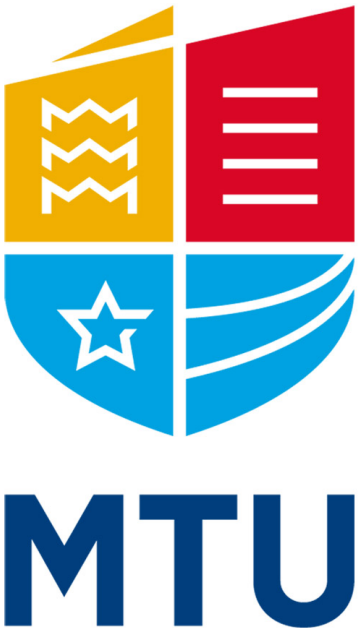


NoteReader, an Application Suite for Cataloging Study Material and Notes

by

Emmett Fitzharris

This thesis has been submitted in partial fulfillment for the degree of Bachelor of Science (Hons) in Software Development

in the
Faculty of Engineering and Science
Department of Computer Science

June 2025

Declaration of Authorship

This report, NoteReader, an Application Suite for Cataloging Study Material and Notes, is submitted in partial fulfillment of the requirements of Bachelor of Science (Hons) in Software Development at Munster Technological University Cork. I, Emmett Fitzharris, declare that this thesis titled, NoteReader, an Application Suite for Cataloging Study Material and Notes and the work represents substantially the result of my own work except where explicitly indicated in the text. This report may be freely copied and distributed provided the source is explicitly acknowledged. I confirm that:

- This work was done wholly or mainly while in candidature Bachelor of Science (Hons) in Software Development at Munster Technological University Cork.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at Munster Technological University Cork or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this project report is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: Emmett Fitzharris

Date: May 2025

MUNSTER TECHNOLOGICAL UNIVERSITY CORK

Abstract

Faculty of Engineering and Science
Department of Computer Science

Bachelor of Science (Hons)

by Emmett Fitzharris

When revising a topic, the first approach is usually to review the notes you took when studying the material. These notes are often shorthand and are usually incomplete. As a result it is important to reference back to the original material for full understanding.

Managing storage of documents alongside digital note taking causes friction in the learning process, and without good self imposed organization it is easy to lose reference material.

The purpose of this project is to create a suite of applications which assist in cataloging digital notes alongside study material.

There are a large number of existing note taking solutions. None however treat document reading as a first class priority. Likewise there are a large number of document readers available, however none treat note taking as a first class priority.

Current solutions (such as Microsoft OneNote) require large file sizes, synced between devices using expensive cloud storage providers, often requiring costly monthly subscriptions.

Acknowledgements

Project Supervisors:

- **Semester 1:** Kapal Dev
- **Semester 2:** Deirdre Dunlea

Technologies Used

- compose-pdf: zt64
  License: MIT
- compose-rich-editor: MohamedRejeb
  License: Apache License 2.0
- Calibre
  License: GNU General Public License v3
- LibreOffice
  License: Mozilla Public License Version 2.0
- Apache PDFBox
  License: Apache License 2.0
- Apache POI
  License: Apache License 2.0
- docx4j
  License: License: Apache License 2.0
- JetBrains Compose Multiplatform
  License: Apache License 2.0

Contents

List of Figures

List of Tables

Abbreviations

API	**A**pplication **P**rogramming **I**nterface **ACM**
Association for **C**omputing **M**achinery	
AI	**A**rtificial **I**ntelligence
CCS	**C**omputing **C**lassification **S**ystem
CSV	**C**omma-**S**eparated **V**alues
DOCX	**D**ocument **O**pen **X**ML
EPUB	**E**lectronic **PUB**lication
HTML	**H**yper**T**ext **M**arkup **L**anguage
ISO	**I**nternational **O**rganization for **S**tandardization
LFS	**L**arge **F**ile **S**torage
MD	**M**ark**D**own
MVP	**M**inimum **V**iable **P**roduct
OCR	**O**ptical **C**haracter **R**ecognition
PDF	**P**ortable **D**ocument **F**ormat
PC	**P**ersonal **C**omputer
PPTX	**P**ower**P**oint **T**ext **X**ML
SQL	**S**tructured **Q**uery **L**anguage
TXT	**T**e**X**t **T**ext Format
UI	**U**ser **I**nterface
UML	**U**nified **M**odeling **L**anguage
UX	**U**ser **E**xperience
WPF	**W**indows **P**resentation **F**oundation **XLSX**
E**x**cel **S**preadsheet **X**ML	
XML	E**x**tensible **M**arkup **L**anguage

Dedicated to my family, who have graciously supported my return to education...

Chapter 1

Introduction

Imagine never having to search through disorganized notes or wonder if you saved that crucial PDF from last week's lecture. NoteReader will address the core challenges of disjointed learning resources and disconnected note-taking, offering a unified platform that promotes better learning, better organization, and better revision.

NoteReader is a study solution that provides an all-in-one interface for document reading and note-taking. By supporting a wide range of file types, including PDFs, Word documents, PowerPoint slides, and eBooks, allowing users to seamlessly organize their study material and notes side-by-side.

The core interface of NoteReader is the Split-View. Each time the user opens a document a dedicated folder is created in the user's file system to store notes linked directly to the document and its page number. The document is displayed on the left pane of the window, while a text editor is displayed on the right. On returning to NoteReader, previously viewed documents will be accessible via a catalogue/ library.

The note editor will support Markdown and HTML, allowing users to create rich, formatted notes. As users navigate through the document, new notes are created automatically for each page. If a user returns to a previous page, the corresponding note is instantly loaded in the editor. Optionally pages can be merged to allow for a more seamless experience.

Notes are stored in lightweight plain-text files, they are inherently portable ensuring compatibility with other tools like Obsidian. Notes are saved in real time and can be automatically synced to version control providers like GitHub, giving users access to version-controlled notes across devices.

Features such as document cataloguing, page-specific notes, and a robust tagging system make it easy for users to locate specific information. Advanced features like handwriting

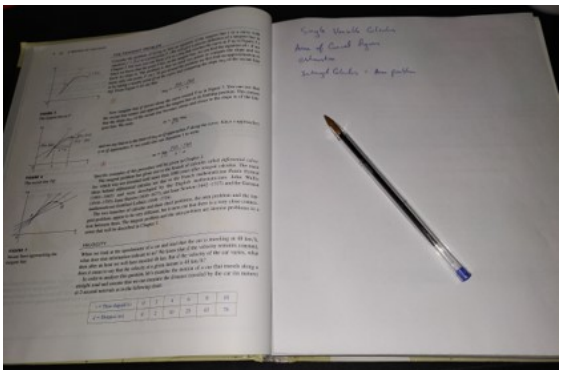


FIGURE 1.1: A book with a blank sheet of paper slotted between the pages to take notes on

support and time-stamped notes for videos and audiobooks are planned as potential stretch goals.

1.1 Motivation

You are revising for an exam. You open a lecture slide on your laptop, but where are your notes? Are they on Google Docs, in OneNote, or saved as scattered files on your desktop? For many students, this process is repetitive and inefficient. Modern note-taking tools do not treat source material as a first-class priority. Instead, they focus on creating isolated notes with minimal connection to the material being studied.

Picture a book with a blank sheet of paper slotted in between each page. As you read, you take notes on this paper and as you flip to the next page, a new sheet awaits you. When it's time to revise, you go back to that page of the book and the sheet containing your notes on the topic are right there waiting for you.

No flipping between refill pads, copy-books, documents on your laptop, or in the cloud. No wondering if the notes you took are complete, or if you missed an important point. Everything is in one place.

Now combine this with digital document management and cataloguing, tagging, searching and linking. It should be trivial to find your notes when you need to refer back.

This is the feeling note reader wants to achieve.

Note reader aims to reduce the friction associated with storing and accessing notes and study material. Reduce the cognitive load of note-taking and encourage real-time engagement with the material.

1.2 Contribution

The application proposed in this project is composed of several key features, such as document reading and rendering, text editing, cataloguing and multiplatform support. To orchestrate these components in a seamless manner, I will leverage the skills and knowledge gained throughout my degree program.

I will apply user-centred design principles through requirements engineering to ensure NoteReader addresses the clear needs of its intended users. This includes gathering, prioritizing, and validating functional and non-functional requirements.

Adherence to object-oriented design principles (like encapsulation, abstraction, inheritance, and polymorphism) will ensure a cohesive, modular system. Core functionalities like document viewing, note synchronization, and user interaction will be implemented as modular, reusable components.

By following object-oriented design principles, NoteReader will be maintainable and extensible. Changes to one part of the system (like file type support) can be isolated to specific classes or components without affecting the rest of the application. This modular approach ensures scalability as new features are introduced.

NoteReader will be developed using the agile Scrum framework, focusing on iterative development. Emphasis will be placed on early delivery of a minimum viable product (MVP) with successive prototype releases. These iterations will allow for continuous feedback loops and improvement.

Knowledge of agile methodologies will enable me to manage project scope, prioritize features, and ensure timely delivery of new prototypes. Skills in backlog management, sprint planning, and stakeholder collaboration will be demonstrated through well-documented sprints and the inclusion of a feedback process for user testing.

Since NoteReader is a multiplatform solution, user accessibility across devices is a core requirement. The UI must adapt to multiple screen sizes and input methods (desktop, tablet, mobile). By using Kotlin Multiplatform, I will leverage platform-specific adaptations for mobile and desktop environments.

The application will maintain a consistent user experience (UX) across Windows, macOS, Linux, and mobile devices. Designing UI components that work equally well on different devices will showcase the skills I gained from mobile development modules. The ability to seamlessly handle document navigation and note creation across platforms is central to NoteReader's usability.

One of NoteReader's unique value propositions is the use of version control services (like GitHub and GitLab) as a method of synchronizing notes. This method ensures that users have access to their notes across multiple devices while maintaining a revision history of their notes.

This feature demonstrates my ability to develop tools that leverage existing cloud infrastructure to deliver a seamless synchronization experience. My knowledge of Git API integration will allow me to build features for automatic sync, conflict resolution, and file version tracking.

A No-SQL file-system-based approach will be used to store and organize notes, metadata, and links to source documents. This approach provides flexibility in handling diverse data formats (PDFs, Word documents, PowerPoints, etc.) and ensures that all associated notes are easily searchable.

The catalogue file (or index) will be synced alongside the notes themselves in the Git repository. This ensures that the state of the catalogue remains synchronized with user notes, even when accessed from multiple devices.

By using Git for note synchronization, the catalogue must be resilient to merge conflicts and data corruption. The system will be designed to track changes to catalogue files, and where conflicts arise, the system will attempt to resolve them automatically or rebuilding, alerting the user only if manual intervention is required.

1.3 Structure of This Document

This document is organized into the following sections:

Section	**Description**
Introduction	Outlines the motivation behind the project, the key contributions, and an overview of the document's structure.
Background	Provides context for the project, including its thematic position within computer science, a review of existing solutions, and relevant literature on the topic of digital note-taking and learning efficiency.
Problem Definition	Discusses the core challenges that the project seeks to address, outlining existing issues with current solutions and presenting the objectives and functional requirements of the proposed application.
Implementation Approach	Details the methodology used to develop the application, including user flows, architectural design, technologies, risk assessment, and the implementation timeline.
Implementation	Details the actual implementation of the project and difficulties encountered.
Testing and Evaluation	Breaks down the methods used to evaluate progress
Conclusions and Future Work	Summarizes the findings and learnings of the project, reflects on the challenges encountered, and outlines potential future developments.
Appendices	Includes supplementary materials such as code snippets, wireframe models, and diagrams to support the main text.

Table 1.1: Structure of This Document

Chapter 2

Background

2.1 Thematic Area within Computer Science

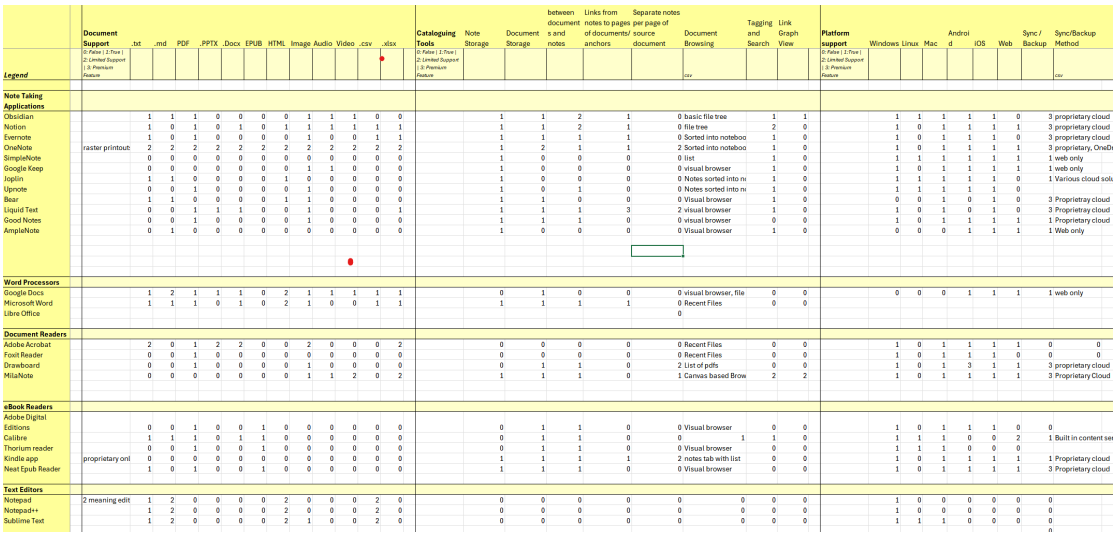
The project of developing NoteReader, lies at the intersection of educational technology (EdTech) and software development. EdTech is a rapidly evolving field characterized by the integration of modern technologies into learning environments to enhance educational outcomes. Current trends in EdTech include the use of AI-driven personalized learning systems, immersive experiences through AR/VR, and data-driven analytics for adaptive learning paths.

However, unlike many modern EdTech initiatives that prioritize AI or AR/VR enhancements, this project addresses a more fundamental aspect of learning, the effective organization and access of study materials. By tackling the challenge of document-linked note-taking, NoteReader aims to simplify the learning process, reduce cognitive load, and enable efficient study workflows. Foundational learning strategies are essential, even as new technologies emerge.

To achieve its goals, this project draws upon principles from several key areas of computer science. These areas provide the theoretical and practical foundations required to develop a robust, scalable, and user-friendly application.

