TEIA - a digital and collaborative adaptation of the *Choose Your Own Adventure* experience

PEDRO NUNO DA SILVA GONÇALVES
Licenciado

Trabalho de Projeto para obtenção do grau de Mestre em Engenharia Informática e Multimédia, na Área de Especialização de Engenharia Eletrónica e Telecomunicações e de Computadores

Orientador:
Doutor Arnaldo Abrantes

Júri:
Presidente: Doutor Rui Jesus
Vogais:
Doutor Pedro Santos
Doutor Arnaldo Abrantes

Setembro de 2024

TEIA - a digital and collaborative adaptation of the *Choose Your Own Adventure* experience

PEDRO NUNO DA SILVA GONÇALVES
Licenciado

Trabalho de Projeto para obtenção do grau de Mestre em Engenharia Informática e Multimédia, na Área de Especialização de Engenharia Eletrónica e Telecomunicações e de Computadores

Orientador:
Doutor Arnaldo Abrantes

Júri:
Presidente: Doutor Rui Jesus
Vogais:
Doutor Pedro Santos
Doutor Arnaldo Abrantes

Setembro de 2024

Acknowledgmentes

I would like to express my deepest gratitude to everyone who has supported me throughout the completion of this project.

First and foremost, I am incredibly thankful to my family and friends for their unwavering support and encouragement.

I am especially grateful to my supervisor, Arnaldo Abrantes, for his invaluable guidance and expertise. His insightful feedback, constructive criticism, and patience have been crucial in shaping this thesis, and lasted longer than what was required of him. I deeply appreciate his dedication and the time he invested in helping me navigate the challenges of this project.

I would also like to extend my sincere appreciation to Saynode, where I had the privilege of working during my research. The knowledge shared by my colleagues and the company's flexible environment were instrumental in my work.

Lastly, I would like to thank everyone else who, in one way or another, has contributed to my personal and academic growth during this journey.

Contributions

One of the key outputs of this project is the web application, TEIA. The complete source code for TEIA is available on GitHub at the following repository:

GitHub Repository

Additionally, TEIA is currently hosted online, allowing users to access the functional prototype that resulted from the implementation described in this document, directly through their browsers and without the need for any installations or configurations. This live deployment provides a functional prototype of the web application. You can explore the TEIA web app at the following link:

TEIA Web Application

Also, the mobile application (exclusive for Android) is available for the public.

TEIA Android Application

Statement of integrity

I declare that this project work is the result of my personal and independent research. Its content is original, and all sources listed in the bibliographic references were consulted and are duly mentioned in the text. I further declare that all scientific and technical references relevant to the development of the work are duly cited and included in the bibliographic references.

The author

Pedro Nuno da Silva Gonçalves

Lisbon, September, 2024

TEIA - a digital and collaborative adaptation of the *Choose Your Own Adventure* experience

Abstract

TEIA, the project in mind, intends to bring the highly acclaimed *Choose Your Own Adventure* books into a digital, collaborative format. CYOA books (as I will refer to them throughout this document) are single-player reading experiences, written in second-person, where the reader is the protagonist of the story and makes decisions that affect the course of the story. TEIA, a web-app, will host this experience virtually, adapting it to include a new agent - the writer. While CYOA books come with a story written before-hand, TEIA will take advantage of its digital context and provide a real-time storytelling experience, in which writers asynchronously contribute to the next chapter in the story and readers venture through it at their own pace. To improve on the readers' experience, TEIA will support text interactions, to allow decision-making and other features customizable by the writers. To improve on the writers' experience, Teia will provide them with powerful *Artificial Intelligence* tools to stimulate their creativity.

Key-words: storytelling; real-time collaboration; non-linear narrative; interactive reading

Glossary

Chapter: In the context of TEIA, a Chapter is an indexed segment of a story (check *Story*), and consists of a set of pages (check *Page*). The chapter is the story unit the writers will periodically publish in real-time sessions. They work on one chapter, publish it, and then start working on the next one.

Connection: In this document's context, two pages are connected, when they are two vertices connected by one path, in the chapter's rooted-tree (check *Rooted-tree*). Note that by being connected, two pages are not necessarily linked (check *Link*).

CRDT (Conflict-Free Replicated Data Type): A type of data structure that supports simultaneous changes from different agents. Generally, it is a data structure that is replicated across multiple computers in a network. But in this document's context, text CRDT is a data type designed for maintaining concurrent text editing across distributed systems. It's a simpler, more modern approach to text collaboration than OT (check *OT*).

CYOA (Choose Your Own Adventure): A book written from second-person point of view, in which the reader is the protagonist and makes meaningful decisions that affect the course of the narrative.

Delta: A simple, yet expressive format that can be used to describe a document's contents and changes. The format is human readable, and easily parsible by machines.

Generative AI (Generative Artificial Intelligence): A type of artificial intelligence capable of generating original content, whether it be text, images or anything else.

Link: In this document's context, two pages are linked when one is mentioned in the other, through a choice snippet (check *Snippet*). Note that a link is different from a connection (check *Connection*).

OT (Operational Transformation): A technology for supporting a range of collaboration functionalities in advanced collaborative software systems. A far more complex approach to text collaboration than CRDT (check *CRDT*).

Page: In TEIA's context, a Page is a single body of text from which the readers (check *Reader*) cannot escape unless they interact with snippets (check *Snippet*). Pages are connected to each other in the chapter's rooted tree (check *Chapter*). TEIA will refer to that rooted tree when the reader triggers a choice snippet.

Reader: In this document's context, a reader is a role TEIA users can assume. Readers are responsible for reading chapters of the active story and interacting with the text to trigger events.

Real-time Storytelling: The asynchronous interaction between writers and readers that allows stories to be created "on the go". In TEIA, groups of users are created and roles are defined before-hand. When the group starts, the writers start working on the first chapter right away, while the readers patiently wait. When the first chapter is published, the readers are notified. Meanwhile, the writers can start working on the next chapter, but they can only publish it once all readers are finished with the previous one. And so on, until the story is over.

Rooted-tree: In graph theory, a rooted-tree is an acyclic graph, in which any two vertices are connected by exactly one path. In this document, rooted-trees are used to represent narratives, and therefore are treated as directed rooted-trees. This means all paths have a designated course that establishes the flow of the narrative.

Snippet: In TEIA's context, a snippet is a segment of the page's text (check *Page*) with which the reader (check *Reader*) can interact. They should click the segment to trigger the interaction. The most basic and fundamental type of snippet is the choice snippet, that allows the reader to make decisions and navigate to the next page. Snippets are disguises in the rest of the text.

Story: In the context of TEIA, a Story is a group of sequential chapters (check *Chapter*). This is the top-level entity of TEIA's CYOA digital adaptation (check *CYOA*). It has a name and authors. The authors are the writers that worked on its real-time storytelling session (check *Real-time Storytelling*).

UI (User Interface): The space where interactions between humans and machines occur. The goal of this interaction is to allow control of the machine from the human end, while the machine simultaneously feeds back information.

UX (User Experience): How a user interacts with and experiences a product, system or service. It includes a person's perceptions of utility, ease of use, and efficiency. Improving user experience is important when creating and refining products, because negative user experience can diminish the use of the product and, therefore, any desired positive impacts.

Writer: In this document's context, a writer is a role TEIA users can assume. Writers are responsible for creating chapters for the active story and for integrating interaction triggers (check *Snippets*) in the text.

Table of Contents

List of Figures

List of Tables

1 Introduction

Storytelling has the power to engage and inspire readers. Crafting such narratives requires a blend of creativity and structure. The process of building visual tales that readers can immerse themselves in is mesmerizing. But the most immersive format of storytelling is the one in which readers make meaningful decisions that somehow affect the course of the narrative. A fantastic example of such storytelling format are *Choose Your Own Adventure* books.

1.1 The Problem with Non-linear Books

CYOA books are non-linear experiences, written from a second-person point of view, in which the reader assumes the role of the protagonist and makes choices that dictate the outcome of the narrative. Like traditional books, the story begins on page one, but as soon as the first decision presents itself, the reader starts to jump across pages, all over the full extent of the book. For a better understanding of this concept, follows the example of a passage with two branches:

> *"As you approach the mountain from the north, a cave starts to form in the distance. Hoping it leads to the other side of the mountain isn't entirely unreasonable, but going around the mountain might be the safer route. You must, however, reach the elfish village farther south, one way or the other. If you want to venture into the cave, jump to page 375. If, instead, you wish to take the safer route around, skip to page 134."*

Most CYOA books arrange narrative passages randomly throughout the full extent of the book, meaning the reader might read the passage in page 63, then jump to page 375, then back to page 134.

