VYSOKÁ ŠKOLA POLYTECHNICKÁ JIHLAVA

Aplikovaná informatika

STUDY ASSISTANT FOR EFFICIENT LEARNING AND EXAM PREPARATION

Bakalářská práce

Autor práce: Oleksandra Rudoi

Vedoucí práce: PaedDr. František Smrčka, Ph.D.

Jihlava 2026

Vysoká škola polytechnická Jihlava

Tolstého 16, 586 01 Jihlava

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Autor práce:	**Oleksandra Rudoi**
Studijní program:	Aplikovaná informatika
Garant studijního programu:	Ing. Lenka Kuklišová Pavelková, Ph.D.
Název práce:	**Studijní asistent pro iOS podporující efektivní přípravu a opakování učiva**
Vedoucí práce:	PaedDr. František Smrčka, Ph.D.
Cíl práce:	Cílem bakalářské práce je navrhnout a zrealizovat mobilní aplikaci pro iOS, která usnadní samostatné studium a přípravu na zkoušky. Uživatel v aplikaci zadá název a popis předmětu a připojí vlastní dokumenty (např. PDF, DOCX apod.). Na základě těchto vstupů aplikace vygeneruje studijní obsah, testové otázky, výukové karty a úkoly k opakování. Uživatel si může zvolit režim procvičování, sledování vlastního pokroku nebo se zaměřit se na slabší oblasti. Výsledkem bude nástroj podporující aktivní učení, dlouhodobé zapamatování a efektivní řízení studia. Práce se soustředí na návrh interaktivního uživatelského rozhraní, práci s textovými vstupy, zpracování obsahu a správu lokálních dat.

Abstrakt

Práce se zaměřuje na návrh a realizaci mobilní aplikace pro platformu iOS určené k podpoře samostatného učení ve vysokoškolském prostředí. Pozornost je věnována metodám samostatného a sebeřízeného učení, principům aktivního vybavování a rozloženého opakování a možnostem využití velkých jazykových modelů ve vzdělávání. Na základě této teoretické analýzy byla navržena a implementována aplikace Kricki, která na základě uživatelem zadaného tématu automaticky generuje strukturovaný studijní plán a pro každé podtéma vytváří studijní poznámky, flashcards, kvíz s výběrem z více možností a slovníček pojmů. Aplikace využívá webově ukotvený jazykový model Perplexity Sonar, který snižuje pravděpodobnost faktických chyb v generovaném obsahu. Veškerá data jsou uložena lokálně pomocí SwiftData, což umožňuje offline přístup. Výsledkem práce je funkční prototyp nativní iOS aplikace, který kombinuje automatizované zpracování studijních materiálů s flexibilním generováním různých typů studijních aktivit.

Klíčová slova

samostatné učení; sebeřízené učení; e-learning; umělá inteligence ve vzdělávání; personalizované učení; mobilní studijní aplikace; vysokoškolské vzdělávání

Abstract

This thesis presents the design and implementation of a native iOS application intended to support independent learning in a university environment. The theoretical analysis covers methods of self-directed and self-regulated learning, the principles of active recall and spaced repetition, and the use of large language models in education. Based on these foundations, the application Kricki was designed and implemented. Given a user-provided topic, it automatically generates a structured study plan and, for each subtopic, produces study notes, flashcards, a multiple-choice quiz, and a vocabulary list. The application uses the web-grounded language model Perplexity Sonar to reduce the likelihood of factual errors in the generated content. All data is stored locally using SwiftData, enabling offline access. The result is a working prototype of a native iOS application that combines automated processing of study materials with flexible generation of multiple types of study activities.

Keywords

self-directed learning; self-regulated learning; e-learning; artificial intelligence in education; personalized learning; mobile learning application; higher education

Prohlašuji, že předložená bakalářská práce je původní a zpracoval/a jsem ji samostatně. Prohlašuji, že citace použitých pramenů je úplná, že jsem v práci neporušil/a autorská práva (ve smyslu zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů, v platném znění, dále též „**AZ**“).

Byl/a jsem seznámen/a s tím, že na mou bakalářskou práci se plně vztahuje **AZ**, zejména § 60 (školní dílo).

Podle § 47b zákona o vysokých školách souhlasím se zveřejněním své práce podle Směrnice pro vedení, vypracování a zveřejňování závěrečných prací na VŠPJ, a to bez ohledu na výsledek obhajoby.

Beru na vědomí, že VŠPJ má právo na uzavření licenční smlouvy o užití mé bakalářské práce a prohlašuji, že **s o u h l a s í m** s případným užitím mé bakalářské práce (prodej, zapůjčení apod.).

Jsem si vědom/a toho, že užít své bakalářské práce či poskytnout licenci k jejímu využití mohu jen se souhlasem VŠPJ, která má právo ode mě požadovat přiměřený příspěvek na úhradu nákladů, vynaložených vysokou školou na vytvoření díla (až do jejich skutečné výše), z výdělku dosaženého v souvislosti s užitím díla či poskytnutím licence.

V Jihlavě dne 8. ledna 2026

…….

Podpis studenta/ky

Acknowledgements

I would like to thank my thesis advisor, PaedDr. František Smrčka, Ph.D., for his expert guidance, feedback, and advice during the preparation of this bachelor's thesis.

List of Contents

List of Tables

List of Figures

List of Attachments

List of Code Snippets

List of Abbreviations Used

AI	Artificial Intelligence
API	Application Programming Interface
CEFR	Common European Framework of Reference for Languages
iOS	iPhone Operating System
JSON	JavaScript Object Notation
LLM	Large Language Model
NLP	Natural Language Processing
OCR	Optical Character Recognition
RAG	Retrieval-Augmented Generation
SDK	Software Development Kit
UI	User Interface
UUID	Universally Unique Identifier
UX	User Experience

Introduction

As students prepare for their exams, they often spend a disproportionate amount of time producing study materials rather than actually studying. The emergence of large language models has made it possible to automate much of this preparatory work. Unfortunately, available solutions that utilise these models typically lack a structured approach, cannot work offline, or are not native mobile applications.

This thesis proposes developing a native iOS application that generates a fully structured study plan from a given topic and, at the same time, creates study materials for each subtopic. Specifically, for every subtopic, the application generates four types of content (study notes, flashcards, a multiple-choice quiz, and a vocabulary list) based on the web-grounded LLMs' output. All generated data is saved locally, allowing the user to study without an internet connection. The application is purposefully limited to single-user use and does not include a backend, user accounts, analytics, or synchronisation features. The purpose of the solution is to relieve students of the need to spend time preparing study materials by automating the entire process.

The motivation of this work is rooted in personal experience during university years. Whenever one had to produce study materials for some course using the general-purpose LLMs like ChatGPT, one had to make several prompts to the model in order to structure the material, generate flashcards, create a quiz, etc. Although helpful, this process was tedious and complicated by the difficulty of revising generated outputs. To streamline that process, this project incorporates all these functions into a single tool without superfluous functionality.

This thesis is organised into five chapters. In Chapter 1, an overview of important concepts is provided. The problem statement, target audience, use cases, and requirements are described in Chapter 2. Chapter 3 covers technical details about the system architecture, data model, selected technologies, and UI design. Chapter 4 discusses implementation issues and presents a verification matrix. The final Chapter 5 contains the conclusions and suggestions for further improvements.

1 Theoretical Background

Only theoretical concepts related to decision-making in the study tool design are discussed here. A more general discussion of the applications of AI technology to education is omitted.

1.1 Independent Study and Micro-Learning

At the university level, students need to organise their learning independently. Knowles (1975) describes it as the process in which students set their goals, choose the strategies and evaluate their progress. According to Zimmerman (2002), the requirement of self-regulation is the primary reason why students struggle with study, despite comprehending the topics.

Two insights follow from this definition. Firstly, the content should be provided in the form of small self-contained blocks instead of long continuous pieces due to cognitive overload (Zimmerman, 2002). Secondly, the application should be convenient for studying on-the-go within any period of free time. According to Trowbridge, Waterbury, and Sudbury (2017), learning in small bursts in mobile conditions is referred to as micro-learning. This type of study increases concentration and memory capacity. These principles inform the design of the application presented in this thesis, where the core learning unit is a subtopic.

1.2 Active Recall and Spaced Repetition

There are two techniques especially useful for the retention of information: active recall and spaced repetition.

Active recall is based on the idea of retrieval. It means that the learner is to recall the information, and this task is more effective in building memory than merely reading or repeating the content. Karpicke and Blunt (2011) prove that retrieval practice is more efficient than elaboration when it comes to learning. Furthermore, Biggs (2003) states that an active process of recalling belongs to deep learning.

As for spaced repetition, it refers to the systematic review of material with increasing intervals between repetitions. Ebbinghaus (1913) described it for the first time. More recently, in 2019, Tabibian et al., using data from Duolingo app, showed that an optimal schedule of repetitions has a positive effect on retention. This technique forms the basis of effective flashcard-based learning systems such as Anki.

The application applies both methods directly. Notes and questions are created automatically, and the user can mark each topic as known or unknown for review. At this stage, there is no mechanism for the scheduling of repetitions, although it will be added in further updates.

1.3 Large Language Models in Educational Tools

The advent of Large Language Models (LLMs) trained on a vast array of text data brought new possibilities to the field of education. Three capabilities of LLMs can be especially helpful for building an educational application: summarisation, question generation, and adaptive explanations.

LLMs can condense long texts into shorter versions that preserve key ideas (IBM, n.d.). While quality depends on input and prompt design, this capability is sufficient for generating initial study notes. In the presented application, each subtopic note is produced by giving the model a topic name and letting it generate a structured explanation.

Generating questions is also possible thanks to LLM. A text-based AI can produce factual, multiple choice, and short-answer questions based on the given passage of text (Holmes, Bialik and Fadel, 2019). However, recent studies such as Maity and Deroy (2024) show that the quality of prompts plays a crucial role in question quality. In some cases, an imprecise or insufficiently specific prompt leads to the generation of questions that do not touch upon the central topics of the text and have multiple answers. Therefore, special attention is paid to the prompt construction and testing of the output.