ACM Computing Classification System (CCS) Concepts

Human-centered computing: User interface design
Information systems: Document collection models
Information systems: Search interfaces
Software and its engineering: Software creation and management



	Document Support	.txt	.md	PDF	.PPTX	.Docx	EPUB	HTML	Image	Audio	Video	.csv	.xlsx	Cataloguing Tools	Note Storage	Document Storage	between document s and notes	Links from notes to pages of documents/ anchors	Separate notes per page of source document	Document Browsing	Tagging and Search	Link Graph View	Platform support	Windows	Linux	Mac	Androi d	iOS	Web	Sync / Backup	Sync/Backup Method
Legend	0: False | 1:True | 2: Limited Support | 3: Premium Feature													0: False | 1:True | 2: Limited Support | 3: Premium Feature						csv			0: False | 1:True | 2: Limited Support | 3: Premium Feature								csv
Note Taking Applications																															
Obsidian		1	1	1	0	0	0	0	1	1	1	0	0		1	1	2	1		0 basic file tree	1	1		1	1	1	1	1	0	3	proprietary cloud
Notion		1	0	1	0	1	0	1	1	1	1	1	1		1	1	2	1		0 file tree	2	0		1	0	1	1	1	1	3	proprietary cloud
Evernote		1	0	1	0	0	0	0	1	0	0	1	1		1	1	1	1		0 Sorted into noteboo	1	0		1	0	1	1	1	0	3	proprietary cloud
OneNote	raster printout:	2	2	2	2	2	2	2	2	2	2	2	2		1	2	1	1		2 Sorted into noteboo	1	0		1	0	1	1	1	1	3	proprietary, OneDr
SimpleNote		0	0	0	0	0	0	0	0	0	0	0	0		1	0	0	0		0 list	1	0		1	1	1	1	1	1	1	web only
Google Keep		0	0	0	0	0	0	0	1	1	0	0	0		1	0	0	0		0 visual browser	1	0		1	0	1	1	1	1	1	web only
Joplin		1	1	0	0	0	0	1	0	0	0	0	0		1	0	0	0		0 Notes sorted into no	1	0		1	1	1	1	1	0	1	Various cloud solu
Upnote		0	0	1	0	0	0	0	1	0	0	0	0		1	0	1	0		0 Notes sorted into no	1	0		1	1	1	1	1	0		
Bear		1	1	0	0	0	0	1	1	0	0	0	0		1	1	0	0		0 Visual browser	1	0		0	0	1	0	1	0	3	Proprietray cloud
Liquid Text		0	0	1	1	1	0	0	1	0	0	0	1		1	1	1	3		2 visual browser	1	0		1	0	1	0	1	0	3	Proprietray cloud
Good Notes		0	0	1	0	0	0	0	1	0	0	0	0		1	1	1	0		0 visual browser	0	0		1	0	1	1	1	1	1	Proprietary cloud
AmpleNote		0	1	0	0	0	0	0	0	0	0	0	0		1	0	0	0		0 Visual browser	1	0		0	0	0	1	1	1	1	Web only
Word Processors																															
Google Docs		1	2	1	1	1	0	2	1	1	1	1	1		0	1	0	0		0 visual browser, file	0	0		0	0	0	1	1	1	1	web only
Microsoft Word		1	1	1	0	1	0	2	1	0	0	1	1		1	1	1	1		0 Recent Files	0	0									
Libre Office																				0											
Document Readers																															
Adobe Acrobat		2	0	1	2	2	0	0	2	0	0	0	2		0	0	0	0		0 Recent Files	0	0		1	0	1	1	1	1	0	0
Foxit Reader		0	0	1	0	0	0	0	0	0	0	0	0		0	0	0	0		0 Recent Files	0	0		1	0	1	1	1	0	0	0
Drawboard		0	0	1	0	0	0	0	0	1	0	0	0		0	1	1	0		2 List of pdfs	0	0		1	0	1	3	1	1	3	proprietary cloud
MilaNote		0	0	0	0	0	0	0	0	1	1	2	0	2	1	1	1	0		1 Canvas based Brow	2	2		1	0	1	1	1	1	3	Proprietary Cloud
eBook Readers																															
Adobe Digital Editions		0	0	1	0	0	1	0	0	0	0	0	0		0	1	1	0		0 Visual browser	0	0		1	0	1	1	1	0	0	
Calibre		1	1	1	0	1	1	0	0	0	0	0	0		0	1	1	0		0 1	1	0		1	1	1	0	0	2	1	Built in content se
Thorium reader		0	0	1	0	0	1	0	0	0	0	0	0		0	1	1	0		0 Visual browser	0	0		1	1	1	0	0	0		
Kindle app	proprietary onl	0	0	0	0	0	0	0	0	0	0	0	0		1	1	1	1		2 notes tab with list	0	0		1	0	1	1	1	1	1	Proprietary cloud
Neat Epub Reader		1	0	1	0	0	1	0	0	0	0	0	0		1	1	1	0		0 Visual browser	0	0		1	0	1	1	1	1	3	Proprietary cloud
Text Editors																															
Notepad	2 meaning edit	1	2	0	0	0	0	2	0	0	0	2	0		0	0	0	0		0	0	0		1	0	0	0	0	0	0	
Notepad++		1	2	0	0	0	0	2	0	0	0	2	0		0	0	0	0		0	0	0		1	0	0	0	0	0	0	
Sublime Text		1	2	0	0	0	0	2	1	0	0	2	0		0	0	0	0		0	0	0		1	1	1	0	0	0	0	

FIGURE 2.1: Excerpt from the Existing Solutions Excel File, displaying some key findings (including document file-type support, cataloguing options and platform support)

2.2 A Review of Currently Available Alternatives

This section reviews existing note-taking and document reading solutions to identify gaps in the market that NoteReader aims to fill. Key gaps include the lack of seamless integration between document viewing and note-taking, insufficient support for a wide range of document formats, limited cross-platform compatibility, and reliance on costly subscription-based cloud syncing services.

Addressing these issues, NoteReader aims to provide a unified, subscription-free, and cross-platform solution that enables document-linked notes, multi-format support, and intuitive cataloging and search capabilities. The insights provided here will support the rationale for developing NoteReader and highlight the need for an integrated solution.

2.2.1 Existing Note Taking Solutions

The findings below are based on my research into the feature availability of several note taking applications. This information is documented in an excel sheet. See attached file Existing Solutions Spreadsheet: .

A small excerpt of this file can be found in figure B.1.

1. **Document file type support.**
   Very few applications support a wide range of common document types, and those which do do so using destructive methods, like raster printing of the document to a canvas, without storing the original.

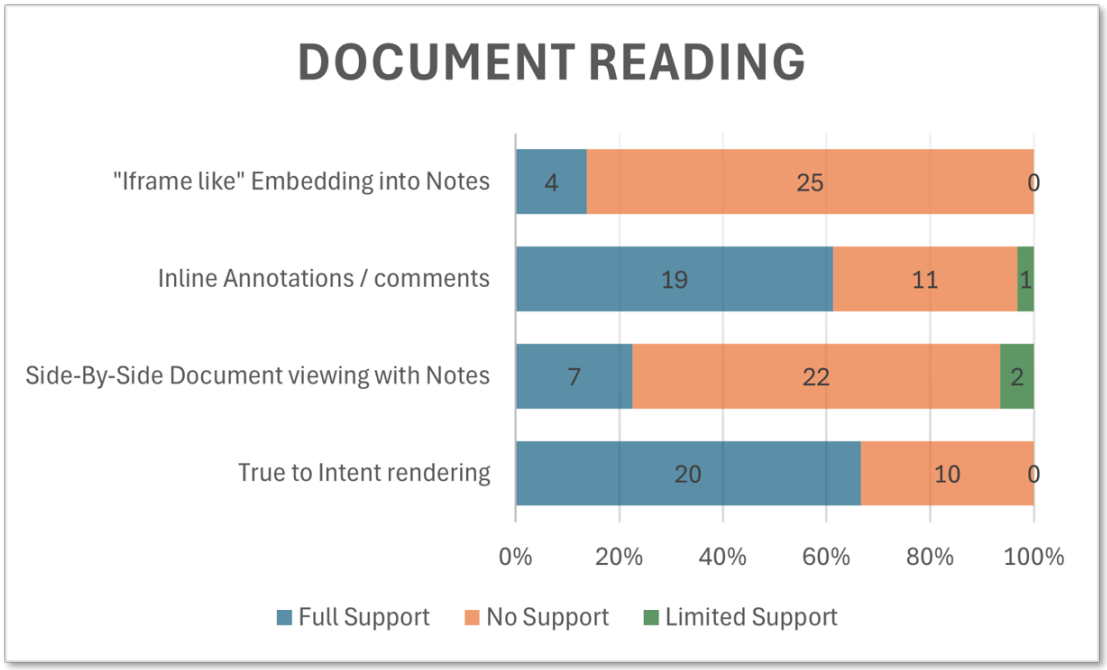


FIGURE 2.2: Graph illustrating the support across reviewed applications for features including "True to Intent Rendering" and "Side by Side Viewing"

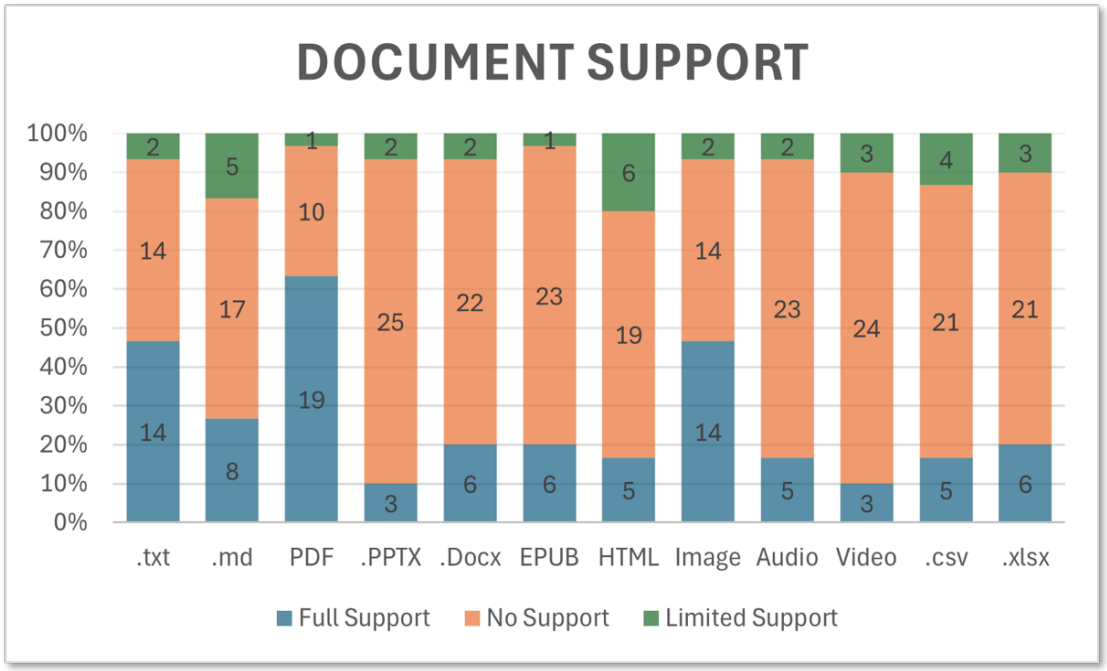


FIGURE 2.3: Graph illustrating the support across reviewed applications for document format

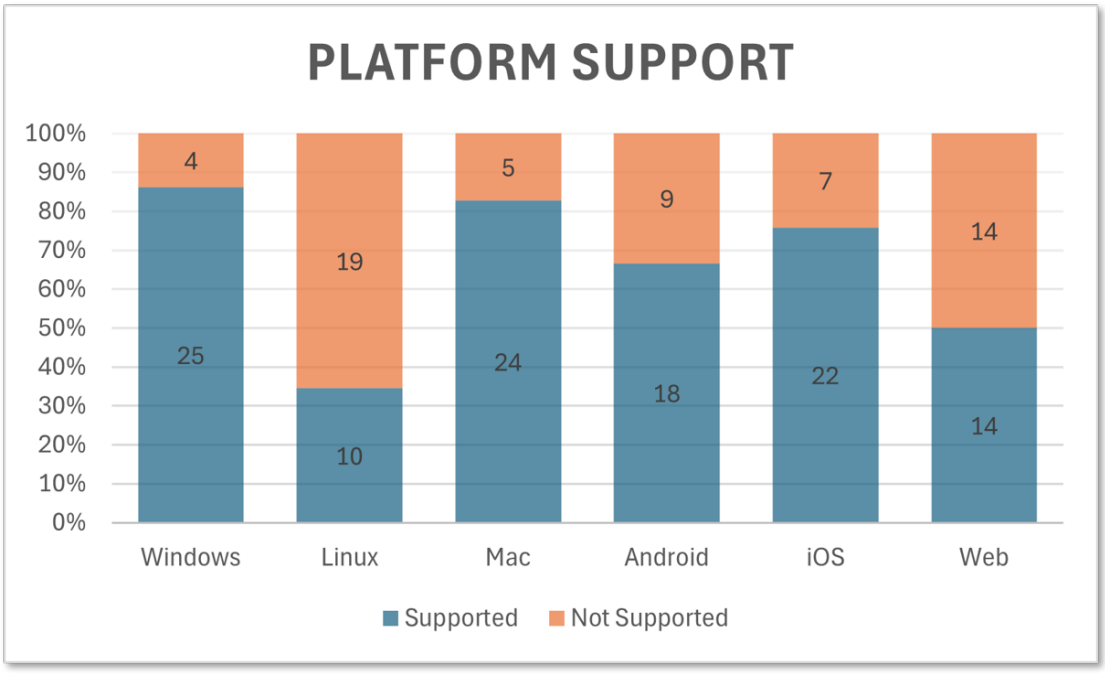


FIGURE 2.4: Graph illustrating the support across review applications for different platforms

2. **File Syncing**

   Most note taking applications require a proprietary cloud service to sync between devices, or back up files.

   Version Control with remote repositories are not commonly used as a method of syncing this data.

3. **Dual functionality**

   Most document readers only allow notes via annotations, while most note taking applications only support document reading as embedded files.

   The ability to have side by side note taking and document reading is uncommon.