While these books offer a unique, immersive experience, their physical format presents certain challenges. The process of jumping to another page is inconvenient and immersion-breaking. Its non-linear structure and unconventional format hinder a seamless reading experience, as it breaks the flow of the narrative. The very essence of CYOA books makes them difficult to navigate, limiting their potential to create deeper immersion.

Furthermore, these books are, unfortunately, a solo experience.

1.2 Transition to Digital

My proposal is to bring CYOA to a digital, more convenient format. TEIA, the project in mind (Portuguese for cobweb), a web application designed in a way that it facilitates the traversal of narrative passages and interaction with the story, will also promote social interaction.

In a digital format, the CYOA experience can be modified to include multiple readers in the same story. It can also be modified to include the option to write a story, instead of reading one. It can even support writer collaboration, so that multiple writers can work on the same story.

TEIA will include these features and more. The main purpose of TEIA is to create interactions between users during a CYOA session. For that, the term “real-time storytelling” comes into play, and will be explored in the next chapter.

1.3 What TEIA Provides

To understand how TEIA will accomplish what it is set out to, it is important to first establish what it will provide. As a digital collaborative adaptation of CYOA books, TEIA should include the following key features:

- **Forming Groups –** TEIA should provide users with a way to group together for a storytelling session.
- **Real-time storytelling –** TEIA will be a platform for both readers and writers. When users group together, they will be able to choose their role in the group – writer or reader. The idea of “real-time storytelling” comes from the fact that there is no story at the start. The story will be created by the writers, chapter by chapter. As the chapters become available, the readers can read them at their own pace.
- **Writing collaboration –** For a good content creation experience, TEIA will have to promote a strong collaboration system that supports multiple writers.
- **Generative artificial intelligence tools –** To improve on the writers’ experience, TEIA will support tools to stimulate their creativity.
- **Narrative interactions –** For a more immersive and interesting reading experience, TEIA equips writers with the ability to add interactions to their text. These interactions can then be triggered by the readers, when clicking on the respective text snippet.

These features are crucial to what is envisioned for TEIA, as a platform for storytelling.

For a better understanding of the app's key aspects, Figure 1 presents key use cases. A user should be able create and join existing groups, choosing a role to perform (writer or reader). Once the group is active, the writers should be able to work on the next chapter of the story, and the readers should be able to read chapters published by the writers. In the use case diagram of Figure 1, the fourth agent is the "Generative AI Tools". It is important to mark it as an agent because TEIA will use third-party AI systems to provide the previously mentioned tools (to be explored further in upcoming chapters).

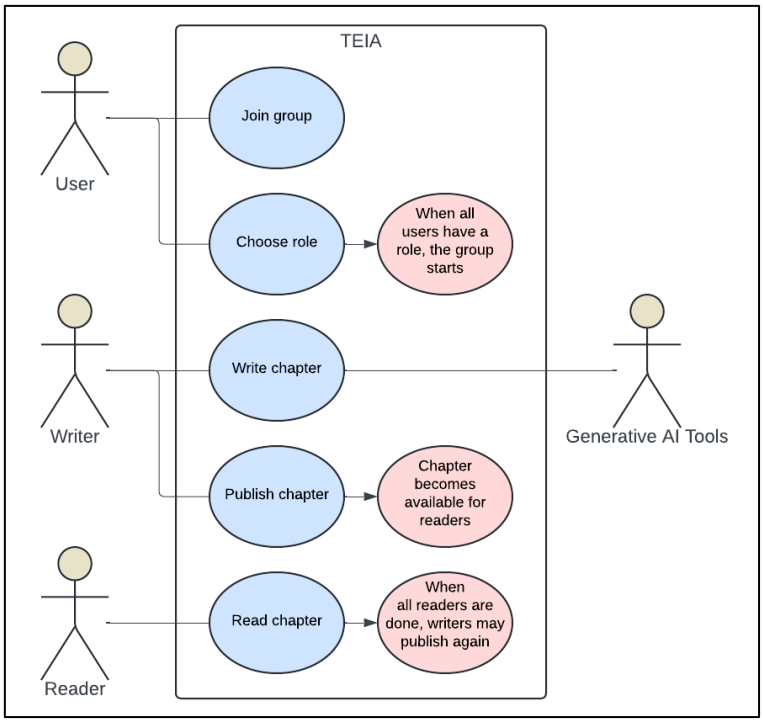


Figure 1 - Fundamental use cases for TEIA

For a better understanding of TEIA's group flow and user interaction, the machine state diagram in Figure 2 pictures the lifecycle of a group and everything involved in the process.

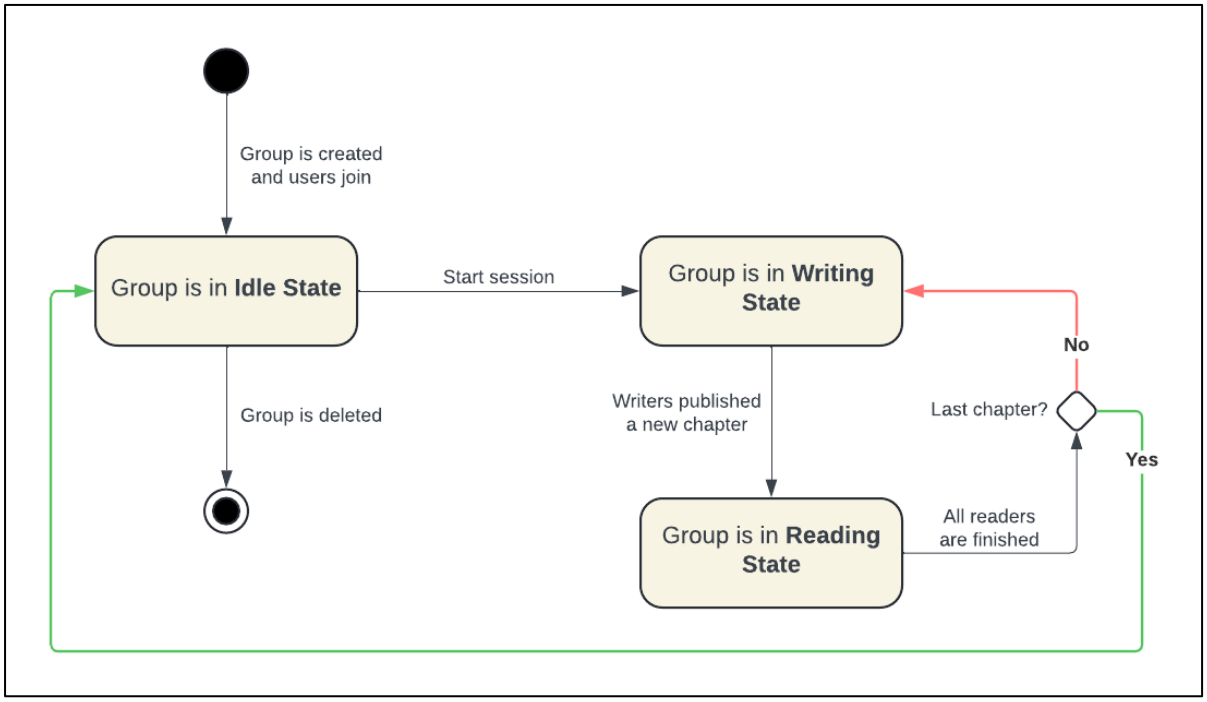


Figure 2 – State machine diagram for the group's lifecycle

A group is created by a user, that becomes the group's admin. Only the admin can delete the group or start a session. During its lifecycle, a group can navigate between three different states, as Figure 2 illustrates.

The **Idle State**, in which no CYOA session is in place, is intended to give other users the opportunity to join the group while the session is not started. When the session starts, the group transitions to another state, and no new users can join until **Idle State** is reached again

The **Writing State** is intended for writers to publish a new chapter. When all the readers are ready for the next chapter, but the writers have not yet finished it, the group will remain in this state. As soon as it is published, the group enters a **Reading State**, in which the readers can

> *Note:* In the **Reading State**, the writers should already be able to start working on the next chapter, but they can't yet publish it.

start experiencing the new chapter.

TEIA is envisioned as a web application, and the user interface/experience is just as important as the software's functionality.