There are two problems with LLMs which are especially relevant for educational purposes. One of them is referred to as hallucination, when the AI generates a text that sounds logical and confident but contains inaccurate information. For example, if a student asks the AI to write a text on the discovery of a famous mathematical theorem, the model may generate plausible but factually incorrect narratives. In case of an educational application, the problem can become fatal due to the fact that the content will be the only source of information.

Another problem is scope drift when the generated text goes outside the scope imposed by the prompt. When a student needs to learn "Preterite vs Imperfect" in Spanish grammar, an uncontrolled model may start explaining the subjunctive mood and future tenses. This can be mitigated with carefully designed prompts and deterministic validation of the generated content.

This thesis does not address broader concerns associated with AI in education, such as AI replacing teachers or reducing student engagement. It considers the technology from a practical standpoint: AI should perform routine tasks, whereas the student is still an active part of the process.

1.4 Web-Grounded Large Language Models

Traditional large language models generate outputs purely based on their learned knowledge without being able to refer to additional information. This limitation affects many applications, including the one presented here. A model trained on data up to a fixed date may give outdated information or miss recent discoveries. There are two solutions to this problem.

The first one is Retrieval-Augmented Generation (RAG). It is when a model retrieves relevant passages from a previously specified corpus of documents and then generates output based on that information (Lewis et al., 2020). RAG works by searching for passages using vector-based database that retrieves passages semantically similar to the question asked. Then, a model is given these passages as a prompt for generating a grounded answer. The method is good when dealing with the previously collected document collection, like Google NoteBookLM, which uses this technique on user uploaded documents. However, it is not flexible regarding the quality and volume of the document set: in case there are no relevant sources, a model will not be able to ground its answer.

Another solution is web-grounding, when information needed for an output is directly retrieved from the internet at the moment of generation. Perplexity's Sonar uses this method. When prompted, the system first searches the web for relevant sources. It then forces the model to refer to those sources during generation (Perplexity AI, 2024). This way, every generated response can be linked to credible and citable internet sources.

For an application intended for study purposesthe choice of a grounded over ungrounded model makes a big difference. Suppose a user asks the application to explain the Krebs cycle for a biochemistry class. An ungrounded LLM may produce a technically correct answer, but it may mix the order of intermediates, call some enzyme a different name or mention wrong ATP molecules numbers. These are typical examples of hallucinations made by an ungrounded model. In contrast, a web-grounded model first looks through the actual biochemistry reference material available online and generates its answer based on it. Errors can still occur but are expected to be less frequent. However, correctness still depends on the quality and reliability of the retrieved sources.

Thus, Perplexity's Sonar has been chosen as a foundation for the application because the latter requires web-grounding, which anchors an output to sources. In requests, an application passes the value of a parameter called temperature, which affects determinism of output to 0.2. However, Perplexity warns in its documentation that the default values are well optimized, and one should avoid extensive tuning of generation parameters (Perplexity AI, 2024).

1.5 Existing Study Tools

Only a few tools exist that dominate higher education and independent study. Here they are compared against the application design outlined above:

Several insights were revealed during the comparative analysis. Firstly, no widely adopted tool combines content creation, a systematic study approach, and a native mobile experience in a single focused workflow. Quizlet and Anki concentrate on repetitive study without content generation, whereas Notion and RemNote offer content generation but are more of organisational tools. ChatGPT can generate content but lacks structured planning and local persistence. Google NotebookLM is the closest competitor, but it processes large inputs as a whole, which may result in shallow coverage when the input is extensive.

The proposed approach is different in that it first produces a study plan, and then creates content for every subtopic separately, meaning that depth scales with the unit of study.

Secondly, the application seems to be unique in terms of web-grounding. It is absent from almost all of the competitors.

In the following tables, "Yes" indicates full support, "No" indicates the feature is absent, and "Partial" indicates partial or limited support.

Tab. 1: Comparison of study tools across the dimensions that matter for independent university study.

Tool	***Auto-generative content***	***Spaced repetition***	***Structured study plan***	***Mobile-native***	***Web-grounded***
Quizlet	*Partial*	**Yes**	No	**Yes**	No
Anki	No	**Yes**	No	**Yes**	No
Notion AI	**Yes**	No	*Partial*	*Partial*	No
RemNote	*Partial*	**Yes**	*Partial*	*Partial*	No
Duolingo	**Yes (language only)**	**Yes**	**Yes**	**Yes**	No
Google NotebookLM	**Yes**	No	No	*Partial*	*Partial*
ChatGPT	**Yes**	No	No	**Yes**	*Partial*
Kricki	**Yes**	*Partial*	**Yes**	**Yes**	**Yes**

Source: own work

2 Analysis

Prior to designing the application, it is necessary to define what problem the application solves, who it does it for, and what exactly needs to be done. The requirements set out in this section will form the basis of all architectural decisions.

2.1 Problem Statement

In preparation for an exam, a student at university would learn from a course outline aided by class notes, slides, and extra materials. For different courses, the procedure of preparation is quite similar:

1. Splitting a course outline into smaller chunks.
2. Finding explanations for those chunks.
3. Learning terms and definitions.
4. Practice remembering them with flashcards or self-quizzing.
5. Estimating their progress by taking quizzes.
6. Coming back to the material they did not fully understand.

Currently, there is no unified approach. Notes are taken in apps like Notion, flashcards are created in Quizlet or Anki, quizzes are handcrafted, and explanations come from books or Wikipedia. Alternatively, AI-based tools such as ChatGPT are increasingly used for generating explanations. However, all of those tools provide solutions to only a small piece of the puzzle above.

Two major problems arise from this fragmentation. Students waste considerable time writing flashcards and quiz questions. Additionally, AI-generated answers are not stored in a structured form, making it hard to revisit them offline.

Kricki aims to reduce such fragmentation by letting users create projects based on course outlines or topics and generate notes, flashcards, quizzes, and vocabulary automatically. Users can learn the material using four modes of operation, all of which are integrated in a single app.

2.2 Target User and Use Scenarios

The target user is a university student preparing for an exam autonomously using their own materials. The user is proficient with mobile apps, but not necessarily with more sophisticated solutions like Anki. The application should work on modern iPhones running iOS 17+ and should have internet access when content is being generated.

Three use cases guide the development:

Scenario A – fast revision before an exam.

A student has an exam in two days and did not start revising yet. They create a project named "Linear Algebra", copy their course outline into the Notes input field, press Create. The application generates a study plan for the course. Then, when the student chooses a subtopic, the application generates notes, flashcards, quizzes, and vocabulary automatically, while the student begins reading them.

Scenario B – long-term language learning.

A student preparing Spanish at the A2 level creates a project named "Spanish A2 Grammar". The application recognizes it as a language course and provides a specialized grammar lesson view with conjugation tables, examples, and common mistakes. Flashcards contain vocabulary as target-language to translation pairs. A student practices a few subtopics per day, for instance, in commutes.

Scenario C – revision of already known topics.

The student revisits the material studied earlier while not having access to the internet, e.g. on a train ride. They choose a topic and run a flashcard session only with items they know nothing about since all information has been stored locally.

2.3 Comparison with Existing Tools

Here, comparison is narrowed down to the features relevant to the requirements specified.

No existing native iOS app combines all of these capabilities in one place. Structured planning, per-subtopic content generation, offline storage, web-grounded responses, and a language-learning module remain fragmented across separate tools.

Kricki will aim to fill this gap in the problem space.

Tab. 2: Feature comparison restricted to the dimensions of the present requirements catalogue

Dimension	*Quizlet*	*Anki*	*NotebookLM*	*ChatGPT*	*Kricki*
Generates a multi-level study plan	No	No	No	*Partial*	**Yes**
Generates notes per subtopic	No	No	*Partial*	**Yes**	**Yes**
Generates flashcards per subtopic	*Partial*	No	*Partial*	*Partial*	**Yes**
Generates a multiple-choice quiz	**Yes**	No	*Partial*	*Partial*	**Yes**
Extracts terms and definitions	*Partial*	No	*Partial*	*Partial*	**Yes**
Stores generated content offline	*Partial*	**Yes**	No	No	**Yes**
Native iOS application	**Yes**	**Yes**	*Partial*	**Yes**	**Yes**
Specialised path for language learning	*Partial*	No	No	No	**Yes**
Web-grounded factual content	No	No	*Partial*	*Partial*	**Yes**
Adapts content quantity to subtopic depth	No	No	No	*Partial*	**Yes**

Source: own work

2.4 Functional Requirements

The table below presents all twenty-one functional requirements of the application. They are enumerated sequentially from F1 to F21 and grouped into five categories: project management, study plan generation, per-subtopic content generation, study modes, and documents with offline access. Each requirement is implemented by one of the modules described in Chapter 4 and was verified manually on a physical device, as summarised in Section 4.10.

Tab. 3: Functional requirements

ID	*Requirement*	*Description*
F1	Create project	The user can create a new project by entering a title and optional notes.
F2	Auto-classify project	The application automatically classifies a new project as General or Language from its title and notes.
F3	Project icon and colour	The user can pick an icon and a colour for each project; both are displayed everywhere the project is shown.
F4	Cascade delete	The user can delete a project; all of its topics, subtopics, flashcards, questions, terms and documents are removed.
F5	Generate study plan	The application generates a multi-level study plan (topics with subtopics) for a project automatically using the AI service.
F6	Adaptive plan size	The number of topics and subtopics is decided by the model and reflects the complexity of the subject – not a fixed quantity.
F7	Regenerate plan	The user can regenerate the study plan, replacing the previous one.
F8	Full content on tap	When the user opens a subtopic, the application generates study notes, flashcards, a multiple-choice quiz, and a list of key terms in one pipeline.
F9	In-progress loader	While generation is in progress the user sees a loader on the subtopic card and the rest of the UI remains responsive.
F10	Generation error feedback	If generation fails for any of the four content types the user is shown a clear error message and can retry.
F11	Persisted content	Generated content is persisted locally so that subsequent visits to the subtopic do not regenerate it.
F12	Language grammar lesson	The notes for a **Language** project are produced as a focused grammar lesson rather than as a generic notes document.
F13	Subtopic-scoped content	All generated content stays scoped to its own subtopic – material from sibling subtopics is explicitly excluded.
F14	Read notes	The user can read the study notes for a subtopic, with markdown formatting and collapsible sections.
F15	Flashcard session	The user can run a flashcard session for a subtopic, choosing between **all cards**, **unknown only**, and **quick review**.
F16	Quiz session	The user can run a multiple-choice quiz for a subtopic; each question shows immediate feedback and an explanation.
F17	Quiz results	At the end of a quiz the user is shown the score, the number of correct/incorrect answers, and a pass/fail indicator.
F18	Vocabulary list	The user can browse the vocabulary list for a subtopic, filter by **all / known / unknown**, and toggle the known flag on each item.
F19	Attach documents	The user can attach reference documents to a project. Generated content takes the documents into account.
F20	Offline access	All generated content is available offline once it has been generated.

