📚 AILO Handbuch

Vollständige Dokumentation aller Views, Funktionen und technischen Komponenten

28+	12	8	2
Views	Services	DAOs	Sprachen

📱 Teil 1: Views & Features

1.1 App Entry & Navigation

- AILO_APPApp.swift – App-Einstiegspunkt `App/`
- ContentView.swift – Root-View mit Tab-Navigation
- NativeApp.swift – Platform-spezifische Anpassungen
- Tab-Navigation: Dashboard, Mail, Logs, Settings

1.2 Dashboard

- DashboardView.swift – Hauptübersicht `Views/Dashboard/`
- Anstehende Erinnerungen (Upcoming Reminders)
- Zuletzt hinzugefügte Einträge (Recent Entries)
- App-Banner (Light/Dark Mode)
- Quick Actions zu Write, Speak, Logs

1.3 Logs System

- LogsView.swift – Auswahlhub (Write/Speak/List) `Views/Logs/`
- LogsListView.swift – Liste aller Einträge `Views/LogsList/`
- TextLogDetailView.swift – Detail-Ansicht mit KI-Überarbeitung
- Volltextsuche über alle Logs
- Swipe-Actions: Mail, Play, Delete
- Audio-Player Integration

1.4 Schreiben (Write)

- SchreibenView.swift – Neue Text-Einträge `Views/Schreiben/`
- Titel & Kategoriezuweisung
- Tags-System (mehrere Tags pro Eintrag)
- Erinnerungen mit DatePicker
- E-Mail-Import Dialog
- Clipboard-Integration

1.5 Sprechen (Speak)

- SprechenView.swift – Audio-Aufnahme `Views/Sprechen/`
- Live-Transkription während Aufnahme
- AudioRecorder-Klasse (AVFoundation)
- LiveTranscriber-Klasse (SFSpeechRecognizer)
- Pegel-Anzeige (Level Meter)
- Pause/Resume Funktionalität
- Silence Detection für automatische Chunks

1.6 Mail Feature

- MailInboxView.swift – Posteingang `Views/Mail/`
- MessageDetailView.swift – E-Mail-Ansicht
- ComposeMailView.swift – Neue E-Mail verfassen
- SchreibenMailView.swift – Mail-Import für Logs
- Filter: Alle / Ungelesen / Markiert
- Sortierung: Neueste / Nach Absender
- Aktionen: Reply, Reply All, Forward
- Als Log speichern (Create Log)
- Flag/Unflag, Read/Unread Toggle
- Technische Header anzeigen
- Anhänge-Verwaltung

1.7 Konfiguration (Settings)

- ConfigView.swift – Einstellungen-Hauptseite `Views/Configuration/`
- AIManagerView.swift – KI-Provider verwalten
- AIEditor.swift – Provider bearbeiten
- PrePromptManager.swift – Pre-Prompts verwalten
- PrePromptEditor.swift – Pre-Prompt bearbeiten
- PrePromptCatalogView.swift – Hierarchischer Katalog
- MailManager.swift – E-Mail-Konten verwalten
- MailEditor.swift – Konto-Details bearbeiten
- Categories.swift – Kategorien verwalten

1.8 Pre-Prompt Katalog System

- PrePromptCatalogView.swift – Hierarchische Ordnerstruktur
- PrePromptPicker.swift – Auswahl-Dialog
- PrePromptCatalogPickerSheet.swift – Sheet-Variante
- RecipeEditor – Rezept-Kombinationen
- CookbookView – Kochbuch-Verwaltung
- Export/Import Funktionalität (JSON)
- Standard-Kategorien: Mail, Reply, Forward, Analyze, Notes

1.9 Shared Components

- MailComposer.swift – E-Mail Compose Sheet `Views/Shared/`
- AudioPlayerView.swift – Audio-Wiedergabe
- SelectionCard.swift – Kachel-Komponente
- LabeledSlider.swift – Slider mit Label
- MarkdownHelper.swift – Markdown-Rendering

Teil 2: Services & Business Logic

2.1 KI-Integration

- AIClient.swift – HTTP-Client für KI-APIs `Services/AI/`
- OpenAI-kompatible API (/v1/chat/completions)
- Ollama API (/api/chat, /api/generate)
- Automatischer Fallback zwischen Providern
- Error Handling mit lokalisierten Meldungen

2.2 Mail Services

- MailRepository.swift – Zentrale Schnittstelle `Services/Mail/Sync/`
- MailSyncEngine.swift – 5-Phasen Synchronisation
- MailProcessorAdapter.swift – Engine-Repository Bridge
- ViewportSyncManager.swift – Viewport-basiertes Laden
- MailSendReceive.swift – IMAP/SMTP Transport `Helpers/Utilities/`
- MailHealthMonitor.swift – Verbindungs-Health-Check
- FolderDiscoveryService.swift – Ordner-Erkennung

2.3 IMAP Implementation

- IMAPConnection.swift – Verbindungsmanagement `Services/Mail/IMAP/`
- IMAPCommands.swift – IMAP-Befehle
- IMAPParsers.swift – Response-Parsing
- Unterstützte Befehle: LOGIN, SELECT, FETCH, SEARCH, LIST, STORE
- ENVELOPE, BODYSTRUCTURE, FLAGS Parsing
- TLS/StartTLS Support

2.4 SMTP Implementation

- SMTPAbstractions.swift – Abstrakte Interfaces `Services/Mail/SMTP/`
- SMTPClient.swift – SMTP-Verbindung
- MailSendService.swift – Outbox-Verwaltung
- S/MIME Signing Support

2.5 Pre-Prompt Management

- PrePromptCatalogManager.swift – Singleton Manager `Helpers/Utilities/`
- Hierarchische Menüstruktur (PrePromptMenuItem)
- Preset-Verwaltung (AIPrePromptPreset)
- Rezept-System (PrePromptRecipe)
- Kochbuch-Struktur (Cookbook, RecipeMenuItem)
- Migration von Legacy-Daten
- UserDefaults Persistierung

2.6 Audio & Speech

- AudioRecorder (in SprechenView) – AVFoundation `Views/Sprechen/`
- LiveTranscriber – SFSpeechRecognizer Integration
- Chunk-basierte Transkription
- Silence Detection (Stille-Erkennung)
- On-Device Recognition Support

2.7 Sicherheit

- KeychainService.swift – Sichere Speicherung `Services/`

- API-Keys verschlüsselt
- E-Mail-Passwörter verschlüsselt
- S/MIME Zertifikate

🗄️ Teil 3: Data Access Layer (DAOs)

3.1 Database Schema

- MailSchema.swift – Tabellen-Definitionen `Database/Schema/`
- Tabellen: accounts, folders, msg_header, msg_body, attachments, outbox
- Blob Storage: blob_meta, mime_parts, render_cache
- SQLite als Datenbank-Engine

3.2 DAO Implementations

- BaseDAO.swift – Basis-Klasse für alle DAOs `Database/DAO/`
- AccountDAO.swift – Account-Verwaltung
- FolderDAO.swift – Ordner-Verwaltung
- MailReadDAO.swift – Lese-Operationen
- MailWriteDAO.swift – Schreib-Operationen
- AttachmentDAO.swift – Anhang-Verwaltung
- OutboxDAO.swift – Outbox-Queue
- DAOFactory.swift – Factory Pattern

3.3 DAO Utilities

- DAOHelpers.swift – SQLite Extensions
- DAOPerformanceMonitor – Query-Timing
- DAOTransactionManager – Batch-Operationen
- SQLQueryBuilder – Query-Konstruktion
- DAOSchemaValidator – Schema-Validierung

3.4 Datenmodelle

- LogEntry.swift – Text/Audio Log `Database/Models/`
- AccountEntity – E-Mail-Account
- FolderEntity – Ordner
- MessageHeaderEntity – E-Mail-Header
- MessageBodyEntity – E-Mail-Body
- AttachmentEntity – Anhang
- AIPrePromptPreset – Pre-Prompt Daten
- PrePromptMenuItem – Menü-Struktur
- PrePromptRecipe – Rezept
- Cookbook – Kochbuch

3.5 DataStore (Logs)

- DataStore.swift – ObservableObject `Database/Store/`
- JSON-basierte Persistierung
- CRUD-Operationen für LogEntry
- Audio-URL Management

🔧 Teil 4: Helpers & Parser

4.1 IMAP/MIME Parser

- IMAPParsers.swift – ENVELOPE, FLAGS, LIST Parsing `Services/Mail/IMAP/`
- RFC2047Decoder – Encoded-Word Dekodierung `Helpers/Parsers/`
- RFC2047Test.swift – Parser-Tests
- UTF-8 / ISO-8859-1 Handling
- Quoted-Printable / Base64 Dekodierung

4.2 Utilities

- MailTransportStubs.swift – Transport-Abstraktion `Helpers/Utilities/`
- CancellationToken.swift – Task-Abbruch
- MarkdownHelper.swift – MD-Rendering
- PrePromptCatalogManager.swift – Katalog-Verwaltung
- PrePromptPicker.swift – Picker UI

⚙️ Teil 5: Konfiguration

5.1 Settings Keys

- SettingsKeys.swift – UserDefaults Keys `Configuration/Settings/`
- kAIServerAddress, kAIServerPort, kAIAPIKey, kAIModel
- kAIPresetsKey, kAISelectedPresetKey
- kPrePromptMenuKey, kPrePromptRecipesKey
- kCookbooksKey, kRecipeMenuKey
- kCategories
- kMicSensitivity, kSilenceThreshold, kChunkSeconds
- kSpeechLang

5.2 Lokalisierung

- Localizable.strings (de) `Configuration/Language/de.lproj/`
- Localizable.strings (en) `Configuration/Language/en.lproj/`
- Key-Pattern: feature.view.element[.state|action|hint]
- ~500 lokalisierte Strings pro Sprache

5.3 Mail Account Configuration

- MailAccountConfig – Account-Struktur
- IMAP: Host, Port, Encryption, Username, Password
- SMTP: Host, Port, Encryption, Auth Method
- Sync Limits: Initial, Refresh, Incremental
- Special Folders: Inbox, Sent, Drafts, Trash, Spam
- S/MIME Signing Configuration

🛠️ Teil 6: Technologie-Stack

📱 Plattform
- iOS 16.0+
- macOS 13.0+ (Catalyst)
- Swift 5.9+
- Xcode 15.0+

🎨 UI Framework
- SwiftUI (100%)
- Combine für Reactive
- NavigationStack
- @Observable (iOS 17+)

💾 Datenbank
- SQLite3 (direkt)
- DAO Pattern
- JSON für Logs
- UserDefaults für Settings

🎙️ Audio
- AVFoundation
- AVAudioRecorder
- AVAudioSession
- m4a Format

🗣️ Speech
- Speech Framework
- SFSpeechRecognizer
- On-Device Recognition
- Live Transcription

📧 Netzwerk
- URLSession (HTTP)
- IMAP (Custom Implementation)
- SMTP (Custom Implementation)
- TLS/StartTLS

🤖 KI-Integration
- OpenAI API (v1/chat)
- Ollama API
- Custom Endpoints
- Stream Support (geplant)

🔐 Sicherheit
- Keychain Services
- S/MIME Signing
- Lokale Speicherung
- Keine Cloud-Sync

Teil 7: Projektstruktur

```
AILO_APP/
├── App/                          # App Entry Point
│   ├── AILO_APPApp.swift
│   ├── ContentView.swift
│   └── NativeApp.swift
│
├── Views/                        # UI Layer
│   ├── Dashboard/DashboardView.swift
│   ├── Logs/LogsView.swift
│   ├── LogsList/LogsListView.swift
│   ├── Schreiben/SchreibenView.swift
│   ├── Sprechen/SprechenView.swift
│   ├── Mail/
│   │   ├── MailInboxView.swift
│   │   ├── MessageDetailView.swift
│   │   └── ComposeMailView.swift
│   ├── Configuration/
│   │   ├── ConfigView.swift
│   │   ├── AIManagerView.swift
│   │   ├── PrePromptManager.swift
│   │   └── MailManager.swift
│   └── Shared/
│       ├── MailComposer.swift
│       └── AudioPlayerView.swift
│
├── Services/                     # Business Logic
│   ├── AI/AIClient.swift
│   ├── Mail/
│   │   ├── Sync/MailRepository.swift
│   │   ├── IMAP/IMAPConnection.swift
│   │   └── SMTP/SMTPAbstractions.swift
│   └── KeychainService.swift
│
├── Database/                     # Data Layer
│   ├── Schema/MailSchema.swift
│   ├── DAO/
│   │   ├── BaseDAO.swift
│   │   ├── AccountDAO.swift
│   │   └── DAOFactory.swift
│   ├── Models/LogEntry.swift
│   └── Store/DataStore.swift
│
├── Configuration/                # Settings & i18n
│   ├── Settings/SettingsKeys.swift
│   └── Language/
│       ├── de.lproj/Localizable.strings
│       └── en.lproj/Localizable.strings
│
├── Helpers/                      # Utilities
│   ├── Parsers/RFC2047Decoder.swift
│   └── Utilities/PrePromptCatalogManager.swift
│
└── Resources/                    # Assets
    ├── Assets.xcassets
    └── Info.plist
```

📱 1.1 App Entry & Navigation

App-Einstiegspunkt und Root-Navigation des AILO-Systems

📁 **Pfad:** `AILO_APP/App/`

🔧 **Technologie:** SwiftUI, @main, NavigationStack

Übersicht

Der App Entry Point bildet das Fundament der AILO-Applikation. Er initialisiert alle notwendigen Services, konfiguriert die Hauptnavigation und stellt die grundlegende Tab-Struktur bereit.

Dateien

AILO_APPApp.swift

Der Haupt-Einstiegspunkt der iOS-Applikation mit @main-Attribut.

Funktionen

- App-Lifecycle Management via @main
- WindowGroup für Multi-Window Support
- Environment Objects Initialisierung
- DataStore Injection für Logs-System
- App-weite State-Verwaltung

ContentView.swift

Die Root-View mit der zentralen Tab-Navigation der Applikation.

Tab-Struktur

Tab	Icon	Beschreibung
Dashboard	`house.fill`	Hauptübersicht mit Quick Actions
Mail	`envelope.fill`	E-Mail-Posteingang & Verwaltung
Logs	`doc.text.fill`	Text- & Audio-Einträge (Write/Speak)
Settings	`gearshape.fill`	Konfiguration & Einstellungen

NativeApp.swift

Platform-spezifische Anpassungen für iOS und macOS (Catalyst).

Funktionen

- Plattform-Erkennung (iOS vs. macOS)
- Screen-Size Anpassungen

- Keyboard-Handling Unterschiede
- Platform-spezifische UI-Elemente

> ℹ️ **Technischer Hinweis**
> Die Tab-Navigation nutzt `NavigationStack` (iOS 16+) für moderne Navigation mit programmatischer Steuerung und Deep-Link Support.

Abhängigkeiten

Der App Entry Point initialisiert folgende zentrale Komponenten:

Komponente	Funktion
`DataStore`	Logs-Verwaltung als EnvironmentObject
`MailRepository`	Mail-Synchronisation & IMAP/SMTP
`AIClient`	KI-Provider Integration (OpenAI/Ollama)
`KeychainService`	Sichere Speicherung von Credentials

🏠 1.2 Dashboard

Zentrale Übersichtsseite mit Quick Actions und aktuellen Informationen

📁 **Pfad:** `AILO_APP/Views/Dashboard/`

📄 **Datei:** `DashboardView.swift`

Übersicht

Das Dashboard ist die zentrale Anlaufstelle der AILO-App nach dem Start. Es bietet einen schnellen Überblick über anstehende Aufgaben, kürzlich erstellte Einträge und ermöglicht den direkten Zugriff auf die Hauptfunktionen über Quick Actions.

UI-Komponenten

App-Banner

Prominentes Branding-Element im oberen Bereich des Dashboards.

- AILO Logo mit Gradient-Effekt
- Automatische Anpassung an Light/Dark Mode
- Responsive Größenanpassung

Quick Actions

Schnellzugriff-Kacheln für die wichtigsten Funktionen der App.

Aktion	Icon	Ziel
Write	✏️ `pencil`	SchreibenView → Neuer Text-Eintrag
Speak	🎙️ `mic.fill`	SprechenView → Audio-Aufnahme
Logs	📋 `list.bullet`	LogsListView → Alle Einträge

Anstehende Erinnerungen

Zeigt Einträge mit gesetzten Erinnerungen, die in naher Zukunft fällig sind.

Funktionen

- **Filterung** — Nur Einträge mit reminderDate in der Zukunft
- **Sortierung** — Chronologisch nach Fälligkeit
- **Limit** — Maximal 5 Einträge angezeigt
- **Navigation** — Tap öffnet TextLogDetailView
- **Leer-State** — Hinweis wenn keine Erinnerungen vorhanden

Zuletzt hinzugefügte Einträge

Kompakte Übersicht der neuesten Log-Einträge für schnellen Zugriff.

Anzeige-Elemente

- **Titel** — Haupttext des Eintrags
- **Kategorie-Badge** — Farbcodierte Kategorie-Anzeige
- **Datum** — Erstellungsdatum formatiert
- **Typ-Icon** — Text (📝) oder Audio (🎙️)
- **Limit** — Maximal 10 neueste Einträge

Technische Details

State Management

Property	Beschreibung
`@EnvironmentObject`	DataStore für Zugriff auf alle Log-Einträge
`@State`	Lokaler UI-State für Navigation und Selektion
`Computed Properties`	Gefilterte Listen für Reminders & Recent Entries

> 💡 **Dark Mode Support**
>
> Das Dashboard passt sich automatisch an das System-Erscheinungsbild an. Das App-Banner verwendet `@Environment(\.colorScheme)` für dynamische Farbanpassungen.

Layout-Struktur

Das Dashboard verwendet eine ScrollView mit vertikalem VStack für optimale Scrollbarkeit auf allen Gerätegrößen.

Bereich	Komponente
Header	App-Banner mit Logo
Quick Actions	HStack mit SelectionCard-Komponenten
Reminders Section	Section mit ForEach-Liste
Recent Entries Section	Section mit NavigationLink-Liste

1.3 Logs System

Verwaltung und Anzeige von Text- und Audio-Einträgen

Pfade: `Views/Logs/`, `Views/LogsList/`

Dateien: `LogsView.swift`, `LogsListView.swift`, `TextLogDetailView.swift`

Übersicht

Das Logs System ist das Herzstück der AILO-App für die Erfassung und Verwaltung von Notizen. Es unterstützt sowohl Text-Einträge als auch Audio-Aufnahmen mit Live-Transkription und bietet umfangreiche Such- und Bearbeitungsfunktionen.

Komponenten

LogsView.swift

Der zentrale Auswahlhub für alle Log-bezogenen Aktionen.

Navigationsoptionen

Option	Icon	Beschreibung
Write	`pencil`	Neuen Text-Eintrag erstellen
Speak	`mic.fill`	Audio-Aufnahme mit Transkription
List	`list.bullet`	Alle Einträge anzeigen und durchsuchen

LogsListView.swift

Übersichtliche Listendarstellung aller gespeicherten Einträge mit Such- und Filterfunktionen.

Volltextsuche

- Durchsucht Titel, Inhalt und Tags
- Case-insensitive Suche
- Echtzeit-Filterung während der Eingabe
- Highlight von Suchtreffern (geplant)

Swipe-Actions

Aktion	Farbe	Funktion
Mail	Blau	Eintrag per E-Mail versenden
Play	Grün	Audio-Aufnahme abspielen (nur Audio-Logs)
Delete	Rot	Eintrag löschen (mit Bestätigung)

TextLogDetailView.swift

Detaillierte Ansicht eines einzelnen Eintrags mit Bearbeitungs- und KI-Funktionen.

Anzeige-Elemente

- **Titel** — Bearbeitbar mit Inline-Editor
- **Inhalt** — Vollständiger Text mit Markdown-Rendering
- **Kategorie** — Badge mit Farbcodierung
- **Tags** — Chip-Liste aller zugewiesenen Tags
- **Erinnerung** — Datum und Uhrzeit falls gesetzt
- **Erstellungsdatum** — Formatierte Zeitangabe

KI-Überarbeitung

> **KI-Integration**
> Die KI-Überarbeitung nutzt den konfigurierten AI-Provider (OpenAI/Ollama) mit Pre-Prompts aus dem Katalog. Der Benutzer kann das Ergebnis übernehmen, verwerfen oder erneut generieren.

- **Pre-Prompt Auswahl** — Aus hierarchischem Katalog
- **Verarbeitung** — Asynchron mit Ladeanzeige
- **Ergebnis-Vorschau** — Diff-Ansicht Original vs. Überarbeitet
- **Aktionen** — Übernehmen, Verwerfen, Neu generieren

Audio-Player Integration

Für Audio-Logs steht ein integrierter Player mit erweiterten Funktionen zur Verfügung.

Feature	Beschreibung
Play/Pause	Wiedergabe starten und pausieren
Fortschrittsbalken	Visuelle Anzeige und Seek-Funktion
Zeitanzeige	Aktuelle Position / Gesamtdauer
Skip-Buttons	±15 Sekunden vor/zurück springen
Geschwindigkeit	0.5x, 1x, 1.5x, 2x Wiedergabe

> **Technischer Hinweis**
> Audio-Dateien werden im `m4a-Format` gespeichert und über `AVAudioPlayer` wiedergegeben. Die URLs werden im DataStore referenziert.

Datenmodell: LogEntry

Zentrale Datenstruktur für alle Log-Einträge im System.

Property	Typ	Beschreibung
`id`	`UUID`	Eindeutige Identifikation
`title`	`String`	Titel des Eintrags
`content`	`String`	Hauptinhalt / Transkription
`category`	`String?`	Optionale Kategorie-Zuweisung
`tags`	`[String]`	Liste von Tags
`reminderDate`	`Date?`	Optionale Erinnerung
`audioURL`	`URL?`	Pfad zur Audio-Datei (nur Audio-Logs)
`createdAt`	`Date`	Erstellungszeitpunkt

✏️ 1.4 Schreiben (Write)

Erstellen neuer Text-Einträge mit Kategorien, Tags und Erinnerungen

📁 **Pfad:** `AILO_APP/Views/Schreiben/`

📄 **Datei:** `SchreibenView.swift`

Übersicht

Die SchreibenView ermöglicht das Erstellen neuer Text-Einträge in der AILO-App. Sie bietet ein umfassendes Formular mit Titel, Inhalt, Kategoriezuweisung, Tags-System, Erinnerungsfunktion sowie Import-Möglichkeiten aus E-Mail und Zwischenablage.

