// MARK: - Models

import Foundation

import SwiftUI

import AVFoundation

import Combine

// MARK: - Data Models

struct AIModel {

let name: String

let client: String

let model: String

let type: AIModelType

}

enum AIModelType {

case openai

case claude

case gemini

case deepseek

case groq

}

struct ChatMessage: Identifiable, Equatable {

let id = UUID()

let speaker: String?

let message: String

let timestamp: Date = Date()

}

struct VoiceStyle {

let name: String

let identifier: String

}

// MARK: - API Clients

protocol AIClient {

func streamResponse(systemPrompt: String, messages: [ChatMessage]) -> AsyncThrowingStream<String, Error>

}

class OpenAIClient: AIClient {

private let apiKey: String

private let baseURL: String

private let model: String

init(apiKey: String, baseURL: String = "https://api.openai.com/v1", model: String = "gpt-4o") {

self.apiKey = apiKey

self.baseURL = baseURL

self.model = model

}

func streamResponse(systemPrompt: String, messages: [ChatMessage]) -> AsyncThrowingStream<String, Error> {

AsyncThrowingStream { continuation in

Task {

do {

let url = URL(string: "\(baseURL)/chat/completions")!

var request = URLRequest(url: url)

request.httpMethod = "POST"

request.setValue("Bearer \(apiKey)", forHTTPHeaderField: "Authorization")

request.setValue("application/json", forHTTPHeaderField: "Content-Type")

let apiMessages = [

["role": "system", "content": systemPrompt]

] + messages.compactMap { msg in

guard let speaker = msg.speaker else { return nil }

let role = speaker.contains("Affirmative") ? "assistant" : "user"

return ["role": role, "content": msg.message]

}

let requestBody: [String: Any] = [

"model": model,

"messages": apiMessages,

"stream": true,

"max\_tokens": 1500

]

request.httpBody = try JSONSerialization.data(withJSONObject: requestBody)

let (data, \_) = try await URLSession.shared.data(for: request)

let lines = String(data: data, encoding: .utf8)?.components(separatedBy: "\n") ?? []

for line in lines {

if line.hasPrefix("data: ") && !line.contains("[DONE]") {

let jsonString = String(line.dropFirst(6))

if let jsonData = jsonString.data(using: .utf8),

let json = try? JSONSerialization.jsonObject(with: jsonData) as? [String: Any],

let choices = json["choices"] as? [[String: Any]],

let delta = choices.first?["delta"] as? [String: Any],

let content = delta["content"] as? String {

continuation.yield(content)

}

}

}

continuation.finish()

} catch {

continuation.finish(throwing: error)

}

}

}

}

}

class ClaudeClient: AIClient {

private let apiKey: String

private let model: String

init(apiKey: String, model: String = "claude-3-7-sonnet-latest") {

self.apiKey = apiKey

self.model = model

}

func streamResponse(systemPrompt: String, messages: [ChatMessage]) -> AsyncThrowingStream<String, Error> {

AsyncThrowingStream { continuation in

Task {

do {

let url = URL(string: "https://api.anthropic.com/v1/messages")!

var request = URLRequest(url: url)

request.httpMethod = "POST"

request.setValue(apiKey, forHTTPHeaderField: "x-api-key")

request.setValue("application/json", forHTTPHeaderField: "Content-Type")

request.setValue("2023-06-01", forHTTPHeaderField: "anthropic-version")

let apiMessages = messages.compactMap { msg -> [String: Any]? in

guard let speaker = msg.speaker else { return nil }

let role = speaker.contains("Affirmative") ? "assistant" : "user"

return ["role": role, "content": msg.message]