F21	Error containment	API errors do not corrupt local state – partial generation results are preserved and the user can retry.

Source: own work

2.5 Non-functional Requirements

Alongside functional requirements, the application has to meet several non-functional ones, limiting the platformconstraining its quality attributes, and shaping the architecture.

Tab. 4: Non-functional requirements

ID	***Requirement***	***Description***
N1	Platform	Native iOS application, deployment target iOS 17, written in Swift 5.10 with SwiftUI.
N2	Local persistence	All user data is stored locally using SwiftData. No server-side storage.
N3	Single external service	The application calls only the Perplexity API; no telemetry, no analytics, no third-party SDKs.
N4	Responsive UI	The user interface remains responsive during AI generation. Generation runs off the main actor.
N5	Reliable generation	AI calls are retried up to three times on transient failures before surfacing an error.
N6	Privacy	The Perplexity API key is read from `Info.plist`; user data never leaves the device except as part of an AI prompt.
N7	Accessibility	Standard iOS Dynamic Type and VoiceOver work for all primary screens.
N8	Maintainability	A clear separation between Views, ViewModels and Services. Each generation type goes through one prompt factory.

Source: own work

2.6 Use Case Model

Since the application does not have any administrative functionality, there is no need for administrator roles. Also, there are no collaborative or shared accounts. Therefore, this system only has one actor, and all data is stored locally. Thus, the use case model becomes rather straightforward.

The following table presents primary use cases of the system.

The use case diagram depicts a single human actor, namely, the Student, interacting with ten use cases located inside the boundaries of the application. The external Perplexity Sonar API gets used in UC2 (generation of the study plan) and UC3 (per-subtopic generation of content).

UC3 represents the central interaction flow, and thus it is thoroughly described below.

Alternative flows: if the AI request fails three times, the application displays an error message in the banner; the loader disappears, and partial results stay saved. If all content has already been generated for a subtopic, no AI call will be performed, and the application navigates to the subtopic.

Tab. 5: Primary use cases

UC	***Use case***	***Description***
UC1	Create project	The student creates a new project, providing a title and optional notes.
UC2	Generate study plan	The application turns the project into a list of topics and subtopics using the AI service.
UC3	Open subtopic	The student taps a subtopic; the application generates all four content types and opens the mode menu.
UC4	Read notes	The student reads the study notes for a subtopic.
UC5	Run flashcard session	The student runs a flashcard session in one of three modes.
UC6	Take quiz	The student takes the multiple-choice quiz for a subtopic and reviews the result.
UC7	Browse vocabulary	The student browses, filters and marks vocabulary items.
UC8	Edit project appearance	The student picks an icon and colour for the project.
UC9	Attach document	The student adds a text document to the project context.
UC10	Delete document	The student removes a project together with all its derived content.

Source: own work

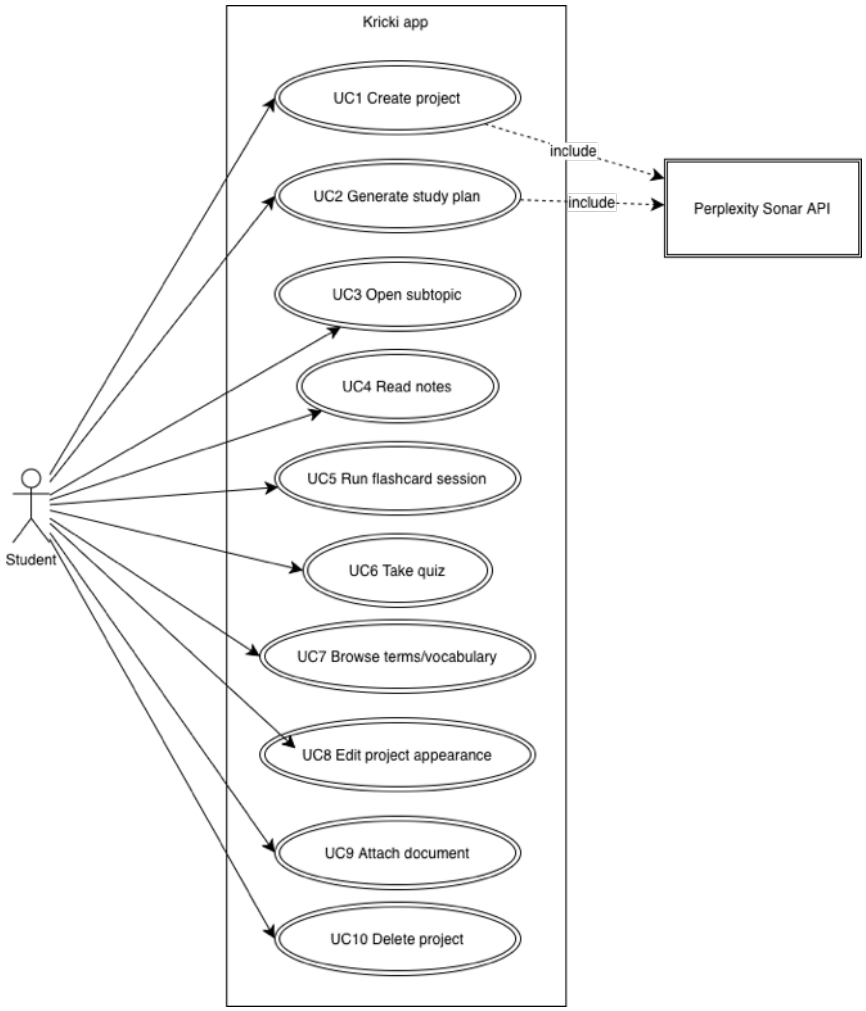


Figure 1: Use case diagram

Source: own work

Tab. 6: Detailed flow for UC3 (Open subtopic)

Step	***Actor***	***Action***
1	Student	Taps a subtopic row in the topic accordion of the project detail screen.
2	Application	Marks the subtopic as generating; shows a loader on its row.
3	Application	If the subtopic has no notes, requests notes generation from the AI service.
4	Application	If the subtopic has no flashcards, requests flashcard generation.
5	Application	If the subtopic has no quiz questions, requests question generation.
6	Application	If the subtopic has no terms, requests term extraction.
7	Application	Persists each successful result to the local SwiftData store as soon as it returns.
8	Application	When all four steps have completed, hides the loader and navigates to the subtopic detail.
9	Student	Lands on the subtopic mode menu; can immediately read notes, run flashcards, or take a quiz.

Source: own work

3 Solution Design

The application follows several guiding principles. These include layered architecture with unidirectional data flow, local-first persistence, two project modes, and a prompt strategy that prioritises scope correctness.

3.1 High-level Architecture

The application follows a layered architecture with a unidirectional data flow: views present data and initiate events. View-models coordinate logic, make calls to services, and update the data store. Services perform domain operations, communicate with AI, and persist data using SwiftData.

Views observe state changes through SwiftUI's observation framework, while SwiftData notifies about changes, which are reflected in the UI reactively. Data flows downwards through service calls, while updates flow upwards via reactive callbacks.

This design choice was made for its clarity, simplicity, separation of concerns, and suitability for a content-driven mobile application. Being designed as offline-first, the application persists every generated AI result immediately and uses the local database for all its screens instead of fetching data from the network.

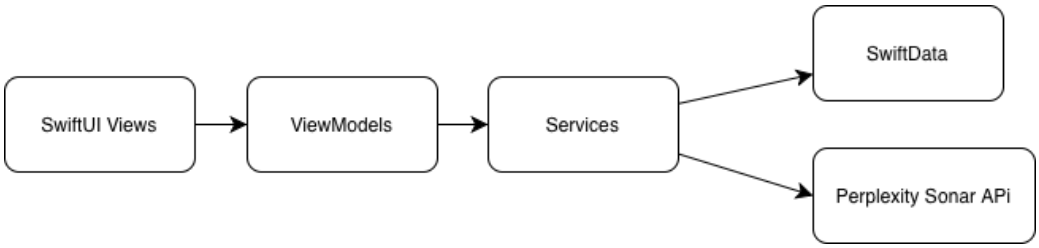


Figure 2: Layered architecture
Source: own work

3.2 Data Model

The data model is implemented using SwiftData-based entity-relationship scheme with seven entities and six one-to-many relationships.

Chained deletion rules for the Project entity ensure all related data is deleted when the project itself is deleted, which fulfils the F4 requirement. Project contains the title, optional notes, the creation timestamp, the subject kind, optional colour and icon fields. Project contains two cascade-delete relationships with its topics and documents. Topic contains the title, display order number, optional AI-generated overview, and owns a cascade-delete relationship with its subtopics. The central entity in this model is Subtopic: it contains the generated Markdown notes in the content field and cascade-deletes relationships with flashcards, questions, and terms.

Document contains a title, body text, and creation timestamp representing the reference document attached to the project. Flashcard contains a front (question) and back (answer)

strings, a known flag for mastery tracking, and an optional lastReviewed timestamp. Question contains the prompt, four possible answer choices represented as a JSON string, the index of the correct answer, and an optional explanation. Term contains a term-definition pair and a known flag. All three leaf entities (flashcards, questions, terms) belong to a single subtopic via an inverse relationship.