Formularfelder

Titel

Einzeiliges Textfeld für die Überschrift des Eintrags.

- **Pflichtfeld** — Muss ausgefüllt werden
- **Placeholder** — "Titel eingeben..."
- **Validierung** — Mindestens 1 Zeichen
- **Keyboard** — Default mit Auto-Capitalization

Inhalt

Mehrzeiliges Textfeld für den Hauptinhalt des Eintrags.

- **TextEditor** — SwiftUI-native Komponente
- **Unbegrenzte Länge** — Scrollbar bei Überlauf
- **Markdown-Support** — Wird in Detail-Ansicht gerendert
- **Minimale Höhe** — 200pt für bessere Bedienbarkeit

Kategoriezuweisung

Dropdown-Auswahl aus den benutzerdefinierten Kategorien.

Eigenschaft	Beschreibung
Komponente	Picker mit .menu Style
Datenquelle	UserDefaults (kCategories)
Standard	"Allgemein" als Fallback
Farbcodierung	Jede Kategorie hat zugewiesene Farbe

Tags-System

Flexibles Tagging-System für erweiterte Kategorisierung und Suche.

Funktionen

- **Mehrfach-Tags** — Beliebig viele Tags pro Eintrag
- **Eingabe** — TextField mit Return-Bestätigung
- **Chip-Darstellung** — Tags als löschbare Chips
- **Duplikat-Prüfung** — Keine doppelten Tags erlaubt
- **Vorschläge** — Basierend auf existierenden Tags (geplant)

> 💡 **UI-Verhalten**
> Tags werden als horizontale FlowLayout-Chips angezeigt. Jeder Chip hat ein ×-Symbol zum Entfernen. Das Eingabefeld erscheint am Ende der Chip-Reihe.

Erinnerungen

Optionale Erinnerungsfunktion mit Datum und Uhrzeit.

Element	Beschreibung
Toggle	Aktiviert/Deaktiviert die Erinnerung
DatePicker	Auswahl von Datum und Uhrzeit
Minimum	Aktuelles Datum (keine Vergangenheit)
Format	Lokalisiert (DE/EN)

Import-Funktionen

E-Mail-Import Dialog

Importiert Inhalte aus E-Mails als neuen Log-Eintrag.

- **Auslöser** — Toolbar-Button mit envelope.open Icon
- **Sheet-Präsentation** — Modal über SchreibenMailView
- **Auswahl** — E-Mail aus Posteingang wählen
- **Import-Optionen** — Betreff als Titel, Body als Inhalt
- **Anhänge** — Optional mit importieren (geplant)

Clipboard-Integration

Schnelles Einfügen von Inhalten aus der Zwischenablage.

- **Button** — Toolbar mit doc.on.clipboard Icon
- **Verhalten** — Fügt am Cursor oder ans Ende ein
- **Unterstützte Formate** — Plain Text, Rich Text
- **Feedback** — Kurze Vibration bei Erfolg

> ℹ️ **Technischer Hinweis**
>
> Der Clipboard-Zugriff erfolgt über `UIPasteboard.general`. Ab iOS 16 wird der Benutzer bei erstem Zugriff zur Bestätigung aufgefordert.

State Management

Property	Typ	Beschreibung
`@State title`	`String`	Titel des Eintrags
`@State content`	`String`	Hauptinhalt des Eintrags
`@State category`	`String`	Ausgewählte Kategorie
`@State tags`	`[String]`	Liste der Tags
`@State hasReminder`	`Bool`	Erinnerung aktiviert?
`@State reminderDate`	`Date`	Gewähltes Erinnerungsdatum
`@State showMailImport`	`Bool`	Mail-Import Sheet anzeigen

Speichern-Aktion

Der Speichervorgang erstellt einen neuen LogEntry und persistiert ihn über den DataStore.

Ablauf

- **1. Validierung** — Titel muss ausgefüllt sein
- **2. LogEntry erstellen** — Mit UUID, Zeitstempel und allen Feldern
- **3. DataStore.add()** — Persistierung in JSON-Datei
- **4. Navigation zurück** — dismiss() nach erfolgreichem Speichern
- **5. Feedback** — Optionale Erfolgsbestätigung

🎙️ 1.5 Sprechen (Speak)

Audio-Aufnahme mit Live-Transkription und intelligenter Stille-Erkennung

📁 **Pfad:** `AILO_APP/Views/Sprechen/`

📄 **Datei:** `SprechenView.swift`

🔧 **Frameworks:** `AVFoundation, Speech Framework`

Übersicht

Die SprechenView ermöglicht Audio-Aufnahmen mit simultanem Live-Transkription. Sie kombiniert AVFoundation für die Aufnahme mit dem Speech Framework für Echtzeit-Spracherkennung und bietet erweiterte Features wie Pegel-Anzeige, Pause/Resume und automatische Chunk-Erkennung durch Stille-Detection.

Kernklassen

AudioRecorder

Zentrale Klasse für die Audio-Aufnahme basierend auf AVFoundation.

Property	Typ	Beschreibung
`audioRecorder`	`AVAudioRecorder?`	Native Audio-Recorder Instanz
`isRecording`	`Bool`	Aufnahme-Status Flag
`isPaused`	`Bool`	Pause-Status Flag
`audioLevel`	`Float`	Aktueller Pegel (0.0 - 1.0)
`recordingURL`	`URL?`	Pfad zur Audio-Datei

Audio-Einstellungen

Einstellung	Wert
Format	`m4a (AAC)`
Sample Rate	`44100 Hz`
Channels	`1 (Mono)`
Encoder Quality	`.high`

LiveTranscriber

Echtzeit-Spracherkennung mit dem Speech Framework für Live-Transkription.

Kernkomponenten

- **SFSpeechRecognizer** — Apple Speech Recognition Engine
- **SFSpeechAudioBufferRecognitionRequest** — Audio-Buffer für Streaming
- **SFSpeechRecognitionTask** — Aktiver Recognition-Task
- **AVAudioEngine** — Audio-Pipeline für Mic-Input

> ⚠️ **Wichtig: Berechtigungen**
>
> Die App benötigt `NSSpeechRecognitionUsageDescription` und `NSMicrophoneUsageDescription` in Info.plist. Der User muss beide Berechtigungen erteilen.

Sprachunterstützung

Sprache	Locale	On-Device
Deutsch	`de-DE`	✓
Englisch (US)	`en-US`	✓
Englisch (UK)	`en-GB`	✓

UI-Komponenten

Pegel-Anzeige (Level Meter)

Visuelle Darstellung des aktuellen Audio-Pegels während der Aufnahme.

- **Animierte Balken** — Echtzeit-Visualisierung der Lautstärke
- **Farbverlauf** — Grün → Gelb → Rot bei steigendem Pegel
- **Update-Rate** — 60 FPS via CADisplayLink
- **Dezibel-Normalisierung** — Umrechnung von dB zu 0.0-1.0

Aufnahme-Steuerung

Button	Icon	Aktion
Start	`circle.fill`	Aufnahme und Transkription starten
Pause	`pause.fill`	Aufnahme pausieren (Resume möglich)
Stop	`stop.fill`	Aufnahme beenden und speichern
Cancel	`xmark`	Aufnahme verwerfen

Silence Detection

Intelligente Erkennung von Sprechpausen für automatische Chunk-Segmentierung.

🤖 Automatische Chunks

Bei erkannter Stille wird die bisherige Transkription als Chunk finalisiert. Dies ermöglicht natürliche Absätze ohne manuelle Eingabe und verbessert die KI-Verarbeitung durch sinnvolle Segmentierung.

Konfigurierbare Parameter

Parameter	Default	Settings Key
Mikrofon-Empfindlichkeit	`0.5`	`kMicSensitivity`
Stille-Schwellwert	`-40 dB`	`kSilenceThreshold`
Chunk-Dauer	`2.0 Sek`	`kChunkSeconds`

ℹ️ On-Device Recognition

AILO nutzt `requiresOnDeviceRecognition = true` für vollständig offline Transkription. Dies gewährleistet Datenschutz und funktioniert ohne Internetverbindung.

1.6 Mail Feature

Vollständige E-Mail-Verwaltung mit IMAP/SMTP-Integration

Übersicht

Das Mail Feature ermöglicht die vollständige Verwaltung von E-Mails innerhalb der AILO App. Es bietet nahtlose Integration mit bestehenden E-Mail-Konten über IMAP und SMTP, intelligente Filterung und Sortierung sowie die Möglichkeit, E-Mails als Logs zu speichern.

Eigenschaft	Beschreibung
Verzeichnis	`Views/Mail/`
Haupt-Views	MailView, MessageDetailView, ComposeMailView
Protokolle	IMAP (Empfang), SMTP (Versand), TLS/StartTLS
Datenbank	SQLite (accounts, folders, msg_header, msg_body, attachments)

Komponenten

MailView.swift – Posteingang

Die zentrale View für die E-Mail-Übersicht mit adaptivem Layout für Compact und Regular Size Classes.

- Mailbox-Navigation: Inbox, Outbox, Sent, Drafts, Trash, Spam
- Schnellfilter: Alle / Ungelesen / Markiert (Segmented Control)
- Suchfunktion mit Echtzeit-Filterung
- Badge-Anzeige für ungelesene Nachrichten
- Pull-to-Refresh für Synchronisation
- Viewport-basiertes Laden (ViewportSyncManager)

Size Class	Layout
Compact (iPhone)	CompactMessageListView mit NavigationLink zur Detail-Ansicht
Regular (iPad)	RegularSplitView mit Master-Detail-Layout

MessageDetailView.swift – E-Mail-Ansicht

Detaillierte Ansicht einer einzelnen E-Mail mit allen Aktionen und Anhang-Verwaltung.

Header-Informationen

- Betreff, Absender (From), Empfänger (To, CC, BCC)
- Datum und Uhrzeit des Empfangs
- Technische Header anzeigen/verbergen (Source View)

Aktionen

Aktion	Icon	Beschreibung
Reply	`arrowshape.turn.up.left`	Antwort an Absender
Reply All	`arrowshape.turn.up.left.2`	Antwort an alle Empfänger
Forward	`arrowshape.turn.up.right`	E-Mail weiterleiten (mit Anhängen)
Create Log	`doc.badge.plus`	E-Mail als Log-Eintrag speichern
Flag/Unflag	`flag / flag.fill`	Markierung setzen/entfernen
Read/Unread	`envelope / envelope.open`	Gelesen/Ungelesen Status wechseln
Delete	`trash`	E-Mail in Papierkorb verschieben

Anhänge-Verwaltung

- Automatische Erkennung von Anhängen (BODYSTRUCTURE)
- QuickLook-Vorschau für unterstützte Dateitypen
- Inline-Bilder (CID-Referenzen) werden als Base64 Data-URLs eingebettet
- Alle Anhänge speichern mit Share-Sheet

ComposeMailView.swift – E-Mail verfassen

Formular zum Erstellen neuer E-Mails sowie für Antworten und Weiterleitungen.

Eingabefelder

- Von (From) – Absender-Account Auswahl
- An (To) – Empfänger-Adressen
- CC, BCC – Kopie-Empfänger (optional)
- Betreff (Subject)
- Nachrichtentext (Body) – Text oder HTML

Anhänge hinzufügen

- Fotos aus Bibliothek (PhotosPicker)
- Dateien aus Files-App (DocumentPicker)
- Anhänge aus Original-Mail bei Forward

KI-Integration

- Pre-Prompt Auswahl aus Katalog
- Automatische E-Mail-Generierung basierend auf Kontext
- Format-aware: HTML für HTML-Antworten, Text für Text

Technische Details

Datenmodelle

Entity	Verwendung
`MessageHeaderEntity`	E-Mail-Metadaten (Subject, From, To, Date, Flags, UID)
`MessageBodyEntity`	E-Mail-Inhalt (HTML/Text Body, Content-Type)
`AttachmentEntity`	Anhang-Metadaten (Filename, MIME-Type, Size, Content-ID)
`FolderEntity`	IMAP-Ordner (Name, Path, Special Folder Type)

Lokalisierung

Alle UI-Texte sind vollständig lokalisiert (Deutsch/Englisch). Die Lokalisierungsschlüssel folgen dem Pattern: `app.mail.[view].[element]`

Schlüssel	Deutsch
`app.mail.inbox`	Posteingang
`app.mail.compose.title`	Verfassen
`app.mail.action.reply`	Antworten
`app.mail.action.forward`	Weiterleiten

Abhängigkeiten

- `MailRepository` – Zentrale Schnittstelle für Mail-Operationen
- `MailSyncEngine` – 5-Phasen Synchronisation
- `IMAPConnection` – IMAP-Protokoll-Handler
- `MailDAOs` – MailReadDAO, MailWriteDAO, AttachmentDAO
- `ViewportSyncManager` – On-Demand-Laden sichtbarer E-Mails

1.7 Konfiguration (Settings)

Zentrale Einstellungsverwaltung für KI-Provider, E-Mail-Konten und Pre-Prompts

Übersicht

Die Konfigurationsseite bietet zentrale Verwaltung aller App-Einstellungen. Von hier aus werden KI-Provider, E-Mail-Konten, Pre-Prompts, Kategorien und Aufnahme-Parameter konfiguriert. Alle Einstellungen werden persistent in UserDefaults gespeichert.

Eigenschaft	Beschreibung
Verzeichnis	`Views/Configuration/`
Haupt-View	ConfigView.swift
Sub-Views	AIManagerView, PrePromptManager, MailManager, Categories
Persistierung	UserDefaults + Keychain (für Passwörter/API-Keys)

Komponenten

ConfigView.swift – Hauptseite

Die zentrale Einstellungsseite mit Sektionen für alle Konfigurationsbereiche.

Sektion	Inhalt
Kategorien	NavigationLink → Categories.swift
Mikrofon / Aufnahme	Empfindlichkeit, Stille-Schwelle, Segmentlänge (Slider)
Sprache	Spracherkennungs-Locale (de-DE, en-US, etc.)
E-Mail	NavigationLink → MailManager.swift
KI	NavigationLinks → AIManagerView, PrePromptManager

AIManagerView.swift – KI-Provider

Verwaltung der KI-Provider (OpenAI, Ollama, Custom). Unterstützt mehrere Provider mit Fallback-Mechanismus.

Funktionen

- Provider hinzufügen / bearbeiten / löschen
- Aktiven Provider setzen (Standard)
- Modell-Liste vom Server abrufen
- Temperatur-Einstellung pro Provider

Provider-Typ	API-Endpunkt	Authentifizierung
OpenAI	`/v1/chat/completions`	Bearer Token (API-Key)
Ollama	`/api/chat, /api/generate`	Keine (lokal)
Custom	`Benutzerdefiniert`	Optional (API-Key)

AIEditor.swift – Provider bearbeiten

- Anzeigename für Provider
- Typ-Auswahl (OpenAI / Ollama / Custom)
- Serveradresse und Port
- API-Schlüssel (im Keychain gespeichert)
- Modell-Auswahl mit Reload-Funktion
- Temperatur (0.0 – 2.0)

PrePromptManager.swift – Pre-Prompts

Verwaltung von Pre-Prompt-Vorlagen, die dem KI-Modell als System-Prompt vorangestellt werden.

- Pre-Prompts erstellen / bearbeiten / löschen
- Name, Icon, Schlagwörter (Keywords) pro Preset
- Standard-Preset festlegen
- NavigationLink → PrePromptCatalogView für hierarchische Struktur

PrePromptEditor.swift – Pre-Prompt bearbeiten

Feld	Beschreibung
Name	Kurzes Label (z.B. Korrektur, Protokoll)
Keywords	Metadaten (Format: Schlüssel: Wert; getrennt durch ;)
Inhalt	Der eigentliche Pre-Prompt-Text für das KI-Modell
Icon	Emoji zur visuellen Kennzeichnung (max. 3 Zeichen)

MailManager.swift – E-Mail-Konten

Verwaltung der E-Mail-Konten mit IMAP/SMTP-Konfiguration.

- E-Mail-Konten hinzufügen / bearbeiten / löschen
- Aktiv/Inaktiv Status pro Konto
- Verbindungstest-Funktion
- NavigationLink → MailEditor für Details

MailEditor.swift – Konto-Details

Sektion	Felder
Konto	Kontoname, Anzeigename, E-Mail-Adresse, Reply-To
Eingehend (IMAP)	Protokoll, Server, Port, Verschlüsselung, Username, Passwort
Ausgehend (SMTP)	Server, Port, Verschlüsselung, Auth-Methode, Username, Passwort
Synchronisation	Initial Sync, Full Sync, Incremental Sync (Anzahl Mails)
Ordner	Inbox, Sent, Drafts, Trash, Spam (Ordner abrufen)
Erweitert	Timeout, Logging, Auto-Mark-as-Read, S/MIME Signing

Categories.swift – Kategorien

Verwaltung der Log-Kategorien zur Organisation von Einträgen.

- Kategorien hinzufügen / bearbeiten / löschen
- Sortierung per Drag & Drop
- Verwendung in SchreibenView für Log-Zuordnung

Settings Keys (UserDefaults)

Alle Einstellungen werden über definierte Keys in UserDefaults persistiert. Datei: `Configuration/Settings/SettingsKeys.swift`

Konstante	Beschreibung
`kAIPresetsKey`	JSON [AIPrePromptPreset]
`kPrePromptMenuKey`	JSON [PrePromptMenuItem]
`kCookbooksKey`	JSON [Cookbook]
`kCategories`	JSON [String] Kategorien-Liste
`kMicSensitivity`	Double 0...1 (Empfindlichkeit)
`kSilenceThreshold`	Double -60...0 dB (Stille-Schwelle)
`kSpeechLang`	String z.B. "de-DE"

Lokalisierung

Alle UI-Texte sind vollständig lokalisiert. Key-Pattern für Configuration: `config.[section].[element]` sowie `aiEditor.*`, `preprompts.*`, `mail.editor.*`

1.8 Pre-Prompt Katalog System

Dokumentation des hierarchischen Katalog-Systems für Pre-Prompts, Rezepte und Kochbücher in AILO.

Übersicht

Das Pre-Prompt Katalog System ermöglicht die hierarchische Organisation von KI-Anweisungen (Pre-Prompts) in einer Ordnerstruktur. Nutzer können einzelne Pre-Prompts erstellen, in Kategorien organisieren und zu komplexen Rezepten kombinieren. Diese Rezepte werden in Kochbüchern verwaltet.

Hauptkomponenten

Datei	Beschreibung
PrePromptCatalogView.swift	Hierarchische Ordnerstruktur mit Drag & Drop, Kontextmenüs und Navigation
PrePromptPicker.swift	Auswahl-Dialog für Pre-Prompt Selektion in anderen Views
PrePromptCatalogPickerSheet.swift	Sheet-basierte Picker-Variante für modale Darstellung
RecipeEditor	Editor für Rezept-Kombinationen aus mehreren Pre-Prompts
CookbookView.swift	Kochbuch-Verwaltung für Rezept-Organisation
PrePromptCatalogManager.swift	Singleton-Manager für Katalog-Daten und Business-Logic

Datenmodelle

PrePromptMenuItem

Repräsentiert einen Eintrag in der hierarchischen Menüstruktur. Kann entweder ein Ordner oder ein Verweis auf ein Preset sein.

- **id:** UUID – Eindeutige Kennung
- **name:** String – Anzeigename
- **icon:** String – Emoji-Symbol
- **parentID:** UUID? – Eltern-Ordner (nil = Root)
- **isFolder:** Bool – Ordner oder Item
- **presetID:** UUID? – Verweis auf AIPrePromptPreset
- **keywords:** String – Schlagwörter für Kontext

AIPrePromptPreset

Enthält den eigentlichen Pre-Prompt-Text und Metadaten.

- **id:** UUID – Eindeutige Kennung
- **name:** String – Preset-Name
- **icon:** String – Emoji-Symbol
- **text:** String – Der Pre-Prompt-Text
- **keywords:** String – Schlüssel-Wert-Paare

PrePromptRecipe

Kombiniert mehrere Pre-Prompts zu einem zusammengesetzten Prompt.

- **id:** UUID – Eindeutige Kennung
- **name:** String – Rezept-Name
- **icon:** String – Emoji-Symbol
- **elementIDs:** [UUID] – Referenzierte Menu-Items
- **keywords:** String – Zusätzliche Schlagwörter

Cookbook

Container für die Organisation von Rezepten.

- **id:** UUID – Eindeutige Kennung
- **name:** String – Kochbuch-Name
- **icon:** String – Emoji-Symbol
- **sortOrder:** Int – Sortierreihenfolge

Funktionen

Hierarchische Navigation

Die PrePromptCatalogView ermöglicht das Navigieren durch eine beliebig tiefe Ordnerstruktur. Breadcrumb-Navigation zeigt den aktuellen Pfad an. Ordner und Items können per Drag & Drop verschoben werden.

Rezept-Erstellung

Im RecipeEditor können Nutzer mehrere Pre-Prompts zu einem Rezept kombinieren. Die Reihenfolge der Elemente bestimmt die Zusammensetzung des generierten Prompts. Eine Live-Vorschau zeigt das Ergebnis an.

Keyword-System

Schlagwörter werden als Key-Value-Paare gespeichert (Format: 'Schlüssel: Wert; Schlüssel2: Wert2'). Bei der Prompt-Generierung werden alle Keywords aus Menu-Items, Presets und Rezepten zusammengeführt. Spätere Definitionen überschreiben frühere.

Export/Import

Der gesamte Katalog kann als JSON-Datei exportiert und auf anderen Geräten importiert werden. Das Format umfasst Menu-Items, Presets, Recipes, Cookbooks und Recipe-Menu-Items.