}

let requestBody: [String: Any] = [

"model": model,

"max\_tokens": 1500,

"system": systemPrompt,

"messages": apiMessages,

"stream": true

]

request.httpBody = try JSONSerialization.data(withJSONObject: requestBody)

let (data, \_) = try await URLSession.shared.data(for: request)

let lines = String(data: data, encoding: .utf8)?.components(separatedBy: "\n") ?? []

for line in lines {

if line.hasPrefix("data: ") {

let jsonString = String(line.dropFirst(6))

if let jsonData = jsonString.data(using: .utf8),

let json = try? JSONSerialization.jsonObject(with: jsonData) as? [String: Any],

let type = json["type"] as? String,

type == "content\_block\_delta",

let delta = json["delta"] as? [String: Any],

let text = delta["text"] as? String {

continuation.yield(text)

}

}

}

continuation.finish()

} catch {

continuation.finish(throwing: error)

}

}

}

}

}

class GeminiClient: AIClient {

private let apiKey: String

private let model: String

init(apiKey: String, model: String = "gemini-2.0-flash") {

self.apiKey = apiKey

self.model = model

}

func streamResponse(systemPrompt: String, messages: [ChatMessage]) -> AsyncThrowingStream<String, Error> {

AsyncThrowingStream { continuation in

Task {

do {

let url = URL(string: "https://generativelanguage.googleapis.com/v1beta/models/\(model):generateContent?key=\(apiKey)")!

var request = URLRequest(url: url)

request.httpMethod = "POST"

request.setValue("application/json", forHTTPHeaderField: "Content-Type")

let fullPrompt = systemPrompt + "\n\n" + (messages.first?.message ?? "Begin.")

let requestBody: [String: Any] = [

"contents": [

["parts": [["text": fullPrompt]]]

]

]

request.httpBody = try JSONSerialization.data(withJSONObject: requestBody)

let (data, \_) = try await URLSession.shared.data(for: request)

if let json = try? JSONSerialization.jsonObject(with: data) as? [String: Any],

let candidates = json["candidates"] as? [[String: Any]],

let content = candidates.first?["content"] as? [String: Any],

let parts = content["parts"] as? [[String: Any]],

let text = parts.first?["text"] as? String {

continuation.yield(text)

}

continuation.finish()

} catch {

continuation.finish(throwing: error)

}

}

}

}

}

// MARK: - Services

class ResearchService: ObservableObject {

private let geminiClient: GeminiClient?

init(apiKey: String?) {

if let apiKey = apiKey {

self.geminiClient = GeminiClient(apiKey: apiKey)

} else {

self.geminiClient = nil

}

}

func getBriefing(for topic: String) async -> (affirmative: String, opposition: String) {

guard let client = geminiClient else {

let fallback = "Research was not available as the Google API key is missing."

return (fallback, fallback)

}

let prompt = """

Please perform a Google search to find the strongest arguments both FOR and AGAINST the topic: '\(topic)'.

Based on your search results, generate two concise, point-form summaries.

The output MUST have two sections. Start the first section with the exact heading '## Affirmative Arguments' and the second with '## Opposition Arguments'.

Response in Chinese

"""

do {

var fullResponse = ""

for try await chunk in client.streamResponse(systemPrompt: "", messages: [ChatMessage(speaker: "user", message: prompt)]) {

fullResponse += chunk

}

if fullResponse.contains("## Opposition Arguments") && fullResponse.contains("## Affirmative Arguments") {

let parts = fullResponse.components(separatedBy: "## Opposition Arguments")

let affirmative = parts[0].replacingOccurrences(of: "## Affirmative Arguments", with: "").trimmingCharacters(in: .whitespacesAndNewlines)

let opposition = parts[1].trimmingCharacters(in: .whitespacesAndNewlines)

return (affirmative, opposition)

} else {

return (fullResponse, fullResponse)

}

} catch {

let errorMessage = "An API error occurred during research: \(error.localizedDescription)"

return (errorMessage, errorMessage)

}

}

}

class TTSService: ObservableObject {

private let synthesizer = AVSpeechSynthesizer()

