ZAI Native Android Chat Client: Architectural Specification & Implementation

An ultra-premium, native Android AI chat client engineered for personal connection to `chat.z.ai` (Open WebUI fork). Designed with the fluid motion of Moonshot Kimi, the typographic elegance of Claude, and strict senior-level Android engineering practices: Kotlin 2.x, Jetpack Compose (BOM latest), Material 3 with a bespoke design system, manual line-by-line OkHttp Server-Sent Events (SSE) streaming, Room offline-first persistence with FTS4 full-text search, and spring-physics animations.

1. Project Directory Structure

```
app/
├── build.gradle.kts
├── proguard-rules.pro
└── src/
    └── main/
        ├── AndroidManifest.xml
        └── java/com/zai/chat/
            ├── ZaiApplication.kt
            ├── config/
            │   └── ZaiConfig.kt
            ├── data/
            │   ├── local/
            │   │   ├── ZaiDatabase.kt
            │   │   ├── dao/
            │   │   │   ├── ChatDao.kt
            │   │   │   └── MessageDao.kt
            │   │   ├── entity/
            │   │   │   ├── ChatEntity.kt
            │   │   │   ├── MessageEntity.kt
            │   │   │   └── MessageFtsEntity.kt
            │   │   └── preferences/
            │   │       ├── SettingsDataStore.kt
            │   │       └── TokenManager.kt
            │   ├── model/
            │   │   ├── Chat.kt
            │   │   ├── Message.kt
            │   │   ├── MessageRole.kt
            │   │   ├── ModelOption.kt
            │   │   └── SearchCitation.kt
            │   └── repository/
            │       ├── ChatRepository.kt
            │       └── ChatRepositoryImpl.kt
            ├── di/
            │   ├── AppModule.kt
            │   ├── DatabaseModule.kt
            │   └── NetworkModule.kt
            ├── network/
            │   ├── ZaiApiService.kt
            │   ├── auth/
```

```
│    │    ├── AuthAuthenticator.kt
│    │    ├── AuthEvent.kt
│    │    └── AuthInterceptor.kt
│    ├── model/
│    │    ├── ChatCompletionRequest.kt
│    │    ├── ChatCompletionChunk.kt
│    │    ├── ChatListResponse.kt
│    │    └── FileUploadResponse.kt
│    └── sse/
│         ├── ManualSseReader.kt
│         └── StreamEvent.kt
└── ui/
    ├── MainActivity.kt
    ├── components/
    │    ├── CodeBlockView.kt
    │    ├── MarkdownRenderer.kt
    │    ├── MessageActionToolbar.kt
    │    ├── MessageBubble.kt
    │    ├── StreamingCursor.kt
    │    ├── ThinkingBlock.kt
    │    └── WebCitationStrip.kt
    ├── screens/
    │    ├── chat/
    │    │    ├── ChatComposer.kt
    │    │    ├── ChatScreen.kt
    │    │    ├── ChatUiEvent.kt
    │    │    ├── ChatUiState.kt
    │    │    └── ChatViewModel.kt
    │    ├── drawer/
    │    │    ├── DrawerChatList.kt
    │    │    └── DrawerViewModel.kt
    │    └── settings/
    │         ├── SettingsScreen.kt
    │         └── SettingsViewModel.kt
    └── theme/
         ├── Color.kt
         ├── Motion.kt
         ├── Shape.kt
         ├── Spacing.kt
         ├── Theme.kt
         └── Type.kt
```

2. Build Configuration & Dependency Catalogs

gradle/libs.versions.toml

```toml
[versions]
agp = "8.5.2"
kotlin = "2.0.20"
ksp = "2.0.20-1.0.25"
composeBom = "2024.09.00"
coroutines = "1.8.1"
```

```
hilt = "2.51.1"
okhttp = "4.12.0"
kotlinxSerialization = "1.7.1"
room = "2.6.1"
datastore = "1.1.1"
securityCrypto = "1.1.0-alpha06"
coil = "2.7.0"
lifecycle = "2.8.4"
navigationCompose = "2.8.0"

[libraries]
androidx-core-ktx = { group = "androidx.core", name = "core-ktx", version = "1.1
androidx-lifecycle-runtime-ktx = { group = "androidx.lifecycle", name = "lifecyc
androidx-lifecycle-viewmodel-compose = { group = "androidx.lifecycle", name = "l
androidx-activity-compose = { group = "androidx.activity", name = "activity-comp
androidx-navigation-compose = { group = "androidx.navigation", name = "navigatio

# Compose
compose-bom = { group = "androidx.compose", name = "compose-bom", version.ref =
compose-ui = { group = "androidx.compose.ui", name = "ui" }
compose-ui-graphics = { group = "androidx.compose.ui", name = "ui-graphics" }
compose-ui-tooling-preview = { group = "androidx.compose.ui", name = "ui-tooling
compose-ui-tooling = { group = "androidx.compose.ui", name = "ui-tooling" }
compose-material3 = { group = "androidx.compose.material3", name = "material3" }
compose-material-icons-extended = { group = "androidx.compose.material", name =

# Coroutines
kotlinx-coroutines-core = { group = "org.jetbrains.kotlinx", name = "kotlinx-cor
kotlinx-coroutines-android = { group = "org.jetbrains.kotlinx", name = "kotlinx-

# Serialization & Networking
kotlinx-serialization-json = { group = "org.jetbrains.kotlinx", name = "kotlinx-
okhttp = { group = "com.squareup.okhttp3", name = "okhttp", version.ref = "okhtt
okhttp-logging = { group = "com.squareup.okhttp3", name = "logging-interceptor",

# Persistence
androidx-room-runtime = { group = "androidx.room", name = "room-runtime", versio
androidx-room-ktx = { group = "androidx.room", name = "room-ktx", version.ref =
androidx-room-compiler = { group = "androidx.room", name = "room-compiler", vers
androidx-datastore-preferences = { group = "androidx.datastore", name = "datasto
androidx-security-crypto = { group = "androidx.security", name = "security-crypt

# Dependency Injection
hilt-android = { group = "com.google.dagger", name = "hilt-android", version.ref
hilt-compiler = { group = "com.google.dagger", name = "hilt-compiler", version.r
androidx-hilt-navigation-compose = { group = "androidx.hilt", name = "hilt-navig

# Media & Image Loading
coil-compose = { group = "io.coil-kt", name = "coil-compose", version.ref = "coi

[plugins]
android-application = { id = "com.android.application", version.ref = "agp" }
kotlin-android = { id = "org.jetbrains.kotlin.android", version.ref = "kotlin" }
kotlin-serialization = { id = "org.jetbrains.kotlin.plugin.serialization", versi
kotlin-compose = { id = "org.jetbrains.kotlin.plugin.compose", version.ref = "ko
ksp = { id = "com.google.devtools.ksp", version.ref = "ksp" }
```

```
hilt-android = { id = "com.google.dagger.hilt.android", version.ref = "hilt" }
```

app/build.gradle.kts

```
plugins {
    alias(libs.plugins.android.application)
    alias(libs.plugins.kotlin.android)
    alias(libs.plugins.kotlin.serialization)
    alias(libs.plugins.kotlin.compose)
    alias(libs.plugins.ksp)
    alias(libs.plugins.hilt.android)
}

android {
    namespace = "com.zai.chat"
    compileSdk = 35

    defaultConfig {
        applicationId = "com.zai.chat"
        minSdk = 26
        targetSdk = 35
        versionCode = 1
        versionName = "1.0.0"

        testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"
        vectorDrawables {
            useSupportLibrary = true
        }
    }

    buildTypes {
        release {
            isMinifyEnabled = true
            isShrinkResources = true
            proguardFiles(
                getDefaultProguardFile("proguard-android-optimize.txt"),
                "proguard-rules.pro"
            )
            signingConfig = signingConfigs.getByName("debug")
        }
        debug {
            applicationIdSuffix = ".debug"
        }
    }

    compileOptions {
        sourceCompatibility = JavaVersion.VERSION_17
        targetCompatibility = JavaVersion.VERSION_17
    }

    kotlinOptions {
        jvmTarget = "17"
        freeCompilerArgs += listOf(
```

```
            "-opt-in=kotlinx.coroutines.ExperimentalCoroutinesApi",
            "-opt-in=androidx.compose.material3.ExperimentalMaterial3Api",
            "-opt-in=androidx.compose.animation.ExperimentalAnimationApi",
            "-opt-in=androidx.compose.foundation.ExperimentalFoundationApi"
        )
    }

    buildFeatures {
        compose = true
    }
}

dependencies {
    implementation(libs.androidx.core.ktx)
    implementation(libs.androidx.lifecycle.runtime.ktx)
    implementation(libs.androidx.lifecycle.viewmodel.compose)
    implementation(libs.androidx.activity.compose)
    implementation(libs.androidx.navigation.compose)

    implementation(platform(libs.compose.bom))
    implementation(libs.compose.ui)
    implementation(libs.compose.ui.graphics)
    implementation(libs.compose.ui.tooling.preview)
    implementation(libs.compose.material3)
    implementation(libs.compose.material.icons.extended)
    debugImplementation(libs.compose.ui.tooling)

    implementation(libs.kotlinx.coroutines.core)
    implementation(libs.kotlinx.coroutines.android)
    implementation(libs.kotlinx.serialization.json)
    implementation(libs.okhttp)
    implementation(libs.okhttp.logging)

    implementation(libs.androidx.room.runtime)
    implementation(libs.androidx.room.ktx)
    ksp(libs.androidx.room.compiler)

    implementation(libs.androidx.datastore.preferences)
    implementation(libs.androidx.security.crypto)

    implementation(libs.hilt.android)
    ksp(libs.hilt.compiler)
    implementation(libs.androidx.hilt.navigation.compose)

    implementation(libs.coil.compose)
}
```

3. Central Configuration & Secure Token Storage

`config/ZaiConfig.kt`

```
package com.zai.chat.config

/**
 * Centralized network and operational configuration for chat.z.ai (Open WebUI f
 * All constants are declared here to facilitate immediate updates whenever
 * the upstream server modifies headers, endpoints, or default models.
 */
object ZaiConfig {
    const val BASE_URL: String = "https://chat.z.ai"
    const val COMPLETIONS_PATH: String = "/api/chat/completions"
    const val CHATS_PATH: String = "/api/v1/chats"
    const val MODELS_PATH: String = "/api/models"
    const val UPLOAD_FILE_PATH: String = "/api/v1/files/"

    // Request Headers mirroring authentic web sessions
    const val USER_AGENT: String = "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15
    const val ORIGIN: String = "https://chat.z.ai"
    const val REFERER: String = "https://chat.z.ai/"
    const val FE_VERSION_HEADER_KEY: String = "X-FE-Version"
    const val FE_VERSION_HEADER_VALUE: String = "prod-20241015-v1"

    // Optional HMAC signing salt if enforced by upstream gateway
    val SIGN_SALT: String? = null

    // Default Fallback Model
    const val MODEL_DEFAULT: String = "GLM-4-Plus"

    // Timeouts
    const val CONNECT_TIMEOUT_SECONDS: Long = 30L
    const val READ_TIMEOUT_SECONDS: Long = 120L
    const val WRITE_TIMEOUT_SECONDS: Long = 60L
}
```

data/local/preferences/TokenManager.kt

```
package com.zai.chat.data.local.preferences

import android.content.Context
import android.content.SharedPreferences
import androidx.security.crypto.EncryptedSharedPreferences
import androidx.security.crypto.MasterKey
import dagger.hilt.android.qualifiers.ApplicationContext
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow
import javax.inject.Inject
import javax.inject.Singleton

@Singleton
class TokenManager @Inject constructor(
    @ApplicationContext private val context: Context
) {
```

```
    private val masterKey = MasterKey.Builder(context)
        .setKeyScheme(MasterKey.KeyScheme.AES256_GCM)
        .build()

    private val prefs: SharedPreferences = EncryptedSharedPreferences.create(
        context,
        "zai_secure_token_prefs",
        masterKey,
        EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
        EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
    )

    private val _tokenFlow = MutableStateFlow<String?>(getStoredToken())
    val tokenFlow: StateFlow<String?> = _tokenFlow.asStateFlow()

    fun getStoredToken(): String? {
        return prefs.getString(KEY_TOKEN, null)
    }

    fun saveToken(token: String) {
        prefs.edit().putString(KEY_TOKEN, token).apply()
        _tokenFlow.value = token
    }

    fun clearToken() {
        prefs.edit().remove(KEY_TOKEN).apply()
        _tokenFlow.value = null
    }

    companion object {
        private const val KEY_TOKEN = "jwt_bearer_token"
    }
}
```

data/local/preferences/SettingsDataStore.kt

```
package com.zai.chat.data.local.preferences

import android.content.Context
import androidx.datastore.core.DataStore
import androidx.datastore.preferences.core.Preferences
import androidx.datastore.preferences.core.booleanPreferencesKey
import androidx.datastore.preferences.core.edit
import androidx.datastore.preferences.core.floatPreferencesKey
import androidx.datastore.preferences.core.stringPreferencesKey
import androidx.datastore.preferences.preferencesDataStore
import com.zai.chat.config.ZaiConfig
import dagger.hilt.android.qualifiers.ApplicationContext
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.map
import javax.inject.Inject
import javax.inject.Singleton
```

```
val Context.dataStore: DataStore<Preferences> by preferencesDataStore(name = "us

@Singleton
class SettingsDataStore @Inject constructor(
    @ApplicationContext private val context: Context
) {
    private object Keys {
        val THEME_MODE = stringPreferencesKey("theme_mode") // "SYSTEM", "LIGHT"
        val FONT_SCALE = floatPreferencesKey("font_scale")
        val DEFAULT_MODEL = stringPreferencesKey("default_model")
        val ENTER_IS_SEND = booleanPreferencesKey("enter_is_send")
        val STREAMING_ENABLED = booleanPreferencesKey("streaming_enabled")
    }

    val themeMode: Flow<String> = context.dataStore.data.map { it[Keys.THEME_MOD
    val fontScale: Flow<Float> = context.dataStore.data.map { it[Keys.FONT_SCALE
    val defaultModel: Flow<String> = context.dataStore.data.map { it[Keys.DEFAUL
    val enterIsSend: Flow<Boolean> = context.dataStore.data.map { it[Keys.ENTER_
    val streamingEnabled: Flow<Boolean> = context.dataStore.data.map { it[Keys.S

    suspend fun setThemeMode(mode: String) {
        context.dataStore.edit { it[Keys.THEME_MODE] = mode }
    }

    suspend fun setFontScale(scale: Float) {
        context.dataStore.edit { it[Keys.FONT_SCALE] = scale }
    }

    suspend fun setDefaultModel(model: String) {
        context.dataStore.edit { it[Keys.DEFAULT_MODEL] = model }
    }

    suspend fun setEnterIsSend(enabled: Boolean) {
        context.dataStore.edit { it[Keys.ENTER_IS_SEND] = enabled }
    }

    suspend fun setStreamingEnabled(enabled: Boolean) {
        context.dataStore.edit { it[Keys.STREAMING_ENABLED] = enabled }
    }
}
```