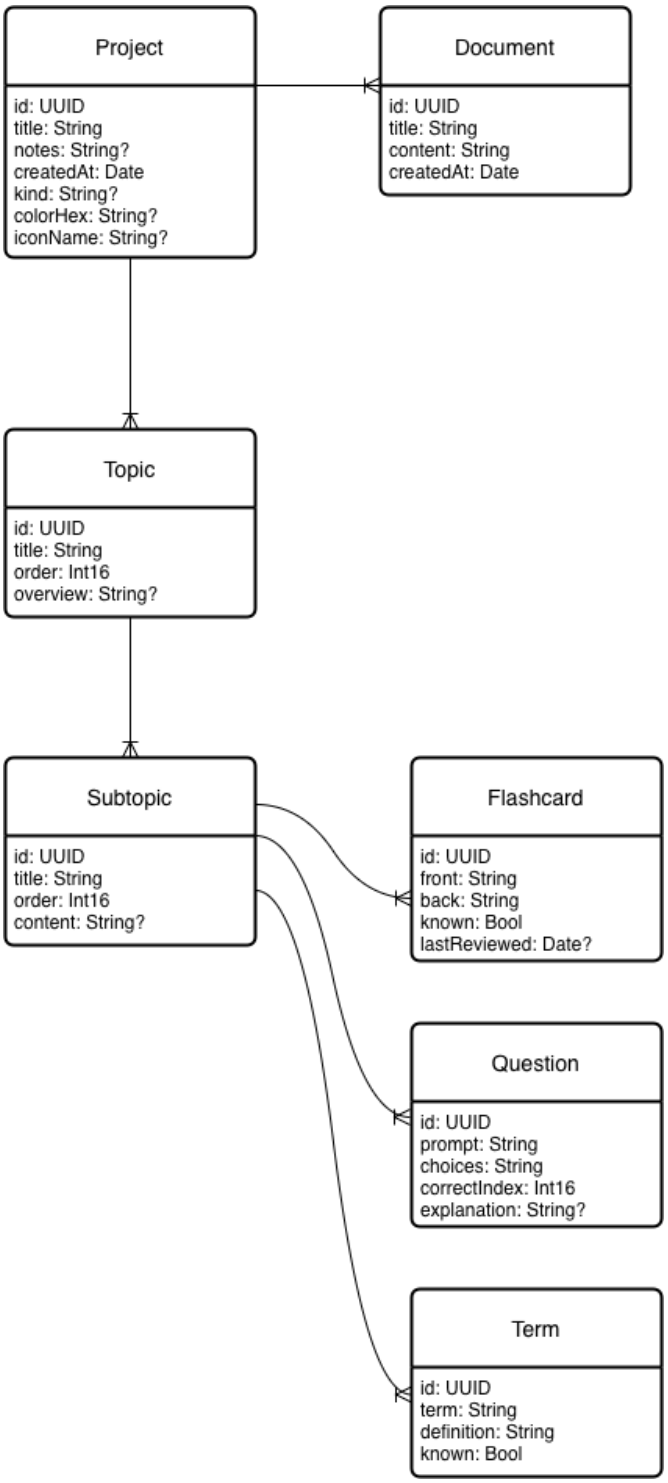


Figure 3: Entity-relationship data model

Source: own work

3.3 Project Modes – General and Language

There are two modes supported by the application: general and language. The mode is determined automatically during project creation by matching language-related keywords in the project name and notes using the SubjectClassifier.

These distinctions have been kept very minimal, since the bulk of subjects can be described with the same schema: a study guide, flashcards, and quizzes. Only language subjects differ meaningfully from concept-based ones in requiring specialized outputs, including conjugation tables, form-meaning explanations, and vocabulary lists. Thus, additional categories were deemed unnecessary.

In the language mode, a specialized output is expected (grammar and vocabulary lessons); in the general mode, standard study notes are requested. Note that only grammar and vocabulary are covered by the language mode; other language aspects are left to the general mode, in case of the need.

3.4 Technology Choices

Each technology has been selected for its ability to satisfy the previously listed requirements.

3.4.1 SwiftUI and the Observation Framework

Since this is a content-focused application that requires little advanced UI customisation, there is no reason to go with UIKit for implementing the interface. Moreover, SwiftData is built-in into iOS 17, providing a convenient observation mechanism, and SwiftUI aligns perfectly with the layered architecture paradigm. View models are marked as @Observable and views use @State for owned view-model instances.

3.4.2 SwiftData

SwiftData (Apple, 2024b) is chosen over Core Data and third-party libraries for several reasons. It is available on iOS 17, uses modern Swift syntax with @Model annotations, integrates seamlessly with SwiftUI, and suits the small data model.

3.4.3 Perplexity Sonar API

The generation functionality is implemented using the Perplexity Sonar API (Perplexity AI, 2024) via its chat completions endpoint. As discussed in Section 1.4, web grounding was the decisive factor in choosing this API over a generic cloud LLM or an on-device model. On-device models were ruled out due to insufficient accuracy across diverse subjects; offline access is instead achieved by persisting all generated content locally (F21). The API follows a standard request structure (model, messages, temperature, max_tokens). The default configuration uses model "sonar", temperature 0.2, and max_tokens 4000.

3.5 User Interface Design

The application consists of six primary screens connected with a linear navigation flow. The student starts the application by opening the home screen where all their projects are listed. On this screen, the student can start a new project or navigate to an existing project's detail screen. The latter shows the generated study plan in the form of an expandable list and leads the student to the subtopic detail screen, where the student chooses one of four study modes. From here, the student is free to switch between study modes using a dropdown menu and does not have to navigate back.

This flow ensures that the student can access any piece of study content in just a few taps, regardless of whether it is a quiz or some notes in Markdown.

The home screen represents the application's entry point and contains all created projects. These projects are represented as cards sorted vertically and colour-coded, providing students with easy differentiation between subjects. Both the colour and icon are selected by the student manually during the project creation process, thus, fulfilling the F3 requirement. At the bottom of the home screen, there is a floating action button allowing the student to create a new project. The greeting message displayed at the top of the screen changes according to the current time of day and enhances the perceived user experience.

Project creation follows a two-step flow. First, the student fills in the project title and optionally the description of the subject along with the attached documents. These notes and attachments will be passed to the prompt, allowing the AI to generate content that will match this particular course as closely as possible. Then the student presses the Start button.

Upon completion of this process, the application navigates the student to the project detail screen, which lists all generated topics and subtopics in an expandable manner. This approach was chosen since it presents the whole study plan at a glance without cluttering the screen with dozens of subtopics. When tapping on one of the subtopics, the application checks whether it has already generated content. If not, the content generation process is triggered, and an appropriate loading animation shows up, indicating that the content is being generated. Once the generation process completes, the student is presented with a subtopic detail screen with four large buttons for each of the four available modes

The four modes do not correspond to separate screens. Instead, they are stacked in a ZStack with horizontal slide transitions, allowing the user to switch modes via a dropdown. This way, the student does not need to navigate back to select a different mode, and there is no need to scroll between cards – everything is accessible at a glance.

The quiz mode presents multiple-choice questions one at a time. For every question, the student gets a brief prompt text and four answer options to choose. Once the student makes their choice, immediate visual feedback is provided in terms of highlighting the correct answer in green and the chosen one, if any, in red. Below the answer options, there is an explanation for the correct answer and, depending on the student's choice, for the answer they chose. The student progresses through the questions by pressing the Next Question button and receives the progress bar at the top of the screen, as well as the total number of questions.

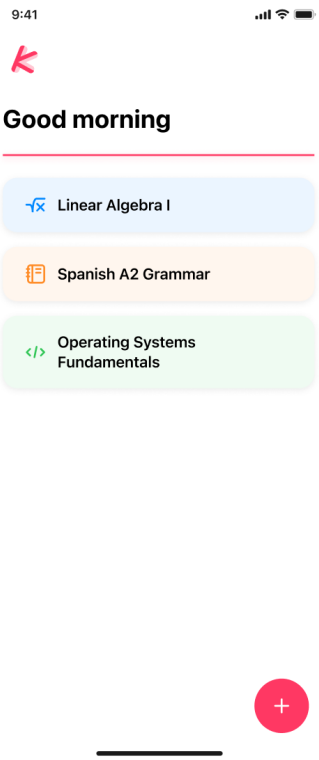
At the end of the quiz, the student sees a summary screen displaying their final score. They see their performance broken down by percentage and by exact answers: how many answers were correct and how many were wrong. Moreover, they can check if they failed or passed the test, according to a dynamic pass/fail threshold.

All the UI components use the same consistent visual language, defined using a collection of reusable components. KrickiNavBar replaces the system navigation bar with a unified interface: a back button, a centred title, and trailing actions. The cardStyle modifier applies shared properties (padding, background colour, and corner radius) to all card-like elements. Medium and subtle shadows (mediumShadow and subtleShadow) are used consistently without requiring additional customisations per component basis. Common UI component styling conventions (e.g., primary button and secondary button) are applied to buttons. Loader on tap interaction pattern is used to signal long running processes. Error message banners are used for informing the user about a generation failure.

Tab. 7: Primary screens and their roles

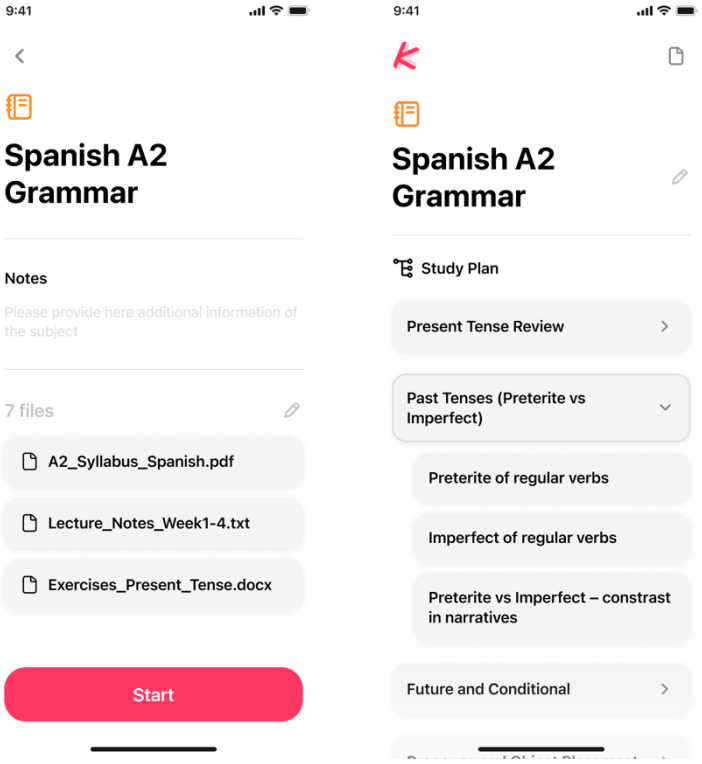
Screen	***Role***
ProjectsView	Home screen with project list and a button to create new projects.
NewProjectView	Two-step project creation: add title, notes, and documents, then start.
ProjectDetailView	Shows the study plan as an accordion of topics with expandable subtopics.
SubtopicDetailView	Mode menu offering four study modes: conspect, flashcards, quiz, vocabulary.
Study mode views	ConspectView, SubtopicFlashcardsView, SubtopicPracticeView, SubtopicTermsView.
DocumentEditorView	Sheet for adding and editing reference documents.