4. **Page of Notes per Page of Document**

   No application fully supports having linked notes with a separate page of notes per page of source document. Those which do support a similar function do so by adding all notes and pages to a large canvas for example Microsoft One Note.

5. **Platform support.**

   Most applications tend to support only a couple of platforms. Linux support is especially rare for multiplatform applications.

6. **Cost**

   Most note taking applications require some form of monthly subscription to use to their full potential. The pricing varies widely, as do the features on offer.

2.3 A Review of literature available on the effectiveness of digital note taking

In this section, I will review available literature on studies performed on the effectiveness of digital note taking, when compared to handwritten notes.

This review aims to better understand the benefits of the digital note-taking process to inform the development of NoteReader's user experience. The literature provides insight into the cognitive, procedural, and organizational aspects of digital note-taking, helping shape the design of the tool to meet the needs of learners.

2.3.1 Note-taking and Handouts in The Digital Age[1]

In this article, the author explores the role of Handouts in lectures as a supplemental tool. The author discusses a number of challenges regarding handouts, pointing out that detailed handouts may lead to students not engaging in lectures, and simply memorizing the power point slides. They also suggest that a complete lack of handouts may result in students transcribing the full lecture material, rather than engaging in real time.

A compromise is suggested, where "skeletal" notes will provide a "scaffolding". This method gives a structure to note taking, without providing all information in the handouts themselves. The conclusion the author reaches here is that the student would be required to engage with the material, but not need to fully transcribe, just fill in the gaps.

This philosophy aligns well with the intended purpose and functionality of NoteReader. Source material acting as a scaffolding for personal notes, would be well supported by an application where pages of notes are displayed alongside pages of source material.

By offering a side-by-side view, NoteReader facilitates a structure where students can engage deeply with the material without the cognitive overload of transcription.

2.3.2 Effectiveness of Digital Note-Taking on Students' Performance in Declarative, Procedural and Conditional Knowledge Learning[2]

This article describes a small trial of 72 Students, 40 taking digital notes, and 32 taking conventional handwritten notes. The author refers to three forms of knowledge Declarative, Procedural and Conditional, to differentiate between understanding of factual information, understanding of processes, and understanding of when to apply knowledge.

Each will be measured separately in their study, and based on performance students are divided into percentiles. Notably this study focused on Computer Science students, in programming only.

Excellent students (those in the 70th percentile) showed improvement in all three knowledge types, but especially in declarative knowledge when using digital notes. Students below the 30th Percentile showed significant improvement in Conditional knowledge. Mid level students did not show significant improvement.

No group showed a decrease in performance when using digital notes. While this is a relatively small study, there is a suggestion that the advantages of digital note taking may be at least as effective as traditional methods.

The additional features proposed by NoteReader may be beneficial in further improving the performance difference of students taking digital notes. The split view link between source material and notes will allow students to study with improved context which may improve conditional understanding, when compared to memorization. This structured support mirrors the scaffolding identified as effective in the previous study[1].

2.3.3 Taking notes in the digital age: Evidence from classroom random control trials[3]

This article describes a study to compare the effectiveness of digital note-taking compared to handwritten notes.

Some key findings were that exam scores showed no statistically significant difference between digital and handwritten notes.

There was however evidence that in subjects with high dependence on graphical or mathematical elements digital note taking may be at a disadvantage. Digital methods of creating diagrams, or mathematical equations tend to be more difficult and slower.

While not in scope for the initial round of development of NoteReader, inclusion of handwriting support and OCR may be a valuable addition to compensate for these weaknesses in a future release.

Chapter 3

Problem - NoteReader, an Application Suite for Cataloging Study Material and Notes

3.1 Problem Definition

In the modern era of digital learning, the process of note-taking has undergone a fundamental shift from traditional paper-based methods to digital mediums. Despite the availability of numerous note-taking applications and document readers, there exists a significant gap in the seamless integration of these two functionalities. Most current solutions treat either note-taking or document reading as the primary focus, with the other as an ancillary feature. This approach introduces friction into the study process, as users are forced to toggle between multiple applications to take notes and reference study materials.

The proposed solution, NoteReader, aims to address this problem by creating an all-in-one, platform-agnostic application that treats both document reading and note-taking as first-class priorities. The primary objective is to develop a cohesive system that allows users to read and annotate documents, create linked notes, and access these materials in an organized and intuitive manner.

3.1.1 Current Issues

- Fragmentation of Tools: Current market solutions (e.g., Microsoft OneNote, Adobe Acrobat) often excel in one area while providing minimal support for the other.

Users are forced to switch between dedicated document readers and standalone note-taking applications, leading to a disjointed study experience.

- Limited Multi-Platform Support: Existing applications frequently prioritize Windows, macOS, or mobile platforms, but fail to offer full support across all devices, especially Linux.
- High Cost of Subscription Services: Many applications rely on proprietary cloud storage systems (e.g., Evernote, Notion) to sync notes, necessitating paid subscriptions for cross-device access. This raises the financial barrier for students and educators.
- Inadequate File Type Support: Current solutions support only a limited number of document formats. For example, EPUBs and PowerPoint slides are often not supported natively or require costly add-ons.
- Loss of Context: Existing applications do not associate notes with the specific page or section of the source material they reference, making it difficult for users to revisit contextual notes.
- Manual Organization and Storage Issues: Users are required to manually organize notes and study materials, often leading to misplaced documents and incomplete notes.
- Data Portability: Many note-taking applications lock users into proprietary formats that limit the ability to transfer or back up notes independently. This can be a critical issue if users decide to migrate to another platform.

3.1.2 Proposed Solutions

NoteReader proposes a unified application that simultaneously handles document reading and note-taking, offering a "split-view" interface where documents are displayed alongside a corresponding note pane. This side-by-side approach ensures that users can create notes that are contextually linked to specific pages or sections of the source document.

- Split-View Interface: Each document opened in NoteReader creates a linked folder containing all notes relevant to that document. As users navigate through the document, the corresponding notes automatically load, ensuring users maintain contextual continuity.

- Multi-Platform Compatibility: Built using Kotlin Multiplatform, NoteReader will be available on Windows, macOS, Linux, and mobile devices, providing true cross-platform accessibility.
- Offline and Cloud-Backed Storage: Notes and documents are stored locally in plain text, ensuring data portability. Optional synchronization with GitHub or other version control platforms provides cloud-based backup and version history.
- File Format Versatility: Support for a wide array of document formats, including PDF, DOCX, EPUB, PPTX, TXT, CSV, and more. This eliminates the need for users to convert files to compatible formats.
- Contextual Note-Linking: Notes are linked directly to specific pages of the source document, allowing users to jump directly to the source context with a single click.
- Lightweight, Portable Note Files: All notes are saved as plain text files, making them accessible and editable with other tools like Obsidian or any plain text editor.
- Tagging, Searching, and Categorization: Users can categorize and tag notes for quick access, supporting efficient search and retrieval of information.
- Version Control for Notes: Sync and track changes to notes using Git-based version control, offering a history of note revisions and easy restoration of previous versions.

3.2 Objectives

3.2.1 Scope

- Create a Seamless Note to Document Link: Achieved through file system, Ensure users can link notes to specific document pages, making it easy to reference the original context.
- Achieve Multi-Platform Support: Use Kotlin Multiplatform, or similar frameworks, to provide cross-platform compatibility across Windows, macOS, Linux, and mobile devices
- Facilitate Data Portability: Ensure that users' notes are stored in plain text files and that the application supports exporting notes for use in other tools like Obsidian to prevent vendor lock in.

- Optimize UI/UX for Cognitive Load: Design an intuitive user interface that reduces cognitive load during note-taking, focusing on minimalist design and efficient navigation.
- Offer Customization and Personalization: Allow users to tag, categorize, and search notes, providing tailored organization options for diverse user needs.
- Minimize Cost and Friction: Avoid reliance on proprietary cloud services or subscription fees by using open formats, local storage, and optional Git based version control and sync.

3.2.2 Stretch-Goals

The following is a list of features which I would consider to be important for a final version of this project, but are not necessarily core to the functionality of a Minimum Viable Product or Prototype release.

These will be explored, time allowing, after the core scope is implemented.

- OCR for Images: Implement Optical Character Recognition (OCR) technology to convert scanned images and PDF documents into searchable and editable text. This functionality will enhance accessibility and improve the discoverability of text-based content in image files.
- Handwriting Support: Enable handwriting support for touchscreen devices and styluses, allowing users to write notes directly on the document. This approach will appeal to users who prefer handwritten notes and facilitate better physio-cognitive engagement with the study material.
- Dictation: Introduce voice-to-text dictation features, allowing users to transcribe spoken words directly into notes. This will benefit users who prefer to dictate notes during lectures or meetings, enabling fast, hands-free note creation.
- Video and Audio, Lecture Recording with Timestamps: Develop functionality for recording live lectures or video content. Users can take time-stamped notes that are linked to specific moments in the video / sound clip, allowing them to quickly revisit relevant sections during playback.

3.2.3 Out of Scope

The following are a list of potentially valuable additions, which fall outside the core purpose of the application. These may be useful features to provide in future updates, but will not be the focus of this project.

- AI Integration: The initial version of NoteReader will not include AI-driven features such as natural language processing (NLP), automated content summarization, or predictive note suggestions.
- Web Interface: The NoteReader application will be a desktop and mobile application only, with no support for web browsers or cloud-based interfaces.
- Real-Time Collaboration: Multi-user editing or real-time collaborative note-taking is outside the scope of the initial implementation.
- Cloud-Only Storage: While cloud sync via GitHub is supported, the system will not offer a proprietary, always-online cloud storage solution.

3.2.4 Stakeholders

- Students: Students across various academic disciplines are the primary beneficiaries, as NoteReader facilitates efficient study workflows and better information retention.
- Researchers and Academics: Researchers who must organize large collections of study materials and notes for literature reviews and projects will benefit from the system's tagging, linking, and search capabilities.
- Educators and Lecturers: Educators can create lecture notes and materials linked to specific document pages, offering students an interactive and guided learning experience.
- Journalists: Journalists who need to organize interview notes, source materials, and articles can use NoteReader's organizational capabilities.

3.3 Functional Requirements

- The system shall provide a split-view interface to allow users to view documents and take notes simultaneously.

- The system shall support the import and display of PDF, EPUB, DOCX, and PPTX files
- The system shall allow users to link notes to specific pages or sections of the source document.
- The system shall support searching and filtering of notes and documents using tags, keywords, and categories.
- The system shall support exporting notes in plain text format for use in other applications.
- The system shall enable users to sync notes to cloud storage providers like GitHub using version control.
- The system shall allow users to tag and categorize notes for efficient organization and retrieval.

3.4 Non-Functional Requirements

- Performance: The system shall load and display documents without noticeable wait times on standard modern devices.
- Usability: The user interface shall be intuitive, with clear user experience flows.
- Scalability: The system shall support large document files (over 25mb), and offer alternative synchronization methods for these files.
- Portability: The system shall support cross-platform functionality on Windows, macOS, Linux, and mobile devices.
- Reliability: The system shall gracefully handle unexpected crashes to prevent data loss by saving notes in real-time and automatically restoring the last working state upon restart.
- Maintainability: The system shall be modular to allow for easy updates and addition of new features.
- Compatibility: The system shall be compatible with widely used file formats (PDF, EPUB, DOCX, PPTX, TXT, etc.).

Chapter 4

Implementation Approach

4.1 User Flow

In order to create a well-designed, purposeful user interface (UI) for the NoteReader application, it is essential to understand and model how users interact with the system. To achieve this, I have created a series of user flow diagrams that represent key workflows within the application. These diagrams illustrate the sequential steps users take to complete specific tasks, ensuring that the interface is intuitive and aligns with user expectations.

The following user flows have been designed:

- **Document Import, Reading, and Note Taking Flow** (see Figure B.4): This flow captures the process a user follows to import a new document into the application, read the document, and take notes alongside it. It begins with the user selecting a document from their file system, proceeds through the rendering of the document in the split-view interface, and describes how notes are created, saved and reloaded.
- **Opening Already Imported Documents from Catalogue Flow** (see Figure B.5): This flow models the steps for accessing previously imported documents via the application's catalogue or library. Users can search for, sort, and select documents from their collection. Once selected, the application loads the document alongside the corresponding notes for the user to resume their work seamlessly.
- **Git-Based File Synchronization Flow** (see Figure B.6): This flow explains how the application handles file synchronization using a Git-based version control system. It includes steps for committing local changes, pulling updates and

pushing notes and metadata to a remote repository to ensure consistency across multiple devices. Conflict resolution and large file size (LFS) are also considered in this flow to provide a smooth syncing experience.

4.1.1 Document Import, Reading, and Note Taking Flow

This user flow begins when the user selects a document for import. The application processes the document, creating a folder in the user's file system to store notes linked to the document. The document is displayed in the split-view interface, with the left pane dedicated to rendering the document and the right pane for note-taking. The user can navigate through the document, take notes, and save them automatically. Notes are stored in lightweight plain-text files for portability.

4.1.2 Opening Already Imported Documents from Catalogue Flow

In this workflow, the user starts by accessing the application's catalogue or library. They can search for a document using keywords, tags, or filters. Once the desired document is located, the application loads it into the split-view interface, along with previously saved notes. This ensures that the user can seamlessly continue their work without losing context.

4.1.3 Git-Based File Synchronization Flow

The synchronization flow focuses on maintaining consistency across multiple devices. Users can commit their local changes (e.g., notes, updates to the catalogue) and push them to a remote Git repository. If working on another device, they can pull updates from the remote repository to retrieve the latest notes and document metadata. The system handles merge conflicts by alerting users and providing options for manual or automatic conflict resolution.

These user flows ensure that NoteReader is user-centric, intuitive, and aligned with the expectations of students, researchers, and other users. Detailed diagrams for each flow are provided in Figures **??**, B.5, and **??**, respectively.