4. Network Layer & Manual Line-by-Line SSE Engine

Network Models: `network/model/ChatCompletionRequest.kt` & `ChatCompletionChunk.kt`

```
package com.zai.chat.network.model

import kotlinx.serialization.SerialName
import kotlinx.serialization.Serializable
import kotlinx.serialization.json.JsonElement
```

```
@Serializable
data class ChatCompletionRequest(
    @SerialName("model") val model: String,
    @SerialName("chat_id") val chatId: String,
    @SerialName("messages") val messages: List<RequestMessage>,
    @SerialName("stream") val stream: Boolean = true,
    @SerialName("web_search") val webSearch: Boolean = false,
    @SerialName("reasoning") val reasoning: Boolean = true,
    @SerialName("file_ids") val fileIds: List<String>? = null
)

@Serializable
data class RequestMessage(
    @SerialName("role") val role: String,
    @SerialName("content") val content: String
)

@Serializable
data class ChatCompletionChunk(
    @SerialName("id") val id: String? = null,
    @SerialName("choices") val choices: List<ChunkChoice> = emptyList(),
    @SerialName("usage") val usage: UsageInfo? = null,
    @SerialName("citations") val citations: List<CitationItem>? = null
)

@Serializable
data class ChunkChoice(
    @SerialName("index") val index: Int = 0,
    @SerialName("delta") val delta: ChunkDelta = ChunkDelta(),
    @SerialName("finish_reason") val finishReason: String? = null
)

@Serializable
data class ChunkDelta(
    @SerialName("role") val role: String? = null,
    @SerialName("content") val content: String? = null,
    @SerialName("reasoning_content") val reasoningContent: String? = null
)

@Serializable
data class UsageInfo(
    @SerialName("prompt_tokens") val promptTokens: Int = 0,
    @SerialName("completion_tokens") val completionTokens: Int = 0,
    @SerialName("total_tokens") val totalTokens: Int = 0
)

@Serializable
data class CitationItem(
    @SerialName("index") val index: Int,
    @SerialName("url") val url: String,
    @SerialName("title") val title: String,
    @SerialName("snippet") val snippet: String? = null
)

@Serializable
data class ChatListItemResponse(
```

```
    @SerialName("id") val id: String,
    @SerialName("title") val title: String,
    @SerialName("updated_at") val updatedAt: Long,
    @SerialName("pinned") val pinned: Boolean = false,
    @SerialName("folder_id") val folderId: String? = null
)

@Serializable
data class ChatDetailResponse(
    @SerialName("id") val id: String,
    @SerialName("title") val title: String,
    @SerialName("chat") val chat: ChatContentPayload
)

@Serializable
data class ChatContentPayload(
    @SerialName("messages") val messages: List<RemoteMessagePayload> = emptyList
)

@Serializable
data class RemoteMessagePayload(
    @SerialName("id") val id: String,
    @SerialName("role") val role: String,
    @SerialName("content") val content: String,
    @SerialName("reasoning") val reasoning: String? = null,
    @SerialName("timestamp") val timestamp: Long = System.currentTimeMillis()
)

@Serializable
data class FileUploadResponse(
    @SerialName("id") val id: String,
    @SerialName("filename") val filename: String
)

@Serializable
data class ModelsListResponse(
    @SerialName("data") val data: List<ModelEntry> = emptyList()
)

@Serializable
data class ModelEntry(
    @SerialName("id") val id: String,
    @SerialName("name") val name: String? = null
)
```

`network/auth/AuthEvent.kt` & Interceptors

```
package com.zai.chat.network.auth

import com.zai.chat.config.ZaiConfig
import com.zai.chat.data.local.preferences.TokenManager
import kotlinx.coroutines.flow.MutableSharedFlow
import kotlinx.coroutines.flow.SharedFlow
```

```
import kotlinx.coroutines.flow.asSharedFlow
import okhttp3.Interceptor
import okhttp3.Response
import javax.inject.Inject
import javax.inject.Singleton

sealed interface AuthEvent {
    data object TokenExpired : AuthEvent
}

@Singleton
class AuthEventManager @Inject constructor() {
    private val _events = MutableSharedFlow<AuthEvent>(extraBufferCapacity = 1)
    val events: SharedFlow<AuthEvent> = _events.asSharedFlow()

    fun emitTokenExpired() {
        _events.tryEmit(AuthEvent.TokenExpired)
    }
}

@Singleton
class AuthInterceptor @Inject constructor(
    private val tokenManager: TokenManager,
    private val authEventManager: AuthEventManager
) : Interceptor {

    override fun intercept(chain: Interceptor.Chain): Response {
        val original = chain.request()
        val builder = original.newBuilder()
            .header("User-Agent", ZaiConfig.USER_AGENT)
            .header("Origin", ZaiConfig.ORIGIN)
            .header("Referer", ZaiConfig.REFERER)
            .header(ZaiConfig.FE_VERSION_HEADER_KEY, ZaiConfig.FE_VERSION_HEADER

        tokenManager.getStoredToken()?.let { token ->
            builder.header("Authorization", "Bearer $token")
        }

        val response = chain.proceed(builder.build())

        if (response.code == 401) {
            authEventManager.emitTokenExpired()
        }

        return response
    }
}
```

network/sse/StreamEvent.kt

```
package com.zai.chat.network.sse

import com.zai.chat.network.model.CitationItem
```

```
import com.zai.chat.network.model.UsageInfo

sealed interface StreamEvent {
    data class ReasoningDelta(val text: String) : StreamEvent
    data class ContentDelta(val text: String) : StreamEvent
    data class Citations(val citations: List<CitationItem>) : StreamEvent
    data class Usage(val usage: UsageInfo) : StreamEvent
    data object Done : StreamEvent
    data class Error(val throwable: Throwable, val partialContent: String, val p
}
```

Manual Line-by-Line SSE Parser: `network/sse/ManualSseReader.kt`

```
package com.zai.chat.network.sse

import com.zai.chat.network.model.ChatCompletionChunk
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.currentCoroutineContext
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.flow
import kotlinx.coroutines.flow.flowOn
import kotlinx.coroutines.isActive
import kotlinx.serialization.json.Json
import okhttp3.ResponseBody
import okio.BufferedSource
import java.io.IOException

class ManualSseReader(
    private val responseBody: ResponseBody,
    private val jsonParser: Json = Json {
        ignoreUnknownKeys = true
        isLenient = true
        coerceInputValues = true
    }
) {
    fun streamEvents(): Flow<StreamEvent> = flow {
        val source: BufferedSource = responseBody.source()
        val accumulatedContent = StringBuilder()
        val accumulatedReasoning = StringBuilder()

        try {
            while (currentCoroutineContext().isActive && !source.exhausted()) {
                val line = source.readUtf8Line() ?: break
                val trimmed = line.trim()

                if (trimmed.isEmpty()) {
                    continue
                }

                if (trimmed.startsWith("data:")) {
                    val payload = trimmed.removePrefix("data:").trim()

                    if (payload == "[DONE]") {
```

```
                        emit(StreamEvent.Done)
                        break
                    }

                    try {
                        val chunk = jsonParser.decodeFromString<ChatCompletionCh

                        // 1. Process reasoning stream
                        val reasoningDelta = chunk.choices.firstOrNull()?.delta?
                        if (!reasoningDelta.isNullOrEmpty()) {
                            accumulatedReasoning.append(reasoningDelta)
                            emit(StreamEvent.ReasoningDelta(reasoningDelta))
                        }

                        // 2. Process content stream
                        val contentDelta = chunk.choices.firstOrNull()?.delta?.c
                        if (!contentDelta.isNullOrEmpty()) {
                            accumulatedContent.append(contentDelta)
                            emit(StreamEvent.ContentDelta(contentDelta))
                        }

                        // 3. Process Web Citations
                        chunk.citations?.let { citations ->
                            if (citations.isNotEmpty()) {
                                emit(StreamEvent.Citations(citations))
                            }
                        }

                        // 4. Process Usage Stats
                        chunk.usage?.let { usage ->
                            emit(StreamEvent.Usage(usage))
                        }
                    } catch (e: Exception) {
                        // Skip malformed individual SSE lines without crashing
                        continue
                    }
                }
            }
        } catch (e: IOException) {
            // Mid-stream network loss: keep partial text and emit typed error w
            emit(
                StreamEvent.Error(
                    throwable = e,
                    partialContent = accumulatedContent.toString(),
                    partialReasoning = accumulatedReasoning.toString()
                )
            )
        } finally {
            try {
                responseBody.close()
            } catch (_: Throwable) {}
        }
    }.flowOn(Dispatchers.IO)
}
```

network/ZaiApiService.kt

```
package com.zai.chat.network

import com.zai.chat.config.ZaiConfig
import com.zai.chat.network.model.ChatCompletionRequest
import com.zai.chat.network.model.ChatDetailResponse
import com.zai.chat.network.model.ChatListItemResponse
import com.zai.chat.network.model.FileUploadResponse
import com.zai.chat.network.model.ModelsListResponse
import com.zai.chat.network.sse.ManualSseReader
import com.zai.chat.network.sse.StreamEvent
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.withContext
import kotlinx.serialization.encodeToString
import kotlinx.serialization.json.Json
import okhttp3.MediaType.Companion.toMediaType
import okhttp3.MultipartBody
import okhttp3.OkHttpClient
import okhttp3.Request
import okhttp3.RequestBody.Companion.asRequestBody
import okhttp3.RequestBody.Companion.toRequestBody
import java.io.File
import java.io.IOException
import javax.inject.Inject
import javax.inject.Singleton

@Singleton
class ZaiApiService @Inject constructor(
    private val okHttpClient: OkHttpClient,
    private val json: Json
) {
    private val jsonMediaType = "application/json; charset=utf-8".toMediaType()

    fun streamChatCompletion(request: ChatCompletionRequest): Flow<StreamEvent>
        val requestBodyString = json.encodeToString(request)
        val httpRequest = Request.Builder()
            .url(ZaiConfig.BASE_URL + ZaiConfig.COMPLETIONS_PATH)
            .header("Accept", "text/event-stream")
            .post(requestBodyString.toRequestBody(jsonMediaType))
            .build()

        val call = okHttpClient.newCall(httpRequest)
        val response = call.execute()

        if (!response.isSuccessful) {
            val errBody = response.body?.string().orEmpty()
            response.close()
            throw IOException("Completions endpoint failed with HTTP ${response.
        }

        val body = response.body ?: throw IOException("Empty response body from
        return ManualSseReader(body, json).streamEvents()
    }
```

```
suspend fun getChats(page: Int): List<ChatListItemResponse> = withContext(Di
    val httpRequest = Request.Builder()
        .url("${ZaiConfig.BASE_URL}${ZaiConfig.CHATS_PATH}/?page=$page")
        .get()
        .build()

    okHttpClient.newCall(httpRequest).execute().use { response ->
        if (!response.isSuccessful) throw IOException("Failed to fetch chats
        val body = response.body?.string() ?: return@use emptyList()
        json.decodeFromString<List<ChatListItemResponse>>(body)
    }
}

suspend fun getChatDetail(chatId: String): ChatDetailResponse = withContext(I
    val httpRequest = Request.Builder()
        .url("${ZaiConfig.BASE_URL}${ZaiConfig.CHATS_PATH}/$chatId")
        .get()
        .build()

    okHttpClient.newCall(httpRequest).execute().use { response ->
        if (!response.isSuccessful) throw IOException("Failed to fetch chat S
        val body = response.body?.string() ?: throw IOException("Empty chat (
        json.decodeFromString<ChatDetailResponse>(body)
    }
}

suspend fun createChat(title: String): ChatListItemResponse = withContext(Di
    val payload = json.encodeToString(mapOf("title" to title))
    val httpRequest = Request.Builder()
        .url("${ZaiConfig.BASE_URL}${ZaiConfig.CHATS_PATH}/new")
        .post(payload.toRequestBody(jsonMediaType))
        .build()

    okHttpClient.newCall(httpRequest).execute().use { response ->
        if (!response.isSuccessful) throw IOException("Failed to create chat
        val body = response.body?.string() ?: throw IOException("Empty chat (
        json.decodeFromString<ChatListItemResponse>(body)
    }
}

suspend fun deleteChat(chatId: String): Boolean = withContext(Dispatchers.IO
    val httpRequest = Request.Builder()
        .url("${ZaiConfig.BASE_URL}${ZaiConfig.CHATS_PATH}/$chatId")
        .delete()
        .build()

    okHttpClient.newCall(httpRequest).execute().use { response ->
        response.isSuccessful
    }
}

suspend fun getAvailableModels(): List<String> = withContext(Dispatchers.IO)
    val httpRequest = Request.Builder()
        .url(ZaiConfig.BASE_URL + ZaiConfig.MODELS_PATH)
        .get()
```

```
            .build()

        try {
            okHttpClient.newCall(httpRequest).execute().use { response ->
                if (!response.isSuccessful) return@withContext listOf(ZaiConfig.I
                val body = response.body?.string() ?: return@withContext listOf(Z
                val decoded = json.decodeFromString<ModelsListResponse>(body)
                decoded.data.map { it.id }
            }
        } catch (_: Exception) {
            listOf(ZaiConfig.MODEL_DEFAULT)
        }
    }

    suspend fun uploadFile(file: File): FileUploadResponse = withContext(Dispatch
        val multipartBody = MultipartBody.Builder()
            .setType(MultipartBody.FORM)
            .addFormDataPart(
                "file",
                file.name,
                file.asRequestBody("application/octet-stream".toMediaType())
            )
            .build()

        val httpRequest = Request.Builder()
            .url(ZaiConfig.BASE_URL + ZaiConfig.UPLOAD_FILE_PATH)
            .post(multipartBody)
            .build()

        okHttpClient.newCall(httpRequest).execute().use { response ->
            if (!response.isSuccessful) throw IOException("Upload failed: ${respo
            val body = response.body?.string() ?: throw IOException("Empty file r
            json.decodeFromString<FileUploadResponse>(body)
        }
    }
}
```

