
🍱 **Repo layout**

```
companionOS-backend/
├── README.md
├── env/
│   ├── .env.sample
│   └── README_ENV.md
├── convex/
│   ├── schema.ts
│   ├── queue.ts
│   ├── settings.ts
│   ├── sessions.ts
│   ├── chats.ts
│   ├── notes.ts
│   └── skills.ts
└── ios/
    ├── App/
    │   ├── AppDelegate.swift
    │   └── SceneDelegate.swift
    ├── Core/
    │   ├── Support/Constants.swift
    │   ├── Support/Keychain.swift
    │   ├── Persistence/LocalCache.swift
    │   ├── Persistence/ConvexClient.swift
    │   ├── Bus/MessageTypes.swift
    │   ├── Bus/CapabilityBus.swift
    │   └── Connectivity/PhoneSession.swift
    ├── Capabilities/
    │   ├── Media/
    │   │   ├── QueueItem.swift
    │   │   ├── QueueService.swift
    │   │   ├── URLRouter.swift
    │   │   ├── Launcher.swift
    │   │   ├── RemoteBridge.swift
    │    │    ├── NowPlayingMirror.swift
    │    │    └── AutoNextMonitor.swift
    │    ├── Actions/
    │    │    ├── ShortcutRunner.swift
    │    │    └── URLActionRouter.swift
    │    ├── Comms/
    │    │    ├── LLMRouter.swift
    │    │    ├── Providers/OpenAIProvider.swift
    │    │    ├── Providers/GoogleAIProvider.swift
    │    │    ├── Providers/LocalHTTPProvider.swift
    │    │    ├── OAuth/ProviderConfig.swift
    │    │    ├── OAuth/OAuthService.swift
    │    │    └── OAuth/TokenStore.swift
    │    ├── Notes/NotesService.swift
    │    └── Search/SearchService.swift
    ├── Extensions/ShareExtension/
    │    ├── ShareViewController.swift
    │    └── Info.plist
    └── Intents/AppIntents.swift
```

README.md (paste)

```
# CompanionOS — Backend (watch-first, OAuth-first)

This repo provides the iOS backend for CompanionOS:
- watchOS ⇄ iOS **Connectivity**
- **Capability Bus** (media, comms/LLMs, actions, notes,
search)
- **OAuth by default** for model providers (Gemini, OpenAI
via proxy), **API key** fallback
- **Convex** persistence (queue, settings, chats, notes,
skills)
- **Privacy-first**: no analytics
```

```
## Quickstart

1. Copy `env/.env.sample` → `env/.env`, fill values.
2. Configure iOS target bundle id + App Group.
3. Configure Google OAuth client (Gemini) and optional OpenAI
OAuth proxy.
4. Build & run on iPhone (paired Watch optional for now).

## OAuth (Default)
- **Gemini**: OAuth 2 (Authorization Code + PKCE)
- **OpenAI**:
  - Preferred: your **OAuth-enabled proxy** (Auth/Token
endpoints)
  - Fallback: API key in Keychain (local only)
- **Local HTTP**: no OAuth (bearer token optional)

## Thread Stickiness
Each provider keeps a **default conversation** per user.
Wrist messages go to that thread unless you explicitly
create/switch threads.

## Messages (Watch ⇄ Phone)
`COSMessage { op, id, domain, action, payload }`
Domains: `media`, `comms`, `actions`, `notes`, `search`.
```
Example: `{"op":"request","domain":"comms","action":"chat","payload":{"router":"gemini","text":"hello"}}`

🌱 **env files**

env/.env.sample

```
CONVEX_DEPLOYMENT_URL=https://YOUR.convex.cloud
CONVEX_AUTH_TOKEN=dev-or-personal-token
APP_GROUP_ID=group.com.your.bundle.companion
```

```
BUNDLE_PREFIX=com.your.bundle

# OAuth: Gemini
GOOGLE_CLIENT_ID=YOUR_IOS_CLIENT_ID.apps.googleusercontent.co
m
GOOGLE_REDIRECT_URI=com.your.bundle:/oauth2redirect/google
# custom scheme

# OAuth: OpenAI via proxy (optional)
OPENAI_AUTH_ENDPOINT=https://your-proxy/auth
OPENAI_TOKEN_ENDPOINT=https://your-proxy/token
OPENAI_CLIENT_ID=your-openai-proxy-client
OPENAI_REDIRECT_URI=com.your.bundle:/oauth2redirect/openai
OPENAI_SCOPES=openid profile offline_access api

# API-key fallbacks (optional)
OPENAI_API_KEY=
GOOGLE_API_KEY=
LOCAL_LLM_BASE_URL=http://127.0.0.1:11434
LOCAL_LLM_BEARER=
```

env/README_ENV.md (short)

- Add URL Schemes for redirects:
  - com.your.bundle → handles /oauth2redirect/google and /oauth2redirect/openai
- Register iOS OAuth client for Gemini in Google Cloud Console:
  - Authorized redirect URI: com.your.bundle:/oauth2redirect/google
- If using OpenAI via proxy, register the same redirect and client there.

🧠 Convex (schema & functions)

convex/schema.ts

```
import { defineSchema, defineTable } from "convex/server";
import { v } from "convex/values";
```

```
export default defineSchema({
  queueItems: defineTable({
    userId: v.string(),
    source: v.string(),
    originalURL: v.string(),
    normalizedURL: v.string(),
    videoId: v.optional(v.string()),
    title: v.string(),
    thumbnailURL: v.optional(v.string()),
    duration: v.optional(v.number()),
    addedAt: v.number(),
    tags: v.optional(v.array(v.string())),
  }).index("by_user_added", ["userId","addedAt"]),

  settings: defineTable({
    userId: v.string(),
    skipForwardSec: v.number(),
    skipBackwardSec: v.number(),
    autoNext: v.boolean(),
    defaultThreads: v.optional(v.record(v.string(),
v.string())), // router -> threadId
  }).index("by_user", ["userId"]),

  sessions: defineTable({
    userId: v.string(),
    currentItemId: v.optional(v.id("queueItems")),
    armedNextId: v.optional(v.id("queueItems")),
    updatedAt: v.number()
  }).index("by_user", ["userId"]),

  chats: defineTable({
    userId: v.string(),
    threadId: v.string(),
    modelRouter: v.string(), // "gemini" | "openai" |
"localHTTP" | "mcp:..."
```

```
    messages: v.array(v.object({
      role: v.union(v.literal("user"),
v.literal("assistant"), v.literal("tool")),
      text: v.string(),
      ts: v.number()
    })),
    name: v.optional(v.string()),
    updatedAt: v.number()
  }).index("by_user_updated", ["userId","updatedAt"]),

  notes: defineTable({
    userId: v.string(),
    text: v.string(),
    createdAt: v.number(),
    tags: v.optional(v.array(v.string()))
  }).index("by_user_created", ["userId","createdAt"]),

  skills: defineTable({
    userId: v.string(),
    name: v.string(),
    kind: v.string(), // "shortcut" | "url" | "mcp"
    spec: v.string(), // JSON blob
    createdAt: v.number()
  }).index("by_user", ["userId"])
});
```