Source: own work



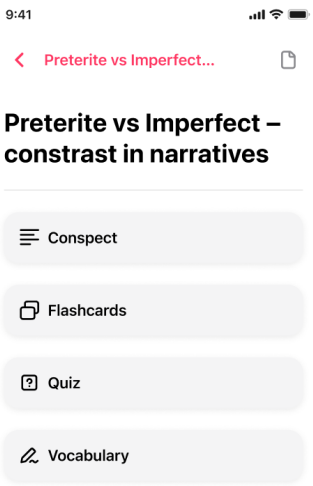
Attachment 1: Home screen with project list

Source: own work



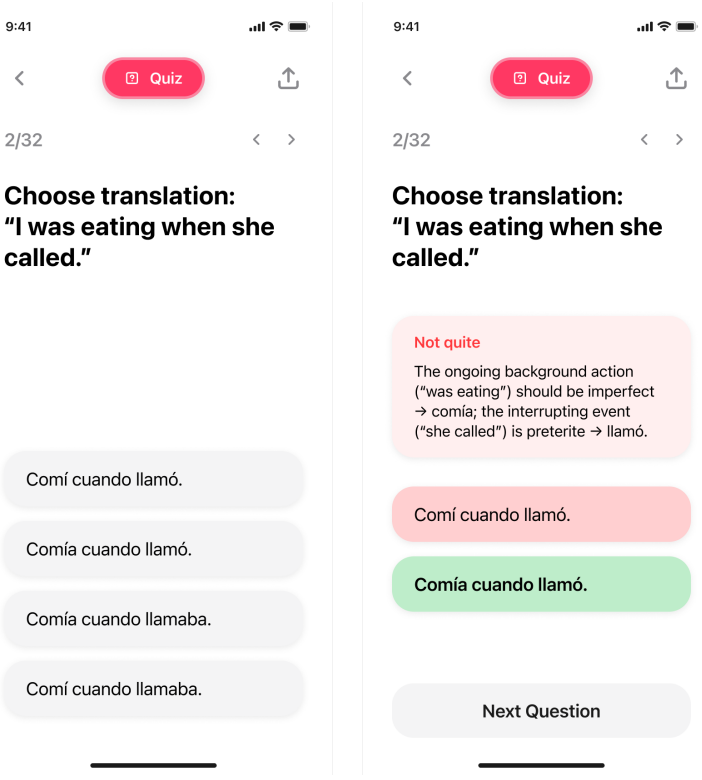
Attachment 2: Project creation (left) and project detail with study plan (right)

Source: own work



Attachment 3: Subtopic detail with four study modes

Source: own work



Attachment 4: Quiz mode – question (left) and answer feedback (right)

Source: own work

3.6 Smart, Scope-Aware Prompts

Generating content from AI heavily depends on prompt design. There are two core principles behind this approach.

3.6.1 Quantity driven by content

The generation does not use fixed output sizes. Instead, the model is instructed to produce as much content as needed to fully cover the given scope. This method prevents both overgeneration (too many unnecessary results) and undergeneration (too few generated results).

3.6.2 Scope enforced through context

In order to prevent scope drifts in the generated content, sibling subtopics are passed to the prompt and instructed explicitly to omit their content. This step is validated in GenerationValidator.

3.6.3 Prompt families

Every type of generated content is generated from a dedicated template. Every template includes a system and a user prompt and strictly limits the scope of the generated content, its quantity, and its formatting.

4 Implementation

The implementation follows the design described in the previous chapter. While the design section talks about what the application is supposed to do, the implementation section describes exactly how it does so – with snippets of code, diagrams and explanations for less obvious design decisions made during the development process.

4.1 Development Environment

The application is written in Swift in Xcode 16.2 with a deployment target of iOS 17. The Swift Package Manager is used for dependency management with only one external dependency – the MarkdownDisplayView package for rendering Markdown content in a WKWebView.

The application includes a unit test suite that covers the key components of the service and model layers. In addition to that, all the functional requirements are manually verified by scenario on a physical device.

4.2 Persistence with SwiftData

There is a single ModelContainer instance created when the application starts. This instance is wrapped by a small struct named PersistenceContainer. There are two variants of it – a production instance with an actual model store backed by a SQLite database, and an in-memory instance used exclusively in SwiftUI previews.

All entities from the data model are implemented as SwiftData @Model classes. Below is an excerpt from the Project class implementation as an example – each field is a stored property, and all relationships are annotated with @Relationship with a cascade deletion rule.

Helpers for accessing data inside SwiftUI views are declared in ModelExtensions.swift. Each entity exposes sorted array properties, such as topicArray in Project and flashcardArray in Subtopic. String-based helpers like subjectKind and projectColor give access to the underlying enums. Finally, the four answer choices for the Question entity are stored as a JSON-encoded string, because they are small, immutable and always accessed together.

The following shows the PersistenceContainer, created once on the main actor. All view-models access the shared container's main context.

```
import SwiftData
import SwiftUI

@MainActor
struct PersistenceContainer {
    static let shared = PersistenceContainer()

    let container: ModelContainer

    init(inMemory: Bool = false) {
        let schema = Schema([Project.self, Topic.self, Subtopic.self,
Document.self, Flashcard.self, Question.self, Term.self])
        let config = ModelConfiguration(schema: schema,
isStoredInMemoryOnly: inMemory)
        do {
            container = try ModelContainer(for: schema, configurations:
[config])
        } catch {
```

```
            fatalError("Failed to create ModelContainer: \(error)")
        }
    }
}
```

Code Snippet 1: PersistenceContainer.swift, lines 1–34

Source: own work

The Project class demonstrates the common entity structure. Each field is a stored property, and relationships use @Relationship with cascade delete rules (F4).

```
@Model
final class Project {
    var id: UUID
    var title: String
    var notes: String?
    var createdAt: Date
    var kind: String?
    var colorHex: String?
    var iconName: String?

    @Relationship(deleteRule: .cascade, inverse: \Topic.project) var topics:
[Topic] = []
    @Relationship(deleteRule: .cascade, inverse: \Document.project) var
documents: [Document] = []

    init(id: UUID = UUID(), title: String, notes: String? = nil, createdAt:
Date = Date(),
         kind: String? = nil, colorHex: String? = nil, iconName: String? =
nil) {
        self.id = id
        self.title = title
        self.notes = notes
        self.createdAt = createdAt
        self.kind = kind
        self.colorHex = colorHex
        self.iconName = iconName
    }
}
```

Code Snippet 2: SwiftDataModels.swift, lines 7–29

Source: own work

4.3 AI Client

All network communication is handled by a single AIClient singleton. A chat completion request is sent to the Perplexity Sonar API and content is extracted from the response. Request and response types are represented by structs compatible with API format.

The method responsible for making an actual request is makeRequest. It takes care of constructing the body, sending it via URLSession, parsing the response and extracting content of the first choice. By default, the model parameter is "sonar", the temperature is 0.2 and maxTokens is set to 4000.

Two techniques are used for improving the reliability of generation. The first one is the retry mechanism which applies to the structured generators like generateFlashcards, generateQuestions, extractTerms and generateLanguageGrammarRead. All these generators will be tried at most three times (for language grammar two retries are done) and a reminder message is appended every time a retry is made to ensure proper formatting of the response. The second technique is type validation which involves sending the parsing results to GenerationValidator.validateFlashcards, validateTerms or validateQuestions. If the result of the validation is null, then another retry is made.

For free-form notes, generateStudyContent is used, and their output is sanitized using GenerationValidator.sanitizeNotes and is saved without making any retries.

All possible errors from API responses and requests are unified under AIError. They are mapped to user-friendly messages and displayed as an error banner.

Request and response types mirror the Perplexity Sonar chat completions API format.

```
struct AIResponse: Codable {
    let choices: [Choice]

    struct Choice: Codable {
        let message: Message

        struct Message: Codable {
            let content: String
        }
    }
}

struct AIRequest: Codable {
    let model: String
    let messages: [Message]
    let temperature: Double
    let maxTokens: Int

    enum CodingKeys: String, CodingKey {
        case model, messages, temperature
        case maxTokens = "max_tokens"
    }

    struct Message: Codable {
        let role: String
        let content: String
    }
}
```

Code Snippet 3: AIClient.swift, lines 4–31

Source: own work

The makeRequest method handles request construction, sending, parsing, and content extraction. It is reused across all generation tasks with different prompts.

```
private func makeRequest(systemPrompt: String, userPrompt: String) async
throws -> String {
        guard let apiKey = apiKey, !apiKey.isEmpty else {
            throw AIError.missingAPIKey
        }

        guard let url = URL(string: baseURL) else {
            throw AIError.invalidURL
        }

        let request = AIRequest(
            model: configuredModel?.trimmingCharacters(in:
.whitespacesAndNewlines).isEmpty == false ?
configuredModel!.trimmingCharacters(in: .whitespacesAndNewlines) :
defaultModel,
            messages: [
                AIRequest.Message(role: "system", content: systemPrompt),
                AIRequest.Message(role: "user", content: userPrompt)
            ],
            temperature: 0.2,
            maxTokens: 4000
        )

        var urlRequest = URLRequest(url: url)
        urlRequest.httpMethod = "POST"
        urlRequest.addValue("Bearer \(apiKey)", forHTTPHeaderField:
"Authorization")
```

```
        urlRequest.addValue("application/json", forHTTPHeaderField:
"Content-Type")

        do {
            urlRequest.httpBody = try JSONEncoder().encode(request)
        } catch {
            throw AIError.encodingFailed
        }

        let (data, response) = try await URLSession.shared.data(for:
urlRequest)

        guard let httpResponse = response as? HTTPURLResponse else {
            throw AIError.invalidResponse
        }

        guard 200...299 ~= httpResponse.statusCode else {
            throw AIError.apiError(statusCode: httpResponse.statusCode)
        }

        do {
            let openAIResponse = try JSONDecoder().decode(AIResponse.self,
from: data)
            guard let content =
openAIResponse.choices.first?.message.content else {
                throw AIError.noContent
            }
            return content
        } catch {
            throw AIError.decodingFailed
        }
}
```