5. Local Persistence: Room Database & FTS Index

Room Entities: `ChatEntity.kt`, `MessageEntity.kt`, `MessageFtsEntity.kt`

```
package com.zai.chat.data.local.entity

import androidx.room.ColumnInfo
import androidx.room.Entity
import androidx.room.ForeignKey
import androidx.room.Fts4
import androidx.room.Index
import androidx.room.PrimaryKey

@Entity(tableName = "chats")
```

```
data class ChatEntity(
    @PrimaryKey
    val id: String,
    val title: String,
    @ColumnInfo(name = "updated_at")
    val updatedAt: Long,
    val pinned: Boolean = false,
    @ColumnInfo(name = "folder_id")
    val folderId: String? = null,
    @ColumnInfo(name = "fully_cached")
    val fullyCached: Boolean = false
)

@Entity(
    tableName = "messages",
    foreignKeys = [
        ForeignKey(
            entity = ChatEntity::class,
            parentColumns = ["id"],
            childColumns = ["chat_id"],
            onDelete = ForeignKey.CASCADE
        )
    ],
    indices = [Index(value = ["chat_id"]), Index(value = ["created_at"])]
)
data class MessageEntity(
    @PrimaryKey
    val id: String,
    @ColumnInfo(name = "chat_id")
    val chatId: String,
    val role: String, // "user", "assistant", "system"
    val content: String,
    val reasoning: String? = null,
    @ColumnInfo(name = "attachments_json")
    val attachmentsJson: String? = null,
    @ColumnInfo(name = "search_results_json")
    val searchResultsJson: String? = null,
    @ColumnInfo(name = "created_at")
    val createdAt: Long = System.currentTimeMillis(),
    @ColumnInfo(name = "token_count")
    val tokenCount: Int? = null,
    @ColumnInfo(name = "is_partial")
    val isPartial: Boolean = false
)

@Entity(tableName = "messages_fts")
@Fts4(contentEntity = MessageEntity::class)
data class MessageFtsEntity(
    @PrimaryKey
    @ColumnInfo(name = "rowid")
    val rowId: Long,
    val content: String,
    val reasoning: String?
)
```

DAOs: ChatDao.kt & MessageDao.kt

```
package com.zai.chat.data.local.dao

import androidx.room.Dao
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query
import androidx.room.Update
import com.zai.chat.data.local.entity.ChatEntity
import kotlinx.coroutines.flow.Flow

@Dao
interface ChatDao {
    @Query("SELECT * FROM chats ORDER BY pinned DESC, updated_at DESC")
    fun getAllChatsFlow(): Flow<List<ChatEntity>>

    @Query("SELECT * FROM chats WHERE id = :chatId LIMIT 1")
    suspend fun getChatById(chatId: String): ChatEntity?

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertOrUpdateChats(chats: List<ChatEntity>)

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertChat(chat: ChatEntity)

    @Update
    suspend fun updateChat(chat: ChatEntity)

    @Query("UPDATE chats SET pinned = :pinned WHERE id = :chatId")
    suspend fun setPinned(chatId: String, pinned: Boolean)

    @Query("DELETE FROM chats WHERE id = :chatId")
    suspend fun deleteChatById(chatId: String)
}
```

```
package com.zai.chat.data.local.dao

import androidx.room.Dao
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query
import androidx.room.Transaction
import com.zai.chat.data.local.entity.MessageEntity
import kotlinx.coroutines.flow.Flow

@Dao
interface MessageDao {
    @Query("SELECT * FROM messages WHERE chat_id = :chatId ORDER BY created_at A
    fun getMessagesForChatFlow(chatId: String): Flow<List<MessageEntity>>

    @Query("SELECT * FROM messages WHERE chat_id = :chatId ORDER BY created_at A
    suspend fun getMessagesForChat(chatId: String): List<MessageEntity>
```

```
    @Query("SELECT * FROM messages WHERE id = :messageId LIMIT 1")
    suspend fun getMessageById(messageId: String): MessageEntity?

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertMessage(message: MessageEntity)

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertMessages(messages: List<MessageEntity>)

    @Query("DELETE FROM messages WHERE id = :messageId")
    suspend fun deleteMessageById(messageId: String)

    @Query("DELETE FROM messages WHERE chat_id = :chatId AND created_at >= :sinc
    suspend fun deleteMessagesSince(chatId: String, sinceTimestamp: Long)

    @Transaction
    @Query("""
        SELECT m.* FROM messages m
        JOIN messages_fts fts ON m.rowid = fts.rowid
        WHERE messages_fts MATCH :searchQuery
        ORDER BY m.created_at DESC
    """)
    fun searchMessages(searchQuery: String): Flow<List<MessageEntity>>
}
```

data/local/ZaiDatabase.kt

```
package com.zai.chat.data.local

import androidx.room.Database
import androidx.room.RoomDatabase
import com.zai.chat.data.local.dao.ChatDao
import com.zai.chat.data.local.dao.MessageDao
import com.zai.chat.data.local.entity.ChatEntity
import com.zai.chat.data.local.entity.MessageEntity
import com.zai.chat.data.local.entity.MessageFtsEntity

@Database(
    entities = [
        ChatEntity::class,
        MessageEntity::class,
        MessageFtsEntity::class
    ],
    version = 1,
    exportSchema = false
)
abstract class ZaiDatabase : RoomDatabase() {
    abstract fun chatDao(): ChatDao
    abstract fun messageDao(): MessageDao
}
```

6. Repository Layer & Data Synchronization

Domain Models: `data/model/`

```
package com.zai.chat.data.model

import kotlinx.serialization.Serializable

enum class MessageRole {
    USER, ASSISTANT, SYSTEM
}

data class Chat(
    val id: String,
    val title: String,
    val updatedAt: Long,
    val pinned: Boolean,
    val folderId: String?,
    val fullyCached: Boolean
)

@Serializable
data class SearchCitation(
    val index: Int,
    val title: String,
    val url: String,
    val snippet: String? = null
)

data class Message(
    val id: String,
    val chatId: String,
    val role: MessageRole,
    val content: String,
    val reasoning: String? = null,
    val attachments: List<String> = emptyList(),
    val citations: List<SearchCitation> = emptyList(),
    val createdAt: Long = System.currentTimeMillis(),
    val tokenCount: Int? = null,
    val isPartial: Boolean = false
)
```

`data/repository/ChatRepository.kt` & Implementation

```
package com.zai.chat.data.repository

import com.zai.chat.data.local.dao.ChatDao
import com.zai.chat.data.local.dao.MessageDao
import com.zai.chat.data.local.entity.ChatEntity
import com.zai.chat.data.local.entity.MessageEntity
import com.zai.chat.data.model.Chat
```

```
import com.zai.chat.data.model.Message
import com.zai.chat.data.model.MessageRole
import com.zai.chat.data.model.SearchCitation
import com.zai.chat.network.ZaiApiService
import com.zai.chat.network.model.ChatCompletionRequest
import com.zai.chat.network.model.RequestMessage
import com.zai.chat.network.sse.StreamEvent
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.flow
import kotlinx.coroutines.flow.flowOn
import kotlinx.coroutines.flow.map
import kotlinx.coroutines.withContext
import kotlinx.serialization.encodeToString
import kotlinx.serialization.json.Json
import java.util.UUID
import javax.inject.Inject
import javax.inject.Singleton

interface ChatRepository {
    fun getChatsStream(): Flow<List<Chat>>
    fun getMessagesStream(chatId: String): Flow<List<Message>>
    suspend fun refreshChats(page: Int = 1)
    suspend fun syncChatMessages(chatId: String)
    suspend fun createChat(firstPrompt: String): String
    suspend fun deleteChat(chatId: String)
    suspend fun togglePinChat(chatId: String, pinned: Boolean)
    fun searchMessages(query: String): Flow<List<Message>>
    suspend fun rollbackMessagesAfter(chatId: String, timestamp: Long)
    fun streamCompletion(
        chatId: String,
        userPrompt: String,
        model: String,
        webSearch: Boolean,
        deepThinking: Boolean
    ): Flow<StreamEvent>
}

@Singleton
class ChatRepositoryImpl @Inject constructor(
    private val apiService: ZaiApiService,
    private val chatDao: ChatDao,
    private val messageDao: MessageDao,
    private val json: Json
) : ChatRepository {

    override fun getChatsStream(): Flow<List<Chat>> {
        return chatDao.getAllChatsFlow().map { entities ->
            entities.map { it.toDomain() }
        }
    }

    override fun getMessagesStream(chatId: String): Flow<List<Message>> {
        return messageDao.getMessagesForChatFlow(chatId).map { entities ->
            entities.map { it.toDomain(json) }
        }
```

```
}

override suspend fun refreshChats(page: Int): Unit = withContext(Dispatchers
    try {
        val remoteChats = apiService.getChats(page)
        val entities = remoteChats.map { remote ->
            val cached = chatDao.getChatById(remote.id)
            ChatEntity(
                id = remote.id,
                title = remote.title,
                updatedAt = remote.updatedAt,
                pinned = remote.pinned,
                folderId = remote.folderId,
                fullyCached = cached?.fullyCached ?: false
            )
        }
        chatDao.insertOrUpdateChats(entities)
    } catch (_: Exception) {
        // Degrade gracefully; Room serves local cache as source of truth
    }
}

override suspend fun syncChatMessages(chatId: String): Unit = withContext(Di
    val chat = chatDao.getChatById(chatId)
    if (chat?.fullyCached == true) return@withContext

    try {
        val detail = apiService.getChatDetail(chatId)
        val entities = detail.chat.messages.map { remoteMsg ->
            MessageEntity(
                id = remoteMsg.id,
                chatId = chatId,
                role = remoteMsg.role.lowercase(),
                content = remoteMsg.content,
                reasoning = remoteMsg.reasoning,
                createdAt = remoteMsg.timestamp
            )
        }
        messageDao.insertMessages(entities)
        chat?.let {
            chatDao.updateChat(it.copy(fullyCached = true))
        }
    } catch (_: Exception) {}
}

override suspend fun createChat(firstPrompt: String): String = withContext(D
    val title = if (firstPrompt.length > 28) firstPrompt.take(28) + "…" else
    val tempId = UUID.randomUUID().toString()
    val now = System.currentTimeMillis()

    // Insert locally immediately for optimistic UI
    chatDao.insertChat(
        ChatEntity(
            id = tempId,
            title = title,
            updatedAt = now,
```

```
            pinned = false,
            fullyCached = true
        )
    )

    try {
        val created = apiService.createChat(title)
        chatDao.deleteChatById(tempId)
        chatDao.insertChat(
            ChatEntity(
                id = created.id,
                title = created.title,
                updatedAt = created.updatedAt,
                pinned = created.pinned,
                fullyCached = true
            )
        )
        created.id
    } catch (_: Exception) {
        tempId
    }
}

override suspend fun deleteChat(chatId: String) = withContext(Dispatchers.IO
    val backup = chatDao.getChatById(chatId)
    chatDao.deleteChatById(chatId)
    try {
        val success = apiService.deleteChat(chatId)
        if (!success && backup != null) {
            chatDao.insertChat(backup)
        }
    } catch (_: Exception) {
        if (backup != null) chatDao.insertChat(backup)
    }
}

override suspend fun togglePinChat(chatId: String, pinned: Boolean) = withCo
    chatDao.setPinned(chatId, pinned)
}

override fun searchMessages(query: String): Flow<List<Message>> {
    val cleanQuery = query.trim().replace("\"", "\"\"") + "*"
    return messageDao.searchMessages(cleanQuery).map { entities ->
        entities.map { it.toDomain(json) }
    }
}

override suspend fun rollbackMessagesAfter(chatId: String, timestamp: Long)
    messageDao.deleteMessagesSince(chatId, timestamp)
}

override fun streamCompletion(
    chatId: String,
    userPrompt: String,
    model: String,
    webSearch: Boolean,
```

```
    deepThinking: Boolean
): Flow<StreamEvent> = flow {
    // Persist User Message to Room
    val userMessageId = UUID.randomUUID().toString()
    val userEntity = MessageEntity(
        id = userMessageId,
        chatId = chatId,
        role = "user",
        content = userPrompt,
        createdAt = System.currentTimeMillis()
    )
    messageDao.insertMessage(userEntity)

    // Read chat history for context
    val history = messageDao.getMessagesForChat(chatId).map {
        RequestMessage(role = it.role, content = it.content)
    }

    val request = ChatCompletionRequest(
        model = model,
        chatId = chatId,
        messages = history,
        stream = true,
        webSearch = webSearch,
        reasoning = deepThinking
    )

    val assistantMessageId = UUID.randomUUID().toString()
    val accumulatedContent = StringBuilder()
    val accumulatedReasoning = StringBuilder()
    var accumulatedCitations = emptyList<SearchCitation>()
    var capturedTokens = 0

    try {
        apiService.streamChatCompletion(request).collect { event ->
            when (event) {
                is StreamEvent.ContentDelta -> {
                    accumulatedContent.append(event.text)
                    emit(event)
                }
                is StreamEvent.ReasoningDelta -> {
                    accumulatedReasoning.append(event.text)
                    emit(event)
                }
                is StreamEvent.Citations -> {
                    accumulatedCitations = event.citations.map {
                        SearchCitation(it.index, it.title, it.url, it.snippe
                    }
                    emit(event)
                }
                is StreamEvent.Usage -> {
                    capturedTokens = event.usage.totalTokens
                    emit(event)
                }
                is StreamEvent.Done -> {
                    // Persist final assistant response
```

```
                        val assistantEntity = MessageEntity(
                            id = assistantMessageId,
                            chatId = chatId,
                            role = "assistant",
                            content = accumulatedContent.toString(),
                            reasoning = accumulatedReasoning.toString().takeIf {
                            searchResultsJson = if (accumulatedCitations.isNotEm
                            createdAt = System.currentTimeMillis(),
                            tokenCount = capturedTokens,
                            isPartial = false
                        )
                        messageDao.insertMessage(assistantEntity)
                        emit(StreamEvent.Done)
                    }
                    is StreamEvent.Error -> {
                        // Mid-stream network loss: persist partial text for inl
                        val partialEntity = MessageEntity(
                            id = assistantMessageId,
                            chatId = chatId,
                            role = "assistant",
                            content = event.partialContent,
                            reasoning = event.partialReasoning.takeIf { it.isNot
                            createdAt = System.currentTimeMillis(),
                            isPartial = true
                        )
                        messageDao.insertMessage(partialEntity)
                        emit(event)
                    }
                }
            }
        } catch (e: Throwable) {
            emit(
                StreamEvent.Error(
                    throwable = e,
                    partialContent = accumulatedContent.toString(),
                    partialReasoning = accumulatedReasoning.toString()
                )
            )
        }
    }.flowOn(Dispatchers.IO)
}

private fun ChatEntity.toDomain(): Chat = Chat(
    id = id,
    title = title,
    updatedAt = updatedAt,
    pinned = pinned,
    folderId = folderId,
    fullyCached = fullyCached
)

private fun MessageEntity.toDomain(json: Json): Message {
    val parsedCitations = searchResultsJson?.let {
        try {
            json.decodeFromString<List<SearchCitation>>(it)
        } catch (_: Exception) {
```

```
            emptyList()
        }
    } ?: emptyList()

    return Message(
        id = id,
        chatId = chatId,
        role = when (role.lowercase()) {
            "user" -> MessageRole.USER
            "assistant" -> MessageRole.ASSISTANT
            else -> MessageRole.SYSTEM
        },
        content = content,
        reasoning = reasoning,
        citations = parsedCitations,
        createdAt = createdAt,
        tokenCount = tokenCount,
        isPartial = isPartial
    )
}
```