4.2 Architecture

4.2.1 Technologies

- Kotlin Multiplatform: Chosen for its ability to support multiple platforms, including Windows, macOS, Linux, and mobile devices. Below is a brief review of a number of other multiplatform frameworks that were considered.
  - **Maui:**
    A multiplatform framework which is part of the .NET ecosystem. This framework supports multiple operating systems, but notably lacks Linux support.
  - **React Native:**
    React Native Platform Support
    Supports all common platforms, though Linux support is community driven
  - **Electron:**
    Chromium based, supports all desktop platforms, but does not support mobile
  - **Flutter:**
    Supported Platforms
    Broad support in Java Script based framework by google
- Document Frameworks: These frameworks are potential solutions for handling document parsing and rendering. The actual solution may change over the course of development.
  - Markdown/HTML Support: Used to format notes and support plain-text storage, ensuring compatibility with other note-taking applications like Obsidian.
  - **.pdf:**
    The PDF format is well defined, under ISO standard 320000-1.
    PDF.js would be one option, for displaying this document type, through a platform independent web view.
  - LibreOffice: Used to convert PowerPoint and Word files to PDF for consistent rendering.
  - **Images (.png, .jpeg, .webp, etc)**
    These can easily be displayed either directly in app, or using a web view.
  - SheetJS: Used to render CSV and XLSX files.
  - Used for rendering eBooks in EPUB format.

Regardless of the specific framework used, the architecture of the solution remains the same. An interface will be defined for document rendering, with each document type being defined as an implementation of that interface.

This ensures the functionality is standardized, and provides a clear pathway for adding additional document format support in future releases.

- Integrations with existing Systems
  - Git, with GitHub/GitLab: Integration with version control for note synchronization.
  - OCR (Optical Character Recognition): Tesseract is proposed for text recognition in image-based PDFs.
  - Handwriting Recognition: Windows Ink support for annotating notes on touch-enabled devices.

4.3 Risk Assessment

TABLE 4.1: Initial risk matrix

Frequency/ Consequence	**1-Rare**	**2-Remote**	**3-Occasional**	**4-Probable**	**5-Frequent**
4-Fatal					
3-Critical					
2-Major					
1-Minor					

TABLE 4.2: Risk Table for Identified Risks

Risk	**Frequency**	**Consequence**
Scope Creep	Occasional	Major
File Corruption	Remote	Fatal
Merge Conflicts	Frequent	Minor
Large File Sizes	Rare	Major

4.3.1 Risk Identification

- Scope Creep: The addition of features like OCR or audio-visual notes early on could divert development focus.
- File Corruption: Data loss could occur due to file corruption during version control sync.

- Merge Conflicts: Users may encounter conflicts when syncing files via Git.
- Large File Sizes: Some file types (e.g., PowerPoint) may be too large for GitHub's file size limitations.
- Change in GitHub terms: A change in the terms of service for remote version control repositories may impact the functionality of the application.

4.3.2 Risk Mitigation Strategies

- Agile Methodology: Use Scrum with two-week sprints to maintain a clear development timeline.
- Data Backup: Ensure automatic backups for critical files during sync.
- File Locking: Implement a file-locking mechanism during note editing to prevent conflicts.
- File Size Limitations: Break large files into smaller components if necessary, or consider cloud storage as a fallback option.
- Support multiple git remote services.

4.4 Methodology

4.5 Implementation Plan Schedule

Agile Scrum: Six sprints of approximately two weeks each will be used to track development progress. These Sprints will start alongside semester 2.

Technology Familiarization: Time between now and the beginning of Sprint 1 will be allocated to learning Kotlin Multiplatform and Git API integration.

- Sprint 1: PDF and Text Editor with Linked Pages.
- Sprint 2: Document and Note Cataloguing and Search.
- Sprint 3: Improved Document Format Support.
- Sprint 4: Version Control and Sync.
- Sprint 5: UI/UX Enhancements.

- Sprint 6: Prototyping and Integration of Additional Features.

The schedule allows for iterative feedback and testing, ensuring that each module is stable before moving forward.

4.6 Evaluation

4.6.1 Functional Testing

- Document Rendering: Ensure support for PDF, DOCX, EPUB, and PPTX files.
- Note Sync: Test sync functionality across GitHub and other remote services.
- Tagging and Search: Verify that tagging and search features are intuitive and fast.

4.6.2 Non-Functional Testing

- Performance: Measure document load times and ensure responsiveness for large files.
- Portability: Ensure the system works across Windows, macOS, Linux, and mobile platforms.
- Reliability: Test for system stability during crashes and validate automatic recovery.

4.7 Prototype

Each sprint will deliver an executable iteration of the project.

A prototype will be developed by the end of Sprint 4. This prototype will feature all core functionality:

- Document Viewing: View and navigate PDF, DOCX, EPUB, and PPTX files.
- Note Editor: Create and store notes linked to specific pages.
- Version Control: Sync notes using GitHub.
- Split-View: User interface supporting simultaneous document view and note-taking.

The prototype will be a minimally viable product (MVP) that focuses on essential functionality, with stretch goals like OCR and handwriting support introduced in later sprints.

For visual reference, several mock ups and diagrams have been produced (See Appendix B.).

- A UML Class diagram detailing the high level architecture is displayed in figure B.1.
- A sample UI for the PC version is represented in figure B.2.
- A sample UI for the Mobile version of the application is represented in figure B.3

Chapter 5

Implementation

5.1 Difficulties Encountered

Developing NoteReader involved navigating a range of technical and architectural challenges. These difficulties can be grouped into three categories: Easy, Medium, and Hard, depending on the complexity of the problem and the extent to which a successful resolution was achieved.

5.1.1 Hard: Framework Immaturity and Document Rendering

One of the most significant challenges was the relative immaturity of Kotlin Multiplatform. The first long-term support stable version was only released in November of 2023. While the framework offers exciting features and remains a solid choice for the requirements of NoteReader, it does introduce several new technical challenges.

5.1.1.1 Nature of Difficulty

1. As the framework is so new, Kotlin Multi-platform has a smaller community, with a smaller based of established resources when compared to more mature alternatives like Flutter or React.
2. I found that official documentation and forum discussions were insufficient to resolve issues, necessitating trial-and-error experimentation.
3. The ecosystem surrounding Kotlin Multiplatform, particularly the third-party libraries available on platforms like Klibs.io (Jetbrains' official repository for community libraries and plug-ins), is still developing. Many libraries were either incomplete, poorly documented, or lacked support for key features required by the

project. This was especially problematic for modules related to document parsing and rendering.

5.1.1.2 Effect on the original project design:

1. The original intended architecture was to implement fully independent rendering module for each document type. Each module would implement a standard interface allowing each rendering path to be slotted in to the same pipeline.

2. True-to-intent rendering of documents is a core principle of the project, so it was important to make sure the document looks like it should. Images should appear where the author placed them, font sizes and other formatting should also match.

3. I had originally intended to use drop in solutions where possible to achieve this, but I found that many of the community built options available on Klibs.io we not sufficient. Most community options were either experimental or lacked critical functionality, such as accurate page rendering or support for embedded multimedia.

5.1.1.3 Approach to solve difficulty

1. I experimented with several different approaches to solve this difficulty. Starting by adapting Java or Android-specific libraries into the project.

2. One solution I attempted was parsing the documents directly and rendering them as html and css in a web view. I abandoned this approach due to scalability concerns, as it would require bespoke implementations for each document type. With the focus being on wide ranging document format support, this would massively increase the development cost of each format.

3. One community library which worked very well was the compose-pdf library created by developer zt64 and offered under an MIT license. This library provided an excellent starting point as a PDF renderer.

4. Ultimately I settled on the approach of converting all supported document type to a temporary PDF file at run-time. The original file remains unchanged and is preserved for archival purposes. The generated PDF is then passed to the compose-pdf renderer for display.

5. Converting various document types to pdf is a very common workflow, so there was a greater level of support available online to help in developing this solution. That said, there remained very few options available for Kotlin Multiplatform Specifically.

6. As the programming language Kotlin is built on Java, it is cross compatible. I was able to use several Apache libraries, including POI to parse Microsoft documents such as PowerPoint and word, along with PDFBox to generate the PDFs.

7. Certain file types were not as easily converted with existing embeddable libraries however. I chose to integrate two free and open source binary programs into the workflow. LibreOffice, is an office suite which includes open source alternatives to popular products like Microsoft Office, Excel, and Power Point. Calibre is an open source e-reader.

   Both LibreOffice and Calibre offer a portable version, each giving access to a wealth of command line tools in pre-compiled binaries, which can be integrated into the Kotlin Multiplatform application.

   Calibre was used to convert e-books formats (such as epub), While LibreOffice was used to convert legacy Microsoft formats.

With the above solution, all of the document types planned for the prototype were implemented. Pdf, Docx, Doc, Pptx, Epub and Html.

This architecture allowed for faithful rendering of diverse document types across all target platforms while maintaining extensibility for future format support.

5.2 Actual Solution Approach

This section presents a reflective comparison between the original solution design, as proposed in Chapter 4, and the final implementation delivered at the end of the development phase. From January to May, the project evolved through iterative development, guided by the original architecture, use cases, and methodology.

However, as with most software projects, practical challenges, such as framework limitations, time constraints, and unexpected technical hurdles, necessitated adjustments to the initial plan. What follows is a structured, component-by-component analysis that documents these changes, explains the underlying causes, and evaluates the decisions made to ensure a functioning and maintainable end product. Each subsection outlines the intended design, contrasts it with what was ultimately delivered, and provides a rationale for any divergences observed.

5.2.1 Architecture Comparison

5.2.1.1 Original Design

The original architecture proposed for NoteReader was structured around a modular, cross-platform framework using Kotlin Multiplatform.

The core logic was intended to be platform-agnostic, with separate implementations for UI and rendering components for each target platform (Windows, macOS, Linux, and Android). The application followed a Model-View-Controller (MVC) paradigm to promote separation of concerns and ease of maintenance.

Several methods were proposed as potential options for rendering documents, often revolving around embedded web views for formats such as PDF and EPUB, and markdown notes were to be stored in plain-text files with metadata managed via a local index or catalogue. Synchronization with GitHub or GitLab was planned through integrated version control utilities.

5.2.1.2 Final Implementation

The final implementation closely adheres to the architectural plan outlined in the original design, particularly in its modular structure and use of Kotlin Multiplatform.

The core logic and data handling remain abstracted from the UI layer, and Compose Multiplatform has been successfully used to build out the desktop front end.

The application includes a dual-pane layout with PDF rendering in one pane and markdown-based note editing in the other. Notes are saved in lightweight plain-text files, and the application state is persisted using a JSON-based structure.

However, only the desktop version has been implemented thus far. Platform-specific renderers for mobile and Linux systems were not developed in this phase due to time constraints and the complexity of supporting diverse file formats across environments. Synchronization functionality via Git was also not implemented.

5.2.1.3 Key Changes and Rationale

The most notable deviation from the original architecture was the decision to focus exclusively on the desktop platform. This decision was driven primarily by limited development time. The project is however designed with expansion in mind, structured

in such a way to make it simple to add additional platforms without changing the core program.

Additionally, the reliance on web views for document rendering proved less mature than expected requiring fallback to slower but more reliable conversion pipelines in some cases.

Another key architectural adjustment was the temporary omission of the Git synchronization module. Although the file structure and catalogue system were built to support version-controlled syncing, integrating Git APIs introduced significant complexity that could not be addressed within the available timeframe.

In summary, while the core architecture remained faithful to the original design, practical limitations led to a reduction in scope and simplification of some components. The system remains extensible and modular, ensuring that postponed features, such as cross-platform UI layers and Git integration, can be added in future development cycles without requiring fundamental refactoring.

5.2.2 Risk Assessment Revisited

5.2.2.1 Original Risk Matrix

TABLE 5.1: Initial risk matrix

Frequency/ Consequence	1-Rare	2-Remote	3-Occasional	4-Probable	5-Frequent
4-Fatal					
3-Critical					
2-Major					
1-Minor					

TABLE 5.2: Risk Table for Identified Risks

Risk	Frequency	Consequence
Scope Creep	Occasional	Major
File Corruption	Remote	Fatal
Merge Conflicts	Frequent	Minor
Large File Sizes	Rare	Major

5.2.2.2 New Risks Identified

- **Framework Learning Curve:** Although Kotlin Multiplatform was selected for its cross-platform promise, working with an emerging framework like Compose

TABLE 5.3: Risk Table for Newly Identified Risks

Risk	Frequency	Consequence
Framework Learning Curve	Occasional	Major
Third-Party Library Limitations	Occasional	Critical
Time Constraints and Parallel Commitments	Frequent	Major

Multiplatform introduced a steeper learning curve than anticipated. Documentation gaps, platform inconsistencies, and immature tooling led to delays, particularly in rendering components and file handling workflows.

- **Third-Party Library Limitations:** Availability of plugins and libraries compatible with kotlin multiplatform was limited due to the immaturity of the platform. Early dependencies, such as those used in the web view approach were found to have constraints not initially evident during the design phase. Issues with third party libraries and the added development cost of bespoke implementations resulted in a need to redesign parts of the file processing logic.
- **Time Constraints and Parallel Academic Commitments:** As this project was conducted alongside academic coursework, available development hours were sometimes inconsistent, especially around assignment and exam periods. This impacted velocity and required periodic rescoping of deliverables to match remaining capacity.

5.2.2.3 Mitigation Measures Adopted

- **Prioritization of Core Features:** To address the effects of scope creep, development efforts were re-centered around the primary user flow: importing, reading, and annotating documents. Secondary features such as Git integration, tagging, and advanced search were deprioritized and marked for future iterations. This ensured that the minimum viable product (MVP) remained deliverable within the timeline.
- **Fallback Conversion Methods:** To mitigate issues with EPUB and DOCX rendering, fallback strategies such as conversion to PDF or plain-text extraction were introduced. While not as elegant as native rendering, this ensured compatibility and allowed users to continue engaging with their content.
- **Interface Abstraction for Platform Readiness:** Although cross-platform support was not fully achieved during this phase, the codebase was structured using interface-driven design. Platform-specific implementations are abstracted, allowing new targets (e.g., Android or macOS) to be added later without refactoring the core application logic.

- **Use of Git for Incremental Backup and Rollback:** Even though user-facing Git integration was not finalized, Git was used throughout development to manage version control of the codebase. This provided resilience against code loss, enabled rollback of unstable changes, and served as a lightweight progress tracking mechanism.
- **User Interface Simplification:** To improve usability despite time and platform constraints, the UI was deliberately simplified. Large buttons, color-coded panes, and minimal navigation steps reduced the learning curve and made the application usable with minimal training or documentation.

5.2.3 Methodology Execution

5.2.3.1 Planned Development Methodology

The project was originally planned using the Agile Scrum methodology, with an emphasis on iterative development and incremental feature delivery. The development timeline was broken down into six two-week sprints, each with a specific focus:

- Sprint 1: Document rendering (PDF) and linked note-taking editor.
- Sprint 2: Catalogue implementation and document/note retrieval.
- Sprint 3: Expanded file format support (EPUB, DOCX, PPTX).
- Sprint 4: Synchronization system design (Git integration).
- Sprint 5: UI/UX refinement and accessibility testing.
- Sprint 6: Prototype integration, testing, and documentation.