1.4 Motivation

CYOA books are special to me, as they were a part of several happy childhood memories. It is a very accessible book format for children – and a gateway to the marvels of storytelling. The motivation for a project such as TEIA comes from many factors.

- **Reviving Nostalgic Formats in a Digital Age**: CYOA books are beloved by many readers for their non-linear nature, but their physical format limits how readers can experience them. Transitioning this format to the digital world makes them more accessible to modern audiences, especially as e-readers and mobile devices become the norm.
- **Increased Individual Interaction**: Traditional CYOA books offer a static form of interaction - readers turn pages to make choices. By transforming them into a digital format, TEIA enables a more immersive experience with features such as multimedia elements (images, sounds, etc.).
- **Collaboration and Social Interaction**: While CYOA books are a one-person activity, TEIA introduces a collaborative element, allowing multiple users to experience a story together and share their findings. It also allows multiple writers to create stories together and get real-time feedback from readers. This aligns with current trends toward social gaming and shared experiences, adding a layer of community engagement that didn't exist with original CYOA books.
- **Customization and Personalization**: A digital format allows any user to create their own stories for others to read, tailoring the experience based on individual preferences or demographics (age, interests, etc.).
- **Educational and Creative Potential**: TEIA can also serve educational purposes, fostering creativity, decision-making skills, and literacy. Teachers or educational institutions could use it to engage students in storytelling and collaborative learning activities.
- **Addressing Modern Content Consumption Habits**: In an era dominated by on-demand, interactive, and mobile entertainment, TEIA fulfils the need for media that is flexible, responsive, and available across platforms.

1.5 TEIA's Place in the Digital World

There is existing software that supports the creation of non-linear stories, but none support real-time social interaction and collaboration. TEIA bridges the gap between traditional CYOA books and modern digital storytelling, while adding a much-needed component of user-to-user interaction and writing collaboration. With its collaborative features, real-time writing capabilities, and support for generative AI, it will redefine the storytelling process for both readers and writers.

The next chapter will explore the existing solutions in this area and outline TEIA's unique proposition value.

2 State of Art

To better understand the place TEIA will take in the digital world, in this chapter I will explore the state-of-the-art approaches to storytelling applications. I will also study state-of-the-art approaches to the previously mentioned challenges TEIA is expected to present during its development.

2.1 Digital Non-Linear Storytelling Platforms

2.1.1 Choice of Games

There are multiple mobile apps that already transitioned specific stories to a digital format. There is a specific company that specializes in this, Choice of Games (Choice of Games LLC, Founded in 2010). Their work in digitally distributing their own stories is exemplar, and their stories are highly immersive. Their work is available on any browser, on mobile apps and even on game distribution services like Steam.

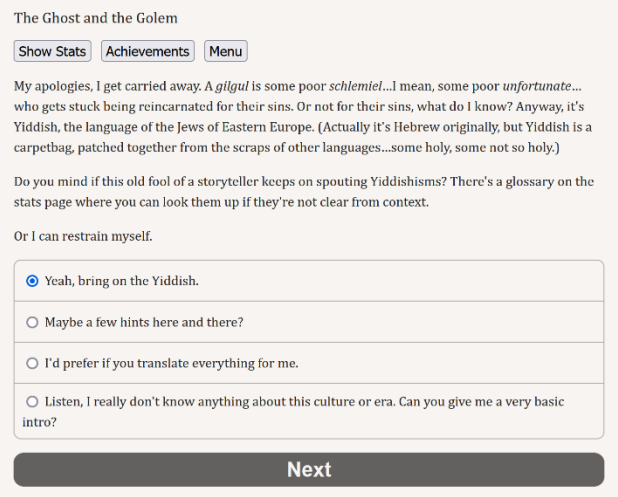


Figure 3 - An example page from a Choice of Games story

Unfortunately, Choice of Games does not provide a way for users to create their own stories, let alone support collaboration between writers or real-time storytelling. Also, the choices are explicit, as seen in Figure 3. I want choices and interactions to be disguised in the text, so that they are experienced only by the readers who find them.

Relevance to TEIA: TEIA should follow Choice of Games' example in availability and accessibility, and be present, at least, in the browser, where it can be accessed from any device with an internet connection.

2.1.2 Twine

Twine (Twine, Founded in 2009) is one of the most well-known platforms for creating and sharing interactive non-linear stories. It is an open-source tool that allows users to write branching narratives and link passages together through hyperlinks. Twine is highly flexible, supporting custom *HTML*, *CSS*, and *JavaScript*, which allows creators to embed multimedia elements and enhance interactivity. The user interface is also very good. Specifically, from the writer's perspective, Twine provides a great user experience. The writer has access to a graph (as seen in Figure 4) that represents the story's passages and links, making it very easy to understand and edit.

However, the writers' experience, although good, is largely dependent on their technical skill and knowledge around some programming languages. While it provides a robust framework for branching non-linear narratives, it lacks built-in collaboration features, and stories are written by exclusively one author. Furthermore, Twine does not natively support group storytelling or any real-time interaction between users.

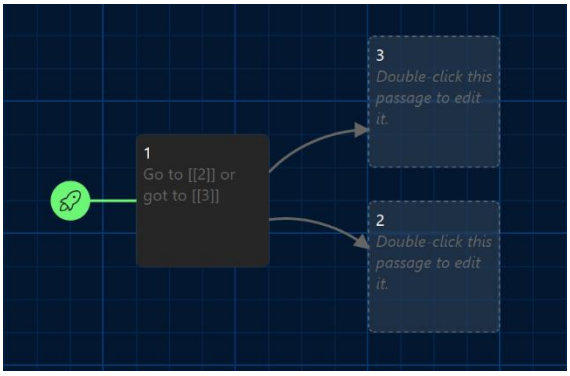


Figure 4 - Twine's story writing interface with 3 passages

Relevance to TEIA: Twine's strength lies in its non-linear storytelling capabilities, but its focus on single authorship and lack of real-time features leaves a gap in the collaborative, interactive, and dynamic storytelling space that TEIA seeks to address.

2.1.3 Inklewriter

Another interactive fiction tool designed to help writers create branching narratives, is Inklewriter (Inklewriter, Founded in 2012). Unlike Twine, the writer has no need for programming knowledge, while still providing features to track variables and create complex decision trees. This allows any user, experienced or not, to add depth and complexity to their stories.

Figure 5 - Inklewriter's story writing interface with 3 passages

Inklewriter also lacks collaborative writing and real-time storytelling, but it also lacks multimedia integration. Additionally, the user interface for creating stories is not the most intuitive. All passages are disposed vertically, like shown in Figure 5, which makes the linking sequences very confusing.

Relevance to TEIA: While Inklewriter does a great job at providing the writer tools to create complex decision trees, vertical organization of the interface is a downside. In a collaborative environment, with multiple writers, it is very important that TEIA presents a clear visualization of the decision tree and tools to easily navigate between passages.

2.1.4 Comparative Analysis of Existing CYOA Platforms

As a final analysis, Table 1 summarizes the pros and cons of each platform previously mentioned, and their respective takeaway for TEIA.

Table 1 - Pros and cons of state of art CYOA platforms

Technology	Pros	Cons	Takeaway
Choice of Games	• Availability and accessibility;	• Static stories; • Explicit choices; • No writing; • No social interaction; • No collaboration; • No AI tools;	TEIA should be a platform accessible from any device, and avoid explicit choices and text interactions.
Twine	• Open-source; • Non-linear writing; • Interactions with the text;	• Requires programming knowledge; • Explicit choices; • No social interaction; • No collaboration; • No AI tools;	TEIA should follow Twine's example in providing the writers with a graph view of the decision tree.
Inklewriter	• Non-linear writing; • Interactions with the text; • Does not require programming knowledge;	• No multimedia integration; • No graph-view for decision tree; • Explicit choices; • No social interaction; • No collaboration; • No AI tools;	TEIA should be accessible to users without programming knowledge, and avoid complex interfaces.

As mentioned across all platforms in Table 1, “explicit choices” is a problem TEIA seeks to solve. Readers should not be handed all the interactions and choices explicitly. Every segment of the text that is clickable should be undistinguishable from the rest of the text. This forces the users to ponder their decisions carefully.

Twine is the closest platform to what I envision for TEIA, although it’s missing most of its core principles, like the real-time storytelling feature, groups and collaboration between writers. It is the platform with the most effective user interface, with the most potential to accommodate multiple writers in a collaboration session.