Code Snippet 4: AIClient.swift, lines 47–96

Source: own work

The retry-with-reminder pattern during flashcard generation. A reminder message is appended on each retry to reinforce the required output format, and a validation gate rejects invalid results.

```
func generateFlashcards(for subtopic: Subtopic, kind: SubjectKind) async
throws -> [(front: String, back: String)] {
        let context = ProjectContextBuilder.buildContext(for: subtopic)
        let basePrompt = PromptFactory.flashcardsPrompt(context: context)
        let system = PromptFactory.flashcardsSystemPrompt(kind: kind)

        for attempt in 1...3 {
            do {
                let prompt = attempt == 1 ? basePrompt : basePrompt +
"\n\nReminder: Output EXACTLY one card per line in the format 'Question |
Answer'. Do not include headings, numbering, or placeholders."
                let response = try await makeRequest(systemPrompt: system,
userPrompt: prompt)
                let pairs = parseFlashcardsResponse(response).map {
($0.front, $0.back) }
                let validated =
GenerationValidator.validateFlashcards(pairs)
                if !validated.isEmpty { return validated }
            } catch {
                if attempt == 3 { throw error }
                continue
            }
        }
        throw AIError.validationFailed(type: "flashcards")
}
```

Code Snippet 5: AIClient.swift, lines 266–284

Source: own work

4.4 Prompt Factory and Context Builder

There is a factory that defines one system prompt and one user prompt per each content generation task in PromptFactory.swift. The scope-aware strategy is applied here to maximize correctness.

For every request, a context is constructed in ProjectContextBuilder. For now, it consists of the following information:

- Project name and description.
- Current topic and subtopic names.
- Existing subtopic notes, if there are any.
- Names of all subtopics in the project, excluding the current subtopic and with an explicit instruction not to generate content for them.
- Documents attached to the project.

This way, it is possible to make sure the content will not deviate from the topic scope. Providing names for all siblings of the current subtopic and instructing not to cover them makes scope deviation even less likely compared to asking without context.

The system prompt for flashcard generation, enforcing scope limits and content-based quantity rather than a fixed number of cards.

```
static func flashcardsSystemPrompt(kind: SubjectKind) -> String {
        let scopeRule = "Stay strictly within THIS subtopic. Do not create
cards about material from other subtopics."
        let quantityRule = "Generate exactly as many cards as the content
warrants — a narrow concept may need 8–12, a broad one 20–35. Never generate
filler or trivially obvious cards."
        switch kind {
        case .language:
            return """
            Create vocabulary flashcards grounded ONLY in this lesson's
content.
            Output: one per line EXACTLY as "Word (target language) |
English translation".
            Each card should test one specific vocabulary item or phrase.
            \(scopeRule)
            \(quantityRule)
            No headings or numbering. Never output placeholders like
"Question | Answer".
            """
        case .general:
            return """
            Create flashcards based on the provided content.
            Output ONLY in this exact format, one per line: Question |
Answer
            Each card should test one specific fact, concept, or
relationship from this subtopic.
            \(scopeRule)
            \(quantityRule)
            No headings or numbering. Never output placeholders like
"Question | Answer".
            """
        }
}
```

Code Snippet 6: PromptFactory.swift, lines 37–60

Source: own work

ProjectContextBuilder.buildContext(for: Subtopic) assembles the AI context, including sibling subtopic names with an explicit exclusion instruction.

```
static func buildContext(for subtopic: Subtopic, maxLength: Int = 8000) ->
String {
        var context = ""
        if let topic = subtopic.topic, let project = topic.project {
            context += "Project: \(project.title)\n"
            context += "Topic: \(topic.title)\n"

            // Include sibling subtopic titles so the AI knows scope
boundaries
            let siblings = topic.subtopicArray.filter { $0.id != subtopic.id
}
            if !siblings.isEmpty {
                context += "Other subtopics in this topic (do NOT cover
these): \(siblings.map(\.title).joined(separator: ", "))\n"
            }
        }
        context += "Subtopic: \(subtopic.title)\n\n"

        if let notes = subtopic.content, !notes.isEmpty {
            context += "Study Notes (Subtopic):\n\(notes)\n\n"
        }

        if let topicOverview = subtopic.topic?.overview,
!topicOverview.isEmpty {
            context += "Topic Overview:\n\(topicOverview)\n\n"
        }

        let terms = subtopic.termArray
        if !terms.isEmpty {
            context += "Vocabulary Terms (existing):\n"
            for term in terms.prefix(80) {
                context += "- \(term.term): \(term.definition)\n"
            }
            context += "\n"
        }

        if let project = subtopic.topic?.project {
            let documents = project.documentArray
            if !documents.isEmpty {
                context += "Documents:\n"
                for document in documents {
                    context += "\(document.title):\n\(document.content)\n\n"
                    if context.count > maxLength {
                        context = String(context.prefix(maxLength))
                        context += "\n[Content truncated...]"
                        break
                    }
                }
            }
        }

        return context
}
```

Code Snippet 7: PromptFactory.swift, lines 336–383

Source: own work

4.5 Full Subtopic Generation Pipeline

Opening a subtopic in the application (UC3) starts a series of generation tasks for each piece of subtopic content – notes, flashcards, questions and vocabulary terms. It is performed by ProjectDetailViewModel.generateAllContent(for:) method. The process consists of four sequential steps:

1. Notes or language grammar lessons.
2. Flashcards.
3. Quiz questions.
4. Vocabulary terms.

Every task result is stored in the database immediately, and UI is updated with a loading bar per each subtopic. The generation pipeline is idempotent: if some content exists in SwiftData, that particular step is skipped. Also, there is a switch between notes and language grammar depending on the project mode.

The following sequence diagram describes the overall flow of events when a student opens a subtopic in their project.

The corresponding flowchart with more emphasis on error handling is shown below. If some generation step fails, an error banner appears, but partially generated content stays in place.

The generateAllContent(for:) method implements the full subtopic generation pipeline: notes, flashcards, quiz, and terms in sequence.

```
func generateAllContent(for subtopic: Subtopic) async throws {
        generatingSubtopicID = subtopic.id
        defer { generatingSubtopicID = nil }

        let kind = project.subjectKind ??
SubjectClassifier.classify(projectTitle: project.title, topics: topics)

        // 1. Notes
        await ensureSubtopicNotes(for: subtopic)

        // 2. Flashcards (if none exist)
        if subtopic.flashcardArray.isEmpty {
            let pairs = try await aiClient.generateFlashcards(for: subtopic,
kind: kind)
            for pair in pairs {
                let card = Flashcard(front: pair.front, back: pair.back)
                card.subtopic = subtopic
                modelContext.insert(card)
            }
            saveContext()
        }

        // 3. Questions (if none exist)
        if subtopic.questionArray.isEmpty {
            let drafts = try await aiClient.generateQuestions(for: subtopic,
kind: kind)
            for draft in drafts {
                let q = Question(prompt: draft.prompt, correctIndex:
Int16(draft.correctIndex), explanation: draft.explanation)
                q.setChoices(draft.choices)
                q.subtopic = subtopic
                modelContext.insert(q)
            }
            saveContext()
        }

        // 4. Terms (if none exist)
        if subtopic.termArray.isEmpty {
            let pairs = try await aiClient.extractTerms(for: subtopic, kind:
kind)
            for pair in pairs {
                let term = Term(term: pair.term, definition:
pair.definition)
                term.subtopic = subtopic
                modelContext.insert(term)
            }
            saveContext()
        }
}
```

Code Snippet 8: ProjectDetailViewModel.swift, lines 201–243

Source: own work

The ensureSubtopicNotes(for:) method branches between general study notes and language grammar lessons based on the project mode.

```
func ensureSubtopicNotes(for subtopic: Subtopic) async {
        if let content = subtopic.content, !content.isEmpty { return }
        let kind = project.subjectKind ??
SubjectClassifier.classify(projectTitle: project.title, topics: topics)
        do {
            let md: String
            if kind == .language {
                md = try await aiClient.generateLanguageGrammarRead(for:
subtopic)
            } else {
                md = try await aiClient.generateStudyContent(for: subtopic,
kind: kind)
            }
            await MainActor.run {
                subtopic.content = GenerationValidator.sanitizeNotes(md)
                self.saveContext()
            }
        } catch {
            await MainActor.run { self.subtopicError =
error.localizedDescription }
        }
    }
```