@Published var isPlaying = false

func generateDebateAudio(messages: [ChatMessage], voices: (affirmative: String, opposition: String)) async -> URL? {

let utterances = await createUtterances(from: messages, voices: voices)

return await synthesizeToFile(utterances: utterances)

}

private func createUtterances(from messages: [ChatMessage], voices: (affirmative: String, opposition: String)) async -> [AVSpeechUtterance] {

var utterances: [AVSpeechUtterance] = []

for message in messages {

guard let speaker = message.speaker, !message.message.isEmpty else { continue }

let voiceIdentifier: String

let speakerIntro: String

if speaker.contains("Affirmative") {

voiceIdentifier = voices.affirmative

speakerIntro = detectLanguage(message.message) == "zh" ? "正方说：" : "Affirmative says: "

} else if speaker.contains("Opposition") {

voiceIdentifier = voices.opposition

speakerIntro = detectLanguage(message.message) == "zh" ? "反方说：" : "Opposition says: "

} else {

continue

}

let fullText = speakerIntro + message.message

let utterance = AVSpeechUtterance(string: fullText)

if let voice = AVSpeechSynthesisVoice(identifier: voiceIdentifier) {

utterance.voice = voice

}

utterance.rate = 0.5

utterance.pitchMultiplier = 1.0

utterance.volume = 1.0

utterances.append(utterance)

}

return utterances

}

private func synthesizeToFile(utterances: [AVSpeechUtterance]) async -> URL? {

let documentsPath = FileManager.default.urls(for: .documentDirectory, in: .userDomainMask)[0]

let audioURL = documentsPath.appendingPathComponent("debate\_audio.caf")

// Note: AVSpeechSynthesizer doesn't directly support file output

// In a real app, you'd need to use AVAudioEngine for recording

// This is a simplified implementation

return audioURL

}

private func detectLanguage(\_ text: String) -> String {

let chineseRange = text.range(of: "[\u{4e00}-\u{9fff}]", options: .regularExpression)

return chineseRange != nil ? "zh" : "en"

}

}

// MARK: - ViewModels

class DebateViewModel: ObservableObject {

@Published var topic = ""

@Published var chatHistory: [ChatMessage] = []

@Published var isDebating = false

@Published var affirmativeModel = "GPT-4o"

@Published var oppositionModel = "Claude 3.7 Sonnet"

@Published var rounds = 2

@Published var affirmativeVoice = "Coral"

@Published var oppositionVoice = "Onyx"

@Published var audioURL: URL?

@Published var audioStatus = ""

@Published var canGenerateAudio = false

@Published var canSaveDebate = false

private let researchService: ResearchService

private let ttsService = TTSService()

private var clients: [String: AIClient] = [:]

let availableModels = [

"GPT-4o", "GPT-4o-mini", "Claude 3.7 Sonnet",

"Gemini-2.0-Flash", "Deepseek-Chat", "Llama-3.3-70b-versatile (Groq)"

]

let voiceStyles = [

"Alloy", "Ash", "Ballad", "Coral", "Echo",

"Fable", "Nova", "Onyx", "Sage", "Shimmer"

]

init(apiKeys: [String: String]) {

self.researchService = ResearchService(apiKey: apiKeys["GOOGLE\_API\_KEY"])

if let openaiKey = apiKeys["OPENAI\_API\_KEY"] {

clients["GPT-4o"] = OpenAIClient(apiKey: openaiKey, model: "gpt-4o")

clients["GPT-4o-mini"] = OpenAIClient(apiKey: openaiKey, model: "gpt-4o-mini")

}

if let claudeKey = apiKeys["ANTHROPIC\_API\_KEY"] {

clients["Claude 3.7 Sonnet"] = ClaudeClient(apiKey: claudeKey)

}

if let geminiKey = apiKeys["GOOGLE\_API\_KEY"] {

clients["Gemini-2.0-Flash"] = GeminiClient(apiKey: geminiKey)