5.2.3.2 Adaptations Made During Implementation

In general Sprints took longer to complete than initially planned.

From an early stage it was clear that the scope of the project needed to be reigned in to focus on only core features. Sprint 2 was changed to Focus on File System and Program State, while Sprint 3 was changed to Expanding file format support, which was the area where most difficulties were encountered. As a result this sprint took much longer than originally hoped.

Git integration and the Catalogue system were pushed to future development cycles.

Overall after all adjustments there were 5 Sprints across the semester:

- **Sprint 1:** Document rendering (PDF) and linked note-taking editor.
- **Sprint 2:** Focus on File System and Program State
- **Sprint 3:** Expanded file format support (EPUB, DOCX, DOC, PPTX, HTML).
- **Sprint 4:** UI/UX refinement.
- **Sprint 5:** Prototype integration and documentation.

5.2.4 Implementation Timeline

5.2.4.1 Original Sprint Schedule

Originally planned as 6 two week sprints.

5.2.4.2 Actual Timeline

- **Sprint 1:** Ending Feb 24th
- **Sprint 2:** Ending Mar 4th
- **Sprint 3:** Ending Apr 8th
- **Sprint 4:** Ending May 5th
- **Sprint 5:** Ending May 20th (on submission of source code)

5.2.4.3 Deviation Analysis and Justification

Difficulties were encountered in rendering documents other than PDF, due to immaturity of the platform and lack of available libraries and plug-ins.

From an early point in development, it was decided to focus on wide ranging File support. Implementation approach needed to change to render these file formats in a manner which respected the authors intent. Exploring and experimenting with available options such as web view and pdf conversion extended the duration of sprints.

5.2.5 Prototyping Outcomes

5.2.5.1 Initial Prototypes

Several wire frame prototypes were produced in advance of beginning development. (See appendix:B)

An interactive prototype was produced in WPF using C#, to act as a visual aid (see figure: B.2)

A visual prototype of a mobile interface was produced in JustInMind (see figure B.3(d)

5.2.5.2 Final Product Interface and Features

- Two Pane layout, featuring Document Reader and Note Editor, with zoom buttons and buttons to navigate between pages (fig:5.1)
- Note editor can be swapped to Markdown rendering Layout (fig: 5.2
- Home Screen, Import Documents, Open the folder where files are contained, open imported documents (fig: 5.3)
- Dark Mode, shows potential for theming to user's preference (fig: 5.4
- Center Column in Reader View provides buttons which allow you to enter a focus mode for either the Reader or Note Taker, making that pane take up the full width of the window. Also contains a button which allows you to swap the two panes.

5.2.5.3 Evolution of the Design and Lessons Learned

The design of NoteReader evolved significantly over the course of development, shaped by technical constraints, time limitations, and practical insights gained through implementation. The foundation of the project however, split-view note taking, document integration, and plain-text note portability remained constant.

One of the most prominent evolutions was the architectural strategy for handling document rendering. The original goal was to have have native support for each format. This approach was determined to be impractical for a prototype. Instead, a fallback architecture was introduced: all supported document types were converted to PDF at runtime and rendered using a unified PDF renderer. This preserves the project's core requirement of true-to-intent document rendering, while simplifying implementation.

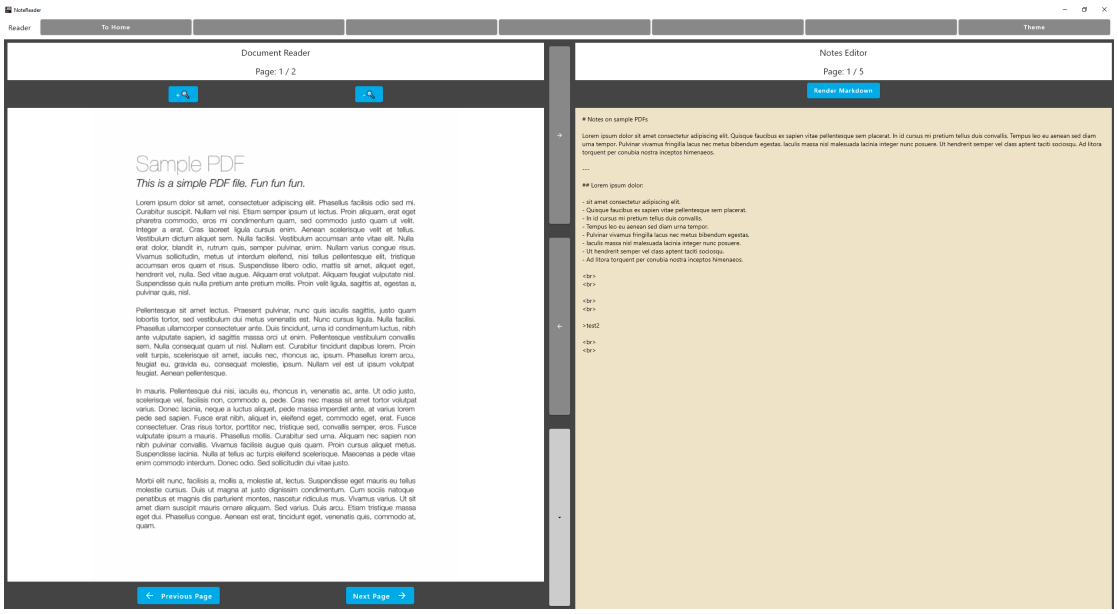


FIGURE 5.1: Reader Page

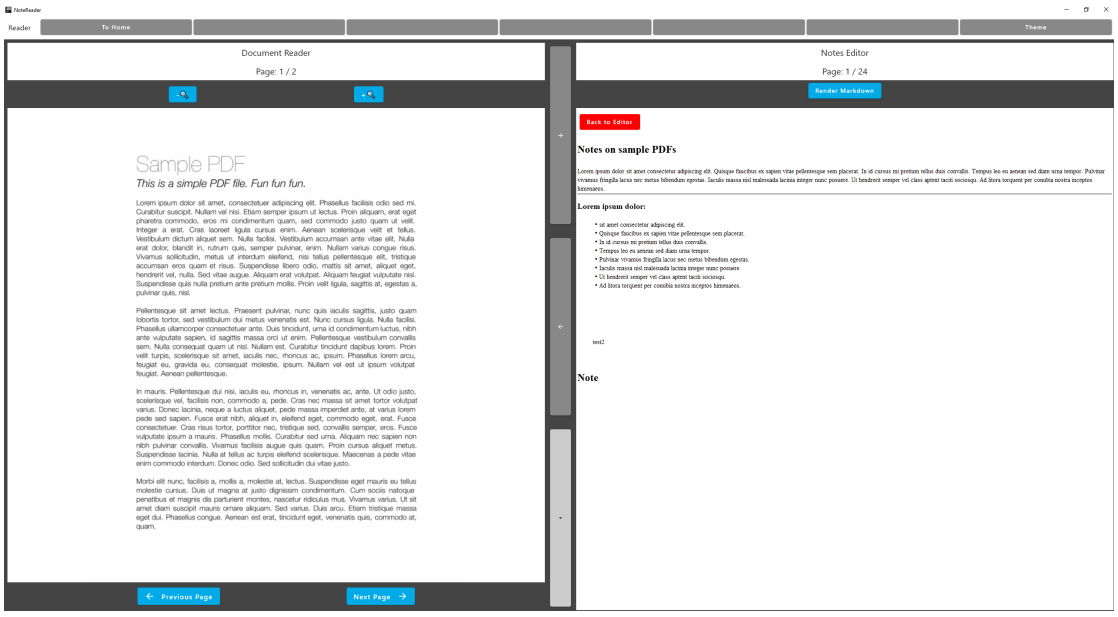


FIGURE 5.2: Markdown View

The scope of the project also shifted to emphasize core functionality. While the initial vision included synchronization, tagging and searching, and full multiplatform support, these features were pushed back to a later development cycle in order to focus on delivering a robust and usable Minimum Viable Product. Emphasis was placed on ensuring the design was fully modular, with string architecture to ensure these features could be integrated in future iterations.

One of the most important lessons learned was the importance of adaptability in software projects. Various difficulties including framework maturity, integration complexity, and

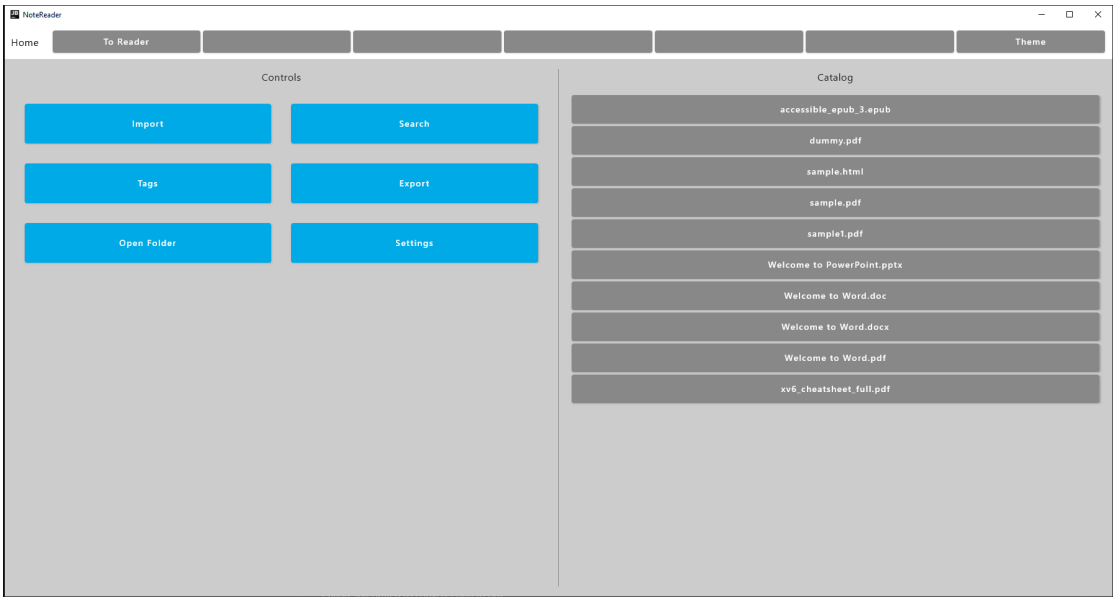


FIGURE 5.3: Home Screen

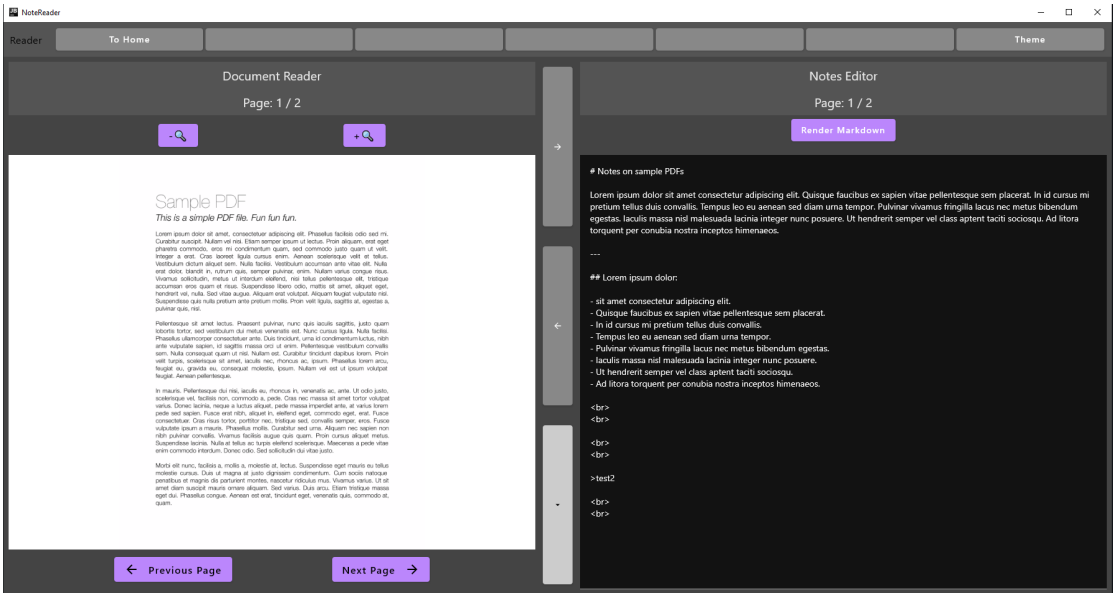


FIGURE 5.4: Dark Mode

time management all influenced the project's direction. Attempting to strictly follow the original implementation plan would have made it difficult to deliver a completed product. The deferral of some requirements allowed the primary vision of the application to be maintained in the first iteration of the product, while preparing to implement those requirements in future.

Chapter 6

Testing and Evaluation

To validate the effectiveness of the NoteReader application, a comprehensive testing process was conducted. A variety of sample files in different formats were used to assess the system's ability to meet its functional and non-functional requirements. These files included PDFs, DOCX, PPTX, EPUB, and HTML documents, and enabled rigorous evaluation of file parsing, rendering fidelity, and performance 6.3.

6.1 Requirements

6.1.1 Functional Requirements

6.1.1.1 The system shall provide a split-view interface to allow users to view documents and take notes simultaneously

Status: Fulfilled
The application successfully implements a split-view interface, where the document is displayed in the left pane and a markdown-compatible note editor appears in the right pane. 6.1.

6.1.1.2 The system shall support the import and display of PDF, EPUB, DOCX, and PPTX files

Status: Fulfilled
The application supports a wide range of file types including PDF, DOCX, DOC, PPTX, EPUB, and HTML. These formats were selected based on their prevalence in academic environments and have been verified to load and render successfully and true to intent.

6.1.1.3 The system shall allow users to link notes to specific pages or sections of the source document.

Status: Fulfilled
The system correctly associates notes with specific pages of the source document. Each note is saved in a file named to reflect its corresponding page number, enabling automatic retrieval and display when users navigate to a particular page.

6.1.1.4 The system shall support searching and filtering of notes and documents using tags, keywords, and categories.

Status: Not Fulfilled
This functionality was not implemented due to time constraints and has been deferred to future development cycles.