7. Design System, Typography & Spring Motion Tokens

ui/theme/Color.kt

```
package com.zai.chat.ui.theme

import androidx.compose.ui.graphics.Color

// Bespoke Palette: OLED, Obsidian, Warm Ivory, Claude/Kimi Accents
val TrueBlack = Color(0xFF000000)
val ObsidianBase = Color(0xFF0D0D11)
val SurfaceContainerDark = Color(0xFF16161D)
val SurfaceContainerHighDark = Color(0xFF1E1E28)
val SurfaceContainerHighestDark = Color(0xFF272734)

val ClaudePeach = Color(0xFFD97757)
val ClaudePeachSubtle = Color(0xFF331E17)
val KimiCyan = Color(0xFF38BDF8)
val KimiCyanSubtle = Color(0xFF0C2433)

val TextPrimaryDark = Color(0xFFECECF1)
val TextSecondaryDark = Color(0xFF9E9EB2)
val TextTertiaryDark = Color(0xFF6B6B7F)

val UserBubbleFillDark = Color(0xFF262633)
val BorderSubtleDark = Color(0xFF2A2A38)

val ThinkingBackgroundDark = Color(0xFF13131A)
val ThinkingBorderDark = Color(0xFF232330)

// Light Theme Tokens
```

```
val PureWhite = Color(0xFFFFFFFF)
val SurfaceLight = Color(0xFFF7F7F8)
val SurfaceContainerLight = Color(0xFFEFEFF2)
val TextPrimaryLight = Color(0xFF1A1A1E)
val TextSecondaryLight = Color(0xFF63636E)
val UserBubbleFillLight = Color(0xFFE4E4EB)
val BorderSubtleLight = Color(0xFFE0E0E6)
```

ui/theme/Spacing.kt

```
package com.zai.chat.ui.theme

import androidx.compose.ui.unit.Dp
import androidx.compose.ui.unit.dp

data class ZaiSpacing(
    val xxxs: Dp = 2.dp,
    val xxs: Dp = 4.dp,
    val xs: Dp = 8.dp,
    val sm: Dp = 12.dp,
    val md: Dp = 16.dp,
    val lg: Dp = 24.dp,
    val xl: Dp = 32.dp,
    val xxl: Dp = 48.dp
)

val LocalSpacing = androidx.compose.runtime.staticCompositionLocalOf { ZaiSpacin
```

ui/theme/Motion.kt

```
package com.zai.chat.ui.theme

import androidx.compose.animation.core.Spring
import androidx.compose.animation.core.SpringSpec
import androidx.compose.animation.core.spring

/**
 * Custom Moonshot/Kimi Spring Motion Specifications.
 * Zero default linear/cubic tweens; everything is spring physics.
 */
object ZaiMotion {
    // Flowy, bouncy spring for drawer and modal interactions
    val FluidSpring: SpringSpec<Float> = spring(
        dampingRatio = Spring.DampingRatioLowBouncy,
        stiffness = Spring.StiffnessLow
    )

    // Snappy spring for button morphs and state switches
    val MorphSpring: SpringSpec<Float> = spring(
        dampingRatio = Spring.DampingRatioNoBouncy,
```

```
        stiffness = Spring.StiffnessMediumLow
    )

    // Gentle spring for thinking expansion panel
    val PanelExpandSpring: SpringSpec<Float> = spring(
        dampingRatio = 0.85f,
        stiffness = 250f
    )
}
```

ui/theme/Type.kt

```
package com.zai.chat.ui.theme

import androidx.compose.material3.Typography
import androidx.compose.ui.text.TextStyle
import androidx.compose.ui.text.font.FontFamily
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.unit.sp

// Assistant responses use larger line-height (15-16sp with 24sp line height) fo
val ZaiTypography = Typography(
    headlineMedium = TextStyle(
        fontFamily = FontFamily.SansSerif,
        fontWeight = FontWeight.SemiBold,
        fontSize = 24.sp,
        lineHeight = 32.sp,
        letterSpacing = (-0.2).sp
    ),
    titleLarge = TextStyle(
        fontFamily = FontFamily.SansSerif,
        fontWeight = FontWeight.SemiBold,
        fontSize = 18.sp,
        lineHeight = 24.sp
    ),
    titleMedium = TextStyle(
        fontFamily = FontFamily.SansSerif,
        fontWeight = FontWeight.Medium,
        fontSize = 15.sp,
        lineHeight = 20.sp
    ),
    bodyLarge = TextStyle(
        fontFamily = FontFamily.SansSerif,
        fontWeight = FontWeight.Normal,
        fontSize = 16.sp,
        lineHeight = 24.sp, // Optimized document reading rhythm
        letterSpacing = 0.15.sp
    ),
    bodyMedium = TextStyle(
        fontFamily = FontFamily.SansSerif,
        fontWeight = FontWeight.Normal,
        fontSize = 14.sp,
        lineHeight = 20.sp
```

```
    ),
    labelSmall = TextStyle(
        fontFamily = FontFamily.Monospace,
        fontWeight = FontWeight.Medium,
        fontSize = 11.sp,
        lineHeight = 16.sp,
        letterSpacing = 0.5.sp
    )
)
```

ui/theme/Theme.kt

```
package com.zai.chat.ui.theme

import androidx.compose.foundation.isSystemInDarkTheme
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.darkColorScheme
import androidx.compose.material3.lightColorScheme
import androidx.compose.runtime.Composable
import androidx.compose.runtime.CompositionLocalProvider

private val DarkColorScheme = darkColorScheme(
    primary = ClaudePeach,
    onPrimary = TrueBlack,
    surface = ObsidianBase,
    onSurface = TextPrimaryDark,
    surfaceVariant = SurfaceContainerDark,
    onSurfaceVariant = TextSecondaryDark,
    outline = BorderSubtleDark,
    background = TrueBlack,
    onBackground = TextPrimaryDark
)

private val LightColorScheme = lightColorScheme(
    primary = ClaudePeach,
    onPrimary = PureWhite,
    surface = SurfaceLight,
    onSurface = TextPrimaryLight,
    surfaceVariant = SurfaceContainerLight,
    onSurfaceVariant = TextSecondaryLight,
    outline = BorderSubtleLight,
    background = PureWhite,
    onBackground = TextPrimaryLight
)

@Composable
fun ZaiTheme(
    themeMode: String = "OLED",
    content: @Composable () -> Unit
) {
    val dark = when (themeMode) {
        "LIGHT" -> false
        "DARK", "OLED" -> true
```

```
        else -> isSystemInDarkTheme()
    }

    val colorScheme = if (dark) DarkColorScheme else LightColorScheme

    CompositionLocalProvider(
        LocalSpacing provides ZaiSpacing()
    ) {
        MaterialTheme(
            colorScheme = colorScheme,
            typography = ZaiTypography,
            content = content
        )
    }
}
```

8. Custom Reusable UI Components

Soft Pulsing Cursor: `ui/components/StreamingCursor.kt`

```
package com.zai.chat.ui.components

import androidx.compose.animation.core.LinearEasing
import androidx.compose.animation.core.RepeatMode
import androidx.compose.animation.core.animateFloat
import androidx.compose.animation.core.infiniteRepeatable
import androidx.compose.animation.core.rememberInfiniteTransition
import androidx.compose.animation.core.tween
import androidx.compose.foundation.background
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.shape.CircleShape
import androidx.compose.material3.MaterialTheme
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.alpha
import androidx.compose.ui.unit.dp

@Composable
fun StreamingCursor(modifier: Modifier = Modifier) {
    val transition = rememberInfiniteTransition(label = "cursor_pulse")
    val alpha by transition.animateFloat(
        initialValue = 0.2f,
        targetValue = 1.0f,
        animationSpec = infiniteRepeatable(
            animation = tween(durationMillis = 500, easing = LinearEasing),
            repeatMode = RepeatMode.Reverse
        ),
        label = "cursor_alpha"
    )
```

```
    Box(
        modifier = modifier
            .size(width = 8.dp, height = 15.dp)
            .alpha(alpha)
            .background(MaterialTheme.colorScheme.primary, CircleShape)
    )
}
```

Thinking Block with Shimmer & Spring: `ui/components/ThinkingBlock.kt`

```
package com.zai.chat.ui.components

import androidx.compose.animation.AnimatedVisibility
import androidx.compose.animation.animateContentSize
import androidx.compose.animation.core.LinearEasing
import androidx.compose.animation.core.RepeatMode
import androidx.compose.animation.core.animateFloat
import androidx.compose.animation.core.infiniteRepeatable
import androidx.compose.animation.core.rememberInfiniteTransition
import androidx.compose.animation.core.tween
import androidx.compose.foundation.background
import androidx.compose.foundation.border
import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.heightIn
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.layout.width
import androidx.compose.foundation.rememberScrollState
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.foundation.verticalScroll
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.rounded.KeyboardArrowDown
import androidx.compose.material.icons.rounded.KeyboardArrowUp
import androidx.compose.material.icons.rounded.Psychology
import androidx.compose.material3.Icon
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.geometry.Offset
import androidx.compose.ui.graphics.Brush
```

```
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.text.font.FontStyle
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import com.zai.chat.ui.theme.ClaudePeach
import com.zai.chat.ui.theme.ThinkingBackgroundDark
import com.zai.chat.ui.theme.ThinkingBorderDark
import com.zai.chat.ui.theme.ZaiMotion

@Composable
fun ThinkingBlock(
    reasoningText: String,
    isStreamingReasoning: Boolean,
    durationSeconds: Float = 0f,
    modifier: Modifier = Modifier
) {
    var expanded by remember { mutableStateOf(isStreamingReasoning) }

    // If streaming reasoning starts, expand automatically
    if (isStreamingReasoning && !expanded) {
        expanded = true
    }

    val shape = RoundedCornerShape(12.dp)

    Column(
        modifier = modifier
            .fillMaxWidth()
            .clip(shape)
            .background(ThinkingBackgroundDark)
            .border(1.dp, ThinkingBorderDark, shape)
            .animateContentSize(animationSpec = ZaiMotion.PanelExpandSpring)
    ) {
        // Collapsed / Header Bar
        Row(
            modifier = Modifier
                .fillMaxWidth()
                .clickable { expanded = !expanded }
                .padding(horizontal = 14.dp, vertical = 10.dp),
            verticalAlignment = Alignment.CenterVertically
        ) {
            Icon(
                imageVector = Icons.Rounded.Psychology,
                contentDescription = null,
                tint = ClaudePeach,
                modifier = Modifier.size(18.dp)
            )
            Spacer(modifier = Modifier.width(8.dp))

            if (isStreamingReasoning) {
                ThinkingShimmerText(text = "Thinking deeply…")
            } else {
                Text(
                    text = if (durationSeconds > 0f) "💭 Thought for ${"%.1f".fo
                    style = MaterialTheme.typography.bodyMedium.copy(
```

```
                        fontWeight = FontWeight.Medium,
                        color = MaterialTheme.colorScheme.onSurfaceVariant
                    )
                )
            }

            Spacer(modifier = Modifier.weight(1f))

            Icon(
                imageVector = if (expanded) Icons.Rounded.KeyboardArrowUp else I
                contentDescription = if (expanded) "Collapse" else "Expand",
                tint = MaterialTheme.colorScheme.onSurfaceVariant,
                modifier = Modifier.size(18.dp)
            )
        }

        // Expanded Scrollable Monospace Body
        AnimatedVisibility(visible = expanded) {
            Box(
                modifier = Modifier
                    .fillMaxWidth()
                    .heightIn(max = 240.dp)
                    .padding(start = 14.dp, end = 14.dp, bottom = 12.dp)
            ) {
                val scrollState = rememberScrollState()
                Text(
                    text = reasoningText.ifEmpty { "Generating thoughts…" },
                    style = MaterialTheme.typography.bodyMedium.copy(
                        fontStyle = FontStyle.Italic,
                        fontFamily = androidx.compose.ui.text.font.FontFamily.Mo
                        fontSize = 13.sp,
                        lineHeight = 19.sp,
                        color = MaterialTheme.colorScheme.onSurfaceVariant.copy(
                    ),
                    modifier = Modifier.verticalScroll(scrollState)
                )
            }
        }
    }
}

@Composable
fun ThinkingShimmerText(text: String) {
    val transition = rememberInfiniteTransition(label = "shimmer")
    val translateAnim by transition.animateFloat(
        initialValue = 0f,
        targetValue = 1000f,
        animationSpec = infiniteRepeatable(
            animation = tween(durationMillis = 1400, easing = LinearEasing),
            repeatMode = RepeatMode.Restart
        ),
        label = "shimmer_translate"
    )

    val brush = Brush.linearGradient(
        colors = listOf(
```

```
            ClaudePeach.copy(alpha = 0.4f),
            ClaudePeach.copy(alpha = 1.0f),
            ClaudePeach.copy(alpha = 0.4f)
        ),
        start = Offset(translateAnim - 200f, 0f),
        end = Offset(translateAnim, 0f)
    )

    Text(
        text = text,
        style = MaterialTheme.typography.bodyMedium.copy(
            fontWeight = FontWeight.SemiBold,
            brush = brush
        )
    )
}
```