convex/queue.ts

```
import { query, mutation } from "convex/server";
import { v } from "convex/values";

export const list = query({
  args: { userId: v.string() },
  handler: async (ctx, { userId }) =>
    ctx.db.query("queueItems").withIndex("by_user_added", q
=> q.eq("userId", userId)).order("asc").collect()
});
```

```
export const add = mutation({
  args: {
    userId: v.string(), source: v.string(), originalURL:
v.string(), normalizedURL: v.string(),
    videoId: v.optional(v.string()), title: v.string(),
thumbnailURL: v.optional(v.string()),
    duration: v.optional(v.number()), tags:
v.optional(v.array(v.string()))
  },
  handler: async (ctx, args) => ctx.db.insert("queueItems",
{ ...args, addedAt: Date.now() })
});

export const remove = mutation({
  args: { userId: v.string(), id: v.id("queueItems") },
  handler: async (ctx, { userId, id }) => {
    const doc = await ctx.db.get(id); if (!doc ||
doc.userId !== userId) return null;
    await ctx.db.delete(id); return id;
  }
});

export const move = mutation({
  args: { userId: v.string(), orderedIds:
v.array(v.id("queueItems")) },
  handler: async (ctx, { userId, orderedIds }) => {
    const base = Date.now();
    for (let i = 0; i < orderedIds.length; i++) {
      const id = orderedIds[i]; const row = await
ctx.db.get(id);
      if (row && row.userId === userId) await
ctx.db.patch(id, { addedAt: base + i });
    }
    return true;
  }
});
```

convex/settings.ts

```
import { query, mutation } from "convex/server";
import { v } from "convex/values";

export const get = query({
  args: { userId: v.string() },
  handler: async (ctx, { userId }) => {
    const [row] = await
ctx.db.query("settings").withIndex("by_user", q =>
q.eq("userId", userId)).collect();
    return row ?? { userId, skipForwardSec: 15,
skipBackwardSec: 15, autoNext: true, defaultThreads: {} };
  }
});

export const upsert = mutation({
  args: { userId: v.string(), skipForwardSec: v.number(),
skipBackwardSec: v.number(), autoNext: v.boolean() },
  handler: async (ctx, args) => {
    const [row] = await
ctx.db.query("settings").withIndex("by_user", q =>
q.eq("userId", args.userId)).collect();
    if (!row) return await ctx.db.insert("settings",
{ ...args, defaultThreads: {} });
    await ctx.db.patch(row._id, args); return true;
  }
});

export const setDefaultThread = mutation({
  args: { userId: v.string(), router: v.string(), threadId:
v.string() },
  handler: async (ctx, { userId, router, threadId }) => {
    const [row] = await
ctx.db.query("settings").withIndex("by_user", q =>
q.eq("userId", userId)).collect();
    if (!row) return await ctx.db.insert("settings",
{ userId, skipForwardSec:15, skipBackwardSec:15,
autoNext:true, defaultThreads: { [router]: threadId } });
    const def = row.defaultThreads ?? {}; def[router] =
threadId;
    await ctx.db.patch(row._id, { defaultThreads: def });
return def;
  }
});
```

convex/sessions.ts

```
import { query, mutation } from "convex/server";
import { v } from "convex/values";

export const get = query({
  args: { userId: v.string() },
  handler: async (ctx, { userId }) => {
    const [row] = await
ctx.db.query("sessions").withIndex("by_user", q =>
q.eq("userId", userId)).collect();
    return row ?? { userId, updatedAt: Date.now() };
  }
});

export const setCurrent = mutation({
  args: { userId: v.string(), currentItemId:
v.optional(v.id("queueItems")) },
  handler: async (ctx, { userId, currentItemId }) => {
    const [row] = await
ctx.db.query("sessions").withIndex("by_user", q =>
q.eq("userId", userId)).collect();
    if (!row) return await ctx.db.insert("sessions",
{ userId, currentItemId, updatedAt: Date.now() });
    await ctx.db.patch(row._id, { currentItemId, updatedAt:
Date.now() }); return true;
```

```
  }
});

export const armNext = mutation({
  args: { userId: v.string(), armedNextId:
v.optional(v.id("queueItems")) },
  handler: async (ctx, { userId, armedNextId }) => {
    const [row] = await
ctx.db.query("sessions").withIndex("by_user", q =>
q.eq("userId", userId)).collect();
    if (!row) return await ctx.db.insert("sessions",
{ userId, armedNextId, updatedAt: Date.now() });
    await ctx.db.patch(row._id, { armedNextId, updatedAt:
Date.now() }); return true;
  }
});
```

convex/chats.ts

```
import { query, mutation } from "convex/server";
import { v } from "convex/values";

export const listThreads = query({
  args: { userId: v.string(), router:
v.optional(v.string()) },
  handler: async (ctx, { userId, router }) => {
    const rows = await
ctx.db.query("chats").withIndex("by_user_updated", q =>
q.eq("userId", userId)).order("desc").collect();
    return router ? rows.filter(r => r.modelRouter ===
router) : rows;
  }
});

export const upsertThread = mutation({
  args: { userId: v.string(), router: v.string(), threadId:
v.optional(v.string()), name: v.optional(v.string()) },
  handler: async (ctx, { userId, router, threadId, name }) =>
{
    const tid = threadId ?? crypto.randomUUID();
    const [existing] = await
ctx.db.query("chats").withIndex("by_user_updated", q =>
q.eq("userId", userId)).order("desc").collect();
    const match = existing && existing.threadId === tid ?
existing : null;
    if (!match) return await ctx.db.insert("chats", { userId,
threadId: tid, modelRouter: router, messages: [], updatedAt:
Date.now(), name });
    await ctx.db.patch(match._id, { name: name ??
match.name }); return { threadId: tid };
  }
});

export const append = mutation({
  args: { userId: v.string(), threadId: v.string(), router:
v.string(), role: v.string(), text: v.string() },
  handler: async (ctx, { userId, threadId, router, role,
text }) => {
    const rows = await
ctx.db.query("chats").withIndex("by_user_updated", q =>
q.eq("userId", userId)).order("desc").collect();
    const chat = rows.find(r => r.threadId === threadId &&
r.modelRouter === router)
      ?? await ctx.db.insert("chats", { userId, threadId,
modelRouter: router, messages: [], updatedAt: Date.now() });
    const msg = { role: role as "user"|"assistant"|"tool",
text, ts: Date.now() };
    await ctx.db.patch(chat._id, { messages:
[...chat.messages, msg], updatedAt: Date.now() });
    return msg;
  }
});
```

convex/notes.ts and convex/skills.ts are small; you can mirror the patterns above.