}

if let deepseekKey = apiKeys["DEEPSEEK\_API\_KEY"] {

clients["Deepseek-Chat"] = OpenAIClient(apiKey: deepseekKey, baseURL: "https://api.deepseek.com/v1", model: "deepseek-chat")

}

if let groqKey = apiKeys["GROQ\_API\_KEY"] {

clients["Llama-3.3-70b-versatile (Groq)"] = OpenAIClient(apiKey: groqKey, baseURL: "https://api.groq.com/openai/v1", model: "llama-3.3-70b-versatile")

}

}

@MainActor

func startDebate() async {

guard !topic.isEmpty else { return }

isDebating = true

chatHistory = []

canGenerateAudio = false

canSaveDebate = false

// Add research message

chatHistory.append(ChatMessage(speaker: nil, message: "🔍 Conducting research..."))

// Get research briefing

let briefing = await researchService.getBriefing(for: topic)

// Remove research message and add briefing

chatHistory.removeLast()

chatHistory.append(ChatMessage(speaker: nil, message: "### Affirmative Research Briefing\n\n---\n\(briefing.affirmative)"))

chatHistory.append(ChatMessage(speaker: nil, message: "### Opposition Research Briefing\n\n---\n\(briefing.opposition)"))

chatHistory.append(ChatMessage(speaker: nil, message: "✅ Research complete. The debate will now begin."))

// Create system prompts

let affirmativePrompt = """

You are a world-class debater arguing IN FAVOR of the topic: \(topic).

## Research Briefing (Arguments FOR your stance)

<research>

\(briefing.affirmative)

</research>

Your opening statement should use this research. In subsequent turns, counter your opponent's arguments directly while reinforcing your own. Provide your response in Chinese.

"""

let oppositionPrompt = """

You are a world-class debater arguing AGAINST the topic: \(topic).

## Research Briefing (Arguments AGAINST your stance)

<research>

\(briefing.opposition)

</research>

Your opening statement should use this research. In subsequent turns, counter your opponent's arguments directly while reinforcing your own. Provide your response in Chinese.

"""

// Conduct debate rounds

var affirmativeMessages: [ChatMessage] = []

var oppositionMessages: [ChatMessage] = []

for \_ in 0..<rounds {

// Affirmative turn

await conductTurn(

speaker: "Affirmative (\(affirmativeModel))",

systemPrompt: affirmativePrompt,

modelName: affirmativeModel,

context: affirmativeMessages + oppositionMessages

)

if let lastMessage = chatHistory.last {

affirmativeMessages.append(lastMessage)

}

// Opposition turn

await conductTurn(

speaker: "Opposition (\(oppositionModel))",

systemPrompt: oppositionPrompt,

modelName: oppositionModel,

context: affirmativeMessages + oppositionMessages

)

if let lastMessage = chatHistory.last {

oppositionMessages.append(lastMessage)

}

}

chatHistory.append(ChatMessage(speaker: nil, message: "🏁 Debate finished. You can now generate the full audio and transcript."))

isDebating = false

canGenerateAudio = true

canSaveDebate = true

}

@MainActor

private func conductTurn(speaker: String, systemPrompt: String, modelName: String, context: [ChatMessage]) async {

guard let client = clients[modelName] else {

chatHistory.append(ChatMessage(speaker: speaker, message: "Error: Model not available"))

return

}

let currentMessage = ChatMessage(speaker: speaker, message: "")

chatHistory.append(currentMessage)

var fullResponse = ""

do {

for try await chunk in client.streamResponse(systemPrompt: systemPrompt, messages: context) {

fullResponse += chunk

// Update the last message in chat history

if let index = chatHistory.firstIndex(where: { $0.id == currentMessage.id }) {

chatHistory[index] = ChatMessage(speaker: speaker, message: fullResponse)

}

}

} catch {

fullResponse = "Error: \(error.localizedDescription)"

if let index = chatHistory.firstIndex(where: { $0.id == currentMessage.id }) {

chatHistory[index] = ChatMessage(speaker: speaker, message: fullResponse)

