- You can export it for future use
- Reduces hallucination (AI refers to artifact, not training data)

The Meta-Strategy: Teaching AI to Teach Itself

Advanced Prompt Pattern

```
"I'm going to teach you a new API in stages. After each stage,
I want you to:

1. Summarize what you learned
2. Identify gaps in your understanding
3. Request the specific documentation that would fill those gaps
4. Generate 2-3 test questions to verify you understood correctly
```
```

This way, you're actively participating in your own learning process
rather than passively receiving information.

Ready? Let's start with [API name]."
```

### Why This Is Powerful

- AI actively identifies what it doesn't know
- Creates feedback loop (AI requests relevant docs)
- Self-testing reinforces learning
- You see exactly what the AI understands (and doesn't)

---

## Conclusion: The Philosophy

**Old Mental Model:**
"AI knows everything, I just need to ask the right question."

**New Mental Model for Fresh APIs:**
"AI is a brilliant intern with no domain knowledge. I need to provide
the documentation, validate outputs, and build up knowledge iteratively."

**The Payoff:**
Once you've taught an AI model the API within a conversation context,
it becomes an incredibly effective collaborator—combining its architectural
reasoning with accurate API knowledge.

**Time Investment:**

- Initial teaching: 30-60 minutes
- Iterative refinement: 10-15 minutes per feature
- Long-term payoff: 2-3x faster development vs. docs alone

**The Key Insight:**
AI coding assistants aren't replacements for reading documentation with
new APIs—they're **amplifiers** that help you apply documentation more
effectively once they've learned it.
```