6.1.1.5 The system shall support exporting notes in plain text format for use in other applications.

Status: Fulfilled
All notes are stored in markdown format (.md), enabling seamless export to and reuse in third-party applications such as Obsidian.

6.1.1.6 The system shall enable users to sync notes to cloud storage providers like GitHub using version control.

Status: Not Fulfilled
Version-controlled synchronization using platforms like GitHub was intended but not implemented due to limited development time.

6.1.1.7 The system shall allow users to tag and categorize notes for efficient organization and retrieval.

Status: Not Fulfilled
Although planned, tagging and categorization features were not developed in this phase of the project.

6.1.2 Non-Functional Requirements

6.1.2.1 Performance: The system shall load and display documents without noticeable wait times on standard modern devices.

Status: Partially Fulfilled This Requirement has been partially fulfilled. PDF files are very quick to load, but the requirement to convert other document types can result in noticeable wait times. EPUB is especially long with the sample document taking approximately 5 seconds to load an ebook with 115 pages.

Once the document is converted however, it is very quick to navigate between pages. Pagination and lazy loading was implemented to ensure the impact on memory resources is kept to a minimum and ensure good performance.

6.1.2.2 Usability: The user interface shall be intuitive, with clear user experience flows.

Status: Fulfilled

This requirement is slightly more subjective, though I did design the application with this in mind, and would consider it to have been fulfilled.

I implemented the program in a manner which requires as few clicks as possible to enter a file. importing a document is as simple as selecting "import", then selecting the name of the document.

While in the reader view, The left pane shows the document, while the right shows the note taking window. The note taking window is coloured with a yellow tint, to evoke the thought of a traditional sticky note, so the purpose of the pane is immediately clear.

Buttons are large, and have icons that indicate their purpose, such as a magnifying glass for zooming.

6.1.2.3 Scalability: The system shall support large document files (over 25mb), and offer alternative synchronization methods for these files.

Status: Deferred Although large files can be imported, advanced synchronization strategies for large datasets (e.g., LFS for Git) are not yet implemented and remain a future goal.

6.1.2.4 Portability: The system shall support cross-platform functionality on Windows, macOS, Linux, and mobile devices.

Status: Not Yet Fulfilled While the application has not yet been deployed on multiple platforms, it is architected using Kotlin Multiplatform, enabling straightforward extension to additional targets in future iterations.

6.1.2.5 Reliability: The system shall gracefully handle unexpected crashes to prevent data loss by saving notes in real-time and automatically restoring the last working state upon restart.

Status: Fulfilled All notes are saved in real time as the user edits them. A persistent application state is maintained using a JSON file, which is read on application launch to restore the last session 6.2

6.1.2.6 Maintainability: The system shall be modular to allow for easy updates and addition of new features.

Status: Fulfilled

The application employs a Model-View-Controller (MVC) architecture, supporting clean separation of concerns.

Views are built in Compose, allowing individual components to be replaced or extended with minimal coupling.

New document format support can be easily added by inserting appropriate logic modules into the existing flow.

6.2 Metrics

- **Feature Completeness**: Whether each intended feature was implemented and functioned as expected during manual use.
- **File Compatibility**: Evaluation of whether a variety of common document formats (PDF, DOCX, EPUB, PPTX, HTML) could be opened, rendered, and navigated.
- **Stability and Recovery**: Observations of the application's behaviour when reopened after closing mid-session, including state persistence, and real time saving of notes.

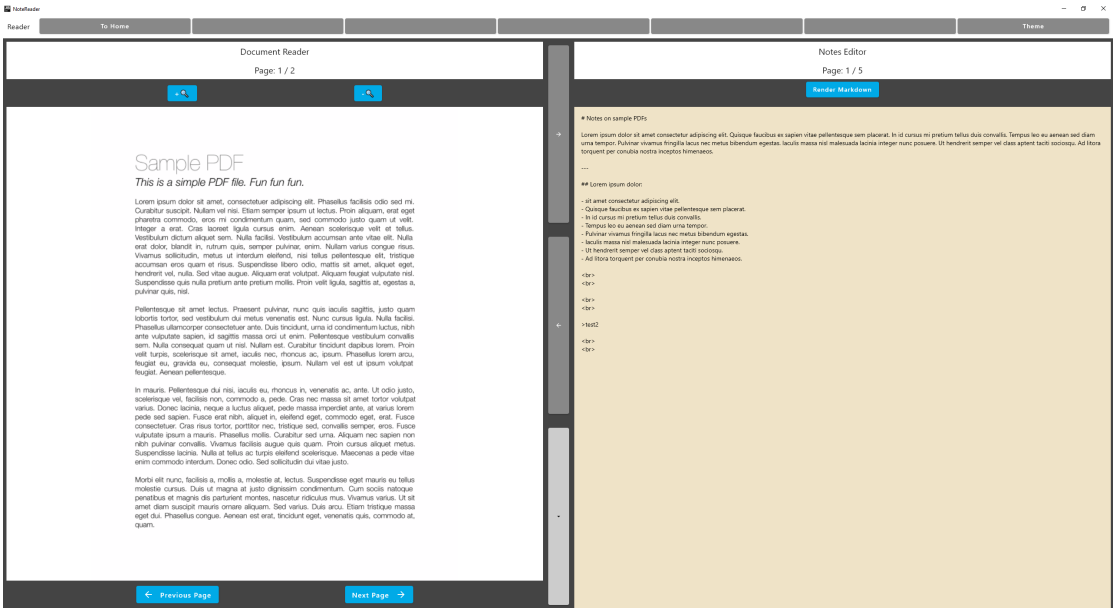


FIGURE 6.1: A screenshot of the NoteReader Application, showing the two pane layout


```
{
  "currentLayout": "READER",
  "currentDocument": {
    "title": "sample1.pdf",
    "path": "C:\\Users\\emmet\\AppData\\Roaming\\NoteReader\\sample1.pdf"
  },
  "currentDocumentFormat": "PDF",
  "currentPage": 0,
  "currentPageCount": 2,
  "currentIdIndex": 79,
  "isDarkTheme": false
}
```


FIGURE 6.2: Sample Program State in Json Format

accessible_epub_3.epub	10/08/2024 14:09	EPUB File	3,957 KB
dummy.pdf	11/02/2025 10:38	Adobe Acrobat D...	13 KB
sample.html	25/09/2024 18:56	LibreWolf Handler	1 KB
sample.pdf	11/02/2025 10:38	Adobe Acrobat D...	19 KB
sample1.pdf	11/02/2025 10:56	Adobe Acrobat D...	71 KB
Welcome to PowerPoint.pptx	25/09/2024 18:47	Microsoft PowerP...	3,740 KB
Welcome to Word.doc	25/09/2024 18:51	Microsoft Word 9...	829 KB
Welcome to Word.docx	25/09/2024 18:51	Microsoft Word D...	700 KB
Welcome to Word.pdf	25/09/2024 18:51	Adobe Acrobat D...	269 KB

FIGURE 6.3: Sample Files used for testing

- **Document-to-Note Linking Accuracy**: Inspection of whether notes correctly reloaded for their associated page, and validating that files are created and sorted appropriately.

These observational metrics provided a practical basis for assessing whether NoteReader met its intended goals and provided a usable experience for real-world document annotation workflows.

6.3 System Testing

System testing was conducted manually, using a range of sample files selected to represent typical academic study materials. The goal was to validate the system's ability to import, render, and annotate documents across a variety of formats. The following test process was followed:

- **Document Import and Rendering**: Sample files (PDF, EPUB, DOCX, PPTX, HTML) were imported to check if they rendered correctly in the split-view layout. Each file was inspected for formatting integrity and legibility.
- **Note Creation and Retrieval**: Notes were created for specific pages, then navigation was tested to confirm that the correct note appeared when returning to that page.
- **Session Persistence**: The program was closed mid-session and reopened to test whether the last viewed document and notes were restored successfully. Crashes were simulated by ending execution of the program directly in the Android Studio IDE.
- **File System Inspection**: The folder structure and file output were manually verified to confirm that notes were stored in plain-text markdown format and logically organized per document.

While these tests did not produce numeric results, they provided confidence that core functionality worked as intended under normal usage conditions.

6.4 Results

The manual testing process confirmed that the core features of NoteReader function as expected across a range of sample documents. Below is a summary of observed results:

- **File Compatibility**: All tested formats PDF, DOCX, PPTX, EPUB, and HTML were successfully rendered using the PDF conversion pipeline. Formatting was preserved in all cases.
- **Note Linking**: Notes were correctly saved per page and reloaded without error.
- **State Restoration**: On restarting the application, the previous session's document and page state were restored, confirming that persistence was functioning as designed. when simulating crashes, all notes were verified to have saved correctly.
- **Interface Stability**: In the final version no crashes were encountered during normal use.

6.4.1 Limitations

- A lack of automated testing may lead to blind spots over time. Manual testing was appropriate for the graphical elements of the application, but subsequent development cycles should focus on unit testing as a priority.
- Testing was conducted only on two devices (Windows 10 and Windows 11).
- No formal load or performance testing was conducted. While considerations were made to performance, such as the implementation of lazy loading and pagination, observations of responsiveness were subjective.

Chapter 7

Discussion and Conclusions

7.1 Solution Review

The final implementation of NoteReader addresses the core problem identified at the beginning of the project: the lack of an integrated solution that treats both document reading and note-taking as first class features.

The completed NoteReader prototype delivers a focused and practical solution to the challenge of fragmented note-taking and document reading workflows.

The application successfully provides a split-view interface that allows users to read documents and take notes simultaneously, preserving contextual relevance and reducing cognitive load.

The system support all targeted document types, with an architecture designed to expand support to further formats. It supports document page to note linking and session persistence. While features like search, tagging, and Git synchronization were not completed, the application architecture supports their future inclusion.

While certain requirements were deferred to later development cycles, the project acts as a solid proof of concept, and illustrates the value of the core features.

7.2 Project Review

The project was approached using an agile methodology, structured around iterative development in five adjusted sprints. Although the original plan envisioned six evenly distributed sprints, difficulties encountered, particularly in areas involving document rendering, required adjustments to the original strategy.

Key challenges included the immaturity of the Kotlin Multiplatform framework, limitations in third-party libraries, and the time-intensive nature of converting diverse document types while preserving author intent.

If this project were to be undertaken again, earlier identification of technological limitations would have enabled a more realistic scope. A stronger focus on UI prototyping and automated testing could also have improved development efficiency and user confidence.

Throughout the project, a number of valuable skills were developed:

- **Cross-platform development**: Learned how to structure shared logic in Kotlin Multiplatform, and abstract platform-specific implementations.
- **Interface design and user experience**: Applied UX design principles to build an intuitive and minimalist interface.
- **Document processing and conversion**: Gained hands-on experience with libraries such as Apache POI, PDFBox, and external tools for document conversion.
- **State persistence and modular architecture**: Designed and implemented a maintainable file structure and session recovery mechanism.

These skills not only contributed to the successful delivery of NoteReader's MVP but will be directly transferable to future software development projects that require cross-platform deployment, structured UI design, or file format interoperability.

7.3 Conclusion

This project set out to explore whether a unified application could bridge the gap between document reading and note-taking in a way that supports better organization, learning, and review. The investigation into current market tools revealed a recurring theme: fragmentation, platform lock-in, and poor contextual linking between notes and source materials.

NoteReader demonstrated a practical solution that merges document viewing and structured note-taking into a single interface. Core features, including split view layout, page specific note linking, plain text file storage, and persistent session state were successfully implemented and validated through manual testing.

The primary conclusion is that the design approach enabled the delivery of a functional MVP with potential for further growth.

Secondary, adopting cutting-edge frameworks (e.g., Kotlin Multiplatform) can accelerate innovation, but must be balanced against the increased risk of instability and limited ecosystem support.

NoteReader stands as a proof of concept that contextual, cross linked note taking is both feasible and desirable, and can serve to save time, prevent data loss and reduce cognitive load.

7.4 Future Work

While the project achieved a core Minimum Viable Product, there are several areas identified for further development and enhancement:

- **Search and Tagging System**: Implementation of a tag-based and full-text search system for fast filtering of notes and documents remains a high-priority feature.
- **Git Integration**: Enable Git-based version control and synchronization to allow for cross-device usage and version history tracking of notes.
- **Cross-Platform Expansion**: Extend support to macOS, Linux, and mobile platforms using Kotlin Multiplatform's shared logic model and Compose Multiplatform's UI capabilities.
- **OCR and Handwriting Support**: Add optical character recognition for image-based documents and enable stylus input for users who prefer handwritten notes.
- **User Customization**: Provide theme support, customizable keyboard shortcuts, and user settings for layout and accessibility.
- **Searchable Catalogue UI**: Build a graphical catalogue browser for easier navigation and visualization of stored documents and notes.
- **Improved EPUB Rendering**: Enhance rendering fidelity for EPUB files using more performant native libraries or better conversion processes.
- **Audio/Video Notes with Timestamps**: Allow users to take notes linked to specific timestamps during lecture recordings or media playback.

If more time and resources were available, these improvements would bring NoteReader closer to a fully featured application suitable for release and daily academic use. The current prototype serves as a solid foundation upon which these features can be gradually layered in a modular and maintainable way.

Bibliography

[1] E. M. Stacy and J. Cain, "Review note-taking and handouts in the digital age."

[2] D. Sun and Y. Li, "Effectiveness of digital note-taking on students' performance in declarative, procedural and conditional knowledge learning," *International Journal of Emerging Technologies in Learning*, vol. 14, pp. 108–119, 2019.

[3] B. Artz, M. Johnson, D. Robson, and S. Taengnoi, "Taking notes in the digital age: Evidence from classroom random control trials," *Journal of Economic Education*, vol. 51, pp. 103–115, 4 2020.