Custom Code Block Composable: `ui/components/CodeBlockView.kt`

```
package com.zai.chat.ui.components

import android.content.ClipData
import android.content.ClipboardManager
import android.content.Context
import android.widget.Toast
import androidx.compose.foundation.background
import androidx.compose.foundation.border
import androidx.compose.foundation.horizontalScroll
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.rememberScrollState
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.rounded.Check
import androidx.compose.material.icons.rounded.ContentCopy
import androidx.compose.material.icons.rounded.WrapText
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.rememberCoroutineScope
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
```

```
import androidx.compose.ui.draw.clip
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.text.font.FontFamily
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import com.zai.chat.ui.theme.SurfaceContainerDark
import com.zai.chat.ui.theme.SurfaceContainerHighDark
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch

@Composable
fun CodeBlockView(
    code: String,
    language: String = "plaintext",
    onInsertToInput: ((String) -> Unit)? = null,
    modifier: Modifier = Modifier
) {
    val context = LocalContext.current
    val scope = rememberCoroutineScope()
    var isCopied by remember { mutableStateOf(false) }
    var wrapLines by remember { mutableStateOf(false) }
    val shape = RoundedCornerShape(10.dp)

    Column(
        modifier = modifier
            .fillMaxWidth()
            .clip(shape)
            .background(SurfaceContainerDark)
            .border(1.dp, MaterialTheme.colorScheme.outline.copy(alpha = 0.3f), s
    ) {
        // Code Block Header
        Row(
            modifier = Modifier
                .fillMaxWidth()
                .background(SurfaceContainerHighDark)
                .padding(horizontal = 12.dp, vertical = 6.dp),
            horizontalArrangement = Arrangement.SpaceBetween,
            verticalAlignment = Alignment.CenterVertically
        ) {
            Text(
                text = language.lowercase(),
                style = MaterialTheme.typography.labelSmall.copy(
                    color = MaterialTheme.colorScheme.onSurfaceVariant
                )
            )

            Row(verticalAlignment = Alignment.CenterVertically) {
                IconButton(
                    onClick = { wrapLines = !wrapLines },
                    modifier = Modifier.size(28.dp)
                ) {
                    Icon(
                        imageVector = Icons.Rounded.WrapText,
                        contentDescription = "Toggle wrap",
                        tint = if (wrapLines) MaterialTheme.colorScheme.primary e
                        modifier = Modifier.size(16.dp)
```

```
                )
            }

            IconButton(
                onClick = {
                    val clipboard = context.getSystemService(Context.CLIPBOA
                    clipboard.setPrimaryClip(ClipData.newPlainText("Code", c
                    Toast.makeText(context, "Copied code snippet", Toast.LEN
                    isCopied = true
                    scope.launch {
                        delay(2000)
                        isCopied = false
                    }
                },
                modifier = Modifier.size(28.dp)
            ) {
                Icon(
                    imageVector = if (isCopied) Icons.Rounded.Check else Ico
                    contentDescription = "Copy code",
                    tint = if (isCopied) MaterialTheme.colorScheme.primary e
                    modifier = Modifier.size(16.dp)
                )
            }
        }
    }

    // Code Body
    val scrollState = rememberScrollState()
    val textModifier = if (wrapLines) {
        Modifier.fillMaxWidth().padding(12.dp)
    } else {
        Modifier
            .horizontalScroll(scrollState)
            .padding(12.dp)
    }

    Text(
        text = code,
        style = MaterialTheme.typography.bodyMedium.copy(
            fontFamily = FontFamily.Monospace,
            fontSize = 13.sp,
            lineHeight = 18.sp,
            color = MaterialTheme.colorScheme.onSurface
        ),
        modifier = textModifier
    )
  }
}
```

Web Citations Strip: `ui/components/WebCitationStrip.kt`

```
package com.zai.chat.ui.components
```

```
import android.content.Intent
import android.net.Uri
import androidx.compose.foundation.clickable
import androidx.compose.foundation.horizontalScroll
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.layout.width
import androidx.compose.foundation.rememberScrollState
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.rounded.Language
import androidx.compose.material3.Icon
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.text.style.TextOverflow
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import com.zai.chat.data.model.SearchCitation
import com.zai.chat.ui.theme.KimiCyan

@Composable
fun WebCitationStrip(
    citations: List<SearchCitation>,
    modifier: Modifier = Modifier
) {
    if (citations.isEmpty()) return
    val context = LocalContext.current

    Column(modifier = modifier.fillMaxWidth()) {
        Row(
            verticalAlignment = Alignment.CenterVertically,
            modifier = Modifier.padding(bottom = 6.dp)
        ) {
            Icon(
                imageVector = Icons.Rounded.Language,
                contentDescription = null,
                tint = KimiCyan,
                modifier = Modifier.size(14.dp)
            )
            Spacer(modifier = Modifier.width(6.dp))
            Text(
                text = "Sources & Citations (${citations.size})",
                style = MaterialTheme.typography.labelSmall.copy(
```

```
                    fontWeight = FontWeight.SemiBold,
                    color = MaterialTheme.colorScheme.onSurfaceVariant
                )
            )
        }

        Row(
            modifier = Modifier
                .fillMaxWidth()
                .horizontalScroll(rememberScrollState()),
            horizontalArrangement = Arrangement.spacedBy(8.dp)
        ) {
            citations.forEach { citation ->
                Surface(
                    shape = RoundedCornerShape(8.dp),
                    color = MaterialTheme.colorScheme.surfaceVariant,
                    modifier = Modifier
                        .clip(RoundedCornerShape(8.dp))
                        .clickable {
                            try {
                                val intent = Intent(Intent.ACTION_VIEW, Uri.pars
                                context.startActivity(intent)
                            } catch (_: Exception) {}
                        }
                ) {
                    Row(
                        modifier = Modifier.padding(horizontal = 8.dp, vertical
                        verticalAlignment = Alignment.CenterVertically
                    ) {
                        Text(
                            text = "[${citation.index}]",
                            style = MaterialTheme.typography.labelSmall.copy(
                                color = KimiCyan,
                                fontWeight = FontWeight.Bold
                            )
                        )
                        Spacer(modifier = Modifier.width(4.dp))
                        Text(
                            text = citation.title.ifBlank { citation.url },
                            style = MaterialTheme.typography.bodyMedium.copy(fon
                            maxLines = 1,
                            overflow = TextOverflow.Ellipsis,
                            modifier = Modifier.width(130.dp)
                        )
                    }
                }
            }
        }
    }
}
```

Action Bar & Bubble View: `ui/components/MessageBubble.kt`

```
package com.zai.chat.ui.components

import android.content.ClipData
import android.content.ClipboardManager
import android.content.Context
import android.content.Intent
import android.widget.Toast
import androidx.compose.animation.AnimatedVisibility
import androidx.compose.animation.fadeIn
import androidx.compose.animation.scaleIn
import androidx.compose.foundation.ExperimentalFoundationApi
import androidx.compose.foundation.combinedClickable
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.layout.width
import androidx.compose.foundation.layout.widthIn
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.rounded.ContentCopy
import androidx.compose.material.icons.rounded.Edit
import androidx.compose.material.icons.rounded.Refresh
import androidx.compose.material.icons.rounded.Share
import androidx.compose.material3.ButtonDefaults
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
import androidx.compose.material3.TextButton
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.platform.LocalContext
import androidx.compose.ui.unit.dp
import com.zai.chat.data.model.Message
import com.zai.chat.data.model.MessageRole
import com.zai.chat.ui.theme.UserBubbleFillDark

@OptIn(ExperimentalFoundationApi::class)
@Composable
fun MessageBubble(
    message: Message,
```

```
    isStreaming: Boolean,
    onEditAndResend: ((Message) -> Unit)? = null,
    onRegenerate: (() -> Unit)? = null,
    onRetryFromHere: ((Message) -> Unit)? = null,
    modifier: Modifier = Modifier
) {
    val context = LocalContext.current
    var showActions by remember { mutableStateOf(false) }

    val isUser = message.role == MessageRole.USER

    Column(
        modifier = modifier
            .fillMaxWidth()
            .padding(horizontal = 16.dp, vertical = 6.dp),
        horizontalAlignment = if (isUser) Alignment.End else Alignment.Start
    ) {
        if (isUser) {
            // USER MESSAGE: Compact right-aligned rounded pill
            Surface(
                shape = RoundedCornerShape(20.dp, 20.dp, 4.dp, 20.dp),
                color = UserBubbleFillDark,
                modifier = Modifier
                    .widthIn(max = 300.dp)
                    .clip(RoundedCornerShape(20.dp, 20.dp, 4.dp, 20.dp))
                    .combinedClickable(
                        onClick = { showActions = !showActions },
                        onLongClick = { showActions = true }
                    )
            ) {
                Text(
                    text = message.content,
                    style = MaterialTheme.typography.bodyLarge.copy(
                        color = MaterialTheme.colorScheme.onSurface
                    ),
                    modifier = Modifier.padding(horizontal = 16.dp, vertical = 1
                )
            }
        } else {
            // ASSISTANT MESSAGE: Edge-to-edge document format (No gray bubble)
            Column(
                modifier = Modifier
                    .fillMaxWidth()
                    .combinedClickable(
                        onClick = { showActions = !showActions },
                        onLongClick = { showActions = true }
                    )
            ) {
                // 1. Thinking block if reasoning content is present
                if (!message.reasoning.isNullOrEmpty()) {
                    ThinkingBlock(
                        reasoningText = message.reasoning,
                        isStreamingReasoning = isStreaming && message.content.isE
                    )
                    Spacer(modifier = Modifier.height(10.dp))
                }
```

```
        // 2. Assistant Document Text with streaming cursor
        Box(modifier = Modifier.fillMaxWidth()) {
            Row {
                Text(
                    text = message.content,
                    style = MaterialTheme.typography.bodyLarge.copy(
                        color = MaterialTheme.colorScheme.onSurface
                    )
                )
                if (isStreaming) {
                    Spacer(modifier = Modifier.width(4.dp))
                    StreamingCursor(modifier = Modifier.align(Alignment.(
                }
            }
        }

        // 3. Citations Strip
        if (message.citations.isNotEmpty()) {
            Spacer(modifier = Modifier.height(8.dp))
            WebCitationStrip(citations = message.citations)
        }

        // 4. Inline Retry on Network Loss
        if (message.isPartial) {
            Spacer(modifier = Modifier.height(8.dp))
            TextButton(
                onClick = { onRetryFromHere?.invoke(message) },
                colors = ButtonDefaults.textButtonColors(contentColor = I
            ) {
                Icon(imageVector = Icons.Rounded.Refresh, contentDescrip
                Spacer(modifier = Modifier.width(6.dp))
                Text(text = "Stream interrupted — Tap to resume")
            }
        }
    }
}

// Action Toolbar (Copy, Edit, Share, Regenerate)
AnimatedVisibility(
    visible = showActions,
    enter = fadeIn() + scaleIn()
) {
    Row(
        modifier = Modifier.padding(top = 4.dp),
        horizontalArrangement = Arrangement.spacedBy(4.dp)
    ) {
        IconButton(
            onClick = {
                val clipboard = context.getSystemService(Context.CLIPBOA
                clipboard.setPrimaryClip(ClipData.newPlainText("Message"
                Toast.makeText(context, "Copied", Toast.LENGTH_SHORT).sh
                showActions = false
            },
            modifier = Modifier.size(32.dp)
        ) {
```

```
                    Icon(imageVector = Icons.Rounded.ContentCopy, contentDescrip
                }

                IconButton(
                    onClick = {
                        val intent = Intent(Intent.ACTION_SEND).apply {
                            type = "text/plain"
                            putExtra(Intent.EXTRA_TEXT, message.content)
                        }
                        context.startActivity(Intent.createChooser(intent, "Shar
                        showActions = false
                    },
                    modifier = Modifier.size(32.dp)
                ) {
                    Icon(imageVector = Icons.Rounded.Share, contentDescription =
                }

                if (isUser) {
                    IconButton(
                        onClick = {
                            onEditAndResend?.invoke(message)
                            showActions = false
                        },
                        modifier = Modifier.size(32.dp)
                    ) {
                        Icon(imageVector = Icons.Rounded.Edit, contentDescriptio
                    }
                } else if (!isStreaming) {
                    IconButton(
                        onClick = {
                            onRegenerate?.invoke()
                            showActions = false
                        },
                        modifier = Modifier.size(32.dp)
                    ) {
                        Icon(imageVector = Icons.Rounded.Refresh, contentDescrip
                    }
                }
            }
        }
    }
}
```