📱 iOS core files (paste-ready)

ios/App/AppDelegate.swift

```
import UIKit

@main
class AppDelegate: UIResponder, UIApplicationDelegate {
  func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions:
[UIApplication.LaunchOptionsKey: Any]?) -> Bool {
    PhoneSession.shared.start()
    CapabilityBus.shared.bootstrap()
    return true
  }
}
```

ios/Core/Support/Constants.swift

```
import Foundation

enum Constants {
  static let appGroup =
ProcessInfo.processInfo.environment["APP_GROUP_ID"] ??
"group.com.your.bundle.companion"
  static let convexURL =
ProcessInfo×processInfo.environment["CONVEX_DEPLOYMENT_URL"]
?? ""
  static let convexAuth =
ProcessInfo×processInfo.environment["CONVEX_AUTH_TOKEN"] ??
""
  static let bundlePrefix =
ProcessInfo.processInfo.environment["BUNDLE_PREFIX"] ??
"com.your.bundle"
```

```
  // OAuth config
  static let googleClientId =
ProcessInfo.processInfo.environment["GOOGLE_CLIENT_ID"]
  static let googleRedirect =
ProcessInfo.processInfo.environment["GOOGLE_REDIRECT_URI"]

  static let openaiAuthEndpoint =
ProcessInfo×processInfo.environment["OPENAI_AUTH_ENDPOINT"]
  static let openaiTokenEndpoint =
ProcessInfo.processInfo.environment["OPENAI_TOKEN_ENDPOINT"]
  static let openaiClientId =
ProcessInfo.processInfo.environment["OPENAI_CLIENT_ID"]
  static let openaiRedirect =
ProcessInfo×processInfo.environment["OPENAI_REDIRECT_URI"]
  static let openaiScopes =
ProcessInfo.processInfo.environment["OPENAI_SCOPES"]?.compone
nts(separatedBy: " ")

  // Fallback API keys
  static let openaiApiKey =
ProcessInfo.processInfo.environment["OPENAI_API_KEY"]
  static let googleApiKey =
ProcessInfo.processInfo.environment["GOOGLE_API_KEY"]
  static let localBaseURL =
ProcessInfo.processInfo.environment["LOCAL_LLM_BASE_URL"] ??
"http://127.0.0.1:11434"
  static let localBearer =
ProcessInfo.processInfo.environment["LOCAL_LLM_BEARER"]
}
```

ios/Core/Support/Keychain.swift (minimal helpers)

```
import Foundation
import Security

enum Keychain {
  static func set(_ value: Data, key: String) {
    let q:[String:Any] = [kSecClass as
String:kSecClassGenericPassword, kSecAttrAccount as
String:key, kSecValueData as String:value, kSecAttrAccessible
as String:kSecAttrAccessibleAfterFirstUnlock]
    SecItemDelete(q as CFDictionary); SecItemAdd(q as
CFDictionary, nil)
  }
  static func get(_ key:String)->Data?{
    let q:[String:Any] = [kSecClass as
String:kSecClassGenericPassword,kSecAttrAccount as
String:key,kSecReturnData as String:true,kSecMatchLimit as
String:kSecMatchLimitOne]
    var out:CFTypeRef?; guard SecItemCopyMatching(q as
CFDictionary,&out)==errSecSuccess else {return nil}; return
out as? Data
  }
  static func setString(_ s:String,key:String)
{ set(Data(s.utf8), key:key) }
  static func getString(_ key:String)->String?{ guard let
d=get(key) else {return nil}; return
String(data:d,encoding:.utf8) }
}
```

ios/Core/Persistence/LocalCache.swift

```
import Foundation

final class LocalCache {
  static let shared = LocalCache()
  private let dir: URL
  private init() {
    dir =
FileManager.default.containerURL(forSecurityApplicationGroupI
dentifier: Constants.appGroup)!
  }
  func url(_ name:String)->URL
{ dir.appendingPathComponent(name) }
  func read<T:Decodable>(_ name:String, as: T.Type)->T?{
    guard let d = try? Data(contentsOf: url(name)) else
{ return nil }
    return try? JSONDecoder().decode(T.self, from: d)
  }
  func write<T:Encodable>(_ name:String, _ value:T){
    if let d = try? JSONEncoder().encode(value) { try?
d.write(to: url(name), options: .atomic) }
  }
}
```

ios/Core/Persistence/ConvexClient.swift

```
import Foundation

final class ConvexClient {
  static let shared = ConvexClient()
  private var base = Constants×convexURL
  private var auth = Constants×convexAuth

  func call<T:Decodable>(_ path:String, _ body:[String:Any])
async throws -> T {
    guard let url = URL(string: "\(base)/\(path)") else
{ throw NSError(domain:"convex", code:0) }
    var req = URLRequest(url:url); req×httpMethod="POST"
    req.addValue("application/json","Content-Type")
    req.addValue(auth, forHTTPHeaderField:"Authorization")
    req×httpBody = try JSONSerialization.data(withJSONObject:
body)
    let (data, resp) = try await URLSession.shared.data(for:
req)
    guard (resp as? HTTPURLResponse)?.statusCode ?? 500 < 300
else { throw NSError(domain:"convex", code:1) }
    return try JSONDecoder().decode(T.self, from: data)
  }
}
```

ios/Core/Bus/MessageTypes.swift

```
import Foundation

public struct COSMessage: Codable {
  public enum Op: String, Codable { case request, response,
event }
  public var op: Op; public var id: String; public var
domain: String; public var action: String
  public var payload: [String:AnyCodable]?
  public var error: COSError?
  public init(op: Op, id: String = UUID()×uuidString, domain:
String, action: String, payload:[String:AnyCodable]?=nil,
error:COSError?=nil){
    self.op=op; self×id=id; self.domain=domain;
self.action=action; self.payload=payload; self.error=error
  }
}
public struct COSError: Codable { public let code: String;
public let message: String }

public struct AnyCodable: Codable {
  public let value: Any
  public init(_ v: Any) { value = v }
  public init(from d: Decoder) throws {
    let c = try d.singleValueContainer()
    if let v = try? c.decode(Bool.self) { value=v; return }
    if let v = try? c.decode(Double.self) { value=v; return }
    if let v = try? c.decode(String.self) { value=v; return }
    if let v = try? c.decode([String:AnyCodable].self)
{ value=v; return }
    if let v = try? c.decode([AnyCodable].self) { value=v;
return }
    throw DecodingError.dataCorruptedError(in:c,
debugDescription:"unsupported")
  }
  public func encode(to e: Encoder) throws {
    var c = e.singleValueContainer()
    switch value {
      case let v as Bool: try c.encode(v)
      case let v as Double: try c.encode(v)
      case let v as String: try c.encode(v)
      case let v as [String:AnyCodable]: try c.encode(v)
      case let v as [AnyCodable]: try c.encode(v)
      default: try c.encodeNil()
    }
  }
}
```

ios/Core/Bus/CapabilityBus.swift

```
import Foundation

protocol Capability { var domain: String { get } func
handle(_ msg: COSMessage) async -> COSMessage }

final class CapabilityBus {
  static let shared = CapabilityBus()
  private var caps: [String: Capability] = [:]
  func register(_ cap: Capability) { caps[cap.domain] = cap }
  func route(_ msg: COSMessage) async -> COSMessage {
    guard let cap = caps[msg.domain] else {
      return COSMessage(op:.response, id:msg.id,
domain:msg.domain, action:msg.action, error:
COSError(code:"no_capability", message:"No handler"))
    }
    return await cap.handle(msg)
  }
  func bootstrap() {
    register(MediaCapability())
    register(CommsCapability())
    register(ActionsCapability())
    register(NotesCapability())
    register(SearchCapability())
  }
}
```

ios/Core/Connectivity/PhoneSession.swift

```
import WatchConnectivity

final class PhoneSession: NSObject, WCSessionDelegate {
  static let shared = PhoneSession(); private override init()
{}
  func start(){ let s=WCSession.default; s.delegate=self;
s.activate() }

  func session(_ session: WCSession, didReceiveMessage msg:
[String : Any], replyHandler: @escaping ([String : Any]) ->
Void) {
    Task {
      guard let data = (msg["data"] as? Data), let cos = try?
JSONDecoder().decode(COSMessage.self, from: data) else {
        replyHandler(["error":"bad_message"]); return
      }
      let res = await CapabilityBus.shared.route(cos)
      let out = (try? JSONEncoder().encode(res)) ?? Data()
      replyHandler(["data": out])
    }
  }
}
```