TEIA introduces a new paradigm for digital non-linear storytelling, offering users a platform that supports multi-writer collaboration, dynamic narrative interactions, and AI-assisted creativity. The next step is to search for state-of-the-art solutions to the two following challenges:

- How will TEIA support collaborative text editing?
- How will TEIA provide Generative AI tools, and which types?

2.2 Collaborative Text Editing Platforms

Approaching the more technical aspects of the collaborative text editing, it becomes more and more evident that a custom collaborative text editor will be needed. Especially when the user experience is so important – particularly the writer’s experience. I want the writers to have simultaneous access to the passage they are editing, and to the graph (decision tree), much like Twine. A split view, essentially. In addition, I want a flexible document structure that allows the writer to add snippets to the text.

> ***Note:*** *At this point in the document, I start using specific terms to refer to certain components of this project. Keep in mind you can always refer to the Glossary for more information. The next paragraph is very important to understand the terminology around TEIA.*

From now on, what I referred to before as “passage” will be renamed to “page”. Which means a story consists of chapters, and chapters consist of pages structured as a rooted tree. A page is a body of text that may contain snippets. Snippets are interactable segments of text hidden in a body of text, that prompt something to the reader when triggered. A choice snippet, for example, would send the reader to the page the snippet is connected to. That is how readers will navigate the stories, from page to page, from chapter to chapter. Figure 7 defines the

relationships between these terms, and shows an overview of how the decision tree should look like.

I tried a few well-known collaborative text editors, and I will explain in detail why they do not meet my requirements, and why I cannot make use of them in my project.

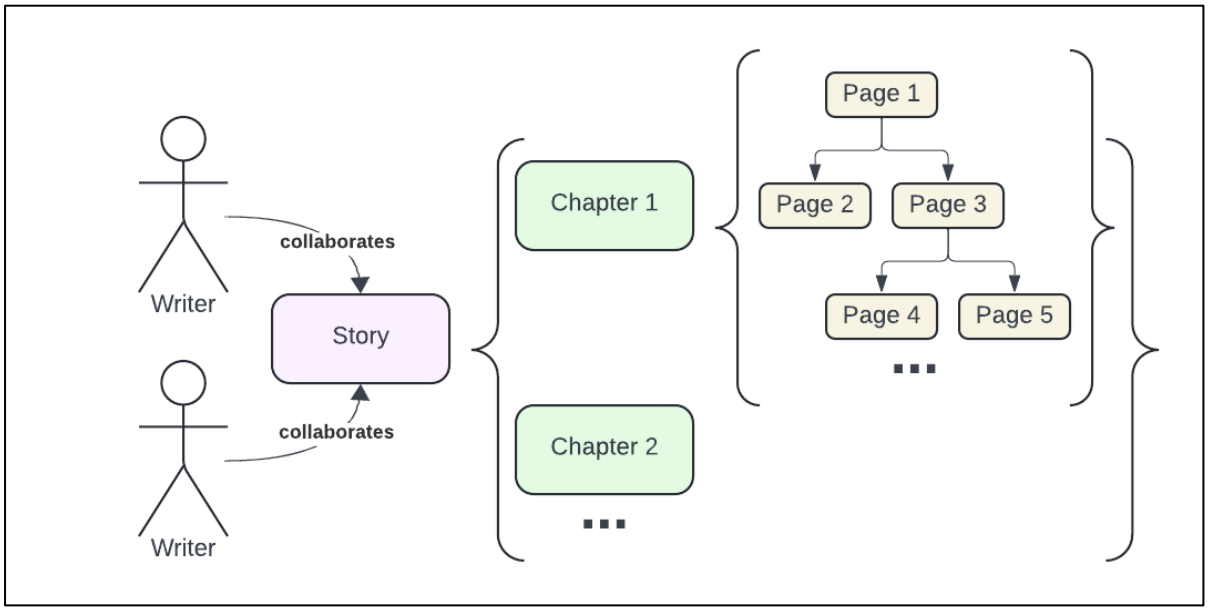


Figure 6 - Structure of a story (TEIA)

> ***Note:*** *Figure 6 shows what I have called and will call a rooted tree of pages. The reason I cannot call it a graph, is because it does not allow back-propagation. If a writer wanted the narrative to jump from page 5 to page 1, currently, it would not be possible (more on this on further chapters).*

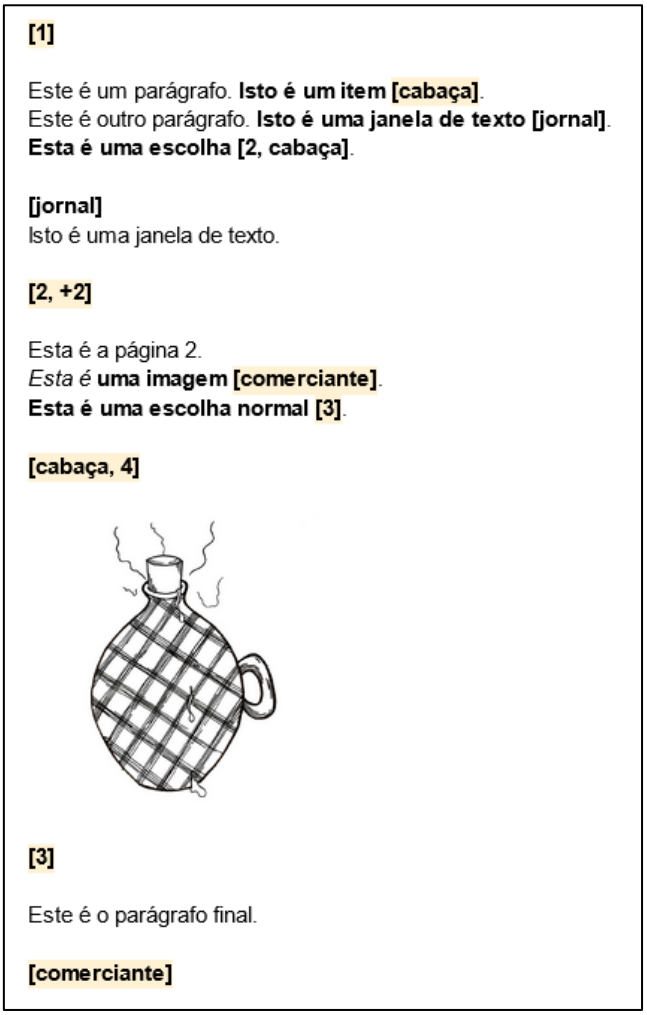
2.2.1 Google Docs

One possibility that I considered was to have writers collaborate in a robust well-known collaborative text editor like Google Docs (Google Docs, Founded in 2006), and then have TEIA serialize their document, so that it could then be showed to the readers. It is a tempting approach, since there are seamless ways to connect third-party applications (like TEIA) to Google Docs. I programmed a simple algorithm to parse information from a document and have it stored in TEIA's own data structures.

However, one of my requirements is that TEIA allows writers to have simultaneous access to the chapter view (decision tree), and the page they are currently editing. That alone excludes Google Docs as an option since it does not support web app embedding (meaning I cannot show a Google Docs view inside my web app). The only possibility would be to send the writers

from TEIA to the Google Docs app, so that they could edit the chapter. That is not very efficient, nor it is elegant.

In any case, to test its functionalities and to experiment with the page text structure, I drafted a chapter shown in Figure 7. This exercise was immensely useful. I quickly realized that I would not be able to use an already existent collaborative text editing tool, because I needed a lot of flexibility. Even with simple mock-up text, the structure swiftly became confusing for several reasons:



[1]

Este é um parágrafo. **Isto é um item [cabaça]**.
Este é outro parágrafo. **Isto é uma janela de texto [jornal]**.
Esta é uma escolha [2, cabaça].

[jornal]
Isto é uma janela de texto.

[2, +2]

Esta é a página 2.
Esta é **uma imagem [comerciante]**.
Esta é uma escolha normal [3].

[cabaça, 4]

[3]

Este é o parágrafo final.

[comerciante]

Figure 7 - Draft chapter using Google Docs

- The writer has almost no flexibility when it comes to paragraphs and line-breaks (they must follow a pattern, so that TEIA is able to parse useful information from the document).
- The indication of where a page ends and another start is not clear enough.
- Attaching images takes too much space and the environment becomes chaotic.

- The use of square brackets (to reference snippets and page identifiers) in the middle of the text flow is not user friendly.
- The parsing algorithm would be incredibly complex, and, worst of all, not very versatile. Any error by the writer would result in the parsing algorithm failing.