9. Chat Screen, Composer & ViewModels

ui/screens/chat/ChatUiState.kt & ChatUiEvent.kt

```
package com.zai.chat.ui.screens.chat

import com.zai.chat.data.model.Message

data class ChatUiState(
```

```
    val currentChatId: String? = null,
    val chatTitle: String = "New Chat",
    val messages: List<Message> = emptyList(),
    val isStreaming: Boolean = false,
    val liveReasoningText: String = "",
    val liveContentText: String = "",
    val selectedModel: String = "GLM-4-Plus",
    val availableModels: List<String> = listOf("GLM-4-Plus", "GLM-4-0520", "GLM-
    val webSearchActive: Boolean = false,
    val deepThinkingActive: Boolean = true,
    val unreadTokensWhileScrolledUp: Int = 0,
    val bannerError: String? = null
)

sealed interface ChatUiEvent {
    data class SendMessage(val text: String) : ChatUiEvent
    data object StopStreaming : ChatUiEvent
    data class ToggleWebSearch(val enabled: Boolean) : ChatUiEvent
    data class ToggleDeepThinking(val enabled: Boolean) : ChatUiEvent
    data class SelectModel(val model: String) : ChatUiEvent
    data class EditAndResend(val messageId: String, val newPrompt: String) : Cha
    data object RegenerateLast : ChatUiEvent
    data class RetryFromMessage(val message: Message) : ChatUiEvent
    data object DismissError : ChatUiEvent
    data class UserScrolledUp(val isScrolledUp: Boolean) : ChatUiEvent
}
```

ui/screens/chat/ChatViewModel.kt

```
package com.zai.chat.ui.screens.chat

import androidx.lifecycle.SavedStateHandle
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.zai.chat.config.ZaiConfig
import com.zai.chat.data.local.preferences.SettingsDataStore
import com.zai.chat.data.model.Message
import com.zai.chat.data.model.MessageRole
import com.zai.chat.data.repository.ChatRepository
import com.zai.chat.network.ZaiApiService
import com.zai.chat.network.sse.StreamEvent
import dagger.hilt.android.lifecycle.HiltViewModel
import kotlinx.coroutines.Job
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.flow.collectLatest
import kotlinx.coroutines.flow.update
import kotlinx.coroutines.launch
import java.util.UUID
import javax.inject.Inject

@HiltViewModel
```

```
class ChatViewModel @Inject constructor(
    private val repository: ChatRepository,
    private val apiService: ZaiApiService,
    private val settingsDataStore: SettingsDataStore,
    savedStateHandle: SavedStateHandle
) : ViewModel() {

    private val _uiState = MutableStateFlow(ChatUiState())
    val uiState: StateFlow<ChatUiState> = _uiState.asStateFlow()

    private var streamingJob: Job? = null
    private var isUserScrolledUp = false

    init {
        val initialChatId: String? = savedStateHandle["chatId"]
        if (initialChatId != null) {
            loadChat(initialChatId)
        }

        viewModelScope.launch {
            settingsDataStore.defaultModel.collectLatest { model ->
                _uiState.update { it.copy(selectedModel = model) }
            }
        }

        viewModelScope.launch {
            val models = apiService.getAvailableModels()
            _uiState.update { it.copy(availableModels = models) }
        }
    }

    fun loadChat(chatId: String) {
        _uiState.update { it.copy(currentChatId = chatId) }
        viewModelScope.launch {
            repository.syncChatMessages(chatId)
        }
        viewModelScope.launch {
            repository.getMessagesStream(chatId).collectLatest { msgs ->
                _uiState.update { it.copy(messages = msgs) }
            }
        }
    }

    fun onEvent(event: ChatUiEvent) {
        when (event) {
            is ChatUiEvent.SendMessage -> handleSendMessage(event.text)
            is ChatUiEvent.StopStreaming -> stopStreaming()
            is ChatUiEvent.ToggleWebSearch -> _uiState.update { it.copy(webSearcl
            is ChatUiEvent.ToggleDeepThinking -> _uiState.update { it.copy(deepTl
            is ChatUiEvent.SelectModel -> _uiState.update { it.copy(selectedMode
            is ChatUiEvent.EditAndResend -> handleEditAndResend(event.messageId,
            is ChatUiEvent.RegenerateLast -> handleRegenerate()
            is ChatUiEvent.RetryFromMessage -> handleRetryFromMessage(event.mess
            is ChatUiEvent.DismissError -> _uiState.update { it.copy(bannerError
            is ChatUiEvent.UserScrolledUp -> {
                isUserScrolledUp = event.isScrolledUp
```

```
            if (!isUserScrolledUp) {
                _uiState.update { it.copy(unreadTokensWhileScrolledUp = 0) }
            }
        }
    }
}

private fun handleSendMessage(prompt: String) {
    if (prompt.isBlank() || _uiState.value.isStreaming) return

    streamingJob?.cancel()
    streamingJob = viewModelScope.launch {
        var chatId = _uiState.value.currentChatId
        if (chatId == null) {
            chatId = repository.createChat(prompt)
            _uiState.update { it.copy(currentChatId = chatId) }
            loadChat(chatId)
        }

        _uiState.update {
            it.copy(
                isStreaming = true,
                liveReasoningText = "",
                liveContentText = "",
                unreadTokensWhileScrolledUp = 0,
                bannerError = null
            )
        }

        repository.streamCompletion(
            chatId = chatId,
            userPrompt = prompt,
            model = _uiState.value.selectedModel,
            webSearch = _uiState.value.webSearchActive,
            deepThinking = _uiState.value.deepThinkingActive
        ).collect { event ->
            when (event) {
                is StreamEvent.ReasoningDelta -> {
                    _uiState.update {
                        it.copy(liveReasoningText = it.liveReasoningText + e
                    }
                }
                is StreamEvent.ContentDelta -> {
                    _uiState.update {
                        val newUnread = if (isUserScrolledUp) it.unreadToken
                        it.copy(
                            liveContentText = it.liveContentText + event.tex
                            unreadTokensWhileScrolledUp = newUnread
                        )
                    }
                }
                is StreamEvent.Done -> {
                    _uiState.update { it.copy(isStreaming = false, liveConte
                }
                is StreamEvent.Error -> {
                    _uiState.update {
```

```
                            it.copy(
                                isStreaming = false,
                                bannerError = "Connection interrupted. Partial r
                            )
                        }
                    }
                    else -> {}
                }
            }
        }
    }

    private fun stopStreaming() {
        streamingJob?.cancel()
        streamingJob = null
        _uiState.update { it.copy(isStreaming = false) }
    }

    private fun handleEditAndResend(messageId: String, newPrompt: String) {
        val target = _uiState.value.messages.firstOrNull { it.id == messageId } 
        val chatId = _uiState.value.currentChatId ?: return
        viewModelScope.launch {
            repository.rollbackMessagesAfter(chatId, target.createdAt)
            handleSendMessage(newPrompt)
        }
    }

    private fun handleRegenerate() {
        val lastUserMsg = _uiState.value.messages.lastOrNull { it.role == Messag
        handleSendMessage(lastUserMsg.content)
    }

    private fun handleRetryFromMessage(message: Message) {
        val chatId = _uiState.value.currentChatId ?: return
        viewModelScope.launch {
            repository.rollbackMessagesAfter(chatId, message.createdAt)
            val lastUserMsg = _uiState.value.messages.lastOrNull { it.role == Me
            if (lastUserMsg != null) {
                handleSendMessage(lastUserMsg.content)
            }
        }
    }
}
```

Flowy Glassmorphic Composer: `ui/screens/chat/ChatComposer.kt`

```
package com.zai.chat.ui.screens.chat

import android.view.HapticFeedbackConstants
import androidx.compose.animation.AnimatedVisibility
import androidx.compose.animation.animateColorAsState
import androidx.compose.animation.core.animateDpAsState
import androidx.compose.foundation.background
```

```
import androidx.compose.foundation.border
import androidx.compose.foundation.clickable
import androidx.compose.foundation.horizontalScroll
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.heightIn
import androidx.compose.foundation.layout.imePadding
import androidx.compose.foundation.layout.navigationBarsPadding
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.layout.width
import androidx.compose.foundation.rememberScrollState
import androidx.compose.foundation.shape.CircleShape
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.foundation.text.BasicTextField
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.rounded.ArrowUpward
import androidx.compose.material.icons.rounded.AttachFile
import androidx.compose.material.icons.rounded.Language
import androidx.compose.material.icons.rounded.Psychology
import androidx.compose.material.icons.rounded.Stop
import androidx.compose.material3.DropdownMenu
import androidx.compose.material3.DropdownMenuItem
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.graphics.SolidColor
import androidx.compose.ui.platform.LocalView
import androidx.compose.ui.text.TextStyle
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import com.zai.chat.ui.theme.ClaudePeach
import com.zai.chat.ui.theme.KimiCyan
import com.zai.chat.ui.theme.SurfaceContainerDark
import com.zai.chat.ui.theme.SurfaceContainerHighDark
import com.zai.chat.ui.theme.ZaiMotion

@Composable
fun ChatComposer(
    isStreaming: Boolean,
```

```
    selectedModel: String,
    availableModels: List<String>,
    webSearchActive: Boolean,
    deepThinkingActive: Boolean,
    onSendMessage: (String) -> Unit,
    onStopStreaming: () -> Unit,
    onToggleWebSearch: (Boolean) -> Unit,
    onToggleDeepThinking: (Boolean) -> Unit,
    onSelectModel: (String) -> Unit,
    modifier: Modifier = Modifier
) {
    var textInput by remember { mutableStateOf("") }
    val view = LocalView.current
    var modelDropdownOpen by remember { mutableStateOf(false) }

    Column(
        modifier = modifier
            .fillMaxWidth()
            .background(SurfaceContainerDark.copy(alpha = 0.95f))
            .border(1.dp, MaterialTheme.colorScheme.outline.copy(alpha = 0.2f))
            .navigationBarsPadding()
            .imePadding()
            .padding(top = 8.dp, bottom = 10.dp, start = 12.dp, end = 12.dp)
    ) {
        // Quick Toggle Chips Row
        Row(
            modifier = Modifier
                .fillMaxWidth()
                .horizontalScroll(rememberScrollState()),
            horizontalArrangement = Arrangement.spacedBy(8.dp),
            verticalAlignment = Alignment.CenterVertically
        ) {
            // Model Picker Chip
            Box {
                Surface(
                    shape = RoundedCornerShape(16.dp),
                    color = SurfaceContainerHighDark,
                    modifier = Modifier.clickable { modelDropdownOpen = true }
                ) {
                    Text(
                        text = selectedModel,
                        style = MaterialTheme.typography.labelSmall.copy(fontWei
                        modifier = Modifier.padding(horizontal = 10.dp, vertical
                    )
                }

                DropdownMenu(
                    expanded = modelDropdownOpen,
                    onDismissRequest = { modelDropdownOpen = false }
                ) {
                    availableModels.forEach { model ->
                        DropdownMenuItem(
                            text = { Text(model) },
                            onClick = {
                                onSelectModel(model)
                                modelDropdownOpen = false
```

```
                }
            )
        }
    }
}

// Web Search Chip
Surface(
    shape = RoundedCornerShape(16.dp),
    color = if (webSearchActive) KimiCyan.copy(alpha = 0.2f) else Su
    modifier = Modifier.clickable { onToggleWebSearch(!webSearchActi
) {
    Row(
        modifier = Modifier.padding(horizontal = 10.dp, vertical = 5
        verticalAlignment = Alignment.CenterVertically
    ) {
        Icon(
            imageVector = Icons.Rounded.Language,
            contentDescription = null,
            tint = if (webSearchActive) KimiCyan else MaterialTheme.
            modifier = Modifier.size(14.dp)
        )
        Spacer(modifier = Modifier.width(4.dp))
        Text(
            text = "Search",
            style = MaterialTheme.typography.labelSmall.copy(
                color = if (webSearchActive) KimiCyan else MaterialT
            )
        )
    }
}

// Deep Thinking Chip
Surface(
    shape = RoundedCornerShape(16.dp),
    color = if (deepThinkingActive) ClaudePeach.copy(alpha = 0.2f) e
    modifier = Modifier.clickable { onToggleDeepThinking(!deepThinki
) {
    Row(
        modifier = Modifier.padding(horizontal = 10.dp, vertical = 5
        verticalAlignment = Alignment.CenterVertically
    ) {
        Icon(
            imageVector = Icons.Rounded.Psychology,
            contentDescription = null,
            tint = if (deepThinkingActive) ClaudePeach else Material
            modifier = Modifier.size(14.dp)
        )
        Spacer(modifier = Modifier.width(4.dp))
        Text(
            text = "Thinking",
            style = MaterialTheme.typography.labelSmall.copy(
                color = if (deepThinkingActive) ClaudePeach else Mat
            )
        )
    }
```

```
        }
    }

    Spacer(modifier = Modifier.height(8.dp))

    // Input Box + Attachment + Morphing Send/Stop Button
    Row(
        modifier = Modifier.fillMaxWidth(),
        verticalAlignment = Alignment.Bottom
    ) {
        IconButton(
            onClick = { /* File picker integration */ },
            modifier = Modifier.size(40.dp)
        ) {
            Icon(
                imageVector = Icons.Rounded.AttachFile,
                contentDescription = "Attach",
                tint = MaterialTheme.colorScheme.onSurfaceVariant
            )
        }

        Box(
            modifier = Modifier
                .weight(1f)
                .clip(RoundedCornerShape(20.dp))
                .background(SurfaceContainerHighDark)
                .padding(horizontal = 14.dp, vertical = 10.dp)
        ) {
            if (textInput.isEmpty()) {
                Text(
                    text = "Ask anything…",
                    style = MaterialTheme.typography.bodyLarge.copy(
                        color = MaterialTheme.colorScheme.onSurfaceVariant.c
                    )
                )
            }
            BasicTextField(
                value = textInput,
                onValueChange = { textInput = it },
                textStyle = TextStyle(
                    color = MaterialTheme.colorScheme.onSurface,
                    fontSize = 16.sp,
                    lineHeight = 22.sp
                ),
                cursorBrush = SolidColor(MaterialTheme.colorScheme.primary),
                modifier = Modifier
                    .fillMaxWidth()
                    .heightIn(max = 140.dp)
            )
        }

        Spacer(modifier = Modifier.width(8.dp))

        // Spring Animated Morphing Send/Stop Button
        val sendButtonBg by animateColorAsState(
            targetValue = if (isStreaming) ClaudePeach else if (textInput.isE
```

```
                animationSpec = ZaiMotion.MorphSpring,
                label = "button_color"
            )

            Box(
                modifier = Modifier
                    .size(40.dp)
                    .clip(CircleShape)
                    .background(sendButtonBg)
                    .clickable {
                        view.performHapticFeedback(HapticFeedbackConstants.KEYBO
                        if (isStreaming) {
                            onStopStreaming()
                        } else if (textInput.isNotBlank()) {
                            onSendMessage(textInput)
                            textInput = ""
                        }
                    },
                contentAlignment = Alignment.Center
            ) {
                Icon(
                    imageVector = if (isStreaming) Icons.Rounded.Stop else Icons
                    contentDescription = if (isStreaming) "Stop" else "Send",
                    tint = if (isStreaming || textInput.isNotBlank()) Color.Blac
                    modifier = Modifier.size(20.dp)
                )
            }
        }
    }
}
```