🎚️ **Media capability (key files)**

QueueItem.swift

```
import Foundation
```

```
public struct QueueItem: Codable, Equatable, Identifiable {
  public let id: UUID
  public var source: String
  public var originalURL: URL
  public var normalizedURL: URL
  public var videoId: String?
  public var title: String
  public var thumbnailURL: URL?
  public var duration: TimeInterval?
  public var addedAt: Date
  public var tags: [String]
  public init(id: UUID = ×init(), source: String,
originalURL: URL, normalizedURL: URL, videoId: String?=nil,
title: String, thumbnailURL: URL?=nil, duration:
TimeInterval?=nil, addedAt: Date = ×init(), tags:[String]=[])
{
    self×id=id; self.source=source;
self.originalURL=originalURL;
self.normalizedURL=normalizedURL; self.videoId=videoId;
self.title=title; self.thumbnailURL=thumbnailURL;
self.duration=duration; self.addedAt=addedAt; self.tags=tags
  }
}
```

URLRouter.swift

```
import Foundation
enum URLRouter {
  static func youtubeWatchURL(videoId:String)->URL{
    URL(string:"youtube://watch?v=\(videoId)") ??
URL(string:"https://www.youtube.com/watch?v=\(videoId)")!
  }
  static func youtubeSearchURL(query:String)->URL{
    let q =
query.addingPercentEncoding(withAllowedCharacters:.urlQueryAl
lowed) ?? query
    return URL(string:"youtube://results?search_query=\
(q)") ?? URL(string:"https://www.youtube.com/results?
search_query=\(q)")!
  }
  static func normalized(from url:URL)->URL { url }
}
```

Launcher.swift

```
import UIKit
enum Launcher {
  static func open(_ url: URL){ DispatchQueue.main.async
{ UIApplication.shared.open(url) } }
  static func play(_ item: QueueItem)
{ open(item.normalizedURL) }
}
```

RemoteBridge.swift

```
import MediaPlayer

final class RemoteBridge {
  static let shared = RemoteBridge()
  private let cmd = MPRemoteCommandCenter×shared()

  init() {
    cmd×nextTrackCommand×isEnabled = true
    cmd×previousTrackCommand×isEnabled = true
  }

  func play() { cmd.playCommand.addTarget { _ in .success } ;
cmd.playCommand.invoke() }
  func pause(){ cmd.pauseCommand.addTarget { _
in .success } ; cmd.pauseCommand.invoke() }
  func nextOrQueueFallback(_ fallback: ()->Void){
    if cmd.nextTrackCommand.isEnabled
{ cmd.nextTrackCommand.invoke() } else { fallback() }
  }
  func prevOrQueueFallback(_ fallback: ()->Void){
    if cmd.previousTrackCommand.isEnabled
{ cmd.previousTrackCommand.invoke() } else { fallback() }
  }
  func seek(to seconds:Double){
    cmd.changePlaybackPositionCommand.addTarget { _
in .success }

cmd.changePlaybackPositionCommand.invoke(MPChangePlaybackPosi
tionCommandEvent(timestamp:0, positionTime: seconds))
  }
}
```

NowPlayingMirror.swift

```
import Foundation
import MediaPlayer
import UIKit

struct CompactNowPlaying: Codable {
  var title:String?
  var appName:String?
  var isPlaying:Bool
  var elapsed:Double?
  var duration:Double?
  var supports:[String:Bool]
}

final class NowPlayingMirror {
  static let shared = NowPlayingMirror()
  func snapshot()->CompactNowPlaying {
    let info =
MPNowPlayingInfoCenter×default()×nowPlayingInfo ?? [:]
    let title = info[MPMediaItemPropertyTitle] as? String
    let elapsed =
info[MPNowPlayingInfoPropertyElapsedPlaybackTime] as? Double
    let duration = info[MPMediaItemPropertyPlaybackDuration]
as? Double
    let rate = info[MPNowPlayingInfoPropertyPlaybackRate] as?
Double ?? 0
    let supports = ["next":
MPRemoteCommandCenter.shared().nextTrackCommand.isEnabled,
                    "prev":
MPRemoteCommandCenter.shared().previousTrackCommand.isEnabled
,
                    "seek":
MPRemoteCommandCenter.shared().changePlaybackPositionCommand.
isEnabled]
    return CompactNowPlaying(title: title, appName: nil,
isPlaying: rate>0, elapsed: elapsed, duration: duration,
supports: supports)
  }
}
```

AutoNextMonitor.swift

```
import Foundation

final class AutoNextMonitor {
  static let shared = AutoNextMonitor()
  private var timer: Timer?
  var enabled = true
  var threshold: Double = 8.0

  func start(){ timer?.invalidate(); timer =
Timer.scheduledTimer(withTimeInterval: 2.0, repeats: true){ _
in self.tick() } }
  private func tick(){
    guard enabled else { return }
    let s = NowPlayingMirror.shared.snapshot()
    if let d = s.duration, let e = s.elapsed, (d-e) <
threshold, s.isPlaying == false {
      Task { await MediaCapability.playNext() }
    }
  }
}
```

QueueService.swift

```
import Foundation

final class QueueService {
  static let shared = QueueService()
  private let cacheName = "queue.json"

  func list(userId:String) async -> [QueueItem] {
    if let items:[QueueItem] =
LocalCache.shared.read(cacheName, as:[QueueItem].self)
{ return items }
    return []
  }

  func syncFromConvex(userId:String) async {
    struct Item:Decodable{
      let
_id:String,userId:String,source:String,originalURL:String,nor
malizedURL:String,videoId:String?,title:String,thumbnailURL:S
tring?,duration:Double?,addedAt:Double
    }
    let rows:[Item] = try! await
ConvexClient.shared.call("query/queue:list",
["userId":userId])
    let mapped = rows.map { r in
      QueueItem(source:r.source,
originalURL:URL(string:r.originalURL)!,
normalizedURL:URL(string:r.normalizedURL)!,
videoId:r.videoId, title:r.title,
thumbnailURL:r.thumbnailURL.flatMap(URL.init(string:)),
duration:r.duration.map(TimeInterval.init))
    }
    LocalCache.shared.write(cacheName, mapped)
  }

  func add(_ item:QueueItem, userId:String) async {
    _ = try? await ConvexClient.shared.call("mutation/
queue:add", [
      "userId":userId, "source":item.source,
      "originalURL": item.originalURL.absoluteString,
      "normalizedURL": item.normalizedURL.absoluteString,
      "videoId": item.videoId as Any, "title": item.title,
      "thumbnailURL": item.thumbnailURL?.absoluteString as
Any, "duration": item.duration as Any
    ]) as [String:String]
  }
}
```