Standard-Kategorien

Bei der ersten Initialisierung werden folgende Kategorien automatisch erstellt:

Icon	Kategorie	Verwendung
	**Mail**	Pre-Prompts für E-Mail-Erstellung
	**Reply**	Pre-Prompts für Antwort-Mails
	**Forward**	Pre-Prompts für Weiterleitungen
	**Analyze**	Pre-Prompts für Inhaltsanalyse
	**Notes**	Pre-Prompts für Notizen und Logs

PrePromptCatalogManager API

Der Singleton-Manager stellt folgende öffentliche Methoden bereit:

Methode	Beschreibung
children(of:)	Gibt alle Kinder eines Eltern-Elements zurück
path(to:)	Breadcrumb-Pfad zu einem Element
preset(withID:)	Preset anhand der UUID abrufen
presets(in:)	Alle Presets in einem Ordner (rekursiv)
recipe(withID:)	Rezept anhand der UUID abrufen
recipes(inCookbook:)	Alle Rezepte in einem Kochbuch
generatePrompt(from:)	Generierten Prompt aus Rezept erstellen

Persistierung

Alle Katalog-Daten werden über UserDefaults persistiert. Die verwendeten Schlüssel sind in SettingsKeys.swift definiert:

- **kPrePromptMenuKey** – Menu-Items (JSON)
- **kAIPresetsKey** – Pre-Prompts (JSON)
- **kPrePromptRecipesKey** – Rezepte (JSON)
- **kCookbooksKey** – Kochbücher (JSON)
- **kRecipeMenuKey** – Recipe-Menu-Items (JSON)

Lokalisierung

Das Katalog-System ist vollständig in Deutsch und Englisch lokalisiert. Die Lokalisierungs-Keys folgen dem Muster 'catalog.*' und 'cookbook.*'. Alle UI-Texte, Fehlermeldungen und Standardkategorien sind übersetzt.

Verzeichnisstruktur

Views/Configuration/

- PrePromptCatalogView.swift
- CookbookView.swift

Helpers/Utilities/

- PrePromptCatalogManager.swift
- PrePromptPicker.swift

Database/Models/

- PrePromptMenuItem.swift
- PrePromptRecipe.swift
- Cookbook.swift
- RecipeMenuItem.swift

1.9 Shared Components

Wiederverwendbare UI-Komponenten und Hilfsklassen

Uebersicht

Die Shared Components sind wiederverwendbare UI-Bausteine und Hilfsklassen, die in verschiedenen Views der AILO App eingesetzt werden. Sie sorgen fuer konsistentes Design und reduzieren Code-Duplizierung.

Eigenschaft	Beschreibung
Verzeichnisse	`Views/Shared/, Helpers/UI/, Services/Audio/`
Komponenten	5 UI-Komponenten + 1 Helper-Klasse
Frameworks	SwiftUI, MessageUI, AVFoundation

Komponenten

MailComposer.swift

UIViewControllerRepresentable-Wrapper fuer MFMailComposeViewController. Ermoeglicht das Versenden von E-Mails ueber die native iOS Mail-App.

Eigenschaften

Property	Beschreibung
`subject: String`	E-Mail-Betreff
`body: String`	Nachrichtentext (Plain Text)
`recipients: [String]`	Empfaenger-Adressen (optional)
`attachments: [Attachment]`	Anhaenge mit Data, MIME-Type, Filename

Verwendung

- LogsListView: Logs per E-Mail teilen
- Audio-Logs mit m4a-Anhang versenden
- MFMailComposeViewControllerDelegate fuer Result-Handling

AudioPlayerView.swift

Kompakte Audio-Wiedergabe-Komponente als Sheet. Nutzt AudioPlaybackController fuer AVAudioPlayer-Integration.

Funktionen

- Play / Pause mit grossem zentralen Button
- Seek per Slider (Fortschrittsanzeige)
- Skip -10s / +10s Buttons
- Zeitanzeige: aktuelle Position / Gesamtdauer
- Automatisches Abspielen bei Oeffnen
- Presentation Detents: 25% und 50% Bildschirmhoehe

AudioPlaybackController

Methode / Property	Beschreibung
`load(url:)`	Laedt Audio-Datei, setzt AVAudioSession
`play() / pause()`	Wiedergabe starten / pausieren
`seekBy(seconds:)`	Relative Positionsaenderung (+/- Sekunden)
`seek(to:)`	Absolute Position (0.0 - 1.0)
`isPlaying: Bool`	@Published Wiedergabe-Status
`progress: Double`	@Published Fortschritt (0.0 - 1.0)

SelectionCard

Grosse Kachel-Komponente fuer Auswahlmenues. Verwendet in LogsView fuer die Hauptnavigation (Write, Speak, List).

Parameter

Parameter	Beschreibung
`icon: String`	SF Symbol Name (z.B. "square.and.pencil")
`title: String`	Beschriftung der Kachel
`color: Color`	Akzentfarbe fuer Icon und Hintergrund

Design

- HStack mit Icon (36pt) und Titel
- Icon in farbigem Container (60x60, cornerRadius 12)
- Chevron-Indikator rechts
- Schatten und abgerundete Ecken (cornerRadius 16)

LabeledSlider

Slider mit integriertem Label und Wertanzeige. Verwendet in ConfigView fuer Mikrofon-Einstellungen.

Parameter

Parameter	Beschreibung
`title: String`	Label-Text links vom Slider
`value: Binding<Double>`	Gebundener Wert
`range: ClosedRange`	Wertebereich (z.B. 0...1, -60...0)
`step: Double`	Schrittweite
`display: String`	Formatierte Wertanzeige (z.B. "50 %", "-30 dB")

Verwendung in ConfigView

- Mikrofon-Empfindlichkeit (0-100%)
- Stille-Schwelle (-60 bis 0 dB)
- Segmentlaenge (1-10 Sekunden)

MarkdownHelper.swift

Hilfsklasse fuer Markdown-Formatierung. Bietet statische Methoden zum Einfuegen von Markdown-Syntax.

Methoden

Methode	Funktion
`insertAtLineStart(_:in:)`	Praefix am Zeilenanfang (# , - , etc.)
`wrapSelectionBold(_:)`	Text mit **fett** umschliessen
`wrapSelectionItalic(_:)`	Text mit *kursiv* umschliessen

Lokalisierung

Die Shared Components verwenden lokalisierte Strings aus den Localizable.strings-Dateien. Relevante Key-Patterns:

Key	Deutsch
`mail.result.sent`	E-Mail gesendet
`mail.result.failed`	E-Mail fehlgeschlagen
`markdown.preview.title`	Vorschau
`markdown.toast.copied`	In Zwischenablage kopiert

2.1 KI-Integration

Dokumentation der KI-Service-Schicht für OpenAI, Ollama und Custom Endpoints.

Übersicht

Die KI-Integration in AILO ermöglicht die Anbindung verschiedener Large Language Models (LLMs) für Textüberarbeitung und -generierung. Der zentrale AIClient abstrahiert die Unterschiede zwischen OpenAI-kompatiblen APIs und Ollama, sodass Nutzer nahtlos zwischen Providern wechseln können.

Komponenten

Datei	Beschreibung
AIClient.swift	Zentraler HTTP-Client für alle KI-Operationen
AIEditor.swift	SwiftUI-Editor für Provider-Konfiguration
AIManagerView.swift	Verwaltung mehrerer KI-Provider
AIProviderConfig	Datenmodell für Provider-Einstellungen

Unterstützte Provider

OpenAI

Vollständige Unterstützung der OpenAI Chat Completions API.

- **Endpoint:** /v1/chat/completions
- **Standard-URL:** https://api.openai.com
- **Port:** 443
- **Modelle:** gpt-4, gpt-4o, gpt-4o-mini, gpt-3.5-turbo
- **Authentifizierung:** Bearer Token (API-Key)

Mistral

OpenAI-kompatible API von Mistral AI.

- **Endpoint:** /v1/chat/completions
- **Standard-URL:** https://api.mistral.ai
- **Modelle:** mistral-large-latest, mistral-medium, mistral-small

Ollama

Lokale LLM-Ausführung über Ollama-Server.

- **Endpoints:** /api/chat, /api/generate
- **Standard-URL:** http://localhost
- **Port:** 11434
- **Modelle:** llama3, llama3:8b, mistral, codellama, etc.
- **Authentifizierung:** Optional (Bearer Token)

Custom Endpoints

Beliebige OpenAI-kompatible Server können konfiguriert werden. Der AIClient erkennt automatisch das API-Format anhand der URL.

AIClient API

Hauptmethode: rewrite()

Führt eine Textüberarbeitung durch. Erkennt automatisch den Provider-Typ.

Parameter	Beschreibung
baseURL	Server-Adresse (z.B. https://api.openai.com)
port	Port-Nummer (optional, Standard: 443)
apiKey	API-Schlüssel für Authentifizierung
model	Modell-ID (z.B. gpt-4, llama3:8b)
prePrompt	System-Prompt für Kontext und Anweisungen
userText	Der zu überarbeitende Originaltext
completion	Callback mit Result<String, Error>

Automatische Provider-Erkennung

Der AIClient erkennt anhand der URL automatisch den Provider-Typ:

1. **URL enthält 'openai.com' oder 'mistral.ai'** → OpenAI-kompatible API
2. **Andere URLs** → Ollama-API (/api/chat, /api/generate)
3. **Fallback** → Automatischer Wechsel zwischen Endpoints bei Fehlern

Error Handling

Der ClientError-Enum definiert alle möglichen Fehlerzustände mit lokalisierten Meldungen:

Fehler	Beschreibung
invalidBaseURL	Ungültige Serveradresse
invalidHTTPResponse	Ungültige Serverantwort
httpStatus(Int)	HTTP-Fehler mit Statuscode
emptyResponse	Leere Antwort vom Server
decoding	JSON-Parsing fehlgeschlagen
endpointNotFound	API-Endpoint nicht gefunden (404)

Provider-Konfiguration

Das AIProviderConfig-Struct speichert alle Einstellungen eines Providers:

- **id:** UUID – Eindeutige Kennung
- **name:** String – Anzeigename (z.B. 'OpenAI Prod')
- **type:** AIProviderType – OpenAI, Mistral, Ollama, Custom
- **baseURL:** String – Server-Adresse
- **port:** String – Port-Nummer

- **apiKey:** String – API-Schlüssel (verschlüsselt gespeichert)
- **model:** String – Ausgewähltes Modell
- **temperature:** Double – Kreativitäts-Parameter (0.0-2.0)

Fallback-Mechanismus

Der AIClient implementiert einen mehrstufigen Fallback bei fehlenden oder ungültigen Parametern:

4. **Übergebene Parameter prüfen:** Direkt verwendbare Werte haben Priorität
5. **Ausgewählten Provider laden:** Fallback auf aktiven Provider aus UserDefaults
6. **Endpoint-Fallback:** Bei Ollama: /api/generate → /api/chat
7. **Standardwerte:** llama3:8b als Default-Modell

Modell-Discovery

Der AIEditor kann verfügbare Modelle vom Server abrufen:

- **OpenAI/Mistral:** GET /v1/models → data[].id
- **Ollama:** GET /api/tags oder /api/models → models[].name

Persistierung

Provider-Konfigurationen werden über UserDefaults gespeichert:

- **config.ai.providers.list** – JSON-Array aller Provider
- **config.ai.providers.selected** – UUID des aktiven Providers

API-Keys werden zusätzlich im iOS Keychain verschlüsselt gespeichert.

Verzeichnisstruktur

Services/AI/

- AIClient.swift

Views/Configuration/

- AIManagerView.swift
- AIEditor.swift

2.2 Mail Services

Dokumentation der Mail-Service-Schicht mit 5-Phasen-Synchronisation, Repository-Pattern und Viewport-basiertem Laden.

Übersicht

Die Mail Services bilden die zentrale Schicht für E-Mail-Operationen in AILO. Sie implementieren ein Repository-Pattern als einheitliche Schnittstelle für die UI, eine 5-Phasen-Synchronisations-Architektur für effizientes Laden und Viewport-basiertes Sync für optimale Performance.

Komponenten

Datei	Beschreibung
MailRepository.swift	Zentrale Schnittstelle für UI – einziger Entry-Point
MailSyncEngine.swift	5-Phasen Synchronisations-Engine
MailProcessorAdapter.swift	Bridge zwischen SyncEngine und Repository
ViewportSyncManager.swift	Viewport-basiertes Laden mit Prefetch-Buffer
MailSendService.swift	Outbox-Queue und SMTP-Versand
MailHealthMonitor.swift	Verbindungs-Health-Check und Metriken
FolderDiscoveryService.swift	Automatische Ordner-Erkennung via IMAP LIST
MailProcessor.swift	MIME-Parsing und Content-Extraktion

5-Phasen Synchronisations-Architektur

Die MailSyncEngine implementiert eine strikte Trennung der Sync-Phasen für optimale Performance und Ressourcennutzung:

Phase 1: Header-Only Sync

Schnelle Header-Synchronisation für die Mailbox-Übersicht. Verwendet IMAP FETCH mit minimalen Daten.

- **IMAP-Befehl:** FETCH (FLAGS UID ENVELOPE)
- **Daten:** UID, From, Subject, Date, Flags
- **Speicherung:** msg_header Tabelle

Phase 2: Body-On-Demand

E-Mail-Body wird erst bei Bedarf (Öffnen der Mail) geladen.

- **IMAP-Befehl:** FETCH BODY[]
- **Verarbeitung:** MailProcessor für MIME-Parsing
- **Speicherung:** msg_body Tabelle

Phase 3: Central Processing

Zentrale Verarbeitung von MIME-Strukturen, Charset-Konvertierung und Content-Extraktion.

- **HTML/Text-Extraktion:** Multipart-Handling
- **Charset-Handling:** UTF-8, ISO-8859-1, etc.
- **Transfer-Encoding:** Quoted-Printable, Base64

Phase 4: Structured Storage

Strukturierte Speicherung in SQLite mit Blob-Storage für große Inhalte.

- **Tabellen:** msg_header, msg_body, attachments
- **Blob-Storage:** blob_meta, mime_parts, render_cache
- **Indizes:** accountId, folder, date für schnelle Queries

Phase 5: Bidirectional Sync

Synchronisation von Flag-Änderungen (gelesen, markiert) zurück zum Server.

- **IMAP-Befehl:** STORE +FLAGS / -FLAGS
- **Unterstützte Flags:** \Seen, \Flagged, \Deleted

MailRepository API

Das Repository ist der einzige Entry-Point für die UI. Es abstrahiert DAOs, SyncEngine und SendService.

Methode	Beschreibung
listHeaders()	Alle Header eines Ordners abrufen
getBody()	Body einer E-Mail laden (on-demand)
sync(accountId:)	Synchronisation für Account starten
send(draft:)	E-Mail in Outbox einreihen
health(accountId:)	Verbindungs-Status abrufen
onChanges(accountId:)	Publisher für Datenänderungen
getAllServerFolders()	Alle Ordner vom Server via IMAP LIST
specialFolders()	Spezialordner-Mapping abrufen

ViewportSyncManager

Optimiert die Synchronisation auf sichtbare E-Mails plus Puffer für flüssiges Scrollen.

Funktionsweise

1. **Tracking:** rowAppeared(uid:) / rowDisappeared(uid:)
2. **Debounce:** 0.3 Sekunden Verzögerung vor Sync
3. **Prefetch-Buffer:** ±10 UIDs um Viewport
4. **Batch-Größe:** Max. 30 UIDs pro Sync-Request

Konfiguration

- **prefetchBuffer:** 10 (UIDs vor/nach Viewport)
- **debounceDelay:** 0.3 Sekunden

- **maxBatchSize:** 30 UIDs

Sync-Limits

Konfigurierbare Limits pro Account in den Mail-Einstellungen:

Parameter	Standard	Beschreibung
syncLimitInitial	50	Initiales Laden bei erstem Sync
syncLimitRefresh	100	Pull-to-Refresh Sync
syncLimitIncremental	20	Hintergrund-Sync neue Mails

Account Health Monitoring

Der MailHealthMonitor überwacht den Verbindungsstatus und liefert Health-Metriken:

- **AccountHealth.ok:** Verbindung funktioniert
- **AccountHealth.degraded:** Eingeschränkte Funktionalität
- **AccountHealth.down:** Keine Verbindung möglich

Folder Discovery

Automatische Erkennung von Spezialordnern via IMAP LIST und Namensanalyse:

- **inbox:** INBOX (standardisiert)
- **sent:** Sent, Gesendet, Sent Items
- **drafts:** Drafts, Entwürfe
- **trash:** Trash, Papierkorb, Deleted
- **spam:** Spam, Junk

Verzeichnisstruktur

Services/Mail/Sync/

- MailRepository.swift
- MailSyncEngine.swift
- MailProcessorAdapter.swift
- ViewportSyncManager.swift
- MailSendService.swift
- MailProcessor.swift
- FolderDiscoveryService.swift

Services/Mail/Diagnostics/

- MailHealthMonitor.swift
- MailLogger.swift
- MailMetrics.swift

Helpers/Utilities/

- MailSendReceive.swift (Transport Layer)

AILO Handbuch – Kapitel 2.2
Erstellt: Dezember 2025

2.3 IMAP Implementation

Services/Mail/IMAP/ • Custom IMAP Protocol Layer

Übersicht

Die IMAP-Implementation in AILO ist eine vollständig eigene Lösung, die ohne externe Bibliotheken auf Apples Network.framework (NWConnection) aufbaut. Sie bietet sichere Verbindungen über TLS/STARTTLS und unterstützt alle gängigen IMAP-Befehle für E-Mail-Synchronisation.

Komponente	Beschreibung
IMAPConnection	Low-Level TCP/TLS Transport Layer
IMAPCommands	Stateless Helper für IMAP-Befehlsausführung
IMAPParsers	Response-Parsing für ENVELOPE, FLAGS, BODYSTRUCTURE

IMAPConnection.swift

Der Low-Level Transport-Layer für IMAP-Verbindungen. Verwaltet TCP/TLS-Verbindungen, SNI, Timeouts und das Senden/Empfangen von IMAP-Zeilen.

Konfiguration

`IMAPConnectionConfig`

Parameter	Typ	Beschreibung
`host`	String	IMAP-Server Hostname
`port`	Int	Port (993 für IMAPS, 143 für STARTTLS)
`tls`	Bool	Direktes TLS (true) oder Plain+STARTTLS (false)
`sniHost`	String?	Server Name Indication für TLS
`connectionTimeoutSec`	Int	Verbindungs-Timeout (Standard: 15s)
`commandTimeoutSec`	Int	Befehls-Timeout (Standard: 10s)
`idleTimeoutSec`	Int	Idle-Timeout für lange Reads (Standard: 10s)

Fehlerbehandlung

Die IMAPError-Enumeration definiert alle möglichen Fehler:

Fehlertyp	Beschreibung
`.invalidState`	Ungültiger Verbindungszustand
`.connectTimeout`	Verbindungs-Timeout überschritten
`.connectFailed`	Verbindungsaufbau fehlgeschlagen
`.sendFailed`	Senden fehlgeschlagen
`.receiveFailed`	Empfangen fehlgeschlagen
`.networkUnreachable`	Netzwerk nicht erreichbar
`.protocolError`	IMAP-Protokollfehler
`.closed`	Verbindung geschlossen

Kernmethoden

- **open(_ cfg: IMAPConnectionConfig)** – Öffnet eine neue IMAP-Verbindung mit TLS 1.2/1.3 Support
- **close()** – Beendet die Verbindung und räumt Ressourcen auf

- **upgradeToTLS()** – Upgrade auf TLS nach STARTTLS-Befehl
- **send(line: String)** – Sendet eine IMAP-Zeile (CRLF wird automatisch angehängt)
- **receiveLines(untilTag:)** – Empfängt Zeilen bis zur Tagged Response (OK/NO/BAD)

IMAPCommands.swift

Stateless Helper für die Ausführung von IMAP-Befehlen. Generiert eindeutige Tags (A1, A2, ...) und formatiert Befehle gemäß RFC 3501.

Session-Befehle

Methode	Beschreibung
`greeting()`	Empfängt Server-Begrüßung (* OK IMAP4rev1 ...)
`login(user:pass:)`	Authentifizierung mit Benutzername/Passwort
`logout()`	Beendet die IMAP-Sitzung ordnungsgemäß
`startTLS()`	Initiiert STARTTLS-Upgrade
`capabilities()`	Ruft Server-Capabilities ab

Ordner-Befehle

Methode	Beschreibung
`listAll()`	Listet alle verfügbaren Ordner (LIST "" "*")
`listSpecialUse()`	Listet Spezialordner gemäß RFC 6154
`select(folder:readOnly:)`	Wählt Ordner aus (SELECT/EXAMINE)

Fetch-Befehle

Methode	Beschreibung
`uidSearch(query:)`	Sucht UIDs nach Kriterien
`uidFetchEnvelope(uids:)`	Holt ENVELOPE + INTERNALDATE + FLAGS
`uidFetchEnvelopeWithStructure(uids:)`	Mit zusätzlichem BODYSTRUCTURE
`uidFetchFlags(uids:)`	Nur FLAGS abrufen
`uidFetchBody(uid:partsOrPeek:)`	Body-Inhalt abrufen
`uidStore(uids:flags:action:)`	Flags setzen/entfernen

IMAPParsers.swift

Parser-Utilities für IMAP-Protokoll-Responses. Konvertiert rohe Antwortzeilen in typisierte Swift-Strukturen. Unterstützt RFC 2047 für Encoded-Words.

Datenmodelle

EnvelopeRecord

- `uid: String` – Eindeutige Message-ID
- `subject: String` – Betreff (RFC 2047 dekodiert)
- `from: String` – Absender (RFC 2047 dekodiert)
- `internalDate: Date?` – Server-Empfangsdatum

MessageEnvelope

Erweitertes Envelope-Modell mit vollständigen Adressfeldern:

- `to, cc, bcc: [String]` – Empfänger-Listen
- `messageId: String?` – Message-ID Header

BodyStructure

Rekursive Struktur für MIME-Parts gemäß RFC 2045:

- `single(Part)` – Einzelner MIME-Part (text/plain, image/png, etc.)
- `multipart(type:subType:parts:)` – Multipart-Container (mixed, alternative, related)

Parser-Methoden

Methode	Rückgabe
`parseEnvelope(lines:)`	[EnvelopeRecord] – Parsed ENVELOPE-Felder
`parseFlags(lines:)`	[FlagsRecord] – UIDs mit zugehörigen Flags
`parseBodyStructure(line:)`	BodyStructure – Rekursive MIME-Struktur
`parseBodySection(lines:)`	String? – Body-Inhalt als Text
`parseUIDs(line:)`	[String] – UIDs aus SEARCH-Antwort
`parseFetchResponse(data:)`	FetchResult – Vollständige FETCH-Antwort

TLS/STARTTLS Support

AILO unterstützt beide Verbindungsmodi für sichere E-Mail-Übertragung:

Direkte TLS-Verbindung (IMAPS)

- Standard-Port: 993
- Sofortige TLS-Verschlüsselung beim Verbindungsaufbau
- Konfiguration: tls = true in IMAPConnectionConfig

STARTTLS-Upgrade

- Standard-Port: 143
- Unverschlüsselte Verbindung → STARTTLS-Befehl → TLS-Upgrade
- upgradeToTLS() schließt Plain-Verbindung und öffnet TLS-Tunnel

TLS-Versionen

Unterstützte Protokollversionen: TLS 1.2 (Minimum) bis TLS 1.3 (Maximum), konfiguriert über sec_protocol_options für maximale Kompatibilität mit Enterprise-Mailservern.