Appendix A

Code Snippets

A.1 Early Prototyping

Following are a number of small code snippets with basic implementation of some core features. This code is written in C# using WPF, and is not intended for use in the final program. These are simply early explorations into the concepts.

```
public void SaveCurrentMarkdown()
{
    var content = MarkdownEditor.Text;
    if (!string.IsNullOrWhiteSpace(content))
    {
        var filePath = Path.Combine(notesFolderPath, $"{CurrentPageIndex}.md");
        markdownHandler.SaveMarkdown(filePath, content);
        UpdateSaveStatus("Saved", false);
    }
}

```

FIGURE A.1: An example of creating a notes markdown file, which uses the current page number as the name of the file, so it can be found when the page is re-opened.

```
using VersOne.Epub;

public class EpubLoader
{
    public List<FlowDocument> LoadEpub(string epubPath)
    {
        var epubBook = EpubReader.ReadBook(epubPath);
        var contentFiles = epubBook.ReadingOrder.OfType<EpubLocalTextContentFile>().Select(chapter => chapter.Content);
        return ConvertToFlowDocuments(contentFiles, epubBook.Content.Images.Local);
    }
    private List<FlowDocument> ConvertToFlowDocuments(IEnumerable<string> contentFiles,
        IReadOnlyCollection<EpubLocalByteContentFile> images)
    {
        if (contentFiles == null || images == null)
            throw new ArgumentNullException("contentFiles or images cannot be null");

        var flowDocuments = new List<FlowDocument>();

        foreach (var content in contentFiles)

......... Specific Implementation omitted for brevity .........

            var htmlPanel = new HtmlPanel();
            htmlPanel.Text = htmlDoc.DocumentNode.OuterHtml;
            document.Blocks.Add(new BlockUIContainer(htmlPanel));

            flowDocuments.Add(document);
        }

        return flowDocuments;
    }

```

FIGURE A.2: An example approach of displaying an EPUB file in a HTML renderer

Appendix B

Wireframe Models and Diagrams

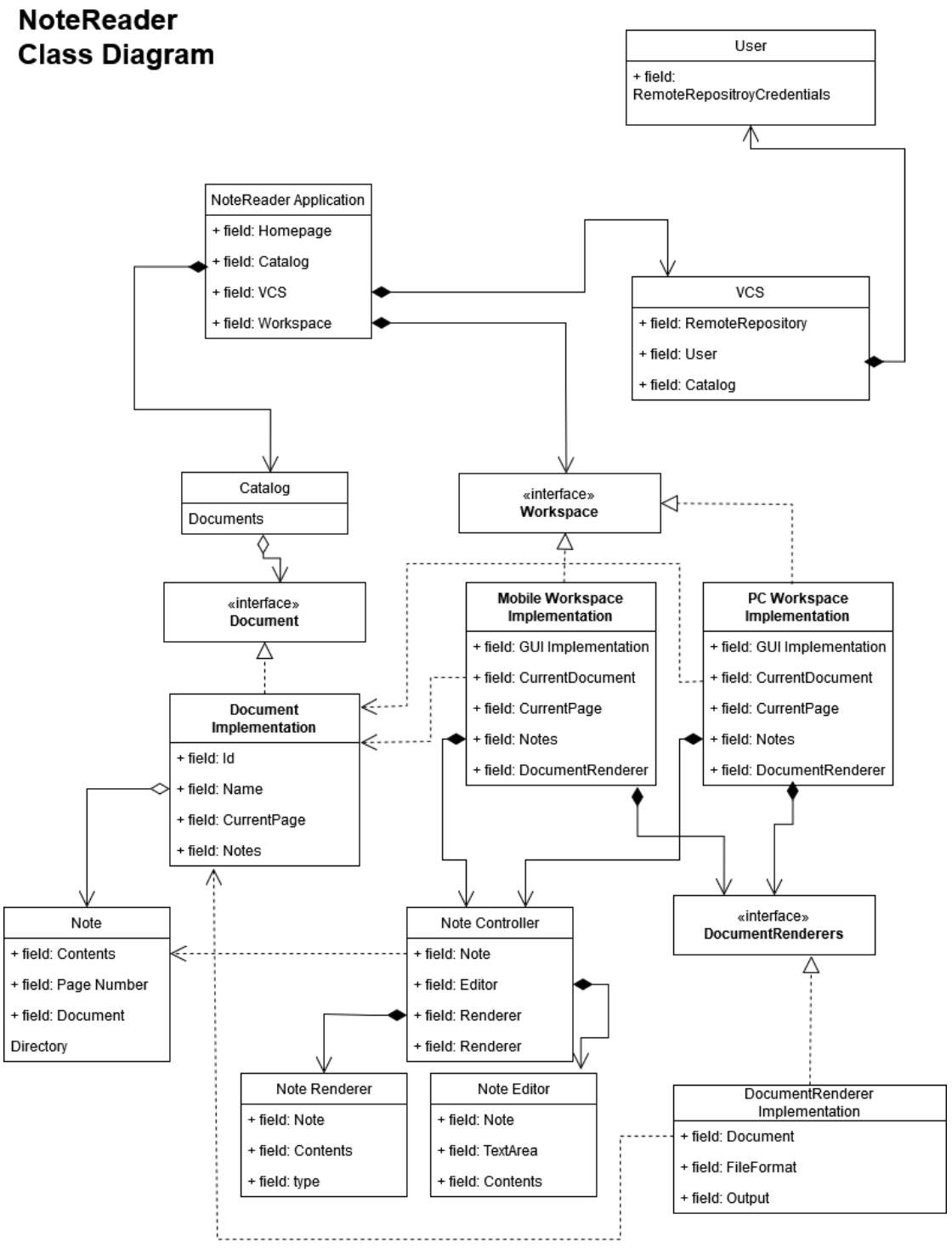
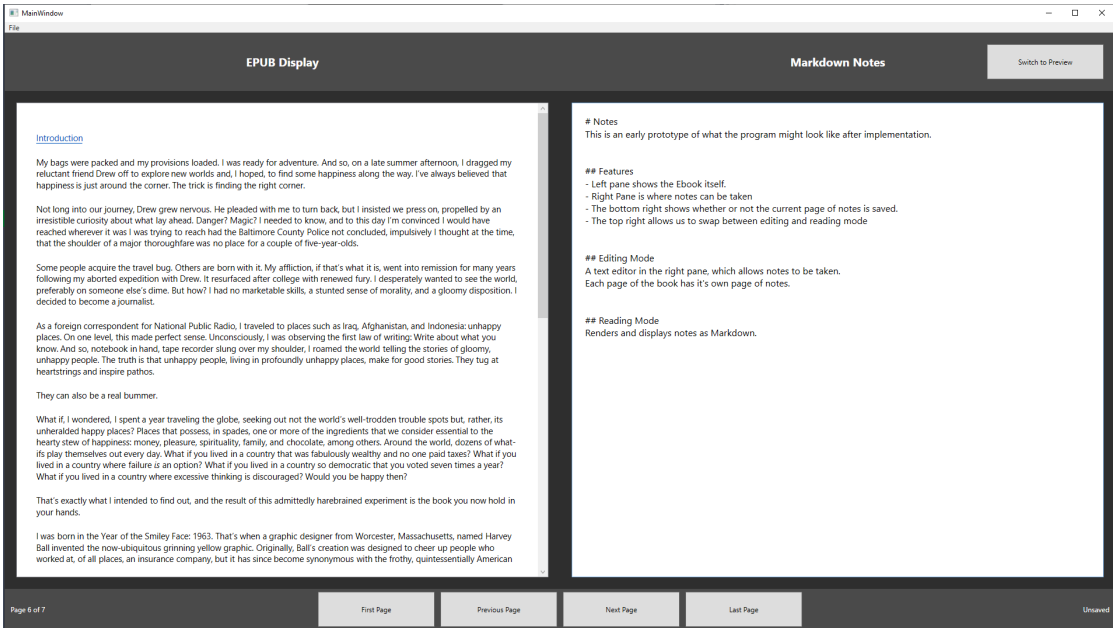
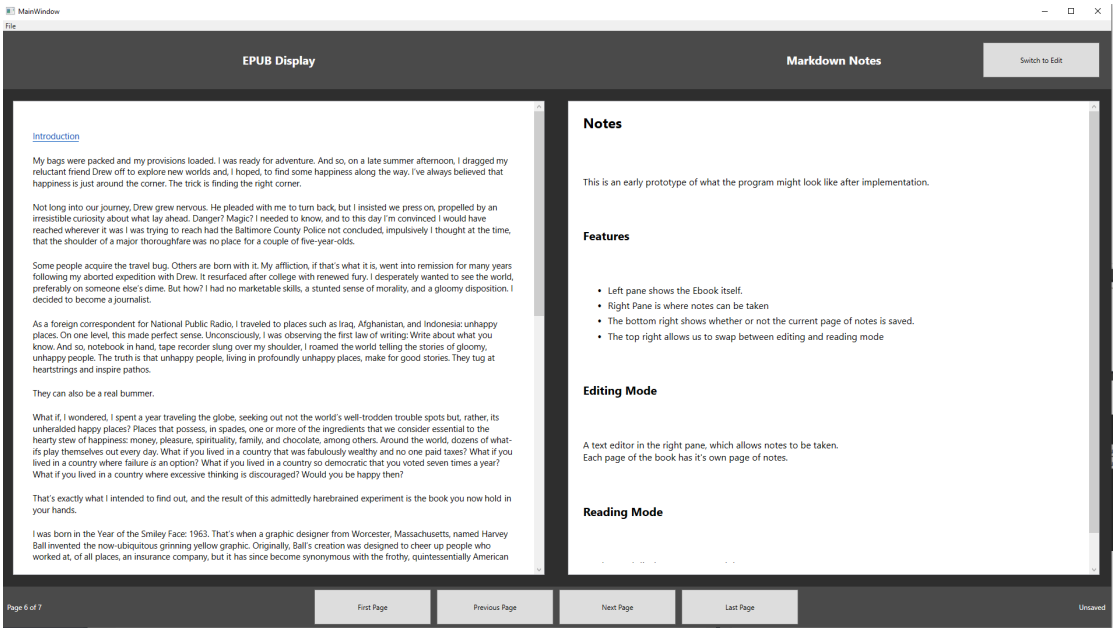


FIGURE B.1: Application Class Diagram, Illustrating high level overview of Architecture

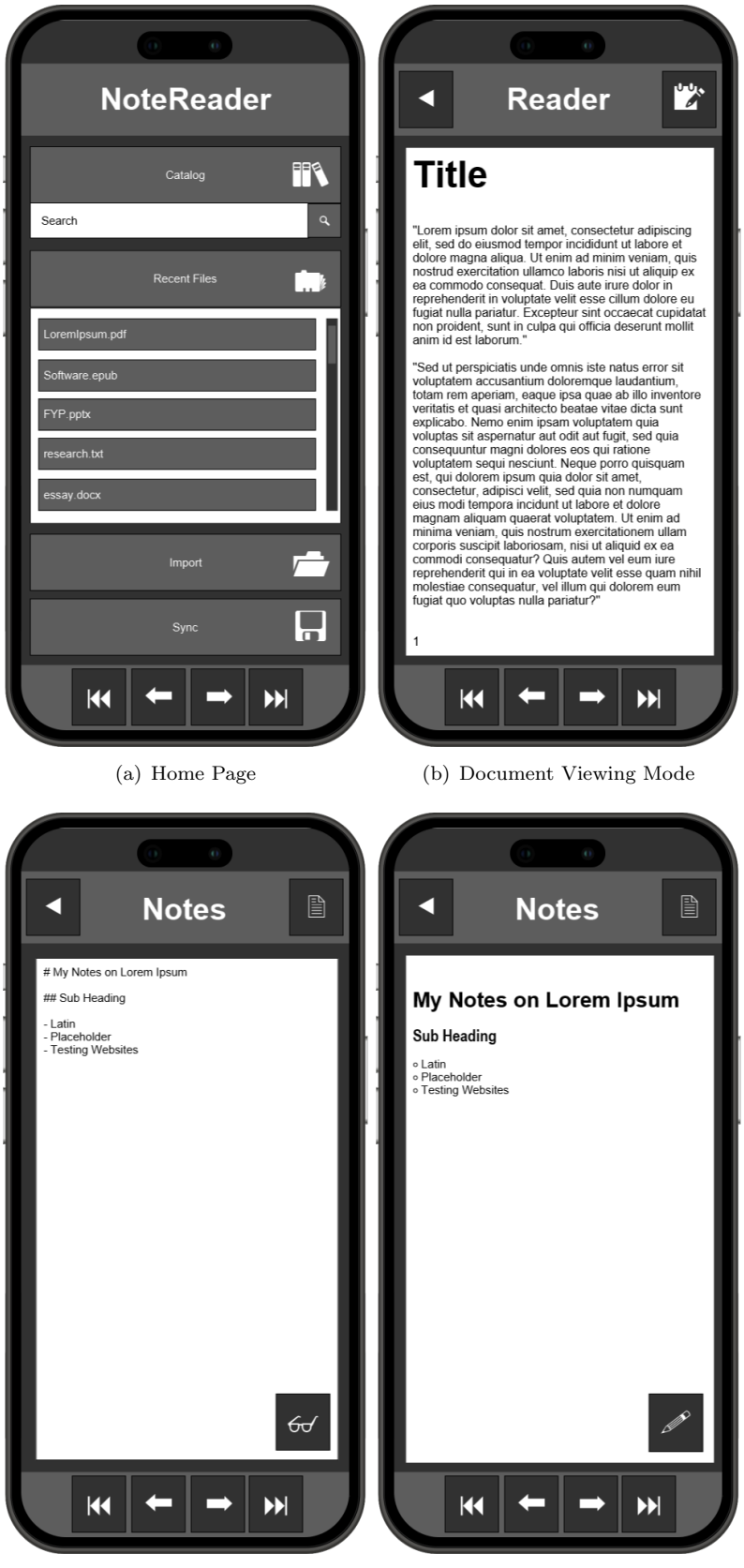


(a) NoteReader in Notes Editing Mode



(b) NoteReader in Notes Rendering Mode

FIGURE B.2: Mock implementation of the NoteReader application, showing EPUB rendering on the left, and Note editor on the right



(a) Home Page (b) Document Viewing Mode

(c) Notes Editing Mode (d) Notes Rendering Mode

FIGURE B.3: Mock implementation of the NoteReader mobile application, showing the three primary modes