MediaCapability.swift (request handler)

```
import Foundation

final class MediaCapability: Capability {
  var domain: String { "media" }

  static func playNext() async {
    // Simplest: open first item (extend to sessions/
armedNext)
    let items = await QueueService.shared.list(userId: "me")
    guard let first = items.first else { return }
    Launcher.play(first)
  }

  func handle(_ msg: COSMessage) async -> COSMessage {
    switch msg.action {
      case "play": RemoteBridge.shared.play()
      case "pause": RemoteBridge.shared.pause()
      case "next": RemoteBridge.shared.nextOrQueueFallback
{ Task { await Self.playNext() } }
      case "prev": RemoteBridge.shared.prevOrQueueFallback
{ /* implement queue back if desired */ }
      case "seek":
        if let s = msg.payload?["seconds"]?.value as? Double
{ RemoteBridge.shared.seek(to:s) }
      case "state":
        let snap = NowPlayingMirror×shared×snapshot()
        let data = try? JSONEncoder().encode(snap)
        return COSMessage(op:.response, id:msg.id,
domain:msg.domain, action:msg.action, payload:
["data":AnyCodable(data?.base64EncodedString() ?? "")])
      default: break
    }
    return COSMessage(op:.response, id:msg.id,
domain:msg.domain, action:msg.action, payload:nil)
  }
}
```

💬 **Comms capability (OAuth-first LLMs)**

Providers/OAuth files

```
ProviderConfig.swift

import Foundation

struct ProviderConfig {
  let id:String
  let authEndpoint:URL?
  let tokenEndpoint:URL?
  let clientId:String?
  let redirectURI:String?
  let scopes:[String]
  let usesOAuth:Bool
  static var gemini: ProviderConfig {
    .init(id:"gemini",
          authEndpoint: URL(string:"https://
accounts.google.com/o/oauth2/v2/auth"),
          tokenEndpoint: URL(string:"https://
oauth2.googleapis.com/token"),
          clientId: Constants.googleClientId,
          redirectURI: Constants.googleRedirect,
          scopes: ["https://www.googleapis.com/auth/
generative-language", "openid", "email", "profile",
"offline_access"],
          usesOAuth: true)
  }

  static var openAIProxy: ProviderConfig {
    guard let a=Constants×openaiAuthEndpoint, let
t=Constants.openaiTokenEndpoint,
          let c=Constants.openaiClientId, let
r=Constants×openaiRedirect else {
      return .init(id:"openai", authEndpoint:nil,
tokenEndpoint:nil, clientId:nil, redirectURI:nil, scopes: [],
usesOAuth:false)
    }
    return .init(id:"openai", authEndpoint: URL(string:a),
tokenEndpoint: URL(string:t), clientId: c, redirectURI: r,
scopes: Constants.openaiScopes ??
["openid","offline_access"], usesOAuth:true)
  }

  static var localHTTP: ProviderConfig {
    .init(id:"localHTTP", authEndpoint:nil,
tokenEndpoint:nil, clientId:nil, redirectURI:nil, scopes: [],
usesOAuth:false)
  }
}
```

TokenStore.swift

```
import Foundation

struct OAuthToken: Codable { let accessToken:String; let
refreshToken:String?; let expiry:Date? }

enum TokenStore {
  static func save(_ t:OAuthToken, provider:String){ let
d=try! JSONEncoder().encode(t); Keychain.set(d,
key:"cos.token.\(provider)") }
  static func load(provider:String)->OAuthToken?{
    guard let d=Keychain.get("cos.token.\(provider)") else
{ return nil }
    return try? JSONDecoder().decode(OAuthToken.self, from:
d)
  }
  static func setAPIKey(_ key:String, provider:String)
{ Keychain.setString(key, key:"cos.apikey.\(provider)") }
  static func getAPIKey(provider:String)->String?
{ Keychain.getString("cos.apikey.\(provider)") }
}
```

OAuthService.swift **(PKCE + ASWebAuthenticationSession; shortened)**

```
import Foundation
import AuthenticationServices
import CryptoKit

final class OAuthService: NSObject {
  static let shared = OAuthService(); private var session:
ASWebAuthenticationSession?

  func signIn(config: ProviderConfig) async throws ->
OAuthToken {
    guard config.usesOAuth, let auth=config.authEndpoint, let
token=config.tokenEndpoint,
          let client=config.clientId, let
redirect=config.redirectURI else { throw
NSError(domain:"oauth", code:0) }
    let (verifier, challenge) = pkce(); let state =
UUID()×uuidString
    var u = URLComponents(url: auth, resolvingAgainstBaseURL:
false)!
    u×queryItems = [
      .init(name:"response_type", value:"code"),
      .init(name:"client_id", value: client),
      .init(name:"redirect_uri", value: redirect),
      .init(name:"scope", value:
config.scopes.joined(separator:" ")),
      .init(name:"state", value: state),
      .init(name:"code_challenge", value: challenge),
      .init(name:"code_challenge_method", value:"S256")
    ]
    let cb = try await presentWebAuth(start: u.url!, scheme:
URL(string: redirect)!.scheme!)
    guard let code = URLComponents(url: cb,
resolvingAgainstBaseURL:false)?.queryItems?×first(where:
{$0.name=="code"})?.value
    else { throw NSError(domain:"oauth", code:1) }

    var req = URLRequest(url: token); req×httpMethod="POST"
    req.addValue("application/x-www-form-urlencoded",
forHTTPHeaderField:"Content-Type")
    req×httpBody = "grant_type=authorization_code&code=\
(code)&client_id=\(client)&redirect_uri=\
(redirect)&code_verifier=\(verifier)".data(using:.utf8)
    let (data,_) = try await URLSession.shared.data(for: req)
    struct R:Decodable{ let access_token:String; let
refresh_token:String?; let expires_in:Double? }
    let r = try JSONDecoder().decode(R.self, from: data)
    return OAuthToken(accessToken:r.access_token,
refreshToken:r.refresh_token, expiry:
```

```
r.expires_in.map{ Date().addingTimeInterval($0) })
  }

  func refresh(config: ProviderConfig, refreshToken:String)
async throws -> OAuthToken {
    guard let token=config.tokenEndpoint, let
client=config.clientId else { throw NSError(domain:"oauth",
code:2) }
    var req = URLRequest(url: token); req×httpMethod="POST"
    req.addValue("application/x-www-form-urlencoded",
forHTTPHeaderField:"Content-Type")
    req×httpBody = "grant_type=refresh_token&refresh_token=\
(refreshToken)&client_id=\(client)".data(using:.utf8)
    let (data,_) = try await URLSession.shared.data(for: req)
    struct R:Decodable{ let access_token:String; let
refresh_token:String?; let expires_in:Double? }
    let r = try JSONDecoder().decode(R.self, from: data)
    return OAuthToken(accessToken:r.access_token,
refreshToken:r.refresh_token ?? refreshToken, expiry:
r.expires_in.map{ Date().addingTimeInterval($0) })
  }

  private func presentWebAuth(start: URL, scheme: String)
async throws -> URL {
    try await withCheckedThrowingContinuation { cont in
      session = ASWebAuthenticationSession(url: start,
callbackURLScheme: scheme) { url, err in
        if let err { cont.resume(throwing: err); return }
        cont.resume(returning: url!)
      }
      session?.prefersEphemeralWebBrowserSession = true
      session?.start()
    }
  }

  private func pkce()->(String,String){
    let verifier = Data((0..<32).map{ _ in
UInt8.random(in:0...255) }).base64URLEncoded()
    let challenge = Data(SHA256.hash(data:
Data(verifier.utf8))).base64URLEncoded()
    return (verifier, challenge)
  }
}
fileprivate extension Data { func base64URLEncoded()->String{
  self.base64EncodedString().replacingOccurrences(of:"+",
with:"-").replacingOccurrences(of:"/",
with:"_")×replacingOccurrences(of:"=", with:"")
}}
```