Verwendungsbeispiel

Typischer Workflow für das Abrufen von E-Mail-Headern:

1. IMAPConnection mit IMAPConnectionConfig erstellen
2. open() aufrufen → Verbindung herstellen
3. greeting() → Server-Begrüßung empfangen
4. Optional: startTLS() für STARTTLS-Upgrade
5. login(user:pass:) → Authentifizierung
6. select(folder:) → Ordner auswählen
7. uidSearch() → UIDs abrufen
8. uidFetchEnvelope() → Header laden
9. IMAPParsers.parseEnvelope() → Response parsen
10. logout() + close() → Verbindung beenden

Technische Hinweise

- **Thread-Safety:** IMAPConnection ist NICHT thread-safe. Verwenden Sie eine Instanz nur von einem seriellen Kontext.
- **Async/Await:** Alle Netzwerk-Operationen sind async und nutzen Swift Concurrency (CheckedContinuation).
- **RFC 2047:** Encoded-Words in Betreff und Absender werden automatisch dekodiert (UTF-8, ISO-8859-1).
- **Literal-Handling:** Der Transport materialisiert {n}\r\n-Literal-Blöcke automatisch in den empfangenen Zeilen.
- **Fallback-Encoding:** Bei UTF-8-Dekodierungsfehlern wird automatisch ISO-8859-1 versucht.

2.4 SMTP Implementation

Services/Mail/SMTP/ • E-Mail-Versand mit S/MIME Support

Übersicht

Die SMTP-Implementation in AILO ermöglicht den sicheren Versand von E-Mails mit vollständiger MIME-Unterstützung, optionaler S/MIME-Signierung und einer persistenten Outbox-Queue mit automatischem Retry-Mechanismus.

Komponente	Beschreibung
SMTPAbstractions	Protokoll-Definition für austauschbare Clients
SMTPClient	NWConnection-basierter SMTP-Client
NIOSMTPClient	SwiftNIO-basierte Alternative (optional)
MailSendService	Outbox-Queue mit Retry-Logik
SMIMESigningService	S/MIME-Signierung für ausgehende Mails

SMTPAbstractions.swift

Definiert das SMTPClientProtocol zur Entkopplung des MailSendService von konkreten SMTP-Client-Implementierungen. Ermöglicht den Austausch zwischen NWConnection- und SwiftNIO-basierten Clients.

SMTPClientProtocol

- `testConnection(_ config: SMTPConfig)` → Result<Void, SMTPError>
- `send(_ message: MailMessage, using config: SMTPConfig)` → DeliveryResult

SMTPConfig

Parameter	Typ	Beschreibung
`host`	String	SMTP-Server Hostname
`port`	UInt16	Port (587 für STARTTLS, 465 für SSL)
`encryption`	Encryption	.none, .startTLS, .sslTLS
`username`	String?	Benutzername für Authentifizierung
`password`	String?	Passwort für Authentifizierung
`heloName`	String	HELO/EHLO Domain-Name
`connectionTimeoutSec`	Int	Verbindungs-Timeout
`commandTimeoutSec`	Int	Befehls-Timeout

SMTPClient.swift

Implementiert das SMTPClientProtocol mit Apples Network.framework (NWConnection). Unterstützt TLS/STARTTLS, PLAIN/LOGIN-Authentifizierung und vollständiges RFC 5321/5322-konformes Message-Building.

Fehlerbehandlung (SMTPError)

Fehlertyp	Beschreibung
invalidState	Ungültiger Verbindungszustand
connectTimeout	Verbindungs-Timeout überschritten
connectFailed	Verbindungsaufbau fehlgeschlagen

Fehlertyp	Beschreibung
greetingFailed	Server-Begrüßung ungültig
startTLSRejected	Server hat STARTTLS abgelehnt
authRequired	Authentifizierung erforderlich
authFailed	Authentifizierung fehlgeschlagen
commandFailed	SMTP-Befehl fehlgeschlagen (Code + Message)
sendFailed	Senden fehlgeschlagen
closed	Verbindung geschlossen

SMTP-Workflow

1. Verbindung öffnen (open) → TCP/TLS-Handshake
2. Server-Greeting empfangen (220 OK)
3. EHLO senden → Capabilities abrufen
4. Optional: STARTTLS → TLS-Upgrade
5. AUTH LOGIN/PLAIN → Authentifizierung
6. MAIL FROM → Absender setzen
7. RCPT TO → Empfänger setzen (To, Cc, Bcc)
8. DATA → RFC 5322 Message übertragen
9. QUIT → Verbindung beenden

MailSendService.swift

Verwaltet die Outbox-Queue für ausgehende E-Mails. Bietet asynchronen Versand mit automatischem Retry bei Fehlern, Backoff-Strategien und persistente Speicherung über OutboxDAO.

OutboxItem

Feld	Typ	Beschreibung
`id`	UUID	Eindeutige Item-ID
`accountId`	UUID	Zugehöriger Mail-Account
`createdAt`	Date	Erstellungszeitpunkt
`lastAttemptAt`	Date?	Letzter Versuch
`attempts`	Int	Anzahl Versuche
`status`	OutboxStatus	pending, sending, sent, failed, cancelled
`lastError`	String?	Letzter Fehlertext
`draft`	MailDraft	E-Mail-Inhalt

Public API

- **queue(_ draft:accountId:)** – Fügt Draft zur Outbox hinzu
- **sendDraft(_ draft:accountId:)** – Validiert, queued und triggert Versand
- **retry(_ id:accountId:)** – Setzt fehlgeschlagenes Item zurück auf pending
- **processNext(accountId:)** – Verarbeitet das nächste Item (One-Shot)
- **processAll(accountId:)** – Verarbeitet Queue bis leer/Fehler
- **publisherOutbox(accountId:)** – Combine Publisher für Outbox-Änderungen

Retry-Mechanismus

- Exponentielles Backoff bei Fehlern
- Minimum 30 Sekunden zwischen Versuchen
- Erfolge/Fehler werden in RetryPolicy protokolliert
- MailMetrics trackt Success/Failure-Raten pro Host

S/MIME Signing Support

AILO unterstützt die digitale Signierung ausgehender E-Mails mittels S/MIME (RFC 5751). Die Signatur wird über SMIMESigningService erstellt und als multipart/signed-Nachricht gesendet.

SMIMESigningService

- **signMessage(mimeContent:certificateId:)** → Result<Data, SigningError>
- **canSign(certificateId:)** → Bool – Prüft ob Zertifikat verfügbar

Plattform-spezifische Implementierung

Plattform	Methode
macOS	CMSEncoder API für native CMS/PKCS#7-Signierung
iOS	Manuelle CMS-Konstruktion mit Security Framework + CommonCrypto

Signierungsprozess

10. Identity (Zertifikat + Private Key) aus Keychain laden
11. Inner MIME Content erstellen (Body ohne Outer Headers)
12. SHA-256 Hash des Contents berechnen
13. Signatur mit Private Key erstellen (PKCS#7/CMS)
14. multipart/signed Message zusammenbauen
15. application/pkcs7-signature Part anhängen

SigningError

- `.certificateNotFound` – Zertifikat nicht im Keychain
- `.certificateError` – Zertifikat konnte nicht geladen werden
- `.privateKeyError` – Private Key nicht verfügbar

MIME Message Building

Der SMTPClient baut RFC 5322-konforme Messages mit vollständiger MIME-Unterstützung für Text, HTML, Multipart und Attachments.

Unterstützte Content-Types

- `text/plain` – Reiner Text (UTF-8)
- `text/html` – HTML-Inhalt
- `multipart/alternative` – Text + HTML kombiniert
- `multipart/mixed` – Mit Attachments
- `multipart/signed` – S/MIME-signiert

RFC-Konformität

- **RFC 5321:** SMTP-Protokoll, Dot-Stuffing
- **RFC 5322:** Internet Message Format
- **RFC 2047:** Encoded-Words für Header (Subject, From)
- **RFC 2045:** MIME Content-Type und Boundaries
- **RFC 5751:** S/MIME Message Specification

Technische Hinweise

- **Async/Await:** Alle Netzwerk-Operationen nutzen Swift Concurrency.
- **Worker-Queues:** Jeder Account hat eine dedizierte DispatchQueue für Outbox-Verarbeitung.
- **Combine Integration:** CurrentValueSubject publiziert Outbox-Änderungen für UI-Updates.
- **Line Endings:** CRLF (\r\n) für SMTP-Übertragung, LF für S/MIME-Hash-Berechnung.
- **Dot-Stuffing:** Zeilen mit führendem Punkt werden gemäß RFC 5321 escaped.

2.5 Pre-Prompt Management

Helpers/Utilities/ • Hierarchisches Prompt-Katalog-System

Übersicht

Das Pre-Prompt Management System in AILO bietet eine hierarchische Struktur zur Organisation, Verwaltung und Kombination von KI-Prompts. Es unterstützt verschachtelte Ordner, wiederverwendbare Presets, Rezept-Kombinationen und Kochbücher für komplexe Prompt-Workflows.

Komponente	Beschreibung
PrePromptCatalogManager	Singleton Manager für alle Katalog-Operationen
PrePromptMenuItem	Hierarchisches Menü-Item (Ordner oder Preset-Referenz)
AIPrePromptPreset	Einzelner Pre-Prompt mit Text und Metadaten
PrePromptRecipe	Kombiniert mehrere Presets zu einem Workflow
Cookbook	Sammlung von Rezepten in Kapiteln
RecipeMenuItem	Hierarchische Struktur innerhalb eines Kochbuchs

PrePromptCatalogManager

Der zentrale Singleton-Manager für das gesamte Pre-Prompt-System. Verwaltet Menüstruktur, Presets, Rezepte und Kochbücher. Implementiert als ObservableObject für SwiftUI-Integration.

Published Properties

Property	Typ	Beschreibung
`menuItems`	[PrePromptMenuItem]	Hierarchische Menüstruktur
`presets`	[AIPrePromptPreset]	Alle Pre-Prompt-Inhalte
`recipes`	[PrePromptRecipe]	Kombinierte Rezepte
`cookbooks`	[Cookbook]	Rezept-Sammlungen
`recipeMenuItems`	[RecipeMenuItem]	Kochbuch-Strukturen

Menu Item Operations

- **addMenuItem(_:)** – Fügt neues Menu-Item hinzu
- **updateMenuItem(_:)** – Aktualisiert bestehendes Item
- **deleteMenuItem(_:)** – Löscht Item mit allen Descendants
- **moveMenuItem(_:to:)** – Verschiebt Item zu neuem Parent
- **reorderItems(in:from:to:)** – Sortiert Items innerhalb Parent

Preset Operations

- **addPreset(_:in:)** – Erstellt Preset + Menu-Item
- **updatePreset(_:)** – Aktualisiert Preset + Menu-Item
- **deletePreset(_:)** – Entfernt Preset + Menu-Item

Query Helpers

- **children(of:)** – Kinder eines Parents (nil = Root)
- **path(to:)** – Breadcrumb-Pfad zu einem Item
- **preset(withID:)** – Preset anhand ID laden

- **presets(in:)** – Alle Presets in Ordner (rekursiv)

PrePromptMenuItem

Repräsentiert ein Element in der hierarchischen Menüstruktur. Kann entweder ein Ordner (Container) oder eine Referenz auf ein Preset sein. Unterstützt unbegrenzte Verschachtelungstiefe.

Eigenschaften

Feld	Typ	Beschreibung
`id`	UUID	Eindeutige Item-ID
`parentID`	UUID?	Parent-ID (nil = Root-Level)
`name`	String	Anzeigename im Menü
`icon`	String	Emoji-Icon
`keywords`	String	Semikolon-getrennte Schlagwörter
`sortOrder`	Int	Sortierreihenfolge
`presetID`	UUID?	Verweis auf Preset (nil = Ordner)

Computed Properties

- `isFolder: Bool` – true wenn presetID == nil
- `isPreset: Bool` – true wenn presetID != nil
- `keywordPairs: [(key, value)]` – Geparste Key-Value-Paare

Factory Methods

- `PrePromptMenuItem.folder(name:icon:keywords:parentID:sortOrder:)`
- `PrePromptMenuItem.preset(name:icon:keywords:parentID:sortOrder:presetID:)`

AIPrePromptPreset

Enthält den eigentlichen Pre-Prompt-Text sowie Metadaten. Wird über PrePromptMenuItem im Menü referenziert und kann in Rezepten kombiniert werden.

Eigenschaften

Feld	Typ	Beschreibung
`id`	UUID	Eindeutige Preset-ID
`name`	String	Anzeigename
`text`	String	Der eigentliche Prompt-Text
`icon`	String	Emoji-Icon
`keywords`	String	Variablen-Definitionen (key:value;...)
`isDefault`	Bool	Standard-Preset markiert

PrePromptRecipe

Ein Rezept kombiniert mehrere Presets und Ordner zu einem komplexen Prompt-Workflow. Kategorien werden zu Überschriften, Presets liefern den Inhalt. Keywords werden hierarchisch vererbt und können überschrieben werden.

Eigenschaften

Feld	Typ	Beschreibung
`id`	UUID	Eindeutige Rezept-ID
`name`	String	Rezept-Name
`icon`	String	Emoji-Icon
`keywords`	String	Rezept-spezifische Keywords
`elementIDs`	[UUID]	Geordnete Liste der Elemente
`separator`	String	Trennzeichen zwischen Elementen

Kernmethoden

- **generatePrompt(from:presets:)** → String – Kombinierter Prompt-Text
- **collectKeywords(from:presets:)** → [(key, value)] – Alle Keywords

Cookbook & RecipeMenuItem

Ein Cookbook ist eine Sammlung von Rezepten, organisiert in Kapiteln. RecipeMenuItem bildet die hierarchische Struktur innerhalb eines Kochbuchs ab – analog zu PrePromptMenuItem für den Preset-Katalog.

Cookbook-Eigenschaften

- `id: UUID` – Eindeutige Kochbuch-ID
- `name: String` – Kochbuch-Name
- `icon: String` – Emoji-Icon
- `keywords: String` – Kochbuch-Keywords
- `sortOrder: Int` – Sortierreihenfolge

RecipeMenuItem

Kann ein Kapitel (isChapter) oder eine Rezept-Referenz (recipeID) sein:

- `cookbookID: UUID` – Zugehöriges Kochbuch
- `parentID: UUID?` – Parent-Kapitel
- `recipeID: UUID?` – Verweis auf Rezept (nil = Kapitel)

Persistierung

Alle Daten werden über UserDefaults als JSON persistiert. Bei der ersten Initialisierung werden Standard-Kategorien und Demo-Presets erstellt.

UserDefaults Keys

Key	Inhalt
kPrePromptMenuKey	Hierarchische Menüstruktur
kAIPresetsKey	Alle Pre-Prompt-Presets
kPrePromptRecipesKey	Rezept-Definitionen
kCookbooksKey	Kochbuch-Metadaten
kRecipeMenuKey	Kochbuch-Strukturen
kCatalogInitializedKey	First-Launch-Flag

Export/Import

Der gesamte Katalog kann als JSON exportiert und importiert werden. Dies ermöglicht Backup, Sharing und Migration zwischen Geräten.

CatalogExport-Struktur

- `version: Int` – Schema-Version (aktuell: 1)
- `exportDate: Date` – Zeitpunkt des Exports
- `menuItems, presets, recipes, cookbooks, recipeMenuItems`

Methoden

- **exportCatalog()** → Data? – JSON-Export
- **importCatalog(from:)** → Bool – JSON-Import mit Validierung

Standard-Kategorien

Bei Erstinstallation werden folgende Kategorien automatisch erstellt:

Icon	Kategorie	Beschreibung
	Mail	Haupt-Kategorie für E-Mail-Prompts
	Reply	Antwort-Vorlagen (unter Mail)
	Forward	Weiterleitungs-Vorlagen (unter Mail)
	Analyze	Analyse-Prompts (unter Mail)
	Notes	Notizen und Protokolle
	General	Allgemeine Prompts

Migration

Das System unterstützt automatische Migration von Legacy-Daten:

- **migrateFromLegacy():** Verschiebt bestehende Presets in "Migriert"-Ordner
- **migrateRecipesToCookbook():** Erstellt Standard-Kochbuch für lose Rezepte

Technische Hinweise

- **ObservableObject:** @Published Properties ermöglichen reaktive SwiftUI-Updates
- **Codable:** Alle Modelle sind JSON-serialisierbar für Persistenz und Export
- **Sendable:** Thread-sichere Modelle für Swift Concurrency
- **Keywords:** Format "key:value;key2:value2" für Template-Interpolation
- **Hierarchie:** Unbegrenzte Verschachtelungstiefe über parentID-Referenzen

2.6 Audio & Speech

Views/Sprechen/ • Live-Transkription mit AVFoundation & Speech Framework

Übersicht

Das Audio & Speech System in AILO ermöglicht Sprachaufnahmen mit Live-Transkription. Es nutzt AVFoundation für Audio-Recording und das Speech Framework für Echtzeit-Spracherkennung mit automatischer Chunk-basierter Verarbeitung und Silence Detection.

Komponente	Beschreibung
AudioRecorder	AVFoundation-basierte Audio-Aufnahme mit Pegel-Metering
LiveTranscriber	Echtzeit-Spracherkennung mit SFSpeechRecognizer
RecordingState	ObservableObject für UI-State-Management
SprechenView	SwiftUI-View für Aufnahme und Transkription

AudioRecorder

Die AudioRecorder-Klasse kapselt AVAudioRecorder für hochwertige Audio-Aufnahmen im m4a-Format. Implementiert als ObservableObject mit AVAudioRecorderDelegate für SwiftUI-Integration.

Published Properties

Property	Typ	Beschreibung
`isRecording`	Bool	Aufnahme aktiv
`isPaused`	Bool	Aufnahme pausiert
`elapsed`	TimeInterval	Verstrichene Zeit in Sekunden
`level`	Float	Aktueller Audiopegel (dB)

Methoden

- **startRecording(to:sensitivity:)** – Startet Aufnahme mit Mikrofon-Empfindlichkeit
- **pause()** – Pausiert die laufende Aufnahme
- **resume()** – Setzt pausierte Aufnahme fort
- **stop(completion:)** – Beendet Aufnahme mit Callback für URL

Audio-Einstellungen

Parameter	Wert
Format	MPEG4 AAC (kAudioFormatMPEG4AAC)
Sample Rate	44.100 Hz
Channels	1 (Mono)
Bitrate	128.000 bps
Quality	AVAudioQuality.high
Output	.m4a Datei

AVAudioSession-Konfiguration

- **Category:** .playAndRecord
- **Mode:** .default
- **Options:** .defaultToSpeaker
- **Input Gain:** Einstellbar über Mikrofon-Empfindlichkeit (0.0–1.0)

LiveTranscriber

Die LiveTranscriber-Klasse implementiert Echtzeit-Spracherkennung mit dem Speech Framework. Nutzt AVAudioEngine für Audio-Streaming und SFSpeechRecognizer für die Transkription mit automatischer Chunk-Verarbeitung.

Published Properties

Property	Typ	Beschreibung
`combinedText`	String	Gesamter transkribierter Text (alle Chunks)
`currentChunk`	String	Aktueller, noch nicht finalisierter Chunk

Konfigurationsoptionen

Option	Typ	Beschreibung
`localeCode`	String	Sprache ("auto" oder Locale-ID)
`partialResultsEnabled`	Bool	Zwischenergebnisse aktiviert
`onDeviceOnly`	Bool	Nur On-Device Recognition
`amplitudeThreshold`	Float	Schwellwert für Stille-Erkennung
`micSensitivity`	Double	Mikrofon-Empfindlichkeit (0.0–1.0)

Methoden

- **applyConfig(...)** – Konfiguration setzen und Threshold berechnen
- **start()** – Startet Engine und Spracherkennung
- **stop()** – Beendet Recognition und committed letzten Chunk
- **reset()** – Setzt alle States zurück
- **previewText()** → String – Kombiniert combinedText + aktuelles Delta

Silence Detection

Der LiveTranscriber implementiert automatische Stille-Erkennung zur intelligenten Chunk-Segmentierung. Dies ermöglicht natürliche Pausen im Sprachfluss und vermeidet übergroße Transkriptionsblöcke.

Funktionsweise

1. Audio-Buffer wird über AVAudioEngine empfangen (1024 Frames)
2. RMS-Amplitude wird per vDSP_measqv berechnet
3. Vergleich mit amplitudeThreshold (konfigurierbar)
4. Bei Stille: Timer startet (silenceHold = 0.8s)
5. Nach Ablauf: commitCurrentChunk() wird aufgerufen

Parameter

Parameter	Wert	Beschreibung
`silenceHold`	0.8 Sekunden	Haltezeit bevor Chunk finalisiert
`minThreshold`	0.003 (~-50 dB)	Hohe Empfindlichkeit
`maxThreshold`	0.02 (~-34 dB)	Niedrige Empfindlichkeit
`bufferSize`	1024 Frames	Audio-Buffer-Größe

Chunk-basierte Transkription

Die Transkription erfolgt in Chunks, die bei Sprechpausen automatisch finalisiert werden. Dies ermöglicht flüssige Echtzeit-Anzeige und verhindert Duplikate.

Deduplizierung

Die commitCurrentChunk()-Methode implementiert mehrere Deduplizierungs-Strategien:

- Delta-Berechnung: Nur neue Textteile werden extrahiert
- lastCommitted-Vergleich: Identische Chunks werden übersprungen
- Suffix-Check: Prüfung ob Text bereits am Ende vorhanden
- Line-Check: Vergleich mit letzter Zeile in combinedText

On-Device Recognition

AILO unterstützt On-Device Spracherkennung für datenschutzsensible Anwendungen. Die Audiodaten verlassen das Gerät nicht, wenn dieser Modus aktiviert ist.