}

}

}

func generateAudio() async {

audioStatus = "Generating audio..."

do {

let url = await ttsService.generateDebateAudio(

messages: chatHistory.filter { $0.speaker != nil },

voices: (affirmative: affirmativeVoice, opposition: oppositionVoice)

)

await MainActor.run {

self.audioURL = url

self.audioStatus = url != nil ? "Audio generated successfully" : "Failed to generate audio"

}

}

}

func saveDebate() -> URL? {

let formatter = DateFormatter()

formatter.dateFormat = "yyyy-MM-dd\_HH-mm-ss"

let timestamp = formatter.string(from: Date())

let fileName = "debate\_\(topic.prefix(30))\_\(timestamp).md"

let documentsPath = FileManager.default.urls(for: .documentDirectory, in: .userDomainMask)[0]

let fileURL = documentsPath.appendingPathComponent(fileName)

var content = "# Debate on: \(topic)\n\n"

content += "> Generated on: \(Date())\n\n---\n\n"

for message in chatHistory {

if let speaker = message.speaker {

content += "\*\*🗣️ \(speaker):\*\*\n\n\(message.message)\n\n---\n\n"

} else {

content += "\*\(message.message)\*\n\n---\n\n"

}

}

do {

try content.write(to: fileURL, atomically: true, encoding: .utf8)

return fileURL

} catch {

print("Error saving debate: \(error)")

return nil

}

}

}

// MARK: - Views

struct ContentView: View {

@StateObject private var viewModel: DebateViewModel

@State private var showingAPIKeySheet = false

init() {

// In a real app, you'd load these from UserDefaults or Keychain

let apiKeys: [String: String] = [

"OPENAI\_API\_KEY": "",

"ANTHROPIC\_API\_KEY": "",

"GOOGLE\_API\_KEY": "",

"DEEPSEEK\_API\_KEY": "",

"GROQ\_API\_KEY": ""

]

\_viewModel = StateObject(wrappedValue: DebateViewModel(apiKeys: apiKeys))

}

var body: some View {

NavigationView {

VStack(spacing: 0) {

// Header

VStack(alignment: .leading, spacing: 16) {

HStack {

Image(systemName: "brain.head.profile")

.font(.title2)

.foregroundColor(.blue)

Text("AI vs. AI Debate")

.font(.title2)

.fontWeight(.bold)

Spacer()

Button("API Keys") {

showingAPIKeySheet = true

}

.font(.caption)

.foregroundColor(.blue)

}

TextField("Enter debate topic...", text: $viewModel.topic)

.textFieldStyle(RoundedBorderTextFieldStyle())

.disabled(viewModel.isDebating)

HStack {

VStack(alignment: .leading, spacing: 8) {

Text("Affirmative")

.font(.caption)

.foregroundColor(.secondary)

Picker("Affirmative Model", selection: $viewModel.affirmativeModel) {

ForEach(viewModel.availableModels, id: \.self) { model in

Text(model).tag(model)

}

}

.pickerStyle(MenuPickerStyle())

.disabled(viewModel.isDebating)

}

VStack(alignment: .leading, spacing: 8) {

Text("Opposition")

.font(.caption)

.foregroundColor(.secondary)

Picker("Opposition Model", selection: $viewModel.oppositionModel) {

ForEach(viewModel.availableModels, id: \.self) { model in

Text(model).tag(model)

}

}

.pickerStyle(MenuPickerStyle())

.disabled(viewModel.isDebating)

}

}

HStack {

VStack(alignment: .leading, spacing: 8) {

Text("Rounds: \(viewModel.rounds)")

.font(.caption)

.foregroundColor(.secondary)

Stepper("", value: $viewModel.rounds, in: 1...5)

.disabled(viewModel.isDebating)

}

Spacer()

if viewModel.isDebating {

ProgressView()

.scaleEffect(0.8)