LLMRouter.swift

```
import Foundation

struct ChatRequest: Codable { let router:String?; let
text:String; let threadId:String?; let command:String?; let
meta:[String:String]? }
struct ChatResponse: Codable { let text:String }

protocol LLMProvider { var id:String { get } func chat(_ req:
ChatRequest, token: String?) async throws -> ChatResponse }

final class LLMRouter {
  static let shared = LLMRouter()
  private var providers:[String:LLMProvider]=[:]; private var
configs:[String:ProviderConfig]=[:]

  func register(_ p: LLMProvider, config: ProviderConfig)
{ providers[p.id]=p; configs[p.id]=config }

  private func token(for provider:String) async throws ->
String? {
    guard let cfg = configs[provider] else { return nil }
    if !cfg.usesOAuth {
      if provider=="openai", let k=Constants.openaiApiKey, !
k.isEmpty { return k }
      return nil
    }
    if let saved = TokenStore×load(provider: provider) {
      if let exp=saved.expiry, exp.timeIntervalSinceNow > 60
{ return saved.accessToken }
      if let rt=saved×refreshToken, let t = try? await
OAuthService.shared.refresh(config: cfg, refreshToken: rt) {
        TokenStore.save(t, provider: provider); return
t.accessToken
      }
    }
    let t = try await OAuthService.shared.signIn(config: cfg)
    TokenStore.save(t, provider: provider)
    return t.accessToken
  }

  func route(_ req: ChatRequest, userId:String) async throws
-> ChatResponse {
    let router = req.router ??
(UserDefaults.standard.string(forKey:"cos.lastRouter") ??
"gemini")
    guard let p = providers[router] else { throw
NSError(domain:"llm", code:404) }
    let tok = try await token(for: router)
    return try await p.chat(req, token: tok)
  }
}
```

GoogleAIProvider.swift (Gemini, OAuth token)

```
import Foundation

final class GoogleAIProvider: LLMProvider {
  let id = "gemini"
  func chat(_ req: ChatRequest, token: String?) async throws
-> ChatResponse {
    guard let token else { throw NSError(domain:"gemini",
code:401) }
    // Simple text-only prompt
    var u = URLRequest(url: URL(string:"https://
generativelanguage.googleapis.com/v1beta/models/gemini-1.5-
pro:generateContent")!)
    u×httpMethod="POST"
    u.addValue("Bearer \(token)",
forHTTPHeaderField:"Authorization")
    u.addValue("application/json","Content-Type")
    u×httpBody = """
      {"contents":[{"parts":[{"text":\(json(req.text))}]}]}
    """.data(using:.utf8)
    let (data, _) = try await URLSession.shared.data(for: u)
    // parse minimal
    struct R:Decodable{ struct C:Decodable{ struct
P:Decodable{ let text:String? }; let parts:[P] }; let
candidates:[struct { let content:C }]}
    let obj = try JSONSerialization.jsonObject(with: data)
as? [String:Any]
    let text = (((obj?["candidates"] as? [Any])?.first as?
[String:Any])?["content"] as? [String:Any])?["parts"] as?
[[String:Any]]
    let out = text?.compactMap{$0["text"] as?
String}.joined(separator:" ") ?? ""
    return ChatResponse(text: out)
  }
  private func json(_ s:String)->String { "\"\
(s.replacingOccurrences(of:"\"", with:"\\\""))\"" }
}
```

OpenAIProvider.swift (proxy OAuth or API key)

```
import Foundation

final class OpenAIProvider: LLMProvider {
  let id = "openai"
  func chat(_ req: ChatRequest, token: String?) async throws
-> ChatResponse {
    let key = token ??
TokenStore.getAPIKey(provider:"openai")
    guard let key else { throw NSError(domain:"openai",
code:401) }
    var u = URLRequest(url: URL(string:"https://
api.openai.com/v1/chat/completions")!)
    u×httpMethod="POST"
    u.addValue("Bearer \(key)",
forHTTPHeaderField:"Authorization")
    u.addValue("application/json","Content-Type")
    u×httpBody = """
      {"model":"gpt-4o-mini","messages":
[{"role":"user","content":\(json(req.text))}]}
    """.data(using:.utf8)
    let (data,_) = try await URLSession.shared.data(for: u)
    let obj = try JSONSerialization.jsonObject(with: data)
as? [String:Any]
    let choices = obj?["choices"] as? [[String:Any]]
    let msg = choices?.first?["message"] as? [String:Any]
    let content = msg?["content"] as? String ?? ""
    return ChatResponse(text: content)
  }
  private func json(_ s:String)->String { "\"\
(s.replacingOccurrences(of:"\"", with:"\\\""))\"" }
}
```

LocalHTTPProvider.swift

```
import Foundation

final class LocalHTTPProvider: LLMProvider {
  let id = "localHTTP"
  func chat(_ req: ChatRequest, token: String?) async throws
-> ChatResponse {
    var r = URLRequest(url: URL(string: "\
(Constants.localBaseURL)/chat")!)
    r×httpMethod="POST"; r.addValue("application/
json","Content-Type")
    if let t = Constants.localBearer, !t.isEmpty
{ r.addValue("Bearer \(t)",
forHTTPHeaderField:"Authorization") }
    r×httpBody = try JSONEncoder().encode(req)
    let (data,_) = try await URLSession.shared.data(for: r)
    return try JSONDecoder().decode(ChatResponse.self, from:
data)
  }
}
```

CommsCapability.swift

```
import Foundation

final class CommsCapability: Capability {
  var domain: String { "comms" }

  init(){
    LLMRouter.shared.register(GoogleAIProvider(),
config: .gemini)
    LLMRouter.shared.register(OpenAIProvider(),
config: .openAIProxy)
    LLMRouter.shared.register(LocalHTTPProvider(),
config: .localHTTP)
  }

  func handle(_ msg: COSMessage) async -> COSMessage {
    switch msg.action {
      case "chat":
        let text = (msg.payload?["text"]?.value as?
String) ?? ""
        let router = msg.payload?["router"]?.value as? String
        let thread = msg.payload?["threadId"]?.value as?
String
        let req = ChatRequest(router: router, text: text,
threadId: thread, command: nil, meta: nil)
        do {
          let res = try await LLMRouter.shared.route(req,
userId:"me")
          return COSMessage(op:.response, id:msg.id,
domain:msg.domain, action:msg.action, payload:
["text":AnyCodable(res.text)])
        } catch {
          return COSMessage(op:.response, id:msg.id,
domain:msg.domain, action:msg.action, error:
COSError(code:"chat_error", message:"\(error)"))
        }
      default:
        return COSMessage(op:.response, id:msg.id,
domain:msg.domain, action:msg.action, error:
COSError(code:"unknown_action", message:"\(msg.action)"))
    }
  }
}
```