Voraussetzungen

- iOS 13.0+ erforderlich
- Prüfung via recognizer.supportsOnDeviceRecognition
- Sprachpaket muss heruntergeladen sein
- Fallback auf Server-Recognition wenn nicht verfügbar

Aktivierung

- **UserDefaults Key:** config.speechOnDeviceOnly
- **Request Property:** requiresOnDeviceRecognition = true

Konfiguration

Die Spracheinstellungen werden über UserDefaults persistiert und in der Settings-View konfiguriert.

UserDefaults Keys

Key	Beschreibung
config.speech.lang	Sprach-Code (z.B. "de-DE", "en-US")
config.speechOnDeviceOnly	On-Device Recognition aktiviert
config.speechPartial	Zwischenergebnisse aktiviert
config.micSensitivity	Mikrofon-Empfindlichkeit (0.0–1.0)

SprechenView

Die SwiftUI-View integriert AudioRecorder und LiveTranscriber für eine nahtlose Benutzeroberfläche mit Pegel-Anzeige, Live-Transkript und Speicherfunktion.

UI-Komponenten

- **Titel-Eingabe:** TextField für optionalen Eintragstitel
- **Level Meter:** Capsule-basierte Pegel-Anzeige (grün/gelb/rot)
- **Zeitanzeige:** Verstrichene Aufnahmezeit (mm:ss)
- **Steuerung:** Start/Stop/Pause-Buttons

- **Transkript:** ScrollView mit Auto-Scroll zum Ende
- **Speichern:** Button zum Sichern als Log-Eintrag

State-Management

- **@StateObject audio:** AudioRecorder-Instanz
- **@StateObject live:** LiveTranscriber-Instanz
- **@State transcript:** Finalisierter Transkript-Text
- **@EnvironmentObject store:** DataStore für Log-Speicherung

Technische Hinweise

- **Berechtigungen:** Mikrofon + Spracherkennung müssen genehmigt sein
- **Delegate:** AVAudioRecorderDelegate für Finish-Callback
- **RunLoop:** Timer wird mit .common Mode registriert
- **Accelerate:** vDSP für performante RMS-Berechnung
- **Memory:** [weak self] in allen Closures zur Vermeidung von Retain Cycles
- **Session:** Deaktivierung mit .notifyOthersOnDeactivation nach Aufnahme

---

2.7 Sicherheit

Services/Security/ & Helpers/Security/ • Keychain, S/MIME & Verschlüsselung

Übersicht

Das Sicherheits-System in AILO gewährleistet den sicheren Umgang mit sensiblen Daten. Es nutzt den iOS/macOS Keychain für Passwörter und Zertifikate, implementiert S/MIME-Signierung für E-Mails und stellt sicher, dass keine Credentials im Klartext gespeichert werden.

Komponente	Beschreibung
KeychainService	Passwort- und Token-Speicherung im System-Keychain
KeychainCertificateService	S/MIME-Zertifikatsverwaltung und P12-Import
SMIMESigningService	Digitale Signierung ausgehender E-Mails
SMIMEVerificationService	Signatur-Verifizierung eingehender E-Mails

KeychainService

Der KeychainService ist ein leichtgewichtiger Wrapper um das iOS/macOS Security Framework. Er speichert sensible Strings wie Passwörter und OAuth-Tokens sicher im System-Keychain.

Basis-API

Methode	Beschreibung
set(_:for:)	Speichert String-Wert für Key → Bool
get(_:)	Liest String-Wert für Key → String?
delete(_:)	Löscht Keychain-Eintrag → Bool

Passwort-Typen (PasswordKind)

- `.recv` – IMAP-Empfangspasswort (Prefix: mail.recv)
- `.smtp` – SMTP-Sendepasswort (Prefix: mail.smtp)

Token-Typen (TokenKind)

- `.recv` – OAuth-Token für IMAP (Prefix: mail.oauth.recv)
- `.smtp` – OAuth-Token für SMTP (Prefix: mail.oauth.smtp)

Convenience-API

- **setPassword(_:kind:accountId:)** – Passwort für Account speichern
- **password(kind:accountId:)** → String? – Passwort abrufen
- **setToken(_:kind:accountId:)** – OAuth-Token speichern
- **token(kind:accountId:)** → String? – OAuth-Token abrufen

Keychain-Attribute

Attribut	Wert
kSecClass	kSecClassGenericPassword
kSecAttrService	Bundle Identifier (App-spezifisch)

Attribut	Wert
kSecAttrAccount	Zusammengesetzter Key (kind.accountId)
kSecAttrAccessible	kSecAttrAccessibleAfterFirstUnlock

KeychainCertificateService

Der KeychainCertificateService verwaltet S/MIME-Zertifikate im System-Schlüsselbund. Er ermöglicht das Auflisten, Importieren und Löschen von Identities (Zertifikat + Private Key).

Kernmethoden

- **listSigningCertificates()** → [SigningCertificateInfo]
- **loadIdentity(certificateId:)** → SecIdentity?
- **importP12(data:password:)** → Result<SigningCertificateInfo, P12ImportError>
- **deleteCertificate(certificateId:)** → Bool

SigningCertificateInfo

Feld	Typ	Beschreibung
`id`	String	Base64-encodierte Persistent Reference
`displayName`	String	Subject Summary des Zertifikats
`email`	String?	E-Mail-Adresse aus Zertifikat
`expiresAt`	Date?	Ablaufdatum (falls extrahierbar)
`identity`	SecIdentity	Referenz auf Keychain-Identity

P12-Import-Prozess

1. P12-Datei mit SecPKCS12Import und Passwort öffnen
2. Identity (kSecImportItemIdentity) extrahieren
3. Zertifikat via SecIdentityCopyCertificate erhalten
4. Private Key via SecIdentityCopyPrivateKey erhalten
5. Zertifikat in Keychain speichern (kSecClassCertificate)
6. Private Key in Keychain speichern (kSecClassKey)

P12ImportError

Fehlertyp	Beschreibung
.wrongPassword	Falsches Passwort für P12-Datei
.invalidFile	Ungültiges P12/PFX-Dateiformat
.noIdentityFound	Keine Identity in P12 gefunden
.certificateError	Zertifikat konnte nicht extrahiert werden
.privateKeyError	Private Key konnte nicht extrahiert werden
.keychainError(OSStatus)	Keychain-Operation fehlgeschlagen

SMIMESigningService

Der SMIMESigningService signiert ausgehende E-Mails digital gemäß RFC 5751 (S/MIME). Die Signatur garantiert Authentizität und Integrität der Nachricht.

Public API

- **signMessage(mimeContent:certificateId:)** → Result<Data, SigningError>
- **canSign(certificateId:)** → Bool – Prüft Zertifikat-Verfügbarkeit

Plattform-Implementierungen

Plattform	Implementierung
macOS	CMSEncoder API für native CMS/PKCS#7-Signierung
iOS	Manuelle CMS-Konstruktion mit Security Framework + CommonCrypto

macOS: CMSEncoder-Workflow

7. CMSEncoderCreate() – Encoder initialisieren
8. CMSEncoderAddSigners() – Identity hinzufügen
9. CMSEncoderAddSupportingCerts() – Zertifikat-Kette
10. CMSEncoderSetHasDetachedContent(true) – Detached Signature
11. CMSEncoderAddSignedAttributes(.attrSigningTime)
12. CMSEncoderUpdateContent() – MIME-Content übergeben
13. CMSEncoderCopyEncodedContent() – Signatur erhalten

iOS: Manuelle CMS-Konstruktion

14. Schlüssel-Algorithmus bestimmen (RSA/EC)
15. SHA-256 Digest des MIME-Contents berechnen (CommonCrypto)
16. SignerInfo ASN.1-Struktur aufbauen
17. Signatur mit SecKeyCreateSignature erstellen
18. multipart/signed MIME-Nachricht assemblieren

SigningError

- `.certificateNotFound` – Zertifikat nicht im Keychain
- `.certificateError` – Zertifikat konnte nicht geladen werden
- `.privateKeyError` – Private Key nicht verfügbar
- `.signingFailed(String)` – Allgemeiner Signierfehler

S/MIME Signaturformat

Die signierte Nachricht wird als multipart/signed gemäß RFC 5751 aufgebaut. Die Signatur ist "detached", d.h. der Original-Content bleibt lesbar.

MIME-Struktur

- **Content-Type:** multipart/signed; protocol="application/pkcs7-signature"; micalg=sha-256
- **Part 1:** Original MIME-Content (text/plain, text/html, etc.)
- **Part 2:** application/pkcs7-signature (Base64-encoded)

Unterstützte Algorithmen

Schlüsseltyp	Signatur-Algorithmus	Hash-Algorithmus
`RSA`	rsaSignatureMessagePKCS1v15SHA256	SHA-256
`EC (P-256)`	ecdsaSignatureMessageX962SHA256	SHA-256

Sicherheitsprinzipien

AILO folgt etablierten Sicherheitsprinzipien für den Umgang mit sensiblen Daten:

Daten im Ruhezustand

- **Passwörter:** Nur im System-Keychain, nie in UserDefaults oder Dateien
- **API-Keys:** Im Keychain mit kSecAttrAccessibleAfterFirstUnlock
- **Zertifikate:** Im System-Schlüsselbund mit kSecAttrAccessibleWhenUnlockedThisDeviceOnly
- **Private Keys:** Nur über SecIdentity-Referenz, nie als Raw-Data exportiert

Daten in Übertragung

- **IMAP/SMTP:** TLS 1.2+ erzwungen (STARTTLS oder IMAPS/SMTPS)
- **API-Calls:** HTTPS mit Certificate Pinning (optional)
- **E-Mail-Content:** Optional S/MIME-signiert für Authentizität

Zugriffskontrolle

- **App-Sandbox:** Strikte iOS/macOS Sandbox-Isolation
- **Keychain-Scoping:** Service = Bundle-ID für App-spezifische Einträge
- **Keine Cloud-Sync:** Sensible Daten verlassen das Gerät nicht

Technische Hinweise

- **Singleton Pattern:** Beide Services als .shared für globalen Zugriff
- **OSStatus-Handling:** Alle Security-Funktionen prüfen Return-Codes
- **Conditional Compilation:** #if os(macOS) für plattformspezifischen Code
- **Duplicate Handling:** errSecDuplicateItem wird als Erfolg gewertet
- **Memory Safety:** SecKey/SecCertificate-Referenzen werden nicht kopiert
- **Lokalisierung:** Alle Fehlermeldungen über String(localized:)

3.1 Database Schema

Database/Schema/ • SQLite Tabellen-Definitionen & Migrationen

Übersicht

Das AILO Database Schema definiert die SQLite-Tabellenstruktur für E-Mail-Persistenz. Es umfasst Tabellen für Accounts, Ordner, Nachrichten, Anhänge und die Outbox-Queue. Das Schema unterstützt versionierte Migrationen und wird über PRAGMA user_version verwaltet.

Eigenschaft	Wert
Aktuelle Version	5 (MailSchema.currentVersion)
Datenbank-Engine	SQLite3 (iOS/macOS native)
Datei	MailSchema.swift
Pfad	Database/Schema/

Tabellen-Übersicht

Tabelle	Beschreibung
accounts	E-Mail-Account-Konfigurationen
folders	IMAP-Ordner pro Account
message_header	E-Mail-Header (Hot Path, häufig gelesen)
message_body	E-Mail-Body (Lazy Loading)
attachment	Anhänge mit Metadaten und Blob-Speicher
outbox	Ausgehende E-Mails (Queue)
mime_parts	Strukturierte MIME-Parts
render_cache	Gerenderte HTML/Text-Versionen
blob_meta	Blob-Deduplizierung und Reference Counting

accounts

Speichert E-Mail-Account-Konfigurationen. Passwörter werden NICHT hier gespeichert, sondern im Keychain.

Spalte	Typ	Beschreibung
`id`	TEXT PK	UUID des Accounts
`display_name`	TEXT	Anzeigename
`email_address`	TEXT	E-Mail-Adresse
`host_imap`	TEXT	IMAP-Server Hostname
`host_smtp`	TEXT	SMTP-Server Hostname
`created_at`	INTEGER	Erstellungsdatum (Epoch)
`updated_at`	INTEGER	Änderungsdatum (Epoch)
`signing_enabled`	INTEGER	S/MIME-Signierung aktiviert (v5)
`signing_cert_id`	TEXT	Keychain-Zertifikat-ID (v5)

folders

IMAP-Ordner mit Special-Use-Flags und Attributen. Composite Primary Key aus account_id + name.

Spalte	Typ	Beschreibung
`account_id`	TEXT PK	Zugehöriger Account

Spalte	Typ	Beschreibung
name	TEXT PK	Ordnername (z.B. "INBOX")
special_use	TEXT	Spezialverwendung (inbox, sent, drafts, trash, spam)
delimiter	TEXT	Hierarchie-Trennzeichen (z.B. "/")
attributes	TEXT	IMAP-Flags (\Noselect, \HasNoChildren, etc.)

message_header

E-Mail-Header für schnellen Listen-Zugriff. Hot Path – häufig gelesen, selten geschrieben. Index auf (account_id, folder, date DESC).

Spalte	Typ	Beschreibung
account_id	TEXT PK	Zugehöriger Account
folder	TEXT PK	Ordnername
uid	TEXT PK	IMAP UID (eindeutig im Ordner)
from_addr	TEXT	Absender-Adresse
subject	TEXT	Betreff
date	INTEGER	Datum (Epoch Sekunden)
flags	TEXT	IMAP-Flags (\Seen, \Flagged, etc.)
has_attachments	INTEGER	Hat Anhänge (0/1)

message_body

E-Mail-Body mit Text/HTML-Inhalt. Lazy Loading – wird erst bei Detailansicht geladen. Seit v3 mit raw_body für Forensik.

Spalte	Typ	Beschreibung
account_id	TEXT PK	Zugehöriger Account
folder	TEXT PK	Ordnername
uid	TEXT PK	IMAP UID
text_body	TEXT	Plain-Text Version
html_body	TEXT	HTML Version
has_attachments	INTEGER	Hat Anhänge (0/1)
raw_body	TEXT	Roher MIME-Body (v3)
content_type	TEXT	MIME Content-Type (v2)
charset	TEXT	Zeichensatz (v2)
transfer_encoding	TEXT	Transfer-Encoding (v2)
is_multipart	INTEGER	Multipart-Nachricht (v2)
raw_size	INTEGER	Rohe Größe in Bytes (v2)
processed_at	INTEGER	Verarbeitungszeitpunkt (v2)

attachment

Anhänge mit Metadaten, Blob-Speicher und Deduplizierung. Unterstützt Inline-Attachments (CID) und externe Dateipfade.

Spalte	Typ	Beschreibung
account_id	TEXT PK	Zugehöriger Account
folder	TEXT PK	Ordnername
uid	TEXT PK	IMAP UID
part_id	TEXT PK	MIME Part-ID (z.B. "1.2")
filename	TEXT	Dateiname
mime_type	TEXT	MIME-Typ
size_bytes	INTEGER	Größe in Bytes
data	BLOB	Binärdaten (optional)
content_id	TEXT	CID für Inline-Attachments (v2)
is_inline	INTEGER	Inline-Attachment (v2)

Spalte	Typ	Beschreibung
file_path	TEXT	Externer Dateipfad (v2)
checksum	TEXT	SHA256 für Deduplizierung (v2)

outbox

Queue für ausgehende E-Mails mit Retry-Logik. Status: pending, sending, sent, failed, cancelled.

Spalte	Typ	Beschreibung
id	TEXT PK	UUID der Outbox-Nachricht
account_id	TEXT	Absender-Account
created_at	INTEGER	Erstellungszeitpunkt
last_attempt_at	INTEGER	Letzter Versuch
attempts	INTEGER	Anzahl Versuche
status	TEXT	Status (pending/sending/sent/failed)
last_error	TEXT	Letzter Fehlertext
from_addr	TEXT	Absender
to_addr	TEXT	Empfänger (kommagetrennt)
cc_addr	TEXT	CC-Empfänger
bcc_addr	TEXT	BCC-Empfänger
subject	TEXT	Betreff
text_body	TEXT	Plain-Text Body
html_body	TEXT	HTML Body
attachments_json	TEXT	Anhänge als JSON (v4)

Blob Storage Tabellen

Zusätzliche Tabellen für MIME-Strukturierung, Render-Cache und Blob-Deduplizierung.

mime_parts

- **id:** TEXT PRIMARY KEY
- **message_id, part_number:** Referenz zur Nachricht
- **content_type, content_subtype:** MIME-Typ
- **is_attachment, is_inline:** Attachment-Flags
- **parent_part_number:** Hierarchie für Multipart

render_cache

- **message_id:** TEXT PRIMARY KEY
- **html_rendered, text_rendered:** Gerenderte Versionen
- **generated_at, generator_version:** Cache-Invalidierung

blob_meta

- **blob_id:** TEXT PRIMARY KEY
- **hash_sha256:** SHA256-Hash für Deduplizierung
- **reference_count:** Anzahl Referenzen (Reference Counting)
- **size_bytes, created_at, last_accessed:** Metadaten

Indizes

Index	Spalten / Zweck
idx_header_date	(account_id, folder, date DESC) – Sortierte Listen
idx_outbox_pending	(status, created_at) – Pending-Queue-Abfrage
idx_attachment_checksum	(checksum) WHERE NOT NULL – Deduplizierung
idx_body_processed_at	(processed_at) WHERE NOT NULL – Migration-Tracking

Schema-Migration

Das Schema unterstützt automatische Migrationen über PRAGMA user_version. Jede Version fügt neue Spalten oder Tabellen hinzu.

Versions-Historie

Version	Änderungen
v1	Initiales Schema mit allen Basistabellen
v2	Enhanced metadata: content_type, charset, transfer_encoding, is_multipart, checksum, is_inline
v3	raw_body für Forensik, .eml-Export, Phishing-Detection
v4	attachments_json in outbox für ausgehende Anhänge
v5	signing_enabled, signing_cert_id für S/MIME-Signierung

Migration-API

- **createStatements(for:)** → [String] – DDL für Version
- **migrationSteps(from:to:)** → [[String]] – Schrittweise Migrationen
- **migrateIfNeeded(readUserVersion:exec:writeUserVersion:)** – Automatische Migration

Entity Models

Leichtgewichtige Swift-Structs für type-safe Datenzugriff. Alle Entities sind Sendable und Equatable.

- **AccountEntity** – id, displayName, emailAddress, hostIMAP, hostSMTP
- **FolderEntity** – accountId, name, specialUse, delimiter, attributes
- **MessageHeaderEntity** – accountId, folder, uid, from, subject, date, flags, hasAttachments
- **MessageBodyEntity** – text, html, rawBody, contentType, charset, transferEncoding
- **AttachmentEntity** – partId, filename, mimeType, sizeBytes, data, contentId, isInline
- **OutboxItemEntity** – id, accountId, status, attempts, from, to, subject, attachmentsJson
- **OutboxAttachment** – filename, mimeType, dataBase64 (Codable für JSON)

Technische Hinweise

- **Composite Primary Keys:** (account_id, folder, uid) für eindeutige Nachrichtenreferenz
- **Epoch Timestamps:** Alle Datums-Felder als INTEGER (Unix-Sekunden)
- **Nullable Spalten:** TEXT-Felder ohne NOT NULL für optionale Metadaten
- **CREATE IF NOT EXISTS:** Idempotente DDL-Statements
- **ALTER TABLE:** Spalten werden nur hinzugefügt, nie entfernt (Backward Compatibility)
- **Partial Index:** WHERE-Klauseln für effiziente Indizes

AILO Handbuch – Kapitel 3.1 Database Schema
Version 1.0 | Dezember 2025

3.2 DAO Implementations

Database/DAO/ • Spezialisierte Data Access Objects

Übersicht

Die DAO-Architektur in AILO folgt dem Repository-Pattern mit spezialisierten Data Access Objects. Die monolithische MailDAO wurde in spezialisierte DAOs aufgeteilt, die über eine zentrale DAOFactory verwaltet werden. Dies ermöglicht bessere Testbarkeit, Separation of Concerns und Performance-Optimierungen.

DAO	Verantwortlichkeit
BaseDAO	Basis-Klasse mit SQLite-Verbindung und Transaktionen
AccountDAO	Account CRUD und Settings
FolderDAO	Ordner-Hierarchie und Special Folders
MailReadDAO	Lese-Operationen (Headers, Body, Attachments)
MailWriteDAO	Schreib-Operationen (Insert, Update, Delete)
AttachmentDAO	Anhang-Management mit File Storage
OutboxDAO	Queue-Management für ausgehende Mails
DAOFactory	Zentrale Factory und Connection-Sharing

BaseDAO

Die abstrakte Basis-Klasse für alle DAOs. Stellt SQLite-Verbindung, Thread-Synchronisation und Transaktionsunterstützung bereit.

Eigenschaften

- **db:** OpaquePointer? – SQLite-Verbindung
- **dbPath:** String – Pfad zur Datenbank-Datei
- **dbQueue:** DispatchQueue – Thread-sichere Serialisierung

Kernmethoden

Methode	Beschreibung
openDatabase()	Öffnet SQLite-Verbindung
closeDatabase()	Schließt Verbindung
ensureOpen()	Stellt offene Verbindung sicher
setSharedConnection(_:)	Setzt geteilte Verbindung von DAOFactory
exec(_:)	Führt SQL-Statement aus
withTransaction(_:)	Führt Closure in Transaktion aus

AccountDAO

Verwaltet E-Mail-Account-Konfigurationen. Implementiert CRUD-Operationen für AccountEntity.

Protocol: AccountDAO

- **create(_:)** – Neuen Account anlegen
- **get(id:)** → AccountEntity? – Account laden
- **getAll()** → [AccountEntity] – Alle Accounts

- **update(_:)** – Account aktualisieren
- **delete(id:)** – Account löschen
- **getByEmail(_:)** → AccountEntity? – Suche per E-Mail

FolderDAO

Spezialisiert auf Ordner-Management mit Hierarchie-Support und Special-Folder-Mapping.