Code Snippet 9: ProjectDetailViewModel.swift, lines 155–172

Source: own work

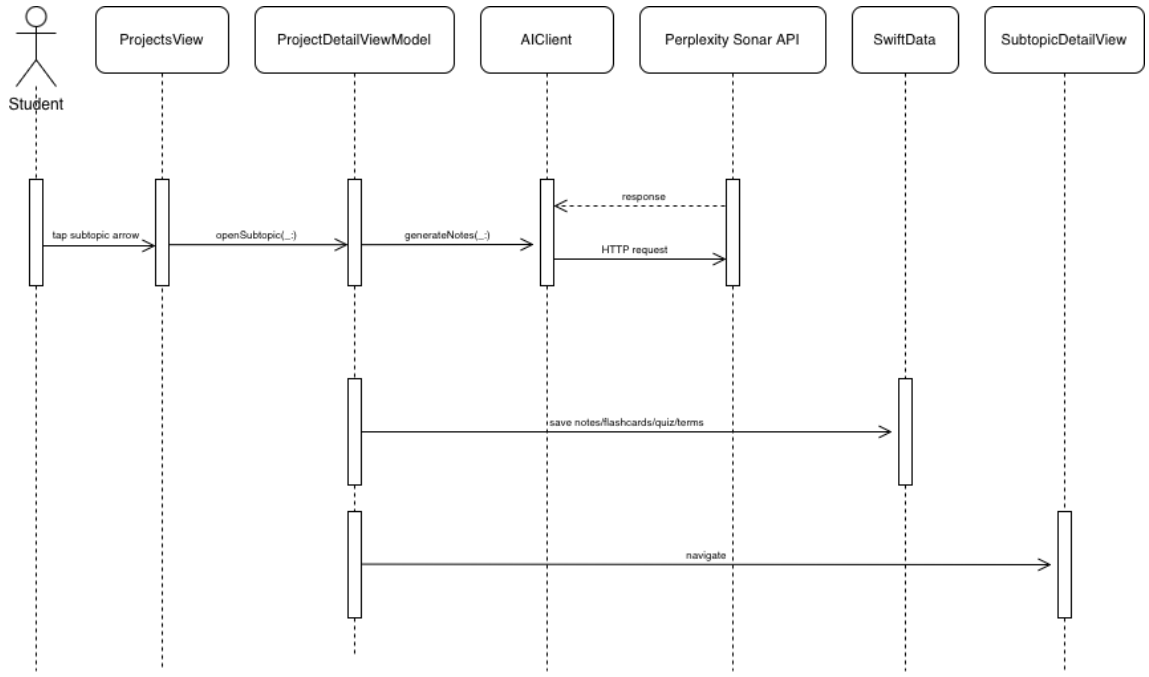


Figure 4: Sequence diagram for opening a subtopic

Source: own work

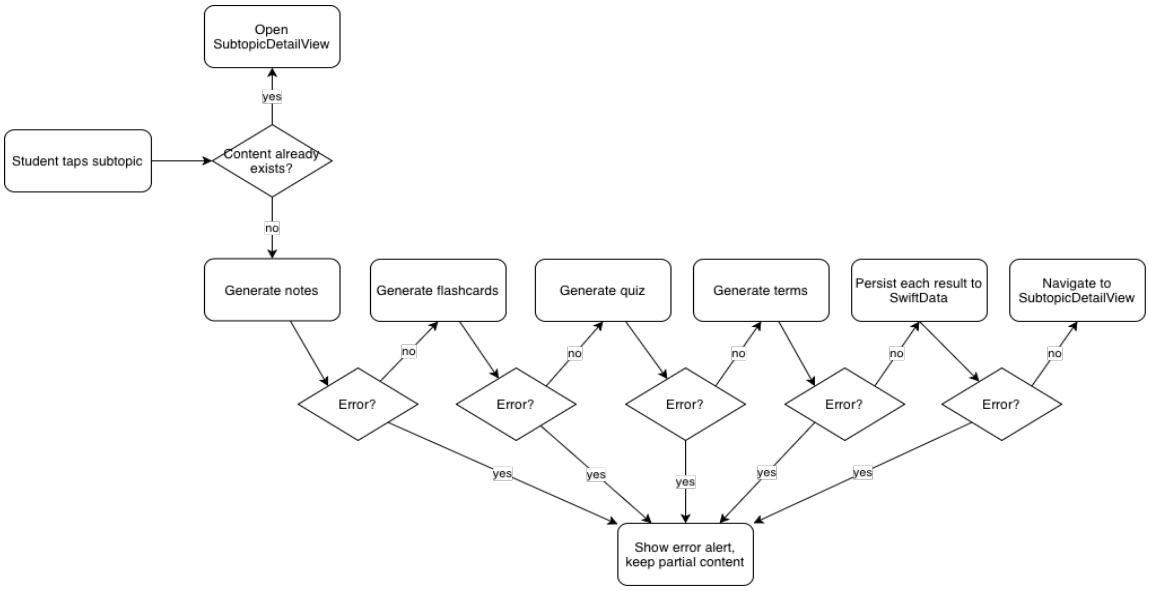


Figure 5: Subtopic generation pipeline flowchart

Source: own work

4.6 Automatic Project Mode Classifier

There is no need to explicitly assign the general project mode to every project, because it can be detected based on certain keywords. It is done in SubjectClassifier using a regular expression that matches any languages from the provided list and certain phrases (e.g. "grammar" or "vocabulary").

This approach does not require any additional API requests and gives results immediately.

The keyword-based SubjectClassifier that determines whether a project is general or language-based without an API call.

```
struct SubjectClassifier {
    static func classify(projectTitle: String?, topics: [Topic]) ->
SubjectKind {
        let title = (projectTitle ?? "").lowercased()
        let topicTitles = topics.map { $0.title.lowercased() }
        let haystack = ([title] + topicTitles).joined(separator: " ")

        let languageKeywords = [
            "language", "vocabulary", "grammar", "conjugation",
"pronunciation", "listening", "speaking",
            "spanish", "french", "german", "italian", "portuguese",
"japanese", "korean", "chinese",
            "arabic", "russian", "hindi", "urdu", "polish", "dutch",
"swedish", "norwegian", "danish",
            "greek", "turkish"
        ]
        if languageKeywords.contains(where: { haystack.contains($0) }) {
return .language }

        return .general
    }
}
```

Code Snippet 10: SubjectClassifier.swift, lines 3–21

Source: own work

4.7 Study Sessions

The application implements two types of study sessions that are controlled by value-type structs and do not interact with the database at all. First of all, there is FlashcardSession. It includes the current flashcard index, set of flashcards seen in the session, and set of successfully recognised cards.

There are three options how to start a session – with all the flashcards in the subtopic, only new or unknown flashcards, and a random quick preview of some of them. PracticeSession is responsible for quiz questions. Similar to flashcards, it stores the current question index, answers already given by the student and total score. The pass threshold is computed by the view-model.

There is also a static vocabulary view where students can see the list of vocabulary terms filtered by known/unknown status.

FlashcardSession is a value-type struct tracking the current card, reviewed set, and correct set during a flashcard study session.

```
import Foundation

struct FlashcardSession {
    let flashcards: [Flashcard]
    var currentIndex: Int
    var reviewedCards: Set<UUID>
    var correctCards: Set<UUID>

    init(flashcards: [Flashcard]) {
        self.flashcards = flashcards
        self.currentIndex = 0
        self.reviewedCards = Set()
        self.correctCards = Set()
    }

    var currentCard: Flashcard? {
        guard currentIndex < flashcards.count else { return nil }
        return flashcards[currentIndex]
    }

    var isComplete: Bool {
        return currentIndex >= flashcards.count
    }

    var progress: Double {
        guard !flashcards.isEmpty else { return 0 }
        return Double(reviewedCards.count) / Double(flashcards.count)
    }

    mutating func nextCard() {
        if currentIndex < flashcards.count - 1 {
            currentIndex += 1
        }
    }

    mutating func markCurrentCard(correct: Bool) {
        guard let currentCard = currentCard else { return }
        reviewedCards.insert(currentCard.id ?? UUID())
        if correct {
            correctCards.insert(currentCard.id ?? UUID())
        }
    }
}
```

Code Snippet 11: FlashcardSession.swift, lines 1–43

Source: own work

PracticeSession is a value-type struct tracking the current question, answers given, and running score during a quiz session.

```swift
import Foundation

struct PracticeSession {
    let questions: [Question]
    var currentIndex: Int
    var answers: [UUID: Int] // QuestionID -> Selected Answer Index
    var score: Int

    init(questions: [Question]) {
        self.questions = questions
        self.currentIndex = 0
        self.answers = [:]
        self.score = 0
    }

    var currentQuestion: Question? {
        guard currentIndex < questions.count else { return nil }
        return questions[currentIndex]
    }

    var isComplete: Bool {
        return currentIndex >= questions.count
    }

    var progress: Double {
        guard !questions.isEmpty else { return 0 }
        return Double(answers.count) / Double(questions.count)
    }

    mutating func answerCurrentQuestion(_ answerIndex: Int) -> Bool {
        guard let currentQuestion = currentQuestion else { return false }

        let questionId = currentQuestion.id ?? UUID()
        answers[questionId] = answerIndex

        let isCorrect = answerIndex == currentQuestion.correctIndex
        if isCorrect {
            score += 1
        }

        return isCorrect
    }

    mutating func nextQuestion() {
        if currentIndex < questions.count - 1 {
            currentIndex += 1
        }
    }

    func finalScore() -> Double {
        guard !questions.isEmpty else { return 0 }
        return Double(score) / Double(questions.count)
    }
}
```

Code Snippet 12: PracticeSession.swift, lines 1–54

Source: own work

4.8 Notable User Interface Features

There are several approaches used in UI development to make the application responsive and easy to use.

4.8.1 Async generation on the main actor

All view-models run on the main actor (@MainActor), which allows state updates from asynchronous AI calls to be applied safely without race conditions.

4.8.2 Loader-on-tap

When a subtopic is being opened by tapping, the loader is shown only for the selected subtopic, leaving the rest open to interaction to prevent redundant requests.

4.8.3 Error banner

In case of failure in content generation, an error banner will appear at the top of the screen and will close automatically after some time, not interrupting other operations.

4.8.4 Custom transitions

Views representing different study modes are stacked in the form of a ZStack with offset-based horizontal transitions rather than the usual NavigationStack push animations. Thus, it is possible to switch between study modes with dropdown menu.

4.8.5 Session-aware back navigation

When the student starts a flashcards or a quiz session, the back button will take the user to session options screen instead of closing the whole study mode.

4.9 Security and Privacy

All data stays on device, and no information is sent anywhere except the content generation prompts to Perplexity Sonar API endpoint. It includes only titles, notes and the context that will be used to generate content. No personal information is included in these prompts. Any errors returned from Perplexity are gracefully handled to ensure the application does not fail and user data is consistent.