} else {

Button("Start Debate") {

Task {

await viewModel.startDebate()

}

}

.buttonStyle(.borderedProminent)

.disabled(viewModel.topic.isEmpty)

}

}

}

.padding()

.background(Color(.systemGroupedBackground))

// Chat History

ScrollViewReader { proxy in

ScrollView {

LazyVStack(alignment: .leading, spacing: 12) {

ForEach(viewModel.chatHistory) { message in

ChatBubbleView(message: message)

.id(message.id)

}

}

.padding()

}

.onChange(of: viewModel.chatHistory.count) { \_ in

if let lastMessage = viewModel.chatHistory.last {

withAnimation {

proxy.scrollTo(lastMessage.id, anchor: .bottom)

}

}

}

}

// Bottom Controls

if viewModel.canGenerateAudio || viewModel.canSaveDebate {

VStack(spacing: 12) {

Divider()

HStack {

if viewModel.canGenerateAudio {

VStack(alignment: .leading, spacing: 8) {

HStack {

Picker("Aff Voice", selection: $viewModel.affirmativeVoice) {

ForEach(viewModel.voiceStyles, id: \.self) { voice in

Text(voice).tag(voice)

}

}

.pickerStyle(MenuPickerStyle())

Picker("Opp Voice", selection: $viewModel.oppositionVoice) {

ForEach(viewModel.voiceStyles, id: \.self) { voice in

Text(voice).tag(voice)

}

}

.pickerStyle(MenuPickerStyle())

}

Button("Generate Audio") {

Task {

await viewModel.generateAudio()

}

}

.buttonStyle(.bordered)

if !viewModel.audioStatus.isEmpty {

Text(viewModel.audioStatus)

.font(.caption)

.foregroundColor(.secondary)

}

}

}

Spacer()

if viewModel.canSaveDebate {

Button("Save Debate") {

\_ = viewModel.saveDebate()

}

.buttonStyle(.bordered)

}

}

}

.padding()

.background(Color(.systemGroupedBackground))

}

}

}

.sheet(isPresented: $showingAPIKeySheet) {

APIKeyConfigView()

}

}

}

struct ChatBubbleView: View {

let message: ChatMessage

var body: some View {

HStack(alignment: .top, spacing: 12) {

if let speaker = message.speaker {

// Speaker avatar

Circle()

.fill(speaker.contains("Affirmative") ? Color.blue : Color.red)

.frame(width: 32, height: 32)

.overlay(

Text(speaker.contains("Affirmative") ? "A" : "O")

.font(.caption)

.fontWeight(.bold)

.foregroundColor(.white)

)

VStack(alignment: .leading, spacing: 4) {

Text(speaker)

.font(.caption)

.fontWeight(.semibold)

.foregroundColor(speaker.contains("Affirmative") ? .blue : .red)

Text(message.message)

.padding(12)

.background(Color(.systemGray6))

.cornerRadius(12)

}

Spacer()

} else {

// System message

HStack {

Spacer()

Text(message.message)

.font(.caption)

.multilineTextAlignment(.center)

.foregroundColor(.secondary)

.padding(8)

.background(Color(.systemGray5))

.cornerRadius(8)

Spacer()

}

}

}

}

}

struct APIKeyConfigView: View {

@Environment(\.dismiss) private var dismiss

@State private var openaiKey = ""

@State private var claudeKey = ""

@State private var googleKey = ""

@State private var deepseekKey = ""

@State private var groqKey = ""

var body: some View {

NavigationView {

Form {

Section("API Configuration") {

SecureField("OpenAI API Key", text: $openaiKey)

SecureField("Anthropic API Key", text: $claudeKey)

SecureField("Google API Key", text: $googleKey)

SecureField("DeepSeek API Key", text: $deepseekKey)

SecureField("Groq API Key", text: $groqKey)

}

Section {

Text("Configure your API keys to enable different AI models. Keys are stored securely on your device.")