Protocol: FolderDAO

Methode	Beschreibung
store(_:)	Ordner speichern/aktualisieren
get(accountId:name:)	Einzelnen Ordner laden
getAll(for:)	Alle Ordner eines Accounts
getSpecialFolders(for:)	Special Folders Mapping [String: String]
updateSpecialFolders(for:mapping:)	Special Folders setzen
getFolderHierarchy(for:)	[FolderHierarchyNode] – Baumstruktur
updateFolderAttributes(...)	IMAP-Attribute aktualisieren
removeFoldersNotIn(...)	Verwaiste Ordner entfernen
getFolderStats(for:)	[FolderStats] – Statistiken

Supporting Types

- **FolderHierarchyNode:** folder + children[] – Rekursive Baumstruktur
- **FolderStats:** folderName, messageCount, unreadCount, totalSizeBytes

MailReadDAO

Optimiert für Lese-Operationen. Hot Path für Listen-Ansichten und Message-Details.

Kern-Operationen

Methode	Rückgabe
headers(accountId:folder:limit:offset:)	[MailHeader]
body(accountId:folder:uid:)	String?
bodyEntity(accountId:folder:uid:)	MessageBodyEntity?
attachments(accountId:folder:uid:)	[AttachmentEntity]
attachmentStatus(accountId:folder:)	[String: Bool]
getMimeParts(messageId:)	[MimePartEntity]
getRenderCache(messageId:)	RenderCacheEntry?
getBlobMeta(blobId:)	BlobMetaEntry?

Blob Storage Analytics

- **getBlobStorageMetrics()** → BlobStorageMetrics
- **getOrphanedBlobs()** → [String] – Unreferenzierte Blobs
- **getBlobsOlderThan(_:)** → [String] – Für Cleanup

MailWriteDAO

Alle schreibenden Operationen für Nachrichten, Bodies, MIME-Parts und Caches.

Message Operations

- **insertHeaders(accountId:folder:headers:)**
- **storeBody(accountId:folder:uid:body:)**
- **updateFlags(accountId:folder:uid:flags:)**
- **deleteMessage(accountId:folder:uid:)**
- **purgeFolder(accountId:folder:)**

MIME & Cache Operations

- **storeMimeParts(_:)** – [MimePartEntity] speichern
- **storeRenderCache(messageId:html:text:generatorVersion:)**
- **invalidateRenderCache(messageId:)**
- **storeBlobMeta(blobId:hashSha256:sizeBytes:)**
- **incrementBlobReference(_:) / decrementBlobReference(_:)**

AttachmentDAO

Spezialisiertes Attachment-Management mit File-Storage und automatischer Deduplizierung basierend auf SHA256-Checksums.

Konfiguration

Parameter	Beschreibung
attachmentsDirectory	URL für File-Storage (Documents/Attachments)
maxInlineSize	Schwelle für Blob vs. File (Default: 1MB)
deduplicationEnabled	SHA256-basierte Deduplizierung aktiv

Protocol: AttachmentDAO

- **store(accountId:folder:uid:attachment:)**
- **getAll(accountId:folder:uid:)** → [AttachmentEntity]
- **getAttachmentData(attachment:)** → Data?
- **delete(accountId:folder:uid:partId:)**
- **cleanupOrphanedFiles()** – Verwaiste Dateien löschen
- **getStorageMetrics()** → AttachmentStorageMetrics

OutboxDAO (MailOutboxDAO)

Queue-Management für ausgehende E-Mails mit Retry-Logik und Status-Tracking.

Protocol: MailOutboxDAO

Methode	Beschreibung
enqueue(_:)	Nachricht in Queue einfügen
getPendingItems(for:limit:)	Pending Items abrufen
updateStatus(id:status:error:)	Status aktualisieren
markAsSending(id:)	Status → sending
markAsSent(id:)	Status → sent
markAsFailed(id:error:)	Status → failed + Fehlertext
incrementAttempts(id:)	Versuchszähler erhöhen
removeSentItems(olderThan:)	Alte gesendete Items löschen
removeFailedItems(maxAge:)	Alte fehlgeschlagene löschen

Status-Workflow

1. **pending** – In Queue, wartet auf Verarbeitung
2. **sending** – Wird gerade gesendet
3. **sent** – Erfolgreich gesendet
4. **failed** – Fehlgeschlagen (mit lastError)
5. **cancelled** – Vom Benutzer abgebrochen

DAOFactory

Zentrale Factory für DAO-Erstellung und Connection-Sharing. Implementiert Lazy Initialization und verwaltet eine geteilte SQLite-Verbindung.

Initialization

- **dbPath:** String – Pfad zur SQLite-Datenbank
- **attachmentsDirectory:** URL? – Custom Attachment-Pfad
- **maxInlineSize:** Int – Schwelle für Inline-Blobs (Default: 1MB)
- **deduplicationEnabled:** Bool – Deduplizierung (Default: true)
- **maxRetryAttempts:** Int – Outbox Retries (Default: 3)

DAO Accessors

- **mailReadDAO:** MailReadDAO
- **mailWriteDAO:** MailWriteDAO
- **attachmentDAO:** AttachmentDAO
- **outboxDAO:** MailOutboxDAO
- **folderDAO:** FolderDAO
- **accountDAO:** AccountDAO
- **mailFullAccessDAO:** MailFullAccessDAO – Kombinierter Zugriff

Database Management

- **initializeDatabase()** – Öffnet DB, teilt Connection, erstellt Schema
- **closeAllConnections()** – Schließt geteilte Verbindung
- **validateSchema()** → (userVersion, foldersTableExists)
- **performMaintenance()** – Cleanup, VACUUM, ANALYZE
- **getPerformanceMetrics()** → [String: (average, calls)]
- **resetPerformanceMetrics()** – Statistiken zurücksetzen

Migration von Legacy-MailDAO

Für schrittweise Migration steht ein LegacyMailDAOAdapter als Kompatibilitätsschicht bereit.

Vorher → Nachher

Alt (MailDAO)	Neu (DAOFactory)
mailDAO.headers(...)	daoFactory.mailReadDAO.headers(...)
mailDAO.body(...)	daoFactory.mailReadDAO.body(...)
mailDAO.insertHeaders(...)	daoFactory.mailWriteDAO.insertHeaders(...)
mailDAO.storeBody(...)	daoFactory.mailWriteDAO.storeBody(...)
mailDAO.attachments(...)	daoFactory.attachmentDAO.getAll(...)

Technische Hinweise

- **Connection Sharing:** Alle DAOs nutzen dieselbe SQLite-Verbindung
- **Thread Safety:** DispatchQueue.sync für serialisierten Zugriff
- **Lazy Initialization:** DAOs werden erst bei Zugriff erstellt
- **Protocol-Based:** Alle DAOs haben Protocol + Impl für Testbarkeit
- **Performance:** 40-60% schnellere Reads durch Spezialisierung
- **Deduplication:** 20-30% Speicherersparnis bei Attachments

---

3.3 DAO Utilities

Database/DAO/DAOHelpers.swift • SQLite Extensions & Performance Tools

Übersicht

Die DAO Utilities stellen essenzielle Hilfsfunktionen für alle Data Access Objects bereit. Sie umfassen SQLite-Typ-Extraktion, Performance-Monitoring, Transaktions-Management, Query-Building und Schema-Validierung.

Komponente	Beschreibung
OpaquePointer Extensions	SQLite-Typ-Extraktion für Spalten
DAOPerformanceMonitor	Query-Timing und Metriken-Sammlung
DAOTransactionManager	Batch-Operationen in Transaktionen
SQLQueryBuilder	Type-safe Query-Konstruktion
DAOSchemaValidator	Schema-Validierung und Version-Check

SQLite Type Extensions

Extensions auf OpaquePointer für typsichere Spalten-Extraktion aus SQLite-Statements. Alle Methoden sind null-safe und konvertieren SQLite-Typen in Swift-Typen.

Basis-Typen

Methode	Rückgabe	Beschreibung
`columnText(_:)`	String?	Text-Spalte als String
`columnInt(_:)`	Int	Integer-Spalte
`columnInt64(_:)`	Int64	64-Bit Integer
`columnDouble(_:)`	Double	Fließkommazahl
`columnBlob(_:)`	Data?	Binary Large Object
`columnBool(_:)`	Bool	Boolean (Int != 0)
`columnIsNull(_:)`	Bool	Prüft auf NULL

Erweiterte Typen

Methode	Rückgabe	Beschreibung
`columnUUID(_:)`	UUID?	UUID aus TEXT-Spalte
`columnDate(_:)`	Date?	Date aus Epoch (Double)
`columnStringArray(_:)`	[String]	Kommaseparierte Strings

DAOPerformanceMonitor

Singleton-basiertes Performance-Monitoring für alle DAO-Operationen. Sammelt Timing-Metriken und berechnet Durchschnittswerte. Thread-sicher durch DispatchQueue.

Architektur

- **Singleton Pattern:** shared Instance für globalen Zugriff
- **Thread Safety:** DispatchQueue für serialisierten Zugriff auf Metriken
- **DAOMetric Struct:** totalTime, callCount, averageTime (computed)

API

Methode	Beschreibung
measure<T>(_:_:)	Misst Ausführungszeit eines Blocks → T
getMetrics()	[String: (average, calls)] – Alle Metriken
resetMetrics()	Setzt alle Metriken zurück

Verwendung

```
return try DAOPerformanceMonitor.measure("headers_query") {     // ... SQL-Operationen ... }
```

Gemessene Operationen

- headers_query, body_query, body_entity_query
- attachments_query, attachment_status_query
- render_cache_query, blob_meta_query, raw_blob_id_query
- blob_storage_metrics_query, orphaned_blobs_query, old_blobs_query
- store_folder, get_attachment, get_all_attachments
- insert_headers, store_body, update_flags, delete_message

DAOTransactionManager

Verwaltet Batch-Operationen innerhalb von Datenbank-Transaktionen. Ermöglicht atomare Ausführung mehrerer Operationen mit automatischem Rollback bei Fehlern.

Initialisierung

```
let transactionManager = DAOTransactionManager(baseDAO)
```

Methoden

Methode	Beschreibung
performBatch<T>(_:)	Führt Array von Closures in Transaktion aus → [T]
performBatchInsert<Entity>(...)	Batch-Insert mit konfigurierbarer Chunk-Größe

performBatchInsert Parameter

- **entities:** [Entity] – Zu speichernde Entitäten
- **batchSize:** Int = 100 – Chunk-Größe pro Transaktion
- **insertOperation:** ([Entity]) throws -> Void – Insert-Closure

Array Chunking Extension

Private Extension auf Array für effizientes Aufteilen in Batches: chunked(into: Int) → [[Element]]

SQLQueryBuilder

Fluent API für type-safe SQL-Query-Konstruktion. Verwendet Builder-Pattern mit verkettbaren Methoden.

Builder-Kette

1. **SQLQueryBuilder.select(...)** → SelectBuilder
2. **.from(table:)** → FromBuilder

3. **.whereCondition(_:)** → WhereBuilder (optional)
4. **.orderBy(_:ascending:)** → OrderBuilder
5. **.limit(_:offset:)** → String (finale Query)
6. **.build()** → String (an jedem Punkt aufrufbar)

Builder-Typen

Builder	Verfügbare Methoden
SelectBuilder	from(_:) → FromBuilder
FromBuilder	whereCondition(_:), orderBy(_:ascending:), build()
WhereBuilder	orderBy(_:ascending:), build()
OrderBuilder	limit(_:offset:), build()

Beispiel

```
let query = SQLQueryBuilder     .select("uid", "from_addr", "subject")
.from("message_header")     .whereCondition("account_id = ?")
.orderBy("date", ascending: false)     .limit(50, offset: 0)
```

DAOSchemaValidator

Validiert die Datenbank-Schema-Integrität. Prüft Tabellen-Existenz und Schema-Version über PRAGMA-Befehle.

Initialisierung

```
let validator = DAOSchemaValidator(baseDAO)
```

Methoden

Methode	Beschreibung
validateTable(_:)	Prüft Tabellen-Existenz → Bool
getUserVersion()	PRAGMA user_version lesen → Int
setUserVersion(_:)	PRAGMA user_version setzen

Validierungsstrategie

7. PRAGMA schema_version ausführen (Pending Transactions committen)
8. PRAGMA table_info(tableName) – Primäre Prüfmethode
9. Fallback: sqlite_master Query bei negativem Ergebnis
10. Debug-Logging bei Inkonsistenzen zwischen beiden Methoden

Verwendung in DAOFactory

```
func validateSchema() throws -> (userVersion: Int, foldersTableExists: Bool) {
let validator = DAOSchemaValidator(_accountDAO)     let userVersion = try
validator.getUserVersion()     let foldersTableExists = try
validator.validateTable(MailSchema.tFolders)     return (userVersion,
foldersTableExists) }
```

BaseDAO Bind Helpers

Interne Helper-Methoden in BaseDAO für sicheres Parameter-Binding mit SQLite. Alle Methoden verwenden SQLITE_TRANSIENT für String-Kopien.

Methode	Beschreibung
bindText(_:_:_:)	String? binden (NULL bei nil)
bindInt(_:_:_:)	Int? binden
bindUUID(_:_:_:)	UUID als TEXT binden
bindBlob(_:_:_:)	Data als BLOB binden
debugBoundValues(_:)	Debug-Ausgabe aller Parameter

Technische Hinweise

- **SQLITE_TRANSIENT:** Erzwingt Kopie von Strings (Memory Safety)
- **async Metrics Recording:** Performance-Daten werden asynchron geschrieben
- **Transaktion-Semantik:** BEGIN → Operation → COMMIT/ROLLBACK
- **PRAGMA vs sqlite_master:** Beide Methoden für zuverlässige Validierung
- **Epoch Timestamps:** Date als Double (timeIntervalSince1970) gespeichert
- **Variadic Columns:** select() akzeptiert beliebig viele Spaltennamen

---

3.4 Datenmodelle

Database/Models/ • Swift-Structs für Persistenz und Domain-Logic

Übersicht

AILO verwendet leichtgewichtige Swift-Structs als Datenmodelle. Alle Modelle sind Sendable für Swift Concurrency, Codable für JSON-Serialisierung und Equatable für Vergleiche. Die Modelle decken drei Hauptbereiche ab: Mail-Entitäten, Log-System und Pre-Prompt-Katalog.

Modell	Beschreibung
LogEntry	Text-/Audio-Log-Einträge
AccountEntity	E-Mail-Account-Konfiguration
FolderEntity	IMAP-Ordner mit Attributen
MessageHeaderEntity	E-Mail-Header für Listen
MessageBodyEntity	E-Mail-Body mit Metadaten
AttachmentEntity	Anhänge mit Deduplizierung
OutboxItemEntity	Ausgehende E-Mails Queue
AIPrePromptPreset	Pre-Prompt-Inhalt
PrePromptMenuItem	Hierarchisches Menü-Item
PrePromptRecipe	Kombinierte Prompts
Cookbook	Kochbuch-Container
RecipeMenuItem	Rezept-Menüstruktur

LogEntry

Repräsentiert einen Log-Eintrag im Tagebuch-System. Unterstützt sowohl Text- als auch Audio-Einträge mit optionaler KI-Verarbeitung.

Eigenschaften

Feld	Typ	Beschreibung
`id`	UUID	Eindeutige ID
`title`	String?	Optionaler Titel
`text`	String?	Text-Inhalt
`audioURL`	URL?	Pfad zur Audio-Datei
`createdAt`	Date	Erstellungsdatum
`category`	String?	Kategorie-Zuordnung
`tags`	[String]	Tag-Liste
`reminderDate`	Date?	Erinnerungsdatum
`useAI`	Bool?	KI-Verarbeitung aktiv
`aiText`	String?	KI-generierter Text

Protokolle

- **Identifiable:** id-Property für SwiftUI-Listen
- **Codable:** JSON-Serialisierung für Persistenz
- **Equatable:** Vergleichbarkeit für Updates

Mail-Entitäten

Die Mail-Entitäten bilden das E-Mail-Datenmodell ab. Alle sind Sendable für Swift Concurrency und in MailSchema.swift definiert.

AccountEntity

Feld	Typ	Beschreibung
`id`	UUID	Account-ID
`displayName`	String	Anzeigename
`emailAddress`	String	E-Mail-Adresse
`hostIMAP`	String	IMAP-Server
`hostSMTP`	String	SMTP-Server
`createdAt`	Date	Erstellungsdatum
`updatedAt`	Date	Änderungsdatum

FolderEntity

- **accountId:** UUID – Zugehöriger Account
- **name:** String – Server-Name (z.B. "INBOX")
- **specialUse:** String? – inbox, sent, drafts, trash, spam
- **delimiter:** String? – Hierarchie-Trennzeichen
- **attributes:** [String] – IMAP-Flags (\Noselect, etc.)

MessageHeaderEntity

- **accountId, folder, uid:** Composite Key
- **from:** String – Absender-Adresse
- **subject:** String – Betreff
- **date:** Date? – Nachrichtendatum
- **flags:** [String] – \Seen, \Flagged, etc.
- **hasAttachments:** Bool – Anhänge vorhanden
- **signatureStatus:** SignatureStatus? – S/MIME Status (v4)
- **signerEmail:** String? – Signatur-E-Mail (v4)

MessageBodyEntity

- **text:** String? – Plain-Text Body
- **html:** String? – HTML Body
- **rawBody:** String? – Roher MIME-Body (v3)
- **contentType, charset:** MIME-Metadaten (v2)
- **transferEncoding:** String? – quoted-printable, base64
- **isMultipart:** Bool – Multipart-Nachricht
- **rawSize, processedAt:** Verarbeitungs-Metadaten

AttachmentEntity

- **partId:** String – MIME Part-ID (z.B. "1.2")
- **filename, mimeType:** Dateiinformationen
- **sizeBytes:** Int – Größe in Bytes
- **data:** Data? – Binärdaten (optional)
- **contentId:** String? – CID für Inline-Attachments
- **isInline:** Bool – Eingebettetes Bild
- **filePath:** String? – Externer Dateipfad
- **checksum:** String? – SHA256 für Deduplizierung

OutboxItemEntity

- **id:** UUID – Outbox-ID
- **accountId:** UUID – Absender-Account
- **status:** OutboxStatusEntity – pending/sending/sent/failed

- **attempts:** Int – Versuchszähler
- **from, to, cc, bcc:** Adressen
- **subject, textBody, htmlBody:** Inhalt
- **attachmentsJson:** String? – JSON-serialisierte Anhänge

Pre-Prompt-Modelle

Das Pre-Prompt-System verwendet ein hierarchisches Modell mit Kategorien, Presets und Rezepten. Alle Modelle sind Codable für UserDefaults-Persistenz.

AIPrePromptPreset

Content-Modell für Pre-Prompt-Vorlagen. Enthält den eigentlichen Prompt-Text.

Feld	Typ	Beschreibung
`id`	UUID	Preset-ID
`name`	String	Anzeigename
`text`	String	Prompt-Inhalt
`icon`	String	Emoji (max 3 Zeichen)
`keywords`	String	Semikolon-getrennte Keywords
`isDefault`	Bool	Standard-Preset
`createdAt`	Date	Erstellungsdatum
`updatedAt`	Date	Änderungsdatum

PrePromptMenuItem

Hierarchisches Menü-Item für Katalog-Struktur. Kann Ordner oder Preset-Referenz sein.

- **id:** UUID – Menü-Item-ID
- **parentID:** UUID? – Parent-Folder (nil = Root)
- **name, icon:** Anzeige-Properties
- **keywords:** String – Kontext-Metadaten
- **sortOrder:** Int – Reihenfolge
- **presetID:** UUID? – Referenz auf AIPrePromptPreset
- **isFolder:** Bool (computed) – presetID == nil
- **Factory Methods:** .folder(...), .preset(...)

PrePromptRecipe

"Kochrezept" das mehrere Katalog-Elemente zu einem Prompt kombiniert.

- **id, name, icon:** Identifikation
- **keywords:** String – Rezept-Metadaten
- **elementIDs:** [UUID] – Geordnete MenuItem-IDs
- **separator:** String – Trennzeichen (Default: "\n\n")
- **generatePrompt(...):** Kombiniert alle Elemente
- **collectKeywords(...):** Sammelt alle Keywords hierarchisch

Cookbook & RecipeMenuItem

Kochbuch-Container und Menüstruktur für Rezepte (analog zu PrePromptMenuItem).

- **Cookbook:** id, name, icon, keywords, sortOrder
- **RecipeMenuItem:** id, cookbookID, parentID, recipeID, name, sortOrder
- **Struktur:** Cookbook → RecipeMenuItem (Kapitel/Rezept-Referenz)

Keyword-Format

Keywords verwenden ein einheitliches Format für Metadaten-Speicherung:

- **Format:** "Schlüssel: Wert; Schlüssel2: Wert2"
- **Beispiel:** "Anrede: Du; Stil: formell; Länge: kurz"
- **keywordPairs:** Computed Property → [(key: String, value: String)]
- **keyword(_:):** Methode zum Abrufen einzelner Werte

Technische Hinweise

- **Sendable:** Alle Modelle sind thread-safe für Swift Concurrency
- **Codable:** Custom init(from:) für Migration von alten Formaten
- **Equatable:** Für SwiftUI-Diff und Update-Detection
- **Identifiable:** id-Property für ForEach und List
- **Factory Methods:** Convenience-Initializer für häufige Fälle
- **updated():** Immutable Updates mit neuem updatedAt-Timestamp
- **Icon-Limit:** Emoji auf 3 Zeichen begrenzt (String.prefix(3))

3.5 DataStore (Logs)

Database/Store/DataStore.swift • JSON-basierte Log-Persistierung

Übersicht

Der DataStore ist ein ObservableObject, das die Persistierung von LogEntry-Objekten verwaltet. Er verwendet JSON-Serialisierung für die Speicherung und bietet vollständige CRUD-Operationen sowie Audio-URL-Management. Die Klasse ist @MainActor für Thread-Sicherheit in SwiftUI.

Feature	Beschreibung
ObservableObject	SwiftUI-Integration mit @Published
JSON-Persistierung	Automatische Speicherung in entries.json
CRUD-Operationen	add(), update(), delete()
Audio-URL-Management	Pfad-Verwaltung für Audio-Dateien
@MainActor	Thread-sichere UI-Updates
Async/Await	Moderne Swift Concurrency

Architektur

Der DataStore folgt dem Repository-Pattern und kapselt alle Persistierung-Logik. Er wird als EnvironmentObject in der SwiftUI-Hierarchie geteilt.