In summary, Google Docs is not the solution I'm looking for, because I cannot embed the document view side-by-side with the chapter rooted tree view. Furthermore, it results in an unacceptably chaotic snippet management, due to its lack of editing customization, and provides an overall bad non-linear writing experience.

2.2.2 FirePad

FirePad (FirePad, Founded in 2013) offers real-time text collaboration with no server-side code, relying on (Firebase, Founded in 2011) Realtime Database for real-time data synchronization. It's open-source, and looks like the perfect candidate. And it would have been if it was compatible with Flutter, which it is not. Flutter is the web development framework I will use to develop TEIA, because of my experience with it, in the mobile app environment (Flutter, Founded in 2017). Flutter's ecosystem, while it is growing swiftly, does not currently provide a ready-to-use package that supports FirePad.

It is a very powerful tool with great features, although it also seems very restrictive in the way it structures data, which is a big disadvantage for me. In any case, adapting a web-first tool like FirePad to Flutter's environment would require significant redevelopment, defeating the purpose of using an existing solution.

2.2.3 Git-based

A git-based system provides asynchronous collaboration, particularly useful for software development and technical writing. These systems track changes in a document as versions designated "commits" and collaborators can work on separate branches before merging their work to a main branch. These merges are accepted by a user with the required permissions to do so.

Unfortunately, one of TEIA's requirements is real-time collaborative text-editing. That will enable users to see each other's changes without having to create commits and accepting

merge requests. If another writer is editing the page I'm currently viewing, I want to see their changes happen live.

2.2.4 A Custom Approach

The tools discussed in this chapter, although powerful and very handful in other use cases, do not meet my requirements for TEIA, for their respective reasons. Synchronization is always a sensitive and complex topic in informatics, but to customize the writers' experience as intended, TEIA needs to integrate its own collaborative text editing system.

This approach will rely on existing collaborative text editing mechanisms that I will explore in the next chapter.

2.3 Collaborative Text Editing Mechanisms

A good collaborative text editing experience requires quite a robust solution because there are plenty of corner cases to consider if the document's convergence and user-intent-preservation are to be protected. This chapter will explore the most important solutions in this area, and study which one is the best solution for TEIA. Nanyang Technological University published a paper highlighting the differences between these solutions, that have now been split into three parts (Real Differences between OT and CRDT under a General Transformation Framework for Consistency Maintenance in Co-Editors, 2019) (Real Differences between OT and CRDT in Building Co-, 2019) (Real Differences between OT and CRDT in Correctness and Complexity for Consistency Maintenance in Co-Editors, 2019).

2.3.1 OT vs CRDT

The most famous and effective approaches are **OT** (Operational Transformation) and **CRDT** (Conflict-Free Replicated Data Type). The concept of Operational Transformation was first introduced in the early 1990s, while CRDT first showed up at late 2000s.

- **OT** – relies on an active server connection. The server processes the operations performed by the users and coordinates the state of the document. In case of conflicts,

caused by concurrent operations coming from different users, the operations are transformed server-side, to include all the changes.

- **CRDT** – supports peer-to-peer implementation, totally exempt from server-side code. A more recent, but quickly ascending solution. The wonder of CRDTs is that it generally comes down to how data is structured, and not how it is processed. They rely on data structures that allow for independent and concurrent modifications, while ensuring document convergence and intent-preservation.

In summary, CRDTs is very easy to implement when compared to OT, and it does not require a server. Another key difference is the fact that OT seeks to guarantee immediate consistency. In the CRDT world, eventual consistency is acceptable.

For Teia, the CRDT is the perfect approach for the following reasons:

1. Eventual consistency is acceptable, because the users will rarely be editing the same part of the document, at the same time.
2. Server-side code would be needed to transform operations, as per the OT approach. It would add to the infrastructure's complexity and financial costs.
3. The complex document structure won't become a problem because the document will be short.

To quote Joseph Gentle, one of the engineers that worked on Google Wave, the technology behind Google Docs:

> *"Unfortunately, implementing OT sucks. There's a million algorithms with different trade-offs, mostly trapped in academic papers. The algorithms are really hard and time consuming to implement correctly. […] Wave took 2 years to write and if we rewrote it today, it would take almost as long to write a second time."*

> *"[...] CRDTs are the future."*

2.3.2 How CRDTs Ensure Commutativity and Idempotency

In the following chapters, I will explain in more depth how CRDTs work specifically for text editing collaboration. For now, and with resource to an amazing article (A simple approach to building a real-time collaborative text editor, 2017), I will expound on the CRDT approach based in a simpler scenario. Kiwis.

Take the scenario in Figure 8 – Reiner, Annie and Bertholdt all have one kiwi at the start. They can add or remove kiwis from their own bowl (called the *operations*). If one of them performs an *operation*, that operation will be broadcasted to the other peers, and they should do the exact same.

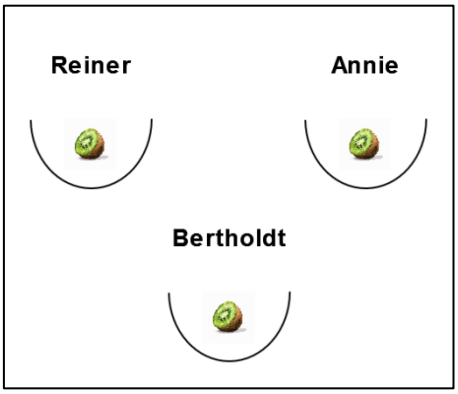


Figure 8 - Initial stage of the kiwi sharing scenario 1

This scenario would result in convergency (that is, all their bowls would have the very same number of kiwis) if Reiner, Annie and Bertholdt never performed an operation simultaneously. But they will (ingrateful test subjects).

As Figure 9 shows, Reiner intends to add a kiwi to his bowl and Annie intends to remove the existing one, as well as Bertholdt.

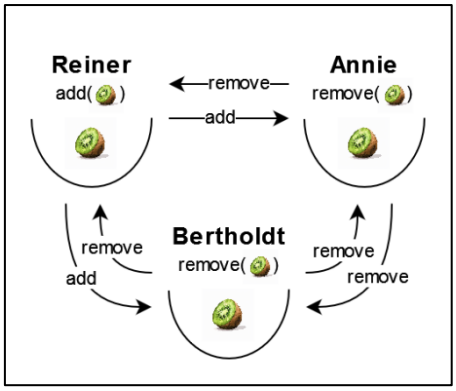


Figure 9 - Second stage of the kiwi sharing scenario 1

Depending on the timings of the operations, there are multiple possible outcomes to this scenario. Both Annie and Bertholdt can end up with either 0 or 1 kiwi in their respective bowl, depending on which operations are executed first.

Take Annie for example – the local remove operation will be executed first. So, she'll have 1 kiwi. Now there are two possibilities:

1. If Reiner's operation arrives first, she will add another kiwi, and then remove it again (as per Bertholdt's operation).
2. If Bertholdt's operation arrives first, she will try to remove a kiwi (unsuccessfully, since she has no kiwis left), and only then add a new kiwi (as per Reiner's operation).

Figure 10 illustrates a possible outcome, given possibility number 2. The scenario did not converge.

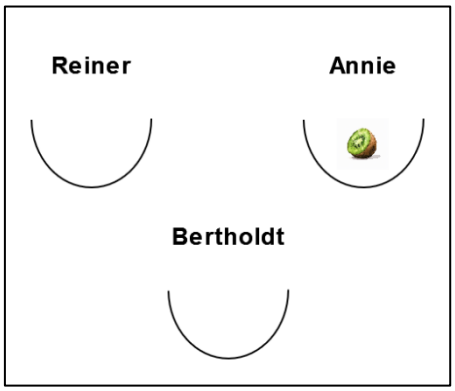


Figure 10 - Final stage of the kiwi sharing scenario 1

What if each kiwi in each bowl has a unique identifier? Well, that's the soul proposal of CRDT. Take Figure 11 – the same scenario as Figure 9, except now the kiwis are identified as **#1**. The operations now must specify which kiwi they are referring to. Note, for example, that Reiner intends to add a kiwi with **#2**, because he already has one with **#1**.