Chat Screen: `ui/screens/chat/ChatScreen.kt`

```
package com.zai.chat.ui.screens.chat

import androidx.compose.animation.AnimatedVisibility
import androidx.compose.animation.fadeIn
import androidx.compose.animation.fadeOut
import androidx.compose.animation.slideInVertically
import androidx.compose.animation.slideOutVertically
import androidx.compose.foundation.background
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.layout.width
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.foundation.lazy.rememberLazyListState
```

```
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.rounded.Close
import androidx.compose.material.icons.rounded.KeyboardArrowDown
import androidx.compose.material.icons.rounded.Menu
import androidx.compose.material.icons.rounded.Warning
import androidx.compose.material3.Badge
import androidx.compose.material3.BadgedBox
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.FloatingActionButton
import androidx.compose.material3.FloatingActionButtonDefaults
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Scaffold
import androidx.compose.material3.SnackbarHost
import androidx.compose.material3.SnackbarHostState
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
import androidx.compose.material3.TopAppBar
import androidx.compose.material3.TopAppBarDefaults
import androidx.compose.runtime.Composable
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.collectAsState
import androidx.compose.runtime.derivedStateOf
import androidx.compose.runtime.getValue
import androidx.compose.runtime.remember
import androidx.compose.runtime.rememberCoroutineScope
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.text.style.TextOverflow
import androidx.compose.ui.unit.dp
import com.zai.chat.data.model.Message
import com.zai.chat.data.model.MessageRole
import com.zai.chat.ui.components.MessageBubble
import com.zai.chat.ui.theme.ClaudePeach
import com.zai.chat.ui.theme.TrueBlack
import kotlinx.coroutines.launch

@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun ChatScreen(
    viewModel: ChatViewModel,
    onOpenDrawer: () -> Unit,
    modifier: Modifier = Modifier
) {
    val state by viewModel.uiState.collectAsState()
    val listState = rememberLazyListState()
    val scope = rememberCoroutineScope()
    val snackbarHostState = remember { SnackbarHostState() }

    // LazyColumn reverseLayout = true ensures zero jump on incoming tokens
    val isScrolledUp by remember {
        derivedStateOf { listState.firstVisibleItemIndex > 0 }
    }
```

```
LaunchedEffect(isScrolledUp) {
    viewModel.onEvent(ChatUiEvent.UserScrolledUp(isScrolledUp))
}

Scaffold(
    topBar = {
        TopAppBar(
            title = {
                Text(
                    text = state.chatTitle,
                    style = MaterialTheme.typography.titleMedium.copy(fontWe
                    maxLines = 1,
                    overflow = TextOverflow.Ellipsis
                )
            },
            navigationIcon = {
                IconButton(onClick = onOpenDrawer) {
                    Icon(imageVector = Icons.Rounded.Menu, contentDescriptio
                }
            },
            colors = TopAppBarDefaults.topAppBarColors(containerColor = True
        )
    },
    snackbarHost = { SnackbarHost(snackbarHostState) },
    bottomBar = {
        ChatComposer(
            isStreaming = state.isStreaming,
            selectedModel = state.selectedModel,
            availableModels = state.availableModels,
            webSearchActive = state.webSearchActive,
            deepThinkingActive = state.deepThinkingActive,
            onSendMessage = { viewModel.onEvent(ChatUiEvent.SendMessage(it))
            onStopStreaming = { viewModel.onEvent(ChatUiEvent.StopStreaming)
            onToggleWebSearch = { viewModel.onEvent(ChatUiEvent.ToggleWebSea
            onToggleDeepThinking = { viewModel.onEvent(ChatUiEvent.ToggleDee
            onSelectModel = { viewModel.onEvent(ChatUiEvent.SelectModel(it))
        )
    },
    containerColor = TrueBlack,
    modifier = modifier.fillMaxSize()
) { paddingValues ->
    Box(
        modifier = Modifier
            .fillMaxSize()
            .padding(paddingValues)
    ) {
        // Error Banner
        state.bannerError?.let { err ->
            Surface(
                color = ClaudePeach.copy(alpha = 0.15f),
                shape = RoundedCornerShape(8.dp),
                modifier = Modifier
                    .fillMaxWidth()
                    .padding(8.dp)
                    .align(Alignment.TopCenter)
```

```
    ) {
        Row(
            modifier = Modifier.padding(horizontal = 12.dp, vertical
            verticalAlignment = Alignment.CenterVertically
        ) {
            Icon(imageVector = Icons.Rounded.Warning, contentDescrip
            Spacer(modifier = Modifier.width(8.dp))
            Text(text = err, style = MaterialTheme.typography.bodyMe
            IconButton(onClick = { viewModel.onEvent(ChatUiEvent.Dis
                Icon(imageVector = Icons.Rounded.Close, contentDescr
            }
        }
    }
}

// Inverted Message Stream (reverseLayout = true)
LazyColumn(
    state = listState,
    reverseLayout = true,
    modifier = Modifier.fillMaxSize()
) {
    // If actively streaming, show the live transient chunk first
    if (state.isStreaming) {
        item(key = "transient_streaming_assistant_msg") {
            val transientMsg = Message(
                id = "transient_msg",
                chatId = state.currentChatId ?: "",
                role = MessageRole.ASSISTANT,
                content = state.liveContentText,
                reasoning = state.liveReasoningText.takeIf { it.isNo
            )
            MessageBubble(
                message = transientMsg,
                isStreaming = true
            )
        }
    }

    // Render persisted messages from Room (in reverse order for rev
    items(
        items = state.messages.reversed(),
        key = { it.id }
    ) { msg ->
        MessageBubble(
            message = msg,
            isStreaming = false,
            onEditAndResend = { viewModel.onEvent(ChatUiEvent.EditAn
            onRegenerate = { viewModel.onEvent(ChatUiEvent.Regenerat
            onRetryFromHere = { viewModel.onEvent(ChatUiEvent.RetryF
        )
    }
}

// Scroll-to-Bottom FAB with live tokens badge
AnimatedVisibility(
    visible = isScrolledUp,
```

```
                enter = fadeIn() + slideInVertically { it },
                exit = fadeOut() + slideOutVertically { it },
                modifier = Modifier
                    .align(Alignment.BottomEnd)
                    .padding(16.dp)
            ) {
                BadgedBox(
                    badge = {
                        if (state.unreadTokensWhileScrolledUp > 0) {
                            Badge(containerColor = ClaudePeach) {
                                Text("+${state.unreadTokensWhileScrolledUp}")
                            }
                        }
                    }
                ) {
                    FloatingActionButton(
                        onClick = {
                            scope.launch {
                                listState.animateScrollToItem(0)
                            }
                        },
                        containerColor = MaterialTheme.colorScheme.surfaceVarian
                        contentColor = MaterialTheme.colorScheme.onSurface,
                        elevation = FloatingActionButtonDefaults.elevation(4.dp)
                        modifier = Modifier.size(44.dp)
                    ) {
                        Icon(imageVector = Icons.Rounded.KeyboardArrowDown, cont
                    }
                }
            }
        }
    }
}
```

10. Drawer Navigation, Room FTS Live Search & Settings

ui/screens/drawer/DrawerViewModel.kt

```
package com.zai.chat.ui.screens.drawer

import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.zai.chat.data.model.Chat
import com.zai.chat.data.model.Message
import com.zai.chat.data.repository.ChatRepository
import dagger.hilt.android.lifecycle.HiltViewModel
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.flow.collectLatest
import kotlinx.coroutines.flow.debounce
import kotlinx.coroutines.flow.distinctUntilChanged
```

```
import kotlinx.coroutines.flow.flatMapLatest
import kotlinx.coroutines.flow.flowOf
import kotlinx.coroutines.flow.update
import kotlinx.coroutines.launch
import javax.inject.Inject

data class DrawerUiState(
    val chats: List<Chat> = emptyList(),
    val searchQuery: String = "",
    val searchResults: List<Message> = emptyList()
)

@HiltViewModel
class DrawerViewModel @Inject constructor(
    private val repository: ChatRepository
) : ViewModel() {

    private val _uiState = MutableStateFlow(DrawerUiState())
    val uiState: StateFlow<DrawerUiState> = _uiState.asStateFlow()

    private val queryFlow = MutableStateFlow("")

    init {
        viewModelScope.launch {
            repository.getChatsStream().collectLatest { chatList ->
                _uiState.update { it.copy(chats = chatList) }
            }
        }

        viewModelScope.launch {
            repository.refreshChats()
        }

        viewModelScope.launch {
            queryFlow
                .debounce(250)
                .distinctUntilChanged()
                .flatMapLatest { query ->
                    if (query.isBlank()) flowOf(emptyList()) else repository.sea
                }
                .collectLatest { results ->
                    _uiState.update { it.copy(searchResults = results) }
                }
        }
    }

    fun onSearchQueryChanged(q: String) {
        _uiState.update { it.copy(searchQuery = q) }
        queryFlow.value = q
    }

    fun togglePin(chatId: String, currentPin: Boolean) {
        viewModelScope.launch {
            repository.togglePinChat(chatId, !currentPin)
        }
    }
```

```
    fun deleteChat(chatId: String) {
        viewModelScope.launch {
            repository.deleteChat(chatId)
        }
    }
}
```

ui/screens/drawer/DrawerChatList.kt

```
package com.zai.chat.ui.screens.drawer

import androidx.compose.foundation.background
import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxHeight
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.layout.width
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.rounded.Add
import androidx.compose.material.icons.rounded.Delete
import androidx.compose.material.icons.rounded.PushPin
import androidx.compose.material.icons.rounded.Search
import androidx.compose.material.icons.rounded.Settings
import androidx.compose.material3.DismissDirection
import androidx.compose.material3.DismissValue
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.OutlinedTextField
import androidx.compose.material3.OutlinedTextFieldDefaults
import androidx.compose.material3.SwipeToDismiss
import androidx.compose.material3.Text
import androidx.compose.material3.rememberDismissState
import androidx.compose.runtime.Composable
import androidx.compose.runtime.collectAsState
import androidx.compose.runtime.getValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.text.font.FontWeight
```

```
import androidx.compose.ui.text.style.TextOverflow
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import com.zai.chat.data.model.Chat
import com.zai.chat.ui.theme.ClaudePeach
import com.zai.chat.ui.theme.SurfaceContainerDark
import com.zai.chat.ui.theme.SurfaceContainerHighDark
import java.text.SimpleDateFormat
import java.util.Date
import java.util.Locale

@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun DrawerChatList(
    viewModel: DrawerViewModel,
    onSelectChat: (String) -> Unit,
    onNewChat: () -> Unit,
    onOpenSettings: () -> Unit,
    modifier: Modifier = Modifier
) {
    val state by viewModel.uiState.collectAsState()

    Column(
        modifier = modifier
            .fillMaxHeight()
            .width(320.dp)
            .background(SurfaceContainerDark)
            .padding(16.dp)
    ) {
        // Top Action Header
        Row(
            modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.SpaceBetween,
            verticalAlignment = Alignment.CenterVertically
        ) {
            Text(
                text = "Chats",
                style = MaterialTheme.typography.titleLarge.copy(fontWeight = Fo
            )
            Row {
                IconButton(onClick = onNewChat) {
                    Icon(imageVector = Icons.Rounded.Add, contentDescription = "
                }
                IconButton(onClick = onOpenSettings) {
                    Icon(imageVector = Icons.Rounded.Settings, contentDescriptio
                }
            }
        }

        Spacer(modifier = Modifier.height(8.dp))

        // FTS Search Field
        OutlinedTextField(
            value = state.searchQuery,
            onValueChange = { viewModel.onSearchQueryChanged(it) },
            placeholder = { Text("Search messages…", style = MaterialTheme.typog
```

```
        leadingIcon = { Icon(Icons.Rounded.Search, contentDescription = null
        shape = RoundedCornerShape(12.dp),
        colors = OutlinedTextFieldDefaults.colors(
            focusedContainerColor = SurfaceContainerHighDark,
            unfocusedContainerColor = SurfaceContainerHighDark,
            focusedBorderColor = Color.Transparent,
            unfocusedBorderColor = Color.Transparent
        ),
        modifier = Modifier.fillMaxWidth()
    )

    Spacer(modifier = Modifier.height(12.dp))

    // Items or Search Results
    LazyColumn(
        modifier = Modifier.fillMaxWidth().weight(1f),
        verticalArrangement = Arrangement.spacedBy(4.dp)
    ) {
        if (state.searchQuery.isNotBlank()) {
            items(state.searchResults, key = { it.id }) { searchMsg ->
                Column(
                    modifier = Modifier
                        .fillMaxWidth()
                        .clip(RoundedCornerShape(8.dp))
                        .background(SurfaceContainerHighDark)
                        .clickable { onSelectChat(searchMsg.chatId) }
                        .padding(10.dp)
                ) {
                    Text(
                        text = searchMsg.content,
                        style = MaterialTheme.typography.bodyMedium,
                        maxLines = 2,
                        overflow = TextOverflow.Ellipsis
                    )
                }
            }
        } else {
            items(state.chats, key = { it.id }) { chat ->
                val dismissState = rememberDismissState(
                    confirmValueChange = { value ->
                        if (value == DismissValue.DismissedToStart) {
                            viewModel.deleteChat(chat.id)
                            true
                        } else false
                    }
                )

                SwipeToDismiss(
                    state = dismissState,
                    directions = setOf(DismissDirection.EndToStart),
                    background = {
                        Box(
                            modifier = Modifier
                                .fillMaxWidth()
                                .fillMaxHeight()
                                .clip(RoundedCornerShape(8.dp))
```

```
                                    .background(Color.Red.copy(alpha = 0.8f))
                                    .padding(horizontal = 16.dp),
                                contentAlignment = Alignment.CenterEnd
                            ) {
                                Icon(Icons.Rounded.Delete, contentDescription =
                            }
                        },
                        dismissContent = {
                            ChatItemRow(
                                chat = chat,
                                onClick = { onSelectChat(chat.id) },
                                onTogglePin = { viewModel.togglePin(chat.id, cha
                            )
                        }
                    )
                }
            }
        }
    }
}

@Composable
fun ChatItemRow(
    chat: Chat,
    onClick: () -> Unit,
    onTogglePin: () -> Unit
) {
    val dateStr = remember(chat.updatedAt) {
        val sdf = SimpleDateFormat("MMM d", Locale.getDefault())
        sdf.format(Date(chat.updatedAt))
    }

    Row(
        modifier = Modifier
            .fillMaxWidth()
            .clip(RoundedCornerShape(8.dp))
            .background(SurfaceContainerHighDark)
            .clickable { onClick() }
            .padding(horizontal = 12.dp, vertical = 10.dp),
        verticalAlignment = Alignment.CenterVertically
    ) {
        Column(modifier = Modifier.weight(1f)) {
            Text(
                text = chat.title,
                style = MaterialTheme.typography.bodyMedium.copy(fontWeight = Fo
                maxLines = 1,
                overflow = TextOverflow.Ellipsis
            )
            Text(
                text = dateStr,
                style = MaterialTheme.typography.labelSmall.copy(
                    fontSize = 11.sp,
                    color = MaterialTheme.colorScheme.onSurfaceVariant
                )
            )
        }
```

```
        IconButton(onClick = onTogglePin, modifier = Modifier.size(24.dp)) {
            Icon(
                imageVector = Icons.Rounded.PushPin,
                contentDescription = "Pin",
                tint = if (chat.pinned) ClaudePeach else MaterialTheme.colorSche
                modifier = Modifier.size(16.dp)
            )
        }
    }
}
```