ActionsCapability.swift (shortcuts)

```
import Foundation
import UIKit

final class ActionsCapability: Capability {
  var domain: String { "actions" }
  func handle(_ msg: COSMessage) async -> COSMessage {
    if msg.action == "runShortcut", let name = msg.payload?
["name"]?.value as? String {
      let urlStr = "shortcuts://run-shortcut?name=\
(name.addingPercentEncoding(withAllowedCharacters:.urlQueryAl
lowed)!)"
      if let url = URL(string:urlStr)
{ UIApplication.shared.open(url) }
```

```
      return COSMessage(op:.response, id:msg.id,
domain:domain, action:msg.action, payload:
["status":AnyCodable("launched")])
    }
    return COSMessage(op:.response, id:msg.id, domain:domain,
action:msg.action, error: COSError(code:"unknown_action",
message: msg.action))
  }
}
```

NotesCapability.swift, SearchCapability.swift can be minimal stubs following the same pattern.

Share Extension and AppIntents are standard stubs; keep as-is unless you need me to paste them too.

🙋🏽‍♀️ **App Store vibe check (your "Spectacles, testicles, wallet, and watch" blessing)**

Short version: CompanionOS can pass review if you:

- Use **public APIs** (you are).
- Don't fake/automate other apps' UIs (you aren't).
- Clearly disclose OAuth sign-ins and data handling (you will).
- Avoid misleading claims about controlling system features that aren't allowed (say what it supports per app).
- Keep sideloading/dev flags out of the App Store build.

What **might** irritate them:

- Being a "meta-launcher" that feels like a replacement store or universal automation layer. Mitigate by positioning as **accessibility companion** that integrates **Shortcuts/App Intents** and user-initiated deep links.
- If your marketing implies universal control of third-party queues. Phrase it as: "CompanionOS provides *your* cross-app queue and uses supported system controls when available."

Your strategy is solid: build it right, frame it as accessibility, show real user benefit. Review teams respond well to clear, respectful docs + demo videos of accessibility wins.

⚙️ **Defaults per your edits**

- **Default auth path: OAuth** (Gemini + OpenAI-proxy configurations are first-class).
- **API keys**: secondary fallback (stored in Keychain).
- **Sticky threads** per model/provider by default (Convex defaultThreads + Settings cache).
- **Watch is primary**: all commands route over WC to iPhone; tokens never live on the Watch.

You're right—you'll need those. Let's bundle the missing pieces so you can build your way and have a clean story for disability + first-responder use. Below are paste-ready files to drop straight into your repo:

🕰️ **watch/ (companion contracts + sample code)**

watch/contracts.md

~~~
# CompanionOS Watch ⇔ iPhone contracts

**Envelope**
```json
{
  "op": "request" | "response" | "event",
  "id": "uuid",
  "domain": "media" | "comms" | "actions" | "notes" |
"search",
  "action": "play" | "pause" | "chat" | "runShortcut" |
"save" | "query" | "...",
  "payload": { "...": "..." },
  "error": null | { "code": "string", "message": "string" }
}
~~~

Common requests

- Now Playing:

```
{"op":"request","domain":"media","action":"state","payload":
{}}
```

- Play/Pause/Next:

{"op":"request","domain":"media","action":"play"} (or "pause", "next")

- Seek:

```
{"op":"request","domain":"media","action":"seek","payload":
{"seconds":123.4}}
```

- Chat (sticky thread):

```
{"op":"request","domain":"comms","action":"chat","payload":
{"router":"gemini","text":"what's next?"}}
```

- Run Shortcut:

~~~
{"op":"request","domain":"actions","action":"runShortcut","pa
yload":{"name":"Toggle Lights"}}

## `watch/WatchSession.swift`
```swift
import Foundation
import WatchConnectivity

final class WatchSession: NSObject, WCSessionDelegate,
ObservableObject {
  static let shared = WatchSession()
  @Published var lastResponse: Data?
  private override init() { super.init() }

  func start() {
    guard WCSession.isSupported() else { return }
    let s = WCSession.default; s.delegate = self;
s.activate()
  }

  func send(_ message: COSMessage, completion: @escaping
(Result<COSMessage,Error>)->Void) {
    guard WCSession.default.isReachable else {
~~~

```
      completion(.failure(NSError(domain:"wc", code:0,
userInfo:[NSLocalizedDescriptionKey:"Phone unreachable"])));
return
    }
    let data = try! JSONEncoder().encode(message)
    WCSession.default.sendMessage(["data": data],
replyHandler: { reply in
      if let resData = reply["data"] as? Data, let res = try?
JSONDecoder().decode(COSMessage.self, from: resData) {
        completion(.success(res))
      } else if let err = reply["error"] as? String {
        completion(.failure(NSError(domain:"wc", code:1,
userInfo:[NSLocalizedDescriptionKey:err])))
      } else {
        completion(.failure(NSError(domain:"wc", code:2,
userInfo:[NSLocalizedDescriptionKey:"Bad reply"])))
      }
    }, errorHandler: { completion(.failure($0)) })
  }

  // WCSessionDelegate
  func session(_ session: WCSession,
activationDidCompleteWith activationState:
WCSessionActivationState, error: Error?) {}
  #if os(watchOS)
  func sessionReachabilityDidChange(_ session: WCSession) {}
  #endif
}

// Minimal mirror of iOS COSMessage so the Watch can encode/
decode
struct COSMessage: Codable {
  enum Op: String, Codable { case request, response, event }
  var op: Op
  var id: String
  var domain: String
  var action: String
  var payload: [String:AnyCodable]?
  var error: COSError?
  init(op: Op, domain: String, action: String, payload:
[String:AnyCodable]?=nil) {
    self.op = op; self.id = UUID().uuidString; self.domain =
domain; self.action = action; self.payload = payload
  }
}
struct COSError: Codable { let code: String; let message:
String }
struct AnyCodable: Codable {
  let value: Any
  init(_ v: Any) { value = v }
  init(from d: Decoder) throws {
    let c = try d.singleValueContainer()
    if let v = try? c.decode(Bool.self) { value=v; return }
    if let v = try? c.decode(Double.self) { value=v; return }
    if let v = try? c.decode(String.self) { value=v; return }
    if let v = try? c.decode([String:AnyCodable].self)
{ value=v; return }
    if let v = try? c.decode([AnyCodable].self) { value=v;
return }
    value = NSNull()
  }
  func encode(to e: Encoder) throws {
    var c = e.singleValueContainer()
    switch value {
      case let v as Bool: try c.encode(v)
      case let v as Double: try c.encode(v)
      case let v as String: try c.encode(v)
      case let v as [String:AnyCodable]: try c.encode(v)
      case let v as [AnyCodable]: try c.encode(v)
      default: try c.encodeNil()
    }
  }
```

```
}
```

watch/Samples.swift (one-tap helpers for your UI)

```swift
import Foundation

enum WatchSamples {
  static func requestNowPlaying() -> COSMessage {
    COSMessage(op: .request, domain: "media", action: 
"state")
  }
  static func play() -> COSMessage {
    COSMessage(op: .request, domain: "media", action: "play")
  }
  static func next() -> COSMessage {
    COSMessage(op: .request, domain: "media", action: "next")
  }
  static func chatGemini(_ text:String) -> COSMessage {
    COSMessage(op: .request, domain: "comms", action: "chat",
               payload: ["router": AnyCodable("gemini"), 
"text": AnyCodable(text)])
  }
  static func runShortcut(_ name:String) -> COSMessage {
    COSMessage(op: .request, domain: "actions", action: 
"runShortcut",
               payload: ["name": AnyCodable(name)])
  }
}
```

🧪 Postman Collection (Convex)

Save as tools/CompanionOS-Convex.postman_collection.json. Import to Postman (or Bruno/Insomnia).