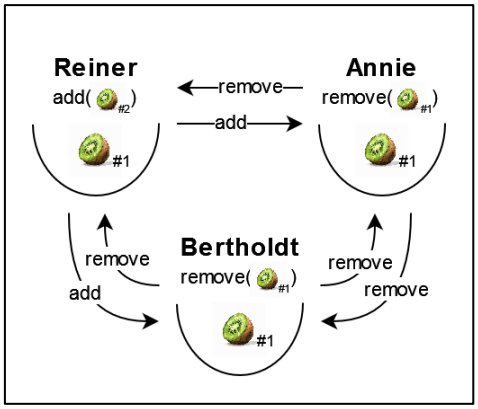


Figure 11 - Initial stage of the kiwi sharing scenario 2

This time, the scenario will converge, and everyone's intent will be preserved. For comparison, take Annie for example – she will start by removing kiwi #1, just like last time. Except now, there's only one possible outcome, independently from which operation arrives first:

1. If Reiner's operation arrives first, she will add kiwi **#2**. Then, she will try to remove **#1**, but nothing happens, because it no longer exists.
2. If Bertholdt's operation arrives first, she will try to remove kiwi #1, but nothing happens. Then, she adds a new kiwi **#2** (as per Reiner's instruction).

The outcome is shown in Figure 12. CRDTs introduced commutativity and idempotency to the system.

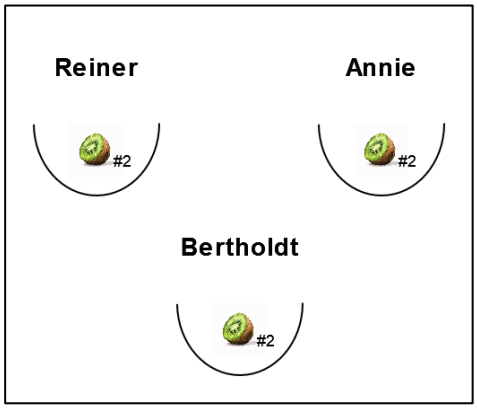


Figure 12 - Final stage of the kiwi sharing scenario 2

2.3.3 Disadvantages of CRDT

Even though its simplicity is very appealing, the disadvantages of CRDT do exist, and they need to be understood. So, why isn't everybody using CRDTs? Why is *Google Docs* still using OT? Well, for several reasons:

1. **Increased Storage Overhead**: CRDTs often require additional metadata to track changes and ensure eventual consistency. This metadata can increase the storage overhead compared to traditional data structures.
2. **Complexity in Design and Implementation**: While CRDTs offer a simpler approach to conflict resolution when compared to OT, designing, and implementing CRDTs correctly can still be a challenge, especially for more complex data types or in scenarios with specific requirements.
3. **Limited Applicability**: CRDTs may not be suitable for all types of data and text collaboration scenarios. Certain types of applications may have performance or scalability issues when using CRDTs.

The storage overhead should not be a problem for a small project, especially one that uses a NoSQL database and contains very small documents. The performance and scalability issues

may become more evident as the documents' size increases, because the unique identifiers will become more complex. Even so, this is the type of project where CRDTs thrive.

2.3.1 Deltas

Another important state-of-the-art concept to know, in the collaborative text editing field, are Deltas (Delta, Rich Text Editor). Deltas are a simple, yet expressive format that can be used to describe the content of a text document, as well as its changes throughout time.

The format is a strict subset of JSON, is human readable. Deltas can describe any Quill document, includes all text and formatting information, without the ambiguity and complexity of HTML.

In fact, Deltas will be one of the most important aspects of my text collaboration solution. They will be used to represent changes to the Quill document, whether it be text inserting changes (*inserts*), text deleting changes (*deletes*) or text retaining changes (*retains*). These are the three Delta operations to be used.

To better understand Deltas, here is an example. Consider the following Delta example:

```
{
  ops: [
    { insert: 'Gandalf', attributes: { bold: true } },
    { insert: ' the ' },
    { insert: 'Grey', attributes: { italic: true } }
  ]
}
```

This Delta contains three operations, all of them *inserts*. The structuring is very self-explanatory. As expected, if this Delta were to be composed to an empty document, the resulting document would be as follows:

Gandalf the *Grey*

Now, to exemplify the other two types of operations, here is another example (this time, composed to the existing document):

```
{
  ops: [
    { retain: 7, attributes: { bold: null, italic: true } },
```

```
    { retain: 5 },
    { insert: "White", attributes: { bold: true } },
    { delete: 4 }
  ]
}
```

The *retain* operation leaves the text unchanged, for the number of characters it specifies, still applying the formatting attributes. The first operation keeps the text as “Gandalf” but changes the formatting from bold to italic. The second operation leaves the “ the “ completely unchanged since it has no formatting attributes. The third operation inserts text in the current position (after retaining the previous characters), that is, after the eleventh index. Lastly, the *delete* operation deletes the upcoming four characters, in this case, “Grey”. So, the document now looks like the following:

Gandalf the **White**

Deltas will be extremely useful while developing a collaborative text editing solution.

2.4 The Landscape of Generative AI

The insertion of Artificial Intelligence in our society has been nothing short of revolutionary, for the good and the bad. It keeps on transforming numerous aspects of our day-to-day life. Although AI comes with an ocean of benefits, the challenges it presents are also quite significant and must not be overlooked.

When it comes to the artistic side of the world, Generative AI has ushered in a new era of possibilities, allowing machines to create good quality content in the form of text, images, music, and more, with impressive fidelity. Two notable advancements in this domain are Stable Diffusion (and the closed-source MidJourney) and ChatGPT. In this chapter, I’ll explore the technical workings of these models, delving into the intricacies that enable their generative capabilities.

2.4.1 What is Generative AI?

Generative AI is a class of artificial intelligence models that can generate new, original content based on the data they trained on in the past, whether it be text, images or anything else.

These models typically rely on deep learning techniques. Most architectures rely on a called “latent space”, which is essentially a high-dimensional abstract space where data is represented in a compressed and encoded form.

There are several unique approaches to generating content.

- **Generative Adversarial Networks (GANs):** Inspired by game theory, GANs consist of two neural networks, a generator and a discriminator. They are trained simultaneously and compete against each other. The generator creates new data instances from random noise vectors in the latent space, while the discriminator evaluates them against real data. The quality of the generated content improves with time, so that it can eventually fool the discriminator with sufficiently realistic samples. Potentially unstable training and lack of diversity in generation (due to their adversarial training nature) are some of GANs’ disadvantages.
- **Variational Autoencoders (VAEs):** Learns to encode data into a latent space, and then decode it back to generate original data. As such, it’s comprised of an encoder and decoder. To capture the inherent uncertainty and variability of data (and for expressive power), VAEs utilise probabilistic techniques, both during inference and while modelling the latent variable.
- **Transformers:** A self-attention mechanism that weights the importance of an input token in a sequence, during training. Transformers like **GPT (Generative Pre-trained Transformer)** generate text by predicting the next word in a sequence, conditioned on the preceding context. However, Transformers have proven to be highly versatile and effective across a wide range of domains. This approach does not explicitly use latent space like VAEs, but it maps input tokens into a continuous vector space, somewhat analogous to a “latent space”. In the next chapters, I will consider Transformers in more detail, with particular interest in (ChatGPT, Founded in 2022).
- **Diffusion Models:** Inspired by non-equilibrium thermodynamics. Unlike equilibrium thermodynamics, it focuses on systems that evolve with time due to external factors like temperature or pressure. Diffusion Models define a Markov chain of diffusion steps to slowly add gaussian noise to data, so that it can then learn to denoise that very same data. The object is to train a neural network that can construct high-fidelity, high-quality samples from gaussian noise, with input guidance. Throughout this document, I will shed some light on this type of architecture, with particular interest in (Stable Diffusion, Founded in 2022).

2.5 Transformers

Until not too many years ago, the advent of a large language model (LLM) that was useful, seemed so distant in time. And yet, here we are - in big part, thanks to Transformers. A technology that can create plausible and sophisticated data at a level that mimics human ability. Financial Times published an article (Generative AI exists because of the transformer, 2023) that introduces Transformers in a way that is very easy to understand.

Transformers are models trained to predict the next piece of data in a sequence. These models can be trained for several types of data, for several applications. It's a rather useful tool - even in other generative AI, like Diffusion Models, Transformers are used as attention mechanisms, for input guidance. And can even be used to predict pixels in images!

ChatGPT is a *generative pre-trained transformer* (GPT), a Transformer fine-tuned for conversational context. It deals with text that (called a *Natural Language Transformer Model)*, and is considered to be a Decoder-only model.

Transformers often comprise an Encoder and a Decoder, but this document will focus mostly on the Encoder block. Understanding how Encoders work helps to understand why GPT models don't need one. Note that the Encoder block is not too different from a Decoder block.