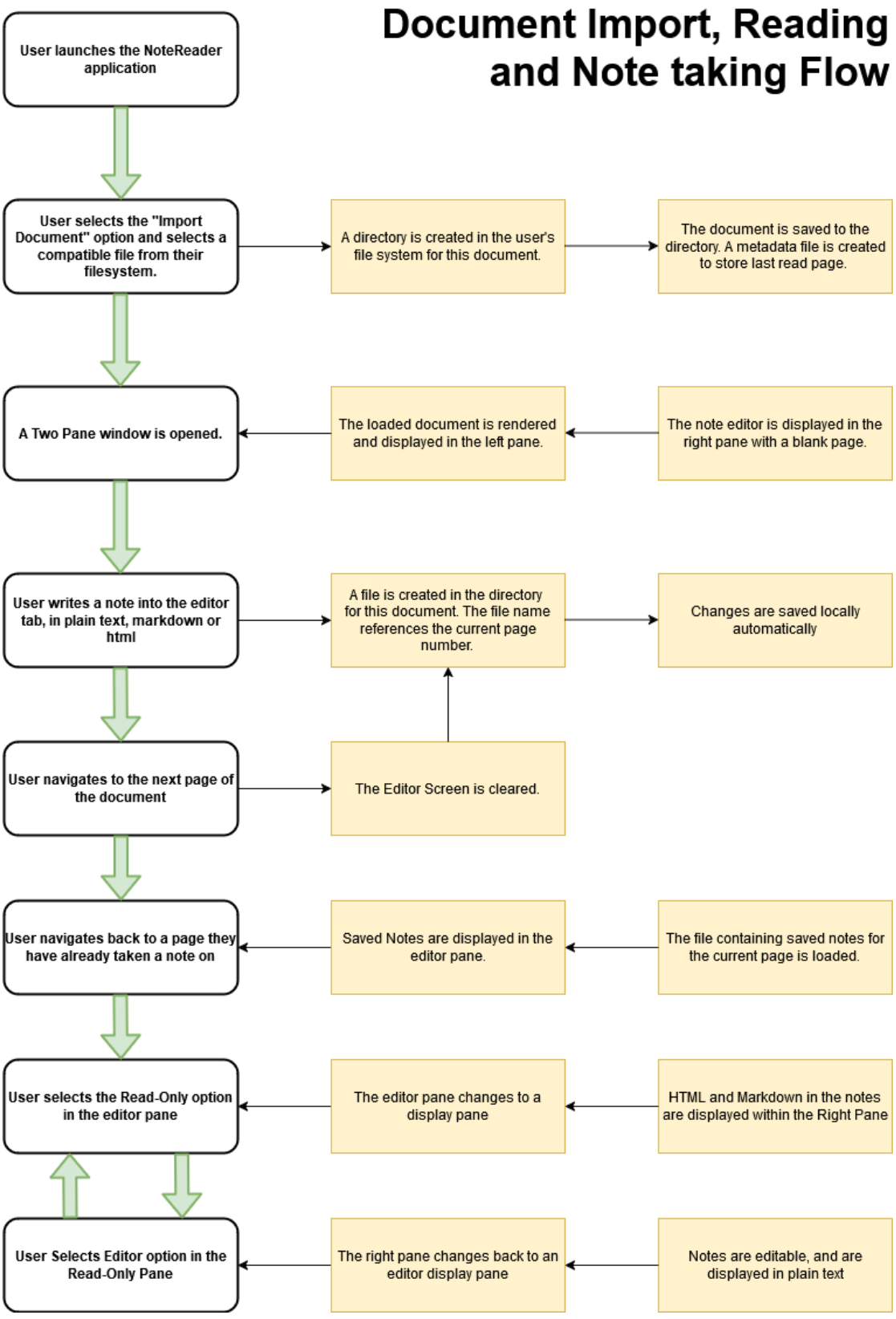


FIGURE B.4: Depicting user flow from opening program, to importing documents, taking notes, and navigating between pages.

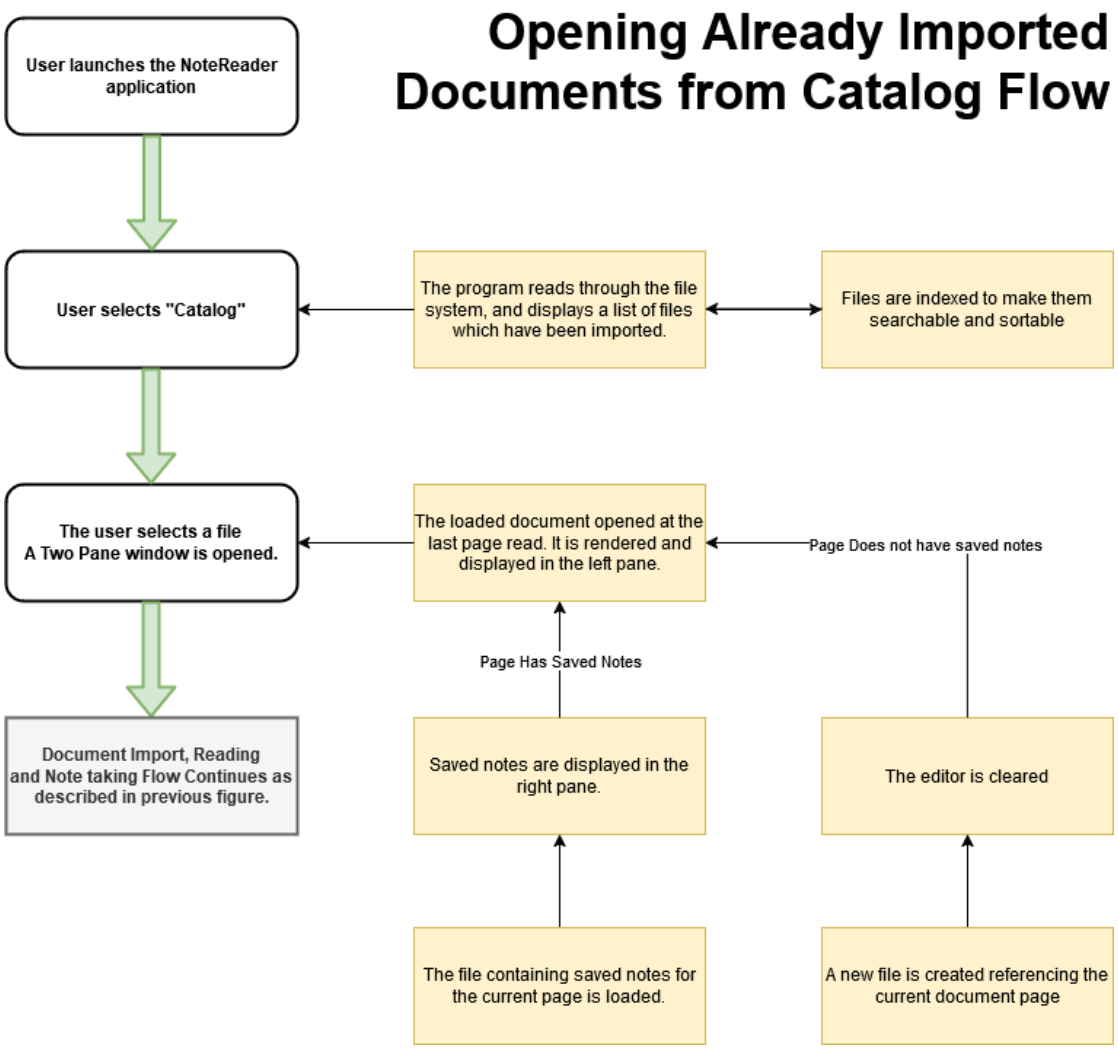


FIGURE B.5: Depicting user flow of re-opening Documents from catalogue.

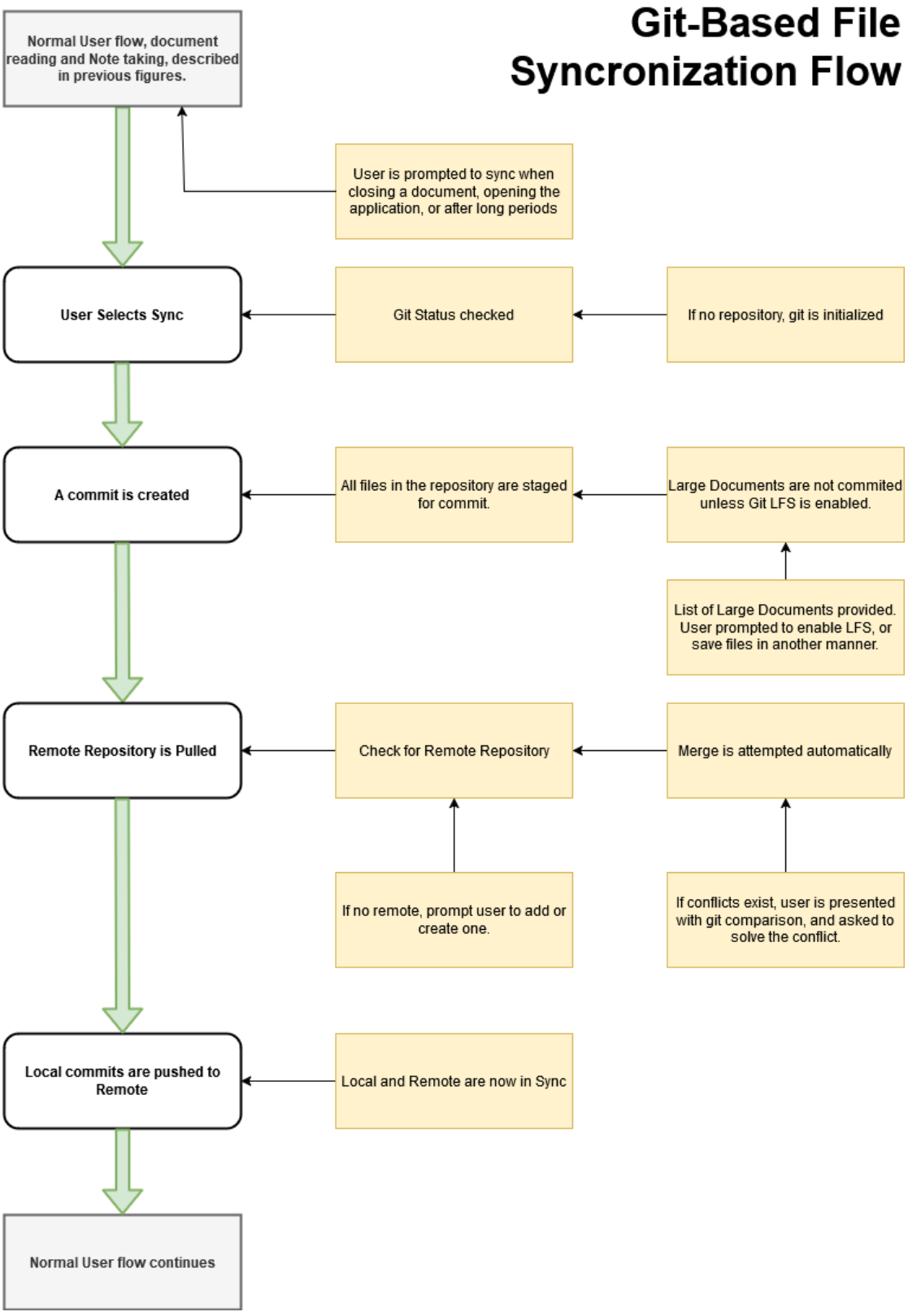


FIGURE B.6: Depicting user flow of syncing files with git.

Appendix C

Miscellaneous

NoteReader (Anóta)

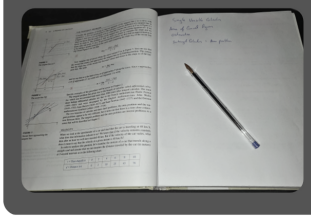
An Application Suite for Cataloging Study Material and Notes

Emmett Fitzharris, BSc Honours in Software Development
Department of Computer Science, MTU Cork, April 2025
Supervisors: Kapal Dev (semester 1), Deirdre Dunlea (semester 2)

1: Introduction

Picture a book, which has a blank sheet of paper slotted between each page.
As you read, you take notes. Flip to the next page, a new sheet awaits you.
When it's time to revise, you go back to that page and your notes on the topic waiting for you.

This is the feeling Note Reader wants to achieve.



2: Problem Statement

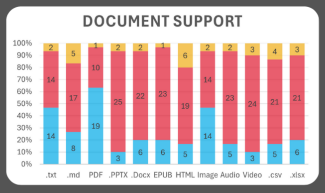
- Fragmentation of Tools
- Limited Platform Support
- High Cost of Subscription Services
- Inadequate File Type Support
- Manual Organization and Storage
- Loss of Context, misplaced documents, wasted time.

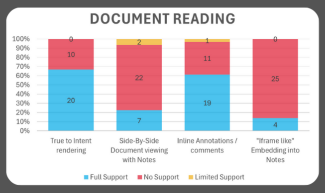
3: Research Methodology

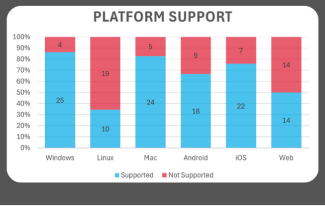
32 programs were reviewed, to determine current levels of support offered by existing solutions. The programs were categorized and checked against a list of features.

Research papers were reviewed to verify the cognitive effectiveness of digital note taking to inform the design

4: Research Results







5: Proposed Solution

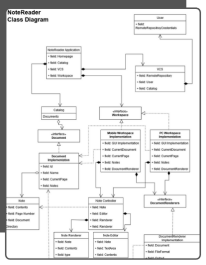
The primary objective is to develop a cohesive, multiplatform system that allows users to read and annotate documents, create linked notes, and access these materials in an organized and intuitive manner, without destructively modifying the source material.

- Side-by-Side document reading and note taking
- Support a wide Range of commonly used document types
- Multi-Platform Design approach
- Git-based file Synchronization

6: Implementation Approach

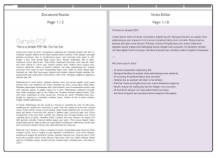
Using Kotlin Multiplatform, develop a core application to ensure consistency across platforms.

Platform specific GUI and Document rendering abstracted by interfaces.



7: Conclusions

- There is a clear gap in the features offered by currently available applications.
- Seamless integration of note taking and document reading is a valuable feature.
- Skeletal handouts approach will be well supported by this application.
- Cognitive benefits, encoding notes rather than transcribing all content.



References:

- *E. M. Stacy and J. Cain, "Review note-taking and handouts in the digital age. 2015*
- *D. Sun and Y. Li, "Effectiveness of digital note-taking on students' performance in declarative, procedural and conditional knowledge learning," International Journal of Emerging Technologies in Learning, vol. 14, pp. 108–119, 2019*
- *B. Artz, M. Johnson, D. Robson, and S. Taengnoi, "Taking notes in the digital age: Evidence from classroom random control trials," Journal of Economic Education, vol. 51, pp. 103–115, 4 2020.*

FIGURE C.1: NoteReader Academic Poster