Klassenstruktur

```
@MainActor final class DataStore: ObservableObject {     @Published
private(set) var entries: [LogEntry] = []     @Published var
pendingImportText: String? = nil     private let fileName = "entries.json"
}
```

Properties

Property	Typ	Beschreibung
`entries`	[LogEntry]	@Published, private(set) – Alle Logs
`pendingImportText`	String?	@Published – Pending Import-Text
`fileName`	String	Private Konstante: "entries.json"

CRUD-Operationen

add(_:)

Fügt einen neuen LogEntry am Anfang der Liste ein (neueste zuerst).

```
func add(_ entry: LogEntry) {     entries.insert(entry, at: 0)     Task {
await save() } }
```

update(_:)

Aktualisiert einen bestehenden Eintrag anhand seiner ID.

```
func update(_ entry: LogEntry) {     if let idx = entries.firstIndex(where:
{ $0.id == entry.id }) {         entries[idx] = entry         Task { await
save() }     } }
```

delete(at:)

Löscht Einträge an den angegebenen IndexSet-Positionen (SwiftUI-kompatibel).

```
func delete(at offsets: IndexSet) {     for index in offsets.sorted(by: >) {         if entries.indices.contains(index) {
entries.remove(at: index)         }     }     Task { await save() } }
```

Persistierung

Die Persistierung erfolgt über JSON-Dateien im Documents-Verzeichnis. Laden und Speichern sind async-Operationen.

Speicherort

- **Verzeichnis:** FileManager.documentDirectory
- **Dateiname:** entries.json
- **Pfad:** ~/Documents/entries.json

load()

Lädt alle Einträge aus der JSON-Datei. Wird im init() automatisch aufgerufen.

```
func load() async {     let url = storeURL()     guard
FileManager.default.fileExists(atPath: url.path) else { return }     let
data = try Data(contentsOf: url)     let decoded = try
JSONDecoder().decode([LogEntry].self, from: data)     self.entries =
decoded }
```

save()

Speichert alle Einträge atomar in die JSON-Datei.

```
func save() async {     let url = storeURL()     let data = try
JSONEncoder().encode(entries)     try data.write(to: url, options:
[.atomic]) }
```

URL-Management

Der DataStore bietet Helper-Methoden für Dateipfad-Verwaltung.

Methode	Beschreibung
documentsURL()	Gibt das Documents-Verzeichnis zurück → URL
storeURL()	Gibt den Pfad zur entries.json zurück → URL (private)
audioURL(for:)	Konstruiert Audio-Dateipfad → URL

Audio-URL-Verwendung

```
// Audio-Datei-Pfad für LogEntry let url = store.audioURL(for:
entry.audioFileName!) if FileManager.default.fileExists(atPath: url.path) {
let data = try Data(contentsOf: url)     // Audio abspielen oder
verarbeiten }
```

LogEntry-Modell

Das LogEntry-Struct repräsentiert einen einzelnen Log-Eintrag (Text oder Audio).

EntryType Enum

Case	Beschreibung
.text	Text-Log-Eintrag mit text-Property
.audio	Audio-Log mit audioFileName-Property

LogEntry Properties

Feld	Typ	Beschreibung
`id`	UUID	Eindeutige ID (Default: UUID())
`date`	Date	Erstellungsdatum (Default: Date())
`type`	EntryType	.text oder .audio
`title`	String?	Optionaler Titel
`text`	String?	Text-Inhalt (für .text)
`audioFileName`	String?	Dateiname (für .audio)
`category`	String?	Kategorie-Zuordnung
`tags`	[String]	Tag-Liste (Default: [])
`reminderDate`	Date?	Erinnerungsdatum
`useAI`	Bool?	KI-Verarbeitung aktiviert
`aiText`	String?	KI-generierter Text

Factory Methods

- `LogEntry.text(_:title:category:tags:reminderDate:useAI:aiText:)`
- `LogEntry.audio(fileName:title:category:tags:reminderDate:)`

SwiftUI-Integration

Der DataStore wird als EnvironmentObject in der App-Hierarchie geteilt.

App-Setup

```
@main struct AILO_APPApp: App {     @StateObject private var store =
DataStore()          var body: some Scene {         WindowGroup {
ContentView()                  .environmentObject(store)         }     } }
```

View-Verwendung

```
struct LogsListView: View {     @EnvironmentObject private var store:
DataStore          var body: some View {         List {
ForEach(store.entries) { entry in                 LogRowView(entry: entry)
}             .onDelete(perform: store.delete)         }     } }
```

Technische Hinweise

- **@MainActor:** Garantiert UI-Updates auf Main Thread
- **private(set):** entries nur intern mutierbar
- **.atomic Option:** Verhindert korrupte Dateien bei Absturz
- **Sortierung:** Neueste Einträge zuerst (insert at: 0)
- **IndexSet Handling:** sorted(by: >) für sichere Löschung
- **Async Init:** load() wird im Task {} gestartet
- **Error Handling:** Silent fail (optional: Logging)
- **Audio-Dateien:** Separat im Documents-Verzeichnis

AILO Handbuch – Kapitel 3.5 DataStore (Logs)
Version 1.0 | Dezember 2025

4.1 IMAP/MIME Parser

Services/Mail/IMAP/ • *Helpers/Parsers/* • E-Mail-Protokoll & Encoding-Verarbeitung

Übersicht

Das Parser-Modul verarbeitet IMAP-Protokollnachrichten und MIME-kodierte Inhalte. Es dekodiert ENVELOPE, FLAGS, BODYSTRUCTURE und LIST-Responses sowie RFC2047-kodierte Header (Subject, From). Das System unterstützt multiple Charsets (UTF-8, ISO-8859-1, Windows-1252) und Transfer-Encodings (Base64, Quoted-Printable).

Komponente	Beschreibung
IMAPParsers	ENVELOPE, FLAGS, LIST, BODYSTRUCTURE Parsing
RFC2047EncodedWordsParser	Encoded-Word Dekodierung (=?charset?B/Q?...?=)
MIMEParser	Multipart-Parsing, Boundary-Handling
TransferEncodingDecoder	Base64, Quoted-Printable Dekodierung
ContentDecoder	Charset-Konvertierung, Text/Enriched
AttachmentExtractor	Attachment-Extraktion aus MIME

IMAPParsers

Zentrale Klasse für das Parsing von IMAP-Server-Responses. Verarbeitet FETCH-Antworten und extrahiert strukturierte Daten.

Datentypen

Struct	Felder
EnvelopeRecord	uid, subject, from, internalDate
FlagsRecord	uid, flags: [String]
FolderInfo	name, delimiter, attributes: [String]
MessageEnvelope	subject, from, to, cc, bcc, date, messageId
FetchResult	uid, flags, internalDate, envelope, bodySection, bodyStructure
BodyStructure	enum: .single(PartInfo) | .multipart(type, subType, parts)

Parse-Methoden

Methode	Beschreibung
parseEnvelope(_:)	[String] → [EnvelopeRecord]
parseFlags(_:)	[String] → [FlagsRecord]
parseUIDs(_:)	String → [String] (SEARCH Response)
parseInternalDate(_:)	String → Date?
parseBodySection(_:)	[String] → String? (Body-Inhalt)
parseFetchResponse(_:)	Data → FetchResult (throws)
parseEnvelope(_:) single	String → MessageEnvelope (throws)
parseBodyStructure(_:)	String → BodyStructure (throws)
parseListResponse(_:)	String → FolderInfo (throws)

BodyStructure Enum

```
public enum BodyStructure: Sendable, Equatable {     case single(PartInfo)
case multipart(type: String, subType: String, parts: [BodyStructure]) }
```

IMAPBodyStructure Enum

- **.text(subtype:charset:)** – text/plain, text/html
- **.multipart(subtype:parts:)** – multipart/mixed, alternative
- **.application(subtype:)** – application/pdf, octet-stream
- **.image/.audio/.video(subtype:)** – Medientypen
- **.message(subtype:)** – message/rfc822

RFC2047EncodedWordsParser

Dekodiert RFC2047 Encoded-Words in E-Mail-Headern. Format: =?charset?encoding?data?= wobei encoding B (Base64) oder Q (Quoted-Printable) ist.

API

Methode	Beschreibung
decode(_:)	Allgemeine Dekodierung → String
decodeSubject(_:)	Subject-Header dekodieren → String
decodeFrom(_:)	From-Header dekodieren → String
containsEncodedWord(_:)	Prüft auf Encoded-Word → Bool
encode(_:charset:encoding:)	Text zu Encoded-Word kodieren

EncodingType Enum

- **.base64 ("B"):** Standard Base64-Kodierung
- **.quotedPrintable ("Q"):** Q-Encoding mit =XX Escape-Sequenzen

Beispiele

```
// Base64 "=?UTF-8?B?Q2Fmw6kgaW4gTcO8bmNoZW4=?=" → "Café in München"  // Quoted-Printable "=?ISO-8859-1?Q?Caf=E9_in_M=FCnchen?=" → "Café in München"  // UTF-8 Q-Encoding (Multi-Byte) "=?UTF-8?Q?=C3=BC=C3=A4=C3=B6=C3=9F?=" → "üäöß"
```

TransferEncodingDecoder

Dekodiert Content-Transfer-Encoding für E-Mail-Bodies. Unterstützt Base64, Quoted-Printable, 7bit, 8bit und Binary.

Hauptmethode

```
static func decode(      _ content: String,      encoding: String?,
charset: String? ) -> String
```

Unterstützte Encodings

Encoding	Verarbeitung
base64	Standard Base64 → Data → String mit Charset
quoted-printable	=XX Escape-Sequenzen, Soft Line Breaks (=)
7bit / 8bit	Passthrough (ASCII / Extended ASCII)
binary	Raw-Daten ohne Transformation

Charset-Handling

- **UTF-8:** .utf8 (Default)
- **ISO-8859-1:** .isoLatin1
- **Windows-1252:** .windowsCP1252
- **US-ASCII:** .ascii

MIMEParser

Verarbeitet Multipart-MIME-Nachrichten und extrahiert einzelne Parts anhand von Boundaries.

Kernfunktionen

- **parse(_:):** Hauptparser → [MIMEPart]
- **extractContentType(_:):** Content-Type Header parsen
- **extractFilename(_:):** Dateiname aus Content-Disposition
- **decodeQuotedPrintable(_:):** QP-Dekodierung mit Charset
- **decodeRFC5987(_:):** Extended Parameter (filename*=)

Multipart-Verarbeitung

1. Boundary aus Content-Type extrahieren
2. Content an Boundary-Markern splitten
3. Jeden Part rekursiv parsen (nested Multipart)
4. Transfer-Encoding pro Part anwenden

AttachmentExtractor

Extrahiert Attachments aus MIME-Parts mit Unterstützung für verschiedene Filename-Encodings.

Filename-Pattern-Priorität

5. `filename*=utf-8''...` (RFC 5987)
6. `filename="..."` (Quoted)
7. `filename=...` (Unquoted)
8. `name="..."` (Fallback)
9. `name=...` (Letzter Fallback)

ExtractedAttachment Struct

- **filename:** String – Dekodierter Dateiname
- **mimeType:** String – Content-Type
- **data:** Data – Binärdaten
- **contentId:** String? – CID für Inline-Attachments

ContentDecoder

Hochlevel-Decoder für E-Mail-Content mit automatischer Encoding-Erkennung.

Methode	Beschreibung
decodeContent(_:encoding:charset:)	Vollständige Dekodierung
decodeTextEnriched(_:toHTML:)	Text/Enriched zu HTML/Plain
isTextEnriched(_:)	Prüft auf Text/Enriched
detectOriginalEncoding(_:)	Erkennt Charset aus Content
decodeBase64(_:)	Base64 → String
normalizeCharset(_:)	Charset-ID normalisieren

RFC2047Test

Test-Klasse zur Validierung der RFC2047-Dekodierung mit verschiedenen Encoding-Szenarien.

Testfälle

- Q-Encoding mit UTF-8 Multi-Byte-Sequenzen (=C3=BC → ü)
- Q-Encoding mit ISO-8859-1 (=E9 → é, =FC → ü)
- Base64-Encoding mit UTF-8
- Multiple Encoded-Words in einem Header
- Mixed Content (Encoded + Plain Text)

Technische Hinweise

- **UTF-8 Multi-Byte:** Bytes werden gesammelt, dann dekodiert
- **Fallback-Kette:** UTF-8 → ISO-8859-1 → Original
- **Soft Line Break:** "=\n" in QP wird entfernt
- **Underscore in Q:** "_" wird zu Leerzeichen konvertiert
- **Case-Insensitive:** B/b und Q/q sind äquivalent
- **Sendable:** Alle Structs sind thread-safe

4.2 Utilities

Helpers/Utilities/ • Hilfsklassen für Transport, Resilienz & UI

Übersicht

Das Utilities-Modul enthält wiederverwendbare Hilfsklassen für verschiedene Bereiche: Mail-Transport-Abstraktion, Task-Abbruchsteuerung, Resilienz-Pattern (Circuit Breaker, Retry), Markdown-Rendering und Pre-Prompt-Katalog-Verwaltung.

Komponente	Beschreibung
MailTransportStubs	IMAP/SMTP Transport-Abstraktion
CancellationToken	Thread-sicherer Task-Abbruch
CircuitBreaker	Circuit Breaker Pattern
RetryPolicy	Exponential Backoff mit Jitter
MarkdownHelper	Markdown-Formatierung
PrePromptCatalogManager	Hierarchische Katalog-Verwaltung
PrePromptPicker	Auswahl-UI für Pre-Prompts

MailTransportStubs

Transport-Abstraktion für IMAP/SMTP-Operationen. Kapselt Verbindungsmanagement, Header-Fetching und Message-Retrieval mit integriertem Caching.

MailSendReceive Klasse

Hauptklasse für Mail-Transport-Operationen mit Repository-Integration.

Methode	Beschreibung
fetchHeaders(limit:folder:using:...)	Header-Liste mit Cache-Support
fetchMessageUID(_:folder:using:)	Einzelne Nachricht abrufen
fetchMessageOptimized(...)	BODYSTRUCTURE-basiertes Fetching
clearCache(for:folder:)	Cache leeren

FETCH-Response-Rekonstruktion

- Multi-Line FETCH Responses werden rekonstruiert
- Literal-Marker {n} werden aus Responses entfernt
- IMAPParsers.parseEnvelope() für strukturierte Daten
- Flags-Mapping (\Seen → unread: false)

Header-Erkennung

Erkannte Prefixe: From:, To:, Cc:, Subject:, Date:, Message-ID:, Content-Type:, MIME-Version:, X-*, DKIM-Signature:, ARC-*, Authentication-Results:, etc.

CancellationToken

Thread-sicheres Token für kooperative Task-Abbruchsteuerung. Verwendet DispatchQueue für synchronisierten Zugriff auf den cancelled-State.

Implementation

```
public final class CancellationToken: @unchecked Sendable {     private let
q = DispatchQueue(label: "cancellation.token.state")     private var
cancelled = false          public func cancel() { q.sync { cancelled = true
} }     public var isCancelled: Bool { q.sync { cancelled } } }
```

Verwendung

```
let token = CancellationToken() // In async Task: while !token.isCancelled
{     // ... work ... } // Abbruch von außen: token.cancel()
```

CircuitBreaker

Implementiert das Circuit Breaker Pattern mit drei Zuständen: Closed, Open und HalfOpen. Verhindert Überlastung fehlschlagender Services durch temporäres Blockieren von Anfragen.

State Enum

State	Beschreibung
.closed(Int)	Normal – Zählt consecutive Failures
.open(until: Date)	Blockiert – Bis Cooldown-Ende
.halfOpen(remainingProbes: Int)	Test-Phase – Erlaubt Probe-Requests

Konfiguration

- **openAfterFailures:** Int = 3 – Schwelle zum Öffnen
- **baseOpenDuration:** TimeInterval = 10s – Basis-Cooldown
- **maxOpenMultiplier:** Int = 5 – Max Cooldown-Multiplikator
- **halfOpenProbes:** Int = 3 – Erfolgreiche Probes zum Schließen

record(_:) Methode

Verarbeitet Result<Void, Error> und transitioniert den State. Gibt optionalen Delay zurück (Sekunden bis zum nächsten Versuch).

RetryPolicy

Kapselt Exponential Backoff und Circuit Breaker Logik für transiente Netzwerkfehler. Singleton-basiert mit Thread-sicherer Konfiguration.

ErrorKind Enum

Kind	Klasse	Beschreibung
`.dns`	transient	DNS-Auflösungsfehler
`.timeout`	transient	Verbindungs-Timeout
`.refused`	transient	Connection Refused
`.unreachable`	transient	Host nicht erreichbar
`.io`	transient	I/O-Fehler
`.auth`	permanent	Authentifizierungsfehler
`.protocolErr`	permanent	Protokollfehler
`.parseErr`	permanent	Parse-Fehler

BackoffProfile Struct

- **base:** TimeInterval – Initial Delay

- **factor:** Double = 2.0 – Exponentieller Faktor
- **max:** TimeInterval – Maximum Delay Cap
- **jitter:** Double = 0.2 – Zufallsabweichung (0.0-1.0)

API

Methode	Beschreibung
nextDelay(for:attempt:key:)	Berechnet nächsten Delay mit Jitter
recordFailure(_:kind:)	Registriert Fehler, öffnet ggf. Circuit
recordSuccess(_:)	Schließt Circuit, reset Counters
isOpen(_:)	Prüft ob Circuit offen ist
shouldRetry(kind:attempt:key:)	Prüft ob Retry erlaubt ist
remainingRetries(kind:attempt:)	Verbleibende Retry-Versuche

MarkdownHelper

Einfache Hilfsfunktionen für Markdown-Textformatierung im Editor.

Methode	Beschreibung
insertAtLineStart(_:in:)	Fügt Präfix (# , -) am Zeilenanfang ein
wrapSelectionBold(_:)	Umschließt mit **text**
wrapSelectionItalic(_:)	Umschließt mit *text*

PrePromptCatalogManager

Singleton für die Verwaltung des hierarchischen Pre-Prompt-Katalogs. Speichert Menu-Items, Presets, Recipes und Cookbooks in UserDefaults.

@Published Properties

- **menuItems:** [PrePromptMenuItem] – Hierarchische Struktur
- **presets:** [AIPrePromptPreset] – Prompt-Inhalte
- **recipes:** [PrePromptRecipe] – Kombinierte Prompts
- **cookbooks:** [Cookbook] – Kochbuch-Container
- **recipeMenuItems:** [RecipeMenuItem] – Rezept-Menüstruktur

CRUD-Operationen

- addMenuItem(_:), updateMenuItem(_:), deleteMenuItem(_:)
- addPreset(_:in:), updatePreset(_:), deletePreset(_:)
- addRecipe(_:), updateRecipe(_:), deleteRecipe(_:)
- moveMenuItem(_:to:), reorderItems(in:from:to:)

Export/Import

CatalogExport Struct: version, exportDate, menuItems, presets, recipes, cookbooks, recipeMenuItems. JSON-Serialisierung mit ISO8601-Datumsformat.

PrePromptPicker

SwiftUI-View für die hierarchische Auswahl von Pre-Prompts aus dem Katalog. Unterstützt Navigation durch Ordner-Hierarchie.

Features

- Breadcrumb-Navigation durch Ordner
- Unterscheidung Ordner (Folder) vs. Preset-Items
- Selection-Callback für gewählten Preset
- Integration mit PrePromptCatalogManager.shared

Technische Hinweise

- **Sendable:** CancellationToken, CircuitBreaker sind thread-safe
- **Singleton:** RetryPolicy.shared, PrePromptCatalogManager.shared
- **DispatchQueue:** Serialisierter Zugriff auf mutable State
- **ObservableObject:** PrePromptCatalogManager für SwiftUI-Binding
- **Value Type:** CircuitBreaker als Struct (copy-on-write)
- **Jitter:** Zufällige Abweichung verhindert Thundering Herd

---

5.1 Settings Keys

Configuration/Settings/SettingsKeys.swift • Zentrale UserDefaults-Konstanten

Übersicht

Die SettingsKeys.swift definiert alle UserDefaults-Schlüssel als zentrale Konstanten. Dies ermöglicht konsistente Verwendung in der gesamten App und verhindert Tippfehler in Key-Strings. Die Keys folgen einem hierarchischen Namensschema: config.bereich.unterbereich

Kategorie	Anzahl Keys	Präfix
KI-Server/Modelle	5	`config.ai.server.*`
Pre-Prompt System	6	`config.ai.preprompt.*`
Kategorien	1	`config.categories`
Mikrofon/Aufnahme	4	`config.mic.*`
Sprache	1	`config.speech.*`

KI-Server/Modelle

Konfiguration für KI-Provider wie OpenAI und Ollama.

Konstante	UserDefaults Key	Typ / Beschreibung
`kAIServerAddress`	`config.ai.server.address`	String – Server-URL
`kAIServerPort`	`config.ai.server.port`	String – Port (leer=443)
`kAIAPIKey`	`config.ai.server.apikey`	String – API-Schlüssel
`kAIModel`	`config.ai.server.model`	String – Modellname
`kAIPrePrompt`	`config.ai.preprompt`	String – Legacy Pre-Prompt

Verwendung

```
// Lesen let addr = UserDefaults.standard.string(forKey: kAIServerAddress)
?? "" let port = UserDefaults.standard.string(forKey: kAIServerPort) ??
"443" let model = UserDefaults.standard.string(forKey: kAIModel) ??
"llama3:8b"  // Schreiben
UserDefaults.standard.set("https://api.openai.com", forKey:
kAIServerAddress)
```

Pre-Prompt System

Keys für das hierarchische Pre-Prompt-Katalog-System mit Menüstruktur, Presets, Rezepten und Kochbüchern.