In Figure 13, I illustrated a very broad diagram for an Encoder of a generic text Transformer architecture. It is often composed of the following four blocks:

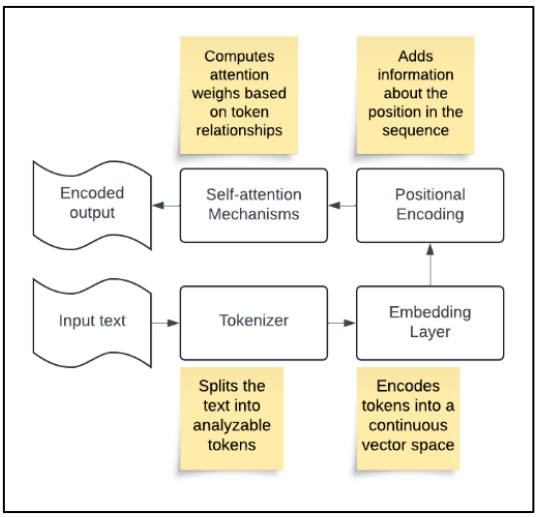


Figure 13 - Transformer encoder architecture

- The **Tokenizer** is an essential step that splits the input text into fractions, basic units that can later be encoded, called “tokens”;
- In the **Embedding Layer**, the system will encode the tokens into a multi-dimensional space, producing a “token embeddings”;
- The **Positional Encoding** block adds information about the position of each token in the sequence, to its respective token embedding;
- Finally, Transformers use **Self-attention Mechanisms** to relate each token with all other tokens. This allows the model to understand how likely two tokens are to occur together in a sequence.

2.5.1 Tokenizer

Tokenizing means splitting a text sequence into smaller units, called *tokens*. The most intuitive way of tokenizing sentences, for the human brain, is to split them word by word, and while that is a considerable approach, there are several other methods that are not only more efficient, but also optimized for certain other criteria. Two aspects that are important to consider are:

- **Token sequence -** the resulting encoded sequence (already translated to tokens). The higher, the more computational power required.
- **Vocabulary –** the different tokens recognized by the model. Demands memory. The higher, the more memory allocation required.

An important notion to have, is that the two previous concepts are not the same. The *token* sequence admits repeated tokens (can be thought of a sentence admitting repeated words), while the vocabulary consists of all the unique tokens the model is aware of.

Table 2 seeks to summarize the two basic types of text Tokenizers, and lists the pros and cons of each one.

Table 2 - Tokenizers' pros and cons

Tokenizer	Description	Pros	Cons
Word-based Tokenizers	Resort to the human-intuitive way of splitting a sequence of text into words.	• Carry more semantic information; • The token sequence is shorter, and the training and inference is faster (due to larger but less tokens); • Well-suited for tasks where word meaning and syntax are critical.	• Struggles with words not seen during training (*OOV*[1]); • Large vocabulary, each different word produces a different token; • Polysemy issues (when context is not given).
Character-based Tokenizers	Generates tokens at character level. Every letter, symbol or punctuation is a token.	• Captures nuances at the character level; • Short vocabulary, composed of only each different character, symbol and punctuation; • Handles *OOV* words and spelling errors effectively.	• Longer token sequence, resulting in slower training and inference; • Loss of semantic information, since characters carry no inherent meaning; • Polysemy issues (when context is not given).

As seen in Table 2, both types of Tokenizers have good and bad features, and might be useful for some applications but not others. But what if we could reach a compromise between the two?

[1] OOV means out-of-vocabulary – words that were not seen by the model during training.

2.5.2 Sub-word Tokenizer

Sub-word tokenization is a middle-ground between Word-based Tokenizers and Character-Based Tokenizers. These models split the text sequence into units called *sub-words*. These units are typically longer than individual characters but shorter than full natural language words.

A Sub-word Tokenizer:

- Produces a smaller vocabulary size, when compared to Word-based models;
- Has shorter token sequence length, when compared to Character-based models;
- Handles *OOV* words and spelling errors effortlessly.
- Develops flexibility across different languages;
- Retains as much semantic meaningfulness as Word-based models;
- Efficiently handles polysemy, because sub-words can contain only a portion of one word, or even different words – a phenomenon very useful for context;

2.5.3 Byte-Pair Encoding

There are several types of Sub-word Tokenizers, but I will focus on the Byte-Pair Encoding method (*BPE*), used, amongst others, by ChatGPT. It was initially developed as an algorithm to compress texts, but was then adapted for tokenization when pretraining GPT models. Hugging Faces (Hugging Faces, 2016) hosts an NLP (Natural Language Processing) course that contains a great article (Byte-Pair Encoding tokenization, 2016) on this topic.

BPE has the particularity of having a predetermined vocabulary size (**V**), decided before the process starts, as Figure 14 implies. Throughout the tokenization process, more and more

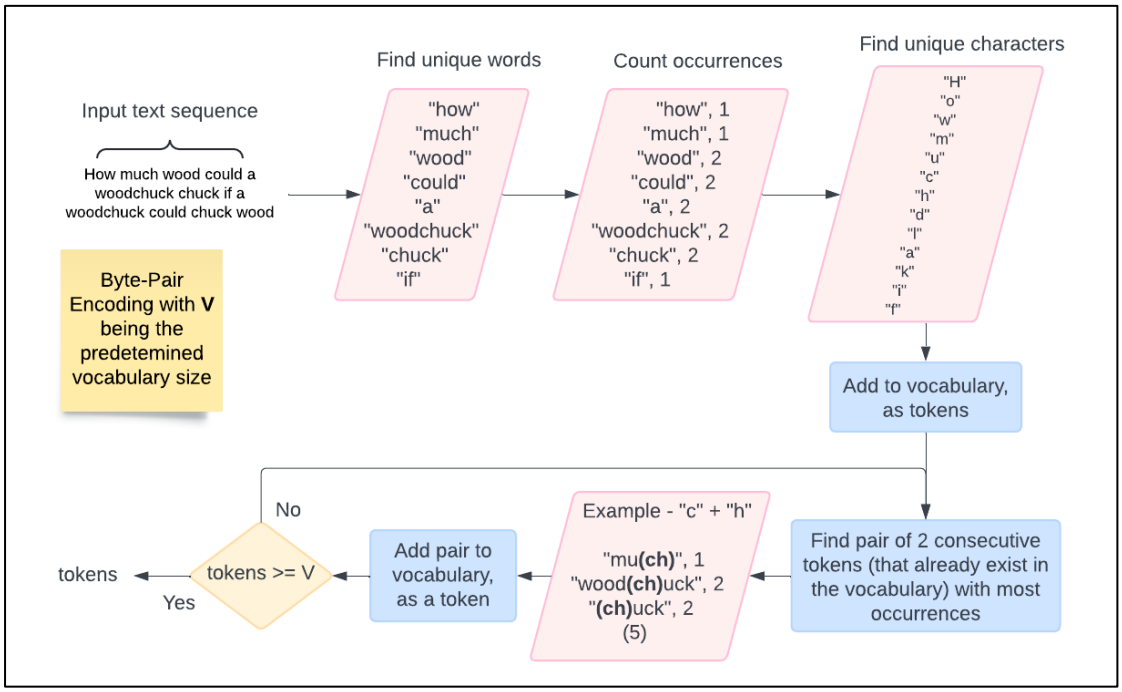


Figure 14 - BPE tokenizer process

tokens will be added to the vocabulary, until **V** is reached. That dictates the end of the process.

Figure 14 illustrates the process a simple *BPE* Tokenizer. It starts by finding all the unique words in the input text sequence, and counting the respective frequence. This information is saved into what's called the **Corpus**. Then, it finds the unique characters in the sequence, and instantly adds them as tokens to the vocabulary. For that reason, the vocabulary of a *BPE* Tokenizer is always expected to be larger than the vocabulary of a Character-based Tokenizer.

With that base vocabulary established, the system then enters a loop until the vocabulary size **V** is reached. It iteratively seeks the pair of consecutive tokens that produce the most occurrences in the current Corpus. These pairs must consist of tokens that have already been added to the vocabulary, either in the beginning, or through previous iterations. The most frequent pair is added to the vocabulary as a token, and the consecutive tokens (the pair) in the current Corpus are replaced by this new token (called **Merge**).

Naturally, the first few iterations will find pairs of tokens both with only one character, since these are the only tokens available in the vocabulary. As the vocabulary grows and iterations get processed, the system will start to find pairs of tokens with more and more characters.