.font(.caption)

.foregroundColor(.secondary)

}

}

.navigationTitle("API Keys")

.navigationBarTitleDisplayMode(.inline)

.toolbar {

ToolbarItem(placement: .navigationBarLeading) {

Button("Cancel") {

dismiss()

}

}

ToolbarItem(placement: .navigationBarTrailing) {

Button("Save") {

// In a real app, save to Keychain

dismiss()

}

}

}

}

}

}

// MARK: - App Entry Point

**import** SwiftUI

**@main**

**struct** ClashAIApp: App {

**var** body: **some** Scene {

WindowGroup {

ContentView()

}

}

}  
  
**import** SwiftUI

**struct** AudioControlsView: View {

@ObservedObject **var** viewModel: DebateViewModel

**var** body: **some** View {

**if** viewModel.canGenerateAudio || viewModel.canSaveDebate {

VStack(spacing: 12) {

Divider()

HStack {

**if** viewModel.canGenerateAudio {

VStack(alignment: .leading, spacing: 8) {

HStack {

Picker("Aff Voice", selection: $viewModel.affirmativeVoice) {

ForEach(viewModel.voiceStyles, id: \.**self**) { voice **in**

Text(voice).tag(voice)

}

}

.pickerStyle(MenuPickerStyle())

Picker("Opp Voice", selection: $viewModel.oppositionVoice) {

ForEach(viewModel.voiceStyles, id: \.**self**) { voice **in**

Text(voice).tag(voice)

}

}

.pickerStyle(MenuPickerStyle())

}

Button("Generate Audio") {

Task {

**await** viewModel.generateAudio()

}

}

.buttonStyle(.bordered)

**if** !viewModel.audioStatus.isEmpty {

Text(viewModel.audioStatus)

.font(.caption)

.foregroundColor(.secondary)

}

}

}

Spacer()

**if** viewModel.canSaveDebate {

Button("Save Debate") {

\_ = viewModel.saveDebate()

}

.buttonStyle(.bordered)

}

}

}

.padding()

.background(Color(.systemGroupedBackground))

}

}

}  
  
**import** SwiftUI

**struct** DebateControlsView: View {

@ObservedObject **var** viewModel: DebateViewModel

**var** body: **some** View {

VStack(alignment: .leading, spacing: 16) {

TextField("Enter debate topic...", text: $viewModel.topic)

.textFieldStyle(RoundedBorderTextFieldStyle())

.disabled(viewModel.isDebating)

HStack {

VStack(alignment: .leading, spacing: 8) {

Text("Affirmative")

.font(.caption)

.foregroundColor(.secondary)

Picker("Affirmative Model", selection: $viewModel.affirmativeModel) {

ForEach(viewModel.availableModels, id: \.**self**) { model **in**

Text(model).tag(model)

}

}

.pickerStyle(MenuPickerStyle())

.disabled(viewModel.isDebating)

}

VStack(alignment: .leading, spacing: 8) {

Text("Opposition")

.font(.caption)

.foregroundColor(.secondary)

Picker("Opposition Model", selection: $viewModel.oppositionModel) {

ForEach(viewModel.availableModels, id: \.**self**) { model **in**

Text(model).tag(model)

}

}

.pickerStyle(MenuPickerStyle())

.disabled(viewModel.isDebating)

}

}

HStack {

VStack(alignment: .leading, spacing: 8) {

Text("Rounds: \(viewModel.rounds)")

.font(.caption)

.foregroundColor(.secondary)

Stepper("", value: $viewModel.rounds, in: 1...5)

.disabled(viewModel.isDebating)

}

Spacer()

**if** viewModel.isDebating {

ProgressView()

.scaleEffect(0.8)

} **else** {

Button("Start Debate") {

Task {

**await** viewModel.startDebate()

}

}

.buttonStyle(.borderedProminent)

.disabled(viewModel.topic.isEmpty)

}

}

}

}

}