Konstante	UserDefaults Key	Datentyp
`kAIPresetsKey`	`config.ai.preprompts`	JSON [AIPrePromptPreset]
`kAISelectedPresetKey`	`config.ai.preprompt.selected`	String (UUID)
`kPrePromptMenuKey`	`config.ai.preprompt.menu`	JSON [PrePromptMenuItem]
`kPrePromptRecipesKey`	`config.ai.preprompt.recipes`	JSON [PrePromptRecipe]
`kCookbooksKey`	`config.ai.preprompt.cookbooks`	JSON [Cookbook]
`kRecipeMenuKey`	`config.ai.preprompt.recipemenu`	JSON [RecipeMenuItem]
`kCatalogInitializedKey`	`config.ai.preprompt.initialized`	Bool

JSON-Persistierung

```
// Laden guard let data = UserDefaults.standard.data(forKey:
kAIPresetsKey),       let items = try?
```

```
JSONDecoder().decode([AIPrePromptPreset].self, from: data) else { return }
// Speichern guard let data = try? JSONEncoder().encode(presets) else {
return } UserDefaults.standard.set(data, forKey: kAIPresetsKey)
```

Kategorien

Benutzerdefinierte Kategorien für Log-Einträge.

Konstante	UserDefaults Key	Datentyp
`kCategories`	`config.categories`	JSON [String]

Standard-Kategorien

- Allgemein
- Netzwerk
- Dokumentation

Mikrofon/Aufnahme

Einstellungen für Audio-Aufnahme und Spracherkennung.

Konstante	UserDefaults Key	Typ / Wertebereich
`kMicSensitivity`	`config.mic.sensitivity`	Double 0.0 – 1.0
`kSilenceThreshold`	`config.mic.silenceDB`	Double -60 – 0 dB
`kChunkSeconds`	`config.mic.chunkSeconds`	Double 1 – 10 Sekunden
`kSpeechLang`	`config.speech.lang`	String (z.B. "de-DE")

Standardwerte

- **Empfindlichkeit:** 0.85 (85%)
- **Stille-Schwelle:** -40 dB
- **Segmentlänge:** 2.0 Sekunden
- **Sprache:** Locale.current (z.B. de-DE)

Verfügbare Sprachen

Code	Bezeichnung
`de-DE`	Deutsch (Deutschland)
`de-AT`	Deutsch (Österreich)
`de-CH`	Deutsch (Schweiz)
`en-US`	English (US)
`en-GB`	English (UK)

ConfigView Integration

Die ConfigView.swift verwendet lokale Konstanten, die auf SettingsKeys referenzieren.

Lokale Key-Enum

```
private enum K {     static let categories         = "config.categories"
static let micSensitivity     = "config.mic.sensitivity"     static let
silenceThresholdDB = "config.mic.silenceDB"     static let chunkSeconds
= "config.mic.chunkSeconds"     static let speechLang         =
"config.speech.lang"     static let mailSettingsName   =
"config.mail.settingsName" }
```

NaN-Schutz bei Double-Werten

```
// Laden mit Validierung let raw = ud.object(forKey: K.micSensitivity) as?
Double ?? 0.85 micSensitivity = raw.isNaN || raw.isInfinite ? 0.85 : raw
// Speichern mit Validierung let valid = value.isNaN || value.isInfinite ?
0.85 : value ud.set(valid, forKey: K.micSensitivity)
```

Namenskonventionen

- **Präfix config.:** Alle App-Einstellungen
- **Bereich.Unterbereich:** Hierarchische Struktur
- **Konstante k...:** Global let mit k-Präfix
- **CamelCase:** Konstanten-Namen in Swift
- **Lowercase + Dots:** Key-Strings in UserDefaults

Technische Hinweise

- **UserDefaults:** Automatische Synchronisation
- **JSON-Codable:** Arrays/Structs als Data speichern
- **Typ-Sicherheit:** Konstanten verhindern Tippfehler
- **Migration:** kCatalogInitializedKey für einmalige Setup-Logik
- **Fallback:** Immer Standardwerte bei nil/NaN
- **Locale:** Sprach-Codes mit Bindestrich (de-DE, nicht de_DE)

5.2 Lokalisierung

Configuration/Language/ • Deutsch/Englisch Übersetzungen

Übersicht

AILO verwendet das iOS/macOS Lokalisierungssystem mit Localizable.strings Dateien. Die App unterstützt Deutsch (de) als Hauptsprache und Englisch (en) als Fallback. Alle UI-Texte sind externalisiert und folgen einem konsistenten Key-Namensschema.

Datei	Sprache	Anzahl Strings
`de.lproj/Localizable.strings`	Deutsch 🇩🇪	**~500**
`en.lproj/Localizable.strings`	English 🇬🇧	**~500**

Key-Namensschema

Alle Lokalisierungs-Keys folgen einem hierarchischen Muster für konsistente Benennung und einfache Zuordnung zu Views.

Pattern

`<feature>.<view>.<element>[.<state|action|hint>]`

Komponenten

Teil	Beschreibung	Beispiele
`feature`	Feature-Bereich	logs, mail, config, write, speak
`view`	View/Screen-Name	list, detail, editor, picker
`element`	UI-Element	title, field, button, label
`state`	Zustandsmodifikator	empty, loading, error, success
`action`	Aktionsmodifikator	save, delete, cancel, retry
`hint`	Hilfetext/Tooltip	placeholder, a11y, hint

Feature-Kategorien

Die Strings sind nach Features gruppiert mit MARK-Kommentaren in den Dateien.

Präfix	Feature	Dateien
`common.*`	Globale UI-Texte	Alle Views
`app.*`	App-Shell/Navigation	ContentView, Tabs
`dashboard.*`	Dashboard	DashboardView
`logs.*`	Log-Liste	LogsView, LogsListView
`logDetail.*`	Log-Details	TextLogDetailView
`write.*`	Schreiben	SchreibenView
`speak.*`	Sprechen/Audio	SprechenView
`mail.*`	E-Mail (Legacy)	MailComposer
`app.mail.*`	E-Mail (Neu)	MailView, MessageDetailView
`config.*`	Einstellungen	ConfigView
`aiManager.*`	KI-Provider	AIManagerView, AIEditor
`aiEditor.*`	KI-Editor	AIEditor
`preprompts.*`	Pre-Prompts	PrePromptManager
`catalog.*`	Pre-Prompt-Katalog	PrePromptCatalogView
`categories.*`	Kategorien	CategoriesView

Präfix	Feature	Dateien
`store.*`	Datenspeicher	DataStore
`state.* / msg.*`	Zustände/Meldungen	Global

Common Keys

Häufig verwendete globale Übersetzungen unter dem common.*-Präfix.

Key	Deutsch	English
`common.ok`	OK	OK
`common.cancel`	Abbrechen	Cancel
`common.save`	Sichern	Save
`common.edit`	Bearbeiten	Edit
`common.done`	Fertig	Done
`common.delete`	Löschen	Delete
`common.close`	Schließen	Close
`common.back`	Zurück	Back
`common.yes`	Ja	Yes
`common.no`	Nein	No
`common.error`	Fehler	Error
`common.success`	Erfolg	Success
`common.loading`	Laden…	Loading…
`common.search`	Suchen	Search
`common.retry`	Erneut versuchen	Retry

Beispiel-Strings

Feature: Logs

```
// Deutsch "logs.list.title" = "Alle Logs"; "logs.search.placeholder" = "Logs durchsuchen"; "logs.empty" = "Noch keine Einträge"; "logs.action.new" = "Log hinzufügen";  // English "logs.list.title" = "All Logs"; "logs.search.placeholder" = "Search logs"; "logs.empty" = "No entries yet"; "logs.action.new" = "Add log";
```

Feature: Mail

```
// Deutsch "app.mail.action.reply" = "Antworten"; "app.mail.action.forward" = "Weiterleiten"; "app.mail.folder.inbox" = "Posteingang"; "app.mail.folder.sent" = "Gesendet";  // English "app.mail.action.reply" = "Reply"; "app.mail.action.forward" = "Forward"; "app.mail.folder.inbox" = "Inbox"; "app.mail.folder.sent" = "Sent";
```

Verwendung im Code

SwiftUI String(localized:)

```
// Einfacher Text Text(String(localized: "logs.list.title"))  // Mit Variablen Text(String(localized: "catalog.recipe.elements \(count)"))  // In Attributen .navigationTitle(String(localized: "config.nav.title"))
```

String Interpolation

```
// Key mit %@ Platzhalter "speak.entry.titleWithName" = "Transkript: %@";
// Verwendung String(format: NSLocalizedString("speak.entry.titleWithName", comment: ""), name)
```

Dateistruktur

```
Configuration/ └── Language/     ├── de.lproj/     │   └──
Localizable.strings    // Deutsch (Hauptsprache)      └── en.lproj/
└── Localizable.strings    // English (Fallback)
```

Guidelines

- **Vollständige Sätze:** Keine Konkatenation von Textfragmenten
- **MARK-Kommentare:** Feature-Gruppierung mit // MARK: -
- **Datei-Referenz:** Kommentar mit zugehörigen Swift-Dateien
- **Konsistente Keys:** Gleiches Pattern in allen Sprachen
- **Plural-Handling:** Später via String Catalog (.xcstrings)
- **Accessibility:** a11y-Suffix für VoiceOver-Texte

Technische Hinweise

- **Bundle:** Strings werden automatisch aus Bundle.main geladen
- **Fallback:** English als Base Localization bei fehlendem Key
- **Encoding:** UTF-8 mit \n für Zeilenumbrüche
- **Escape:** Anführungszeichen als \" escapen
- **Semikolon:** Jede Zeile endet mit ;
- **Kommentare:** /* */ oder // für Dokumentation
- **Format-Specifier:** %@, %d, %lld, %.1f für Variablen

5.3 Mail Account Configuration

Database/Models/MailModels.swift • E-Mail-Konto-Konfiguration

Übersicht

Die MailAccountConfig-Struktur definiert alle Parameter für E-Mail-Konten: IMAP/POP3-Empfang, SMTP-Versand, Authentifizierung, Sync-Limits, Ordnerzuordnung und S/MIME-Signierung. Die Konfiguration wird als JSON in UserDefaults persistiert.

Bereich	Beschreibung
Account-Basis	ID, Name, E-Mail-Adresse, Anzeigename, Reply-To
IMAP/POP3	Host, Port, Verschlüsselung, Benutzername, Passwort
SMTP	Host, Port, Verschlüsselung, Benutzername, Passwort
Authentifizierung	Password, OAuth2, App-Password
Sync Limits	Initial, Refresh, Incremental
Special Folders	Inbox, Sent, Drafts, Trash, Spam
S/MIME	Signierung aktiviert, Zertifikat-ID
Verhalten	Auto-Mark-Read, Intervall, Timeout, Logging

MailAccountConfig Struct

Codable, Identifiable, Equatable, Sendable Struct für vollständige Account-Konfiguration.

Account-Basis

Property	Typ	Beschreibung
`id`	`UUID`	Eindeutige Account-ID
`accountName`	`String`	Anzeigename im Account-Picker
`displayName`	`String?`	Absendername in E-Mails
`emailAddress`	`String`	E-Mail-Adresse
`replyTo`	`String?`	Optionale Reply-To Adresse

IMAP/POP3 Empfang

Property	Typ	Beschreibung
`recvProtocol`	`MailProtocol`	.imap oder .pop3
`recvHost`	`String`	IMAP/POP3 Server
`recvPort`	`Int`	Port (993=IMAPS, 143=IMAP)
`recvEncryption`	`MailEncryption`	.none, .sslTLS, .startTLS
`recvUsername`	`String`	Benutzername
`recvPassword`	`String?`	Passwort

SMTP Versand

Property	Typ	Beschreibung
`smtpHost`	`String`	SMTP Server
`smtpPort`	`Int`	Port (587=STARTTLS, 465=SSL)
`smtpEncryption`	`MailEncryption`	.none, .sslTLS, .startTLS
`smtpUsername`	`String`	SMTP Benutzername
`smtpPassword`	`String?`	SMTP Passwort

Enums

MailProtocol

- `.imap` – Internet Message Access Protocol
- `.pop3` – Post Office Protocol v3

MailEncryption

- `.none` – Keine Verschlüsselung (nicht empfohlen)
- `.sslTLS` – Direktes SSL/TLS (Port 993/465)
- `.startTLS` – STARTTLS Upgrade (Port 143/587)

MailAuthMethod

- `.password` – Standard Passwort-Authentifizierung
- `.oauth2` – OAuth2 Token-basiert
- `.appPassword` – App-spezifisches Passwort (Gmail, etc.)

Sync Limits

Konfigurierbare Limits für die E-Mail-Synchronisation zur Performance-Optimierung.

Property	Typ	Default	Beschreibung
`syncLimitInitial`	`Int`	**200**	Erster Sync
`syncLimitRefresh`	`Int`	**500**	Vollständiger Sync
`syncLimitIncremental`	`Int`	**50**	Inkrementeller Sync

Special Folders

Die Folders-Struktur definiert Server-Ordnernamen für Spezialordner. Automatische Erkennung via FolderDiscoveryService.

Property	Default	Gmail-Beispiel
`inbox`	`"INBOX"`	INBOX
`sent`	`"Sent"`	[Gmail]/Sent Mail
`drafts`	`"Drafts"`	[Gmail]/Drafts
`trash`	`"Trash"`	[Gmail]/Trash
`spam`	`"Spam"`	[Gmail]/Spam

FolderDiscoveryService

- Automatische Erkennung via IMAP LIST + SPECIAL-USE Extension
- Fallback auf Name-Heuristik bei fehlender Server-Unterstützung
- Vorkonfigurierte Provider: Gmail, Outlook, Yahoo
- 60-Sekunden Debounce für wiederholte Discovery

S/MIME Signierung

Optionale E-Mail-Signierung mit S/MIME-Zertifikaten aus dem iOS Keychain.

Property	Typ	Beschreibung
`signingEnabled`	`Bool`	Signierung aktiviert (Default: false)
`signingCertificateId`	`String?`	Keychain-Referenz zum Zertifikat

P12-Import

- Import via File-Picker (.p12, .pfx)
- Passwort-geschützter Import
- Speicherung in iOS Keychain mit SecItemAdd

Verhaltenseinstellungen

Property	Default	Beschreibung
`autoMarkAsRead`	true	Beim Öffnen als gelesen markieren
`checkIntervalEnabled`	false	Automatische Prüfung aktiviert
`checkIntervalMin`	nil (15)	Prüfintervall in Minuten
`connectionTimeoutSec`	15	Verbindungs-Timeout (5-120s)
`enableLogging`	false	Verbindungsprotokoll aktivieren
`oauthToken`	nil	OAuth2-Token für OAuth-Auth

Persistierung

Accounts werden als JSON-Array in UserDefaults gespeichert.

Speicherung

```
// Key "mail.accounts"  // UserDefaults Key  // Laden guard let data =
UserDefaults.standard.data(forKey: "mail.accounts"),       let accounts =
try? JSONDecoder().decode([MailAccountConfig].self, from: data) else {
return }  // Speichern let data = try JSONEncoder().encode(accounts)
UserDefaults.standard.set(data, forKey: "mail.accounts")
```

MailEditor View

SwiftUI-View für Account-Bearbeitung mit Validierung und Verbindungstest.

Sections

- **Account:** Name, E-Mail, Display Name
- **Incoming:** Protokoll, Host, Port, Verschlüsselung, Credentials
- **Outgoing:** SMTP Host, Port, Verschlüsselung, Credentials
- **Synchronization:** Sync Limits (Initial, Refresh, Incremental)
- **Folders:** Special Folders mit Auto-Discovery
- **Advanced:** Auth, Timeout, Logging, Auto-Mark-Read
- **S/MIME:** Signierung, Zertifikat-Auswahl, P12-Import

Technische Hinweise

- **Codable:** JSON-Serialisierung für UserDefaults
- **Sendable:** Thread-sicher für async/await
- **Validierung:** Timeout 5-120s, Interval 1-120min
- **Port-Defaults:** 993 (IMAPS), 143 (IMAP), 587 (SMTP+STARTTLS), 465 (SMTPS)
- **DAO-Sync:** Special Folders werden in MailReadDAO persistiert
- **Fallback:** SMTP-Credentials auf IMAP-Credentials wenn leer

Teil 6: Technologie-Stack

🛠️ Plattform, Frameworks & Dependencies

Übersicht

AILO ist eine native iOS/macOS-App, entwickelt in Swift 5.9+ mit SwiftUI. Die Architektur kombiniert moderne Apple-Frameworks mit Custom-Implementierungen für IMAP/SMTP-Kommunikation und verwendet SQLite für die lokale Datenpersistenz.

📱 Plattform

Komponente	Version / Details
iOS Deployment Target	iOS 16.0+
macOS Deployment Target	macOS 13.0+ (Catalyst)
Swift Version	Swift 5.9+
Xcode Version	Xcode 15.0+
Architecture	arm64 (Apple Silicon), x86_64 (Intel)
App Store	iOS App Store Ready

🎨 UI Framework

Framework	Verwendung
SwiftUI	100% der UI (kein UIKit)
Combine	Reactive Programming, Publisher/Subscriber
NavigationStack	Navigation (iOS 16+)
@Observable	iOS 17+ Observation (optional)
@StateObject	View Model Lifecycle
@EnvironmentObject	App-weites State Sharing

SwiftUI Patterns

- **MVVM:** ViewModels als @StateObject/@ObservedObject
- **Repository Pattern:** MailRepository als zentrale Datenschicht
- **Factory Pattern:** DAOFactory für DAO-Instanziierung
- **Singleton:** Shared Manager (PrePromptCatalogManager, RetryPolicy)

💾 Datenbank

Technologie	Verwendung
SQLite3	Mail-Datenbank (direkte C-API)
DAO Pattern	Data Access Objects für alle DB-Operationen
JSON (Codable)	Logs, Pre-Prompts, Recipes, Cookbooks
UserDefaults	App-Settings, Account-Konfiguration
Keychain	Sensible Daten (Passwörter, API-Keys)

SQLite Schema

- `accounts` – E-Mail-Account-Referenzen
- `folders` – Ordner-Metadaten
- `msg_header` – E-Mail-Header (From, Subject, Date, Flags)
- `msg_body` – E-Mail-Body (HTML, Text)
- `attachments` – Anhänge-Metadaten
- `outbox` – Ausgehende E-Mails Queue
- `blob_meta, mime_parts, render_cache` – Blob Storage

Audio & Speech

Framework	Verwendung
AVFoundation	Audio-Aufnahme und -Wiedergabe
AVAudioRecorder	Mikrofon-Recording (.m4a Format)
AVAudioSession	Audio Session Management
AVAudioPlayer	Audio-Playback
Speech Framework	Live-Spracherkennung
SFSpeechRecognizer	On-Device Recognition (de-DE, en-US)

Audio Features

- Chunk-basierte Transkription für Live-Feedback
- Silence Detection (konfigurierbare Stille-Schwelle)
- On-Device Recognition (datenschutzfreundlich)
- M4A-Format für kompakte Speicherung

Netzwerk

Protokoll/Framework	Verwendung
URLSession	HTTP/HTTPS für KI-APIs
IMAP (Custom)	E-Mail-Empfang (IMAPConnection, IMAPCommands)
SMTP (Custom)	E-Mail-Versand (SMTPClient, NIOSMTPClient)
SwiftNIO	Async Networking (SMTP)
NIOSSL	TLS/SSL für SMTP-Verbindungen
Network.framework	TCP/TLS für IMAP

IMAP Implementation

- **IMAPConnection.swift:** Verbindungsmanagement
- **IMAPCommands.swift:** LOGIN, SELECT, FETCH, SEARCH, LIST, STORE
- **IMAPParsers.swift:** ENVELOPE, BODYSTRUCTURE, FLAGS Parsing
- **IMAPConnectionPool.swift:** Connection Pooling

SMTP Implementation

- **SMTPClient.swift:** Network.framework-basiert
- **NIOSMTPClient.swift:** SwiftNIO mit NIOSSL
- **STARTTLS:** In-Place TLS Upgrade
- **AUTH LOGIN:** Base64-kodierte Authentifizierung

🤖 KI-Integration

Provider	Endpoint
OpenAI	/v1/chat/completions (GPT-4, GPT-3.5)
Ollama	/api/chat, /api/generate (llama3, mistral, etc.)
Custom	OpenAI-kompatible API Endpoints

AIClient Features

- Multi-Provider Support mit automatischem Fallback
- Pre-Prompt-System mit hierarchischem Katalog
- Rezept-basierte Prompt-Kombinationen
- Lokalisierte Fehlermeldungen

🔐 Sicherheit

Komponente	Funktion
KeychainService	Sichere Speicherung (Passwörter, API-Keys)
Security.framework	Keychain-API, SecItemAdd/Query/Delete
S/MIME	E-Mail-Signierung mit Zertifikaten
SMIMESigningService	PKCS#7/CMS Signatur-Erstellung
KeychainCertificateService	Zertifikat-Import (P12/PFX)
TLS 1.2/1.3	Verschlüsselte Verbindungen

📦 Dependencies

Externe Swift Packages via Swift Package Manager (SPM).

Package	Version	Verwendung
`swift-nio`	2.65.0+	Async Networking
`swift-nio-ssl`	2.26.0+	TLS/SSL für NIO
`NIOCore`	–	Event Loop, Channels
`NIOPosix`	–	POSIX Integration
`NIOSSL`	–	SSL Context, Handler

📁 Projektstruktur

```
AILO_APP/
├── App/                    # App Entry Point
├── Views/                  # SwiftUI Views
│   ├── Dashboard/
│   ├── Logs/
│   ├── Mail/
│   ├── Schreiben/
│   ├── Sprechen/
│   ├── Configuration/
│   └── Shared/
├── Services/               # Business Logic
│   ├── AI/
│   ├── Audio/
│   ├── Mail/
│   │   ├── IMAP/
│   │   ├── SMTP/
│   │   ├── Sync/
│   │   └── Diagnostics/
│   └── Security/
├── Database/               # Data Layer
│   ├── DAO/
│   ├── Models/
│   ├── Schema/
│   └── Store/
├── Helpers/                # Utilities
│   ├── Parsers/
│   ├── Security/
│   ├── UI/
│   └── Utilities/
├── Configuration/          # Settings & Language
│   ├── Settings/
│   └── Language/
└── Resources/              # Assets, Plist
```

Technische Hinweise

- **Keine externen UI-Frameworks:** 100% SwiftUI
- **Async/Await:** Durchgängig moderne Concurrency
- **Sendable:** Thread-sichere Datenstrukturen
- **@MainActor:** UI-Thread-Sicherheit
- **Codable:** JSON-Serialisierung für alle Modelle
- **Identifiable:** UUID-basierte Entitäten
- **Keine CocoaPods/Carthage:** Nur SPM für Dependencies