Settings Screen: `ui/screens/settings/SettingsScreen.kt`

```
package com.zai.chat.ui.screens.settings

import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.padding
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.rounded.ArrowBack
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.RadioButton
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Slider
import androidx.compose.material3.Switch
import androidx.compose.material3.Text
import androidx.compose.material3.TopAppBar
import androidx.compose.runtime.Composable
import androidx.compose.runtime.collectAsState
import androidx.compose.runtime.getValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.unit.dp
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.zai.chat.data.local.preferences.SettingsDataStore
import dagger.hilt.android.lifecycle.HiltViewModel
import kotlinx.coroutines.flow.SharingStarted
import kotlinx.coroutines.flow.stateIn
import kotlinx.coroutines.launch
import javax.inject.Inject

@HiltViewModel
```

```
class SettingsViewModel @Inject constructor(
    private val settingsDataStore: SettingsDataStore
) : ViewModel() {
    val themeMode = settingsDataStore.themeMode.stateIn(viewModelScope, SharingS
    val fontScale = settingsDataStore.fontScale.stateIn(viewModelScope, SharingS
    val enterIsSend = settingsDataStore.enterIsSend.stateIn(viewModelScope, Shar

    fun setTheme(mode: String) = viewModelScope.launch { settingsDataStore.setTh
    fun setScale(scale: Float) = viewModelScope.launch { settingsDataStore.setFo
    fun setEnterIsSend(enabled: Boolean) = viewModelScope.launch { settingsDataS
}

@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun SettingsScreen(
    onNavigateBack: () -> Unit,
    viewModel: SettingsViewModel = hiltViewModel(),
    modifier: Modifier = Modifier
) {
    val theme by viewModel.themeMode.collectAsState()
    val fontScale by viewModel.fontScale.collectAsState()
    val enterSend by viewModel.enterIsSend.collectAsState()

    Scaffold(
        topBar = {
            TopAppBar(
                title = { Text("Settings") },
                navigationIcon = {
                    IconButton(onClick = onNavigateBack) {
                        Icon(Icons.Rounded.ArrowBack, contentDescription = "Back
                    }
                }
            )
        },
        modifier = modifier.fillMaxSize()
    ) { padding ->
        Column(
            modifier = Modifier
                .fillMaxSize()
                .padding(padding)
                .padding(16.dp)
        ) {
            Text("Theme Mode", style = MaterialTheme.typography.titleMedium.copy
            listOf("OLED", "DARK", "LIGHT").forEach { mode ->
                Row(verticalAlignment = Alignment.CenterVertically) {
                    RadioButton(selected = theme == mode, onClick = { viewModel.
                    Text(text = mode, style = MaterialTheme.typography.bodyMediu
                }
            }

            Spacer(modifier = Modifier.height(20.dp))

            Text("Reading Font Scale (${"%.2f".format(fontScale)}x)", style = Ma
            Slider(
                value = fontScale,
                onValueChange = { viewModel.setScale(it) },
```

```
                valueRange = 0.85f..1.35f,
                modifier = Modifier.fillMaxWidth()
            )

            Spacer(modifier = Modifier.height(20.dp))

            Row(
                modifier = Modifier.fillMaxWidth(),
                verticalAlignment = Alignment.CenterVertically
            ) {
                Text(
                    text = "Press Enter to Send",
                    style = MaterialTheme.typography.titleMedium.copy(fontWeight
                    modifier = Modifier.weight(1f)
                )
                Switch(
                    checked = enterSend,
                    onCheckedChange = { viewModel.setEnterIsSend(it) }
                )
            }
        }
    }
}
```

11. Hilt Dependency Injection Setup

di/NetworkModule.kt & DatabaseModule.kt

```
package com.zai.chat.di

import android.content.Context
import androidx.room.Room
import com.zai.chat.config.ZaiConfig
import com.zai.chat.data.local.ZaiDatabase
import com.zai.chat.data.local.dao.ChatDao
import com.zai.chat.data.local.dao.MessageDao
import com.zai.chat.network.auth.AuthInterceptor
import dagger.Module
import dagger.Provides
import dagger.hilt.InstallIn
import dagger.hilt.android.qualifiers.ApplicationContext
import dagger.hilt.components.SingletonComponent
import kotlinx.serialization.json.Json
import okhttp3.OkHttpClient
import okhttp3.logging.HttpLoggingInterceptor
import java.util.concurrent.TimeUnit
import javax.inject.Singleton

@Module
@InstallIn(SingletonComponent::class)
object NetworkModule {
```

```
    @Provides
    @Singleton
    fun provideJson(): Json = Json {
        ignoreUnknownKeys = true
        isLenient = true
        coerceInputValues = true
        encodeDefaults = true
    }

    @Provides
    @Singleton
    fun provideOkHttpClient(authInterceptor: AuthInterceptor): OkHttpClient {
        val logging = HttpLoggingInterceptor().apply {
            level = HttpLoggingInterceptor.Level.BASIC
        }

        return OkHttpClient.Builder()
            .addInterceptor(authInterceptor)
            .addInterceptor(logging)
            .connectTimeout(ZaiConfig.CONNECT_TIMEOUT_SECONDS, TimeUnit.SECONDS)
            .readTimeout(ZaiConfig.READ_TIMEOUT_SECONDS, TimeUnit.SECONDS)
            .writeTimeout(ZaiConfig.WRITE_TIMEOUT_SECONDS, TimeUnit.SECONDS)
            .retryOnConnectionFailure(true)
            .build()
    }
}

@Module
@InstallIn(SingletonComponent::class)
object DatabaseModule {

    @Provides
    @Singleton
    fun provideDatabase(@ApplicationContext context: Context): ZaiDatabase {
        return Room.databaseBuilder(
            context,
            ZaiDatabase::class.java,
            "zai_local_chat.db"
        ).fallbackToDestructiveMigration().build()
    }

    @Provides
    fun provideChatDao(db: ZaiDatabase): ChatDao = db.chatDao()

    @Provides
    fun provideMessageDao(db: ZaiDatabase): MessageDao = db.messageDao()
}
```

di/AppModule.kt

```
package com.zai.chat.di

import com.zai.chat.data.repository.ChatRepository
```

```
import com.zai.chat.data.repository.ChatRepositoryImpl
import dagger.Binds
import dagger.Module
import dagger.hilt.InstallIn
import dagger.hilt.components.SingletonComponent
import javax.inject.Singleton

@Module
@InstallIn(SingletonComponent::class)
abstract class AppModule {

    @Binds
    @Singleton
    abstract fun bindChatRepository(impl: ChatRepositoryImpl): ChatRepository
}
```

12. Main Activity, Token Reconnect Dialog & Navigation

ui/MainActivity.kt

```
package com.zai.chat.ui

import android.annotation.SuppressLint
import android.os.Bundle
import android.webkit.CookieManager
import android.webkit.WebResourceRequest
import android.webkit.WebView
import android.webkit.WebViewClient
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.material3.AlertDialog
import androidx.compose.material3.DrawerValue
import androidx.compose.material3.ModalNavigationDrawer
import androidx.compose.material3.Text
import androidx.compose.material3.TextButton
import androidx.compose.material3.rememberDrawerState
import androidx.compose.runtime.Composable
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.collectAsState
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.rememberCoroutineScope
import androidx.compose.runtime.setValue
import androidx.compose.ui.Modifier
import androidx.compose.ui.viewinterop.AndroidView
import androidx.hilt.navigation.compose.hiltViewModel
import androidx.navigation.compose.NavHost
import androidx.navigation.compose.composable
import androidx.navigation.compose.rememberNavController
```

```
import com.zai.chat.config.ZaiConfig
import com.zai.chat.data.local.preferences.SettingsDataStore
import com.zai.chat.data.local.preferences.TokenManager
import com.zai.chat.network.auth.AuthEvent
import com.zai.chat.network.auth.AuthEventManager
import com.zai.chat.ui.screens.chat.ChatScreen
import com.zai.chat.ui.screens.chat.ChatViewModel
import com.zai.chat.ui.screens.drawer.DrawerChatList
import com.zai.chat.ui.screens.drawer.DrawerViewModel
import com.zai.chat.ui.screens.settings.SettingsScreen
import com.zai.chat.ui.theme.ZaiTheme
import dagger.hilt.android.AndroidEntryPoint
import kotlinx.coroutines.launch
import javax.inject.Inject

@AndroidEntryPoint
class MainActivity : ComponentActivity() {

    @Inject
    lateinit var tokenManager: TokenManager

    @Inject
    lateinit var authEventManager: AuthEventManager

    @Inject
    lateinit var settingsDataStore: SettingsDataStore

    override fun onCreate(savedInstanceState: Bundle?) {
        enableEdgeToEdge()
        super.onCreate(savedInstanceState)

        setContent {
            val themeMode by settingsDataStore.themeMode.collectAsState(initial
            var showReconnectDialog by remember { mutableStateOf(false) }

            LaunchedEffect(Unit) {
                authEventManager.events.collect { event ->
                    when (event) {
                        is AuthEvent.TokenExpired -> {
                            showReconnectDialog = true
                        }
                    }
                }
            }

            ZaiTheme(themeMode = themeMode) {
                val navController = rememberNavController()
                val drawerState = rememberDrawerState(initialValue = DrawerValue
                val scope = rememberCoroutineScope()
                val drawerViewModel: DrawerViewModel = hiltViewModel()

                ModalNavigationDrawer(
                    drawerState = drawerState,
                    gesturesEnabled = drawerState.isOpen,
                    drawerContent = {
                        DrawerChatList(
```

```
                            viewModel = drawerViewModel,
                            onSelectChat = { chatId ->
                                scope.launch { drawerState.close() }
                                navController.navigate("chat?chatId=$chatId") {
                                    launchSingleTop = true
                                }
                            },
                            onNewChat = {
                                scope.launch { drawerState.close() }
                                navController.navigate("chat") {
                                    launchSingleTop = true
                                }
                            },
                            onOpenSettings = {
                                scope.launch { drawerState.close() }
                                navController.navigate("settings")
                            }
                        )
                    }
                ) {
                    NavHost(navController = navController, startDestination = "cl
                        composable("chat?chatId={chatId}") {
                            val chatViewModel: ChatViewModel = hiltViewModel()
                            ChatScreen(
                                viewModel = chatViewModel,
                                onOpenDrawer = { scope.launch { drawerState.open
                            )
                        }
                        composable("settings") {
                            SettingsScreen(
                                onNavigateBack = { navController.popBackStack()
                            )
                        }
                    }
                }

                if (showReconnectDialog) {
                    SilentTokenReconnectDialog(
                        onTokenExtracted = { token ->
                            tokenManager.saveToken(token)
                            showReconnectDialog = false
                        },
                        onDismiss = { showReconnectDialog = false }
                    )
                }
            }
        }
    }
}

@SuppressLint("SetJavaScriptEnabled")
@Composable
fun SilentTokenReconnectDialog(
    onTokenExtracted: (String) -> Unit,
    onDismiss: () -> Unit
) {
```

```
    AlertDialog(
        onDismissRequest = onDismiss,
        title = { Text("Session Reconnect") },
        text = {
            AndroidView(
                factory = { ctx ->
                    WebView(ctx).apply {
                        settings.javaScriptEnabled = true
                        settings.domStorageEnabled = true
                        CookieManager.getInstance().setAcceptCookie(true)

                        webViewClient = object : WebViewClient() {
                            override fun onPageFinished(view: WebView?, url: Str
                                super.onPageFinished(view, url)
                                // Silent token extraction from Open WebUI local
                                view?.evaluateJavascript(
                                    "(function() { return localStorage.getItem('
                                ) { rawToken ->
                                    val token = rawToken?.trim('"', ' ')
                                    if (!token.isNullOrBlank() && token != "null
                                        onTokenExtracted(token)
                                    }
                                }
                            }
                        }
                        loadUrl(ZaiConfig.BASE_URL)
                    }
                },
                modifier = Modifier.fillMaxSize()
            )
        },
        confirmButton = {
            TextButton(onClick = onDismiss) {
                Text("Cancel")
            }
        }
    )
}
```

13. DevTools Reconnaissance Checklist

To target your personal `chat.z.ai` web session, extract these 6 values from Chrome/Safari DevTools (Network tab):

#	Parameter	Location in DevTools	Paste Destination in Code
1	**Bearer JWT Token**	`Authorization` header in completion request (`Bearer ey...`) or Console `localStorage.getItem('token')`	Paste directly into initial launch dialog or call `tokenManager.saveToken(...)`

2	**Completions Path**	`Request URL` of the streaming call (e.g., `/api/chat/completions`)	`ZaiConfig.COMPLETIONS_PATH`
3	**Front-End Version**	Custom header (e.g., `X-FE-Version` or `X-App-Version`)	`ZaiConfig.FE_VERSION_HEADER_KEY` & `VALUE`
4	**Origin & Referer**	Header matching reverse-proxy CORS requirement (`https://chat.z.ai`)	`ZaiConfig.ORIGIN` & `ZaiConfig.REFERER`
5	**User-Agent**	Exact Desktop/Mobile UA string sent during session establishment	`ZaiConfig.USER_AGENT`
6	**Default Model String**	Value of `model` in JSON POST payload (e.g., `GLM-4-Plus`)	`ZaiConfig.MODEL_DEFAULT`

14. Performance & Streaming Recomposition Notes

1. **LazyColumn** `reverseLayout = true` **Anchor Mechanics**: Standard lists anchor to index 0 at the top, which triggers full-height layout recalculations and jump-frames whenever new text tokens extend the bottom-most message. Setting `reverseLayout = true` pins index 0 to the bottom screen edge, completely eliminating viewport jump and keyboard-push jank.
2. **Isolating Streaming Invalidation via Transient Composables**: Streaming chunks emit up to 50 tokens per second. If the message list rendered the streaming response by updating the Room database or mutating an item inside the messages list, the entire `LazyColumn` would trigger