Variables expected:

- {{baseUrl}} → your Convex deployment (e.g., https://YOUR.convex.cloud)
- {{auth}} → your Convex auth token (same you've set in env)

```
{
  "info": {
    "name": "CompanionOS Convex",
    "_postman_id": "b2f7c6b0-8b62-4af9-a0f1-2c2b2d98af01",
    "description": "Queries & mutations for CompanionOS 
backend (Convex)",
    "schema": "https://schema.getpostman.com/json/collection/
v2.1.0/collection.json"
  },
  "item": [
    {
      "name": "queue:list",
      "request": {
        "method": "POST",
        "header": [
          { "key": "Content-Type", "value": "application/
json" },
          { "key": "Authorization", "value": "{{auth}}" }
        ],
        "url": { "raw": "{{baseUrl}}/query/queue:list", 
"host": ["{{baseUrl}}"], "path": ["query","queue:list"] },
        "body": { "mode": "raw", "raw": "{\"userId\":
\"me\"}" }
      }
    },
    {
      "name": "queue:add",
      "request": {
        "method": "POST",
        "header": [
          { "key": "Content-Type", "value": "application/
json" },
          { "key": "Authorization", "value": "{{auth}}" }
        ],
        "url": { "raw": "{{baseUrl}}/mutation/queue:add",
"host": ["{{baseUrl}}"], "path": ["mutation","queue:add"] },
        "body": {
          "mode": "raw",
          "raw": "{\"userId\":\"me\",\"source\":\"youtube\",
\"originalURL\":\"https://www.youtube.com/watch?
v=dQw4w9WgXcQ\",\"normalizedURL\":\"youtube://watch?
v=dQw4w9WgXcQ\",\"videoId\":\"dQw4w9WgXcQ\",\"title\":
\"Example\"}"
        }
      }
    },
    {
      "name": "settings:get",
      "request": {
        "method": "POST",
        "header": [
          { "key": "Content-Type", "value": "application/
json" },
          { "key": "Authorization", "value": "{{auth}}" }
        ],
        "url": { "raw": "{{baseUrl}}/query/settings:get",
"host": ["{{baseUrl}}"], "path": ["query","settings:get"] },
        "body": { "mode": "raw", "raw": "{\"userId\":
\"me\"}" }
      }
    },
    {
      "name": "settings:setDefaultThread",
      "request": {
        "method": "POST",
        "header": [
          { "key": "Content-Type", "value": "application/
json" },
          { "key": "Authorization", "value": "{{auth}}" }
        ],
        "url": { "raw": "{{baseUrl}}/mutation/
settings:setDefaultThread", "host": ["{{baseUrl}}"], "path":
["mutation","settings:setDefaultThread"] },
        "body": { "mode": "raw", "raw": "{\"userId\":\"me\",
\"router\":\"gemini\",\"threadId\":\"inbox\"}" }
      }
    },
    {
      "name": "chats:upsertThread",
      "request": {
        "method": "POST",
        "header": [
          { "key": "Content-Type", "value": "application/
json" },
          { "key": "Authorization", "value": "{{auth}}" }
        ],
        "url": { "raw": "{{baseUrl}}/mutation/
chats:upsertThread", "host": ["{{baseUrl}}"], "path":
["mutation","chats:upsertThread"] },
        "body": { "mode": "raw", "raw": "{\"userId\":\"me\",
\"router\":\"gemini\",\"threadId\":\"inbox\",\"name\":
\"Inbox\"}" }
      }
    },
    {
      "name": "chats:append",
      "request": {
        "method": "POST",
        "header": [
          { "key": "Content-Type", "value": "application/
json" },
          { "key": "Authorization", "value": "{{auth}}" }
        ],
        "url": { "raw": "{{baseUrl}}/mutation/chats:append",
"host": ["{{baseUrl}}"], "path":
["mutation","chats:append"] },
        "body": { "mode": "raw", "raw": "{\"userId\":\"me\",
\"threadId\":\"inbox\",\"router\":\"gemini\",\"role\":
\"user\",\"text\":\"hello\"}" }
      }
    }
  ],
  "variable": [
    { "key": "baseUrl", "value": "https://
YOUR.convex.cloud" },
    { "key": "auth", "value": "YOUR_AUTH_TOKEN" }
  ]
}
```

📎 **Bonus: curl quick tests (drop in README)**

```
# queue:list
curl -s -X POST "{{baseUrl}}/query/queue:list" \
 -H "Authorization: {{auth}}" -H "Content-Type: application/
json" \
 -d '{"userId":"me"}' | jq .

# chats:upsertThread
curl -s -X POST "{{baseUrl}}/mutation/chats:upsertThread" \
 -H "Authorization: {{auth}}" -H "Content-Type: application/
json" \
 -d
'{"userId":"me","router":"gemini","threadId":"inbox","name":"
Inbox"}' | jq .

# settings:setDefaultThread
curl -s -X POST "{{baseUrl}}/mutation/
settings:setDefaultThread" \
 -H "Authorization: {{auth}}" -H "Content-Type: application/
json" \
 -d '{"userId":"me","router":"gemini","threadId":"inbox"}' | jq .
```

🔐 OAuth config (Gemini + OpenAI-proxy) recap

Gemini

- Google Cloud → Create OAuth client (iOS).
- Redirect URI: com.your.bundle:/oauth2redirect/google
- Scopes: https://www.googleapis.com/auth/generative-language openid email profile offline_access
- Put GOOGLE_CLIENT_ID + GOOGLE_REDIRECT_URI in env/.env.

OpenAI via proxy (if you have one)

- Your proxy exposes /auth + /token (OIDC/OAuth).
- Redirect: com.your.bundle:/oauth2redirect/openai
- Scopes: at least openid offline_access api.
- If no proxy yet, fallback to API key (Keychain) still works.

Watch auth story

- iPhone completes OAuth and stores tokens in Keychain.
- Watch never holds tokens; it just sends requests to the phone → phone routes with valid auth and sticky thread.

🧭 App Store mood check (brief, honest)

- You're squarely in **accessibility companion** territory and using **public APIs** (WCSession, Shortcuts, URL schemes, OAuth). This is review-friendly.
- What might raise eyebrows: positioning as a "universal control surface." Keep copy precise: *"CompanionOS provides watch-first accessibility controls via system Shortcuts, deep links, and supported media commands. It also offers user-owned queues and AI assistance with explicit user auth."*