Currently, Perplexity API key is hardcoded into Info.plist in order to make development easier and eliminate any additional dependencies. Unfortunately, this approach is insecure for a production release. An API key embedded directly into application binaries can be easily extracted by anyone having access to the binaries (including decompiled and unpacked versions). After extraction, API key can be misused in various malicious ways, including unlimited requests that the developer will be charged for and violation of the API provider's policies.

In order to securely release the application, it is necessary to move API key from the client entirely. One approach is to build a backend proxy. It would receive generation requests from the app, attach the API key internally, forward them to Perplexity, and return the responses.

Alternatively, if running backend is not desirable, Arkana obfuscator can be used. Arkana converts secret keys to Swift code and splits and encrypts them. It will be significantly more difficult for an attacker to get access to the key. However, it is still possible with reverse-

engineering. Thus, using Arkana as obfuscation should be viewed as a deterrent rather than a complete protection against API misuse.

4.10 Automated Testing

There is a unit test suite consisting of 46 cases in four test files. Those tests cover four parts of the application that are most vulnerable to bugs: response parsing, content validation, JSON serialization, and session handling.

There is a large test suite for the most complicated part of the project, which is parsing Perplexity Sonar API response. This task consists of creating structures for topics, subtopics, notes, flashcards, quizzes, and vocabulary terms. Moreover, the output format is not always consistent, which means that different types of data might be found in different formats in the same response. Some examples are pipe-separated vs Q:/A: format, inline vs multiline entries, bullet points. That is why those parsers are highly error-prone.

Tests feed the parser with realistic data of all formats that might occur in the API response and check whether it creates the expected amount of items with proper contents. The test suite also covers invalid API responses. These include cases where no valid data is recognised, the correct-answer marker is missing, or the number of answer choices is insufficient.

In addition to the tests for parsers, there is a series of tests that check the sanitizing functionality. Those tests cover topic sanitization, which is used for removing any meta-labels from topics (e.g. Review, Quiz). They also ensure duplicates in topic list are removed using case-insensitive comparison, and no topics with fewer than four subtopics are generated.

The content generated by Perplexity Sonar API is further validated before showing it to the student. These tests cover all validation aspects: deduplication by normalised content, rejection of placeholders and echo answers, minimum length filtering, and multiple-choice checks such as duplicate options and invalid indices.

The fourth and smallest suite of tests covers JSON serialization of flashcard answers and quizzes. The purpose of these tests is to verify that serialized answers include correct values, even the ones containing apostrophes, quotation marks or other special characters. In addition to this, the tests check that in case of malformed JSON, serialization fails gracefully.

There are two value-type structs – FlashcardSession and PracticeSession. They are covered by a comprehensive test suite, which checks the initial state, index and state advancement, correct and wrong answer handling and final scoring.

Tests for AIClient and the view-models are out of scope. They would require mocking URLSession and significantly expanding the test infrastructure.

In addition to the automated test suite, each functional requirement (F1–F21) was verified manually on a physical device by executing a scenario that exercises the corresponding feature. All requirements passed verification.

5 Results and Discussion

The application is built, implemented and functional. What follows is a fair evaluation of its capabilities, weaknesses and potential improvements.

5.1 Result

The result is a functioning native iOS application. It accepts a student-provided subject, generates a topic–subtopic hierarchy, and produces four content types per subtopic.

The workflow requires minimal interaction – creating a project and selecting a subtopic. Behind this workflow, the application classifies the subject, builds a multi-level outline, and generates all materials using the Perplexity Sonar API. Everything is persisted locally and accessible through four study modes.

Functional requirements F1–F21 are satisfied and verified.

5.2 Evaluation

Content-focused prompts turned out to be the crucial design decision made during the implementation. Prompts were redesigned to define scope by task (e.g., "create cards until all facts are covered"). This reduced both filler generation and overly narrow content. Besides, including the list of the subtopics' titles in the context and specifying not to cover those particular topics helped reduce scope drift even further.

The choice of a web-grounded model, as discussed in Section 1.4, helped reduce factual inaccuracies in the generated content, particularly incorrect dates, definitions, and examples.

A single click to generate content was achieved using the generateAllContent(for:) function. Coupled with the loader-on-tap pattern, this allowed for an extremely streamlined experience that resembled browsing a book rather than generating content

Since all data is saved locally, the application remains functional without an internet connection. Thus, all the generated material can be studied in a library, on a train or anywhere else without access to the Internet.

There are several key limitations of the application. First of all, the whole application depends on an external API. Thus, without an internet connection, no content can be generated, and if the Perplexity API becomes unavailable or is altered in some way, the application becomes unusable. Second, web grounding does not guarantee complete accuracy. Some subjects (particularly, those that are very advanced) cannot be completely covered by the model, resulting in content with factual inaccuracies. There is also no option to manually modify or delete any cards, quiz questions and notes. Third, there is no personalisation or progress tracking. The application does not know how proficient the student is or which topics need revision. The known/unknown flag is set manually, not intelligently. Fourth, there is no spaced repetition scheduler in the application yet, so the benefits of spaced repetition are not realised. Finally, all the information is stored locally on a single device, and the application is available only on iOS.

5.3 Future Work

The extensions below are arranged in order of their expected benefits:

1. Spaced-repetition scheduler: add a "next review" date for each card and create a review queue to turn the "unknown only" mode into a full spaced review session.
2. iCloud sync: replace the SwiftData store with a CloudKit-backed version to allow seamless project synchronisation across devices.
3. Project sharing: enable exporting a project from one user and importing it for another.
4. Support for images and PDF files: extend document handling beyond plain text by adding PDF parsing and OCR capabilities.
5. Broader language support: the interface is currently English-only; internationalisation would expand accessibility.
6. Integration testing: extend unit tests with end-to-end tests using a mocked API and in-memory SwiftData container.
7. Local model fallback: introduce an on-device model as a fallback when offline, accepting reduced accuracy.

Although the current version already solves the initial problem by removing the overhead of preparing study materials, these improvements would significantly increase its long-term usefulness.

Conclusion

The aim of this thesis was to design and implement a native iOS application that generates a structured study plan from a user-provided subject. For each subtopic, it produces notes, flashcards, a quiz, and a vocabulary list using a web-grounded language model. All data is stored locally for offline use.

This objective has been achieved. The Kricki application accepts a subject title and optional reference materials, produces a hierarchical structure of topics and subtopics, and generates all four content types for each subtopic with a single interaction. The use of the Perplexity Sonar API improves factual reliability, while SwiftData ensures all content is persisted locally. All functional (F1–F21) and non-functional (N1–N8) requirements have been implemented and verified through testing on a physical device.

The theoretical concepts directly shaped the design. Micro-learning led to subtopic-sized units, active recall informed the flashcard and quiz modes, and web grounding determined the model choice. The analysis of existing tools confirmed a gap. No current solution combines structured planning, automatic content generation, offline capability, and a language-learning mode in a native iOS app.

The result is a working prototype. Testing across subjects such as Linear Algebra, Spanish A2 Grammar, and Operating Systems showed generally useful outputs, though occasional inaccuracies and limited depth for advanced topics remain. The most impactful next steps include adding spaced repetition, enabling cross-device synchronisation, allowing content editing, and moving API key handling to a secure backend before public release.

References

APPLE INC., 2024a. *SwiftUI Documentation [online].* [cit. 2026-01-08]. Available from: https://developer.apple.com/documentation/swiftui

APPLE INC., 2024b. *SwiftData Documentation [online]* [cit. 2026-01-08]. Available from: https://developer.apple.com/documentation/swiftdata

BIGGS, John. *Teaching for Quality Learning at University*. 2nd ed. Buckingham: Open University Press, 2003. ISBN 0-335-21168-2.

EBBINGHAUS, Hermann., 1913. *Memory: A Contribution to Experimental Psychology [online]*. [cit. 2026-04-10]. Available from: https://psychclassics.yorku.ca/Ebbinghaus/index.htm

HOLMES, Wayne, BIALIK, Maya a FADEL, Charles. *Artificial Intelligence in Education: Promises and Implications for Teaching and Learning*. 2019. ISBN 978-1794293700.

IBM, n.d. *Natural Language Processing Guide [online]*. [no date] [cit. 2026-01-08]. Available from: https://www.ibm.com/think/topics/natural-language-processing

KARPICKE, Jeffrey D. a Janell R. BLUNT. *Retrieval Practice Produces More Learning than Elaborative Studying with Concept Mapping*. Science, 2011, 331(6018), 772–775. DOI: 10.1126/science.1199327

KAUSHIK, Abhishek et al., 2025. *Exploring the Impact of Generative Artificial Intelligence in Education: A Thematic Analysis [online]*. [cit. 2026-01-08]. Available from: https://arxiv.org/html/2501.10134v1

KNOWLES, Malcolm S. *Self-Directed Learning: A Guide for Learners and Teachers*. Chicago: Follett Publishing Company, 1975. ISBN 978-0-695-81116-7.

LEWIS, Patrick et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. Advances in Neural Information Processing Systems [online]*. 2020, [cit. 2026-01-08]. Available from: https://arxiv.org/abs/2005.11401

MAITY, Subhankar a Aniket DEROY. *The Future of Learning in the Age of Generative AI: Automated Question Generation and Assessment with Large Language Models [online]*. 2024 [cit. 2026-04-10]. Available from: https://arxiv.org/abs/2410.09576

PERPLEXITY AI, 2024. *Sonar API Documentation [online]*. [cit. 2026-01-08]. Available from: https://docs.perplexity.ai/

TABIBIAN, Behzad et al. *Enhancing Human Learning via Spaced Repetition Optimization. Proceedings of the National Academy of Sciences*, 2019, 116(10), 3988–3993. DOI: 10.1073/pnas.1815156116.

TROWBRIDGE, Stephanie, Clair WATERBURY and Lindsey SUDBURY. *Learning in Bursts: Microlearning with Social Media.* EDUCAUSE Review *[online]*. 2017 [cit. 2026-01-08]. Available from: https://er.educause.edu/articles/2017/4/learning-in-bursts-microlearning-with-social-media

ZIMMERMAN, Barry J. *Becoming a Self-Regulated Learner: An Overview. Theory Into Practice*, 2002, 41(2), 64–70. DOI: 10.1207/s15430421tip4102_2.