I developed a script in Python to apply the BPE encoding on the following sample text sequence:

> *"How much wood could a woodchuck chuck if a woodchuck could chuck wood"*

The result is as follows:

> ***Initial Corpus:***
> *{'H o w': 1, 'm u c h': 1, 'w o o d': 2, 'c o u l d': 2, 'a': 2, 'w o o d c h u c k': 2, 'c h u c k': 2, 'i f': 1}*
> ***Initial Vocabulary:***
> *['H', 'o', 'w', 'm', 'u', 'c', 'h', 'd', 'l', 'a', 'k', 'i', 'f']*
>
> ***Final Corpus:***
> *{'H o w': 1, 'm uch': 1, 'wood': 2, 'c o u l d': 2, 'a': 2, 'wood chuck': 2, 'chuck': 2, 'i f': 1}*
> ***Final Vocabulary:***
> *['H', 'o', 'w', 'm', 'u', 'c', 'h', 'd', 'l', 'a', 'k', 'i', 'f', 'uc', 'wo', 'woo', 'wood', 'ch', 'chuc', 'chuck']*

Keep in mind that the Corpus in this script is represented as a dictionary, where the keys are the words and the values are the frequency. The words begin with their respective characters separated by whitespaces, to represent the fact that the only tokens available at the beginning are unique characters (as seen in the *"Initial Vocabulary"* as well). In the end, some whitespaces disappeared, symbolizing the Merges that happened throughout the iterations.

Note how, for such a small text sequence, and for such a small **V** value, the BPE process so quickly started processing Merges of as many as five characters (*"chuck"*). This Tokenizer proves to retain a lot of semantic meaningfulness, while keeping a low vocabulary size.

2.5.4 Embedding Layers

The object of the Embedding Layers is to turn each token in the tokenized text sequence generated from the Tokenizer Block, into multi-dimensional vectors. This is a crucial component of the Transformer architecture because it allows the representation of sub-words in a continuous space (a latent space), while retaining their semantic meaning.

There are several Word Embedding techniques, like Word2Vec (A Dummy's Guide to Word2Vec, 2022) and BERT (BERT Explained: State of the art language model for NLP, 2018).

In the Positional Encoding block, information about the order of the tokens in the original sequence is embedded in the vector information. This is an important step, as it ensures the preservation of positional context, because the Transformer will process tokens in parallel, not

sequentially. Also, positional encodings introduce asymmetry to the model, so that tokens at different positions are treated differently, thereby improving the model's ability to capture long-range sub-word relationships.

2.5.5 Self-attention Mechanisms

Within the scope of Generative AI, often when Transformers are the topic, self-attention is most recurring term. That's due to the effective results it produces, as well as to the popular paper *"Attention is All You Need"* (Attention Is All You Need, 2023), published by scientists working at Google, on August 2023. But what exactly is self-attention, and how is such a technique so useful?

Let us consider the following sentence:

> *The dog didn't cross the street because it was too tired.*

Who does "it" refer to? The dog or the street? I know, it is obvious to us, humans. It's not obvious to the machine, however. Self-attention mechanisms seek to address this issue, and enable the model to associate "it" with "animal", instead of "street".

As the model processes each token (each position in the input sequence), Self-attention mechanisms allow it to consider other tokens (other positions) for clues that may lead to a better encoding for the token in question.

2.5.6 GPT Models

Models like GPT-2 and GPT-3, used in ChatGPT, are said to be *Decoder-only*. What does that mean? How to they not need an Encoder, if there is input text involved?

The input for *Decoder-only* models is often a sequence of tokens, the same as the input of an *Encoder-Decoder* model, once it comes out of the Tokenizer block.

As mentioned in precedent chapters, the Encoder block transforms the sequence of tokens into multi-dimensional vectors that are subjected to self-attention mechanisms. In the case of *Decoder-only* models, there is no separate Encoder block. Instead, the input sequence of tokens is processed directly by the Decoder block. The Decoder generates the output

sequence by attending to the input sequence through similar self-attention mechanisms to those discussed previously.

Both the Encoder and Decoder blocks always compute self-attention, but the Decoder also computes **context-attention** (as seen in Figure 15). Context-attention attends to the input (in *Encoder-Decoder* models, that would be the encoded vectors coming from the Encoder), when predicting the next token. GPT models connect the input sequence of tokens directly to that context-attention mechanism.

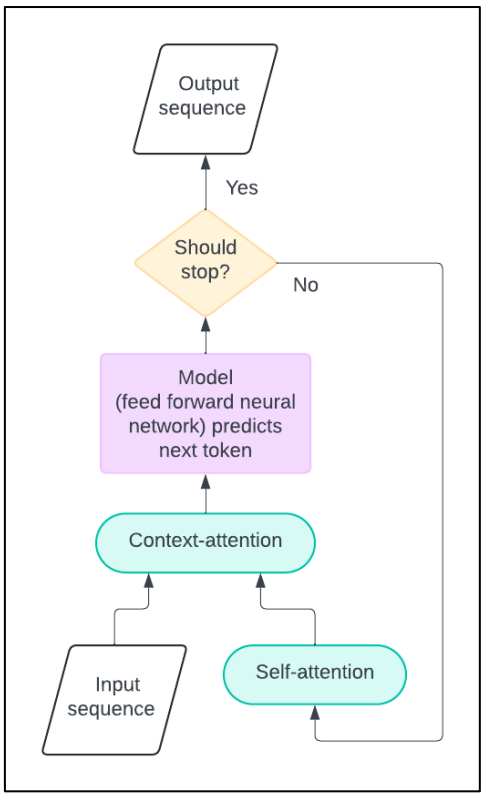


Figure 15 - Simple decoder-only flux gram

GPT models are also know for using a self-attention mechanism called **masked-attention**. While *Encoder-Decoder* models compute bidirectional self-attention, this type of *Decoder-only* models compute unidirectional self-attention, by masking future tokens. In other words, it only attends to the current sub-word and all other sub-words that precede it, ensuring that the model attends heavily to the input. The Decoder is an Autoregressive[2] model – apart from the input sequence, it relies on its own output to generate (predict) the next token.

[2] In an Autoregressive model, the output variable depends linearly on its own previous values.

2.6 Diffusion Models

One of the most useful architectures in Generative AI are Diffusion Models, particularly popular in the image generation domain.

The theoretical foundation for diffusion processes can be traced back to the work on stochastic differential equations performed within the scope of non-equilibrium thermodynamics, as mentioned. These concepts were foundational in understanding how particles diffuse through a medium.

These concepts were first used for machine learning in 2015, focusing on generative models that used diffusion processes to learn data distributions. These models were relatively simple and not widely adopted. It was only by 2020 that researchers demonstrated that Diffusion Models could generate high-quality images by learning to reverse a noising process.

Stable Diffusion is one of the most popular Diffusion Models, for two reasons.

- It is a text-to-image generative AI model that was released in 2022, and quickly got a lot of traction for its ability to generate high-quality images from textual descriptions.
- It is open-source.

For reference, as this document is being written, Figure 16 shows how far text-to-image has come in as little as a couple of years. The image was generated by MidJourney[3], with the following prompt:

Figure 16 - Image generated by MidJourney AI

> *"A painting of a wide natural landscape, where a threatening red-eyed machine stands out from the density and humidity of the swamp."*

[3] MidJourney is a Discord Bot powered to generate images from text prompts. Claimed by many as the best AI image generator yet.

The object of a Diffusion Model is to iteratively add random noise to input data (in this case, images), to train a neural network to do the exact opposite job, remove that very same noise, as accurately and effectively as it possibly can. Having said that, these models are comprised of two procedures – a forward process and a reverse process.

Before I venture into the depths of the Diffusion Model architecture, I should state that some Diffusion models support **Prompt-Embedding**, as is the case of Stable Diffusion. Prompt-Embedding is the procedure of including prompt information (in this case, text information) in the forward process. This is what makes Stable Diffusion so interesting. It is what enables the system to take in a text input from the user, and generate a high-quality image based on that. However, not all Diffusion Models are trained with that in mind. For the sake of keeping the architecture minimal, I will neglect this feature in the following chapters. After the architecture is well understood, I'll come back to it.

2.6.1 The Forward Process

The easiest way to start grasping how a specific process functions, is to clearly understands its inputs and outputs. The forward process intends to add noise to an input sample (an image, in this case).

- *Input* – an image, with a specific resolution (number of pixels). The input image should match the requisites established for the training set, whether that is resolution, aspect ratio, or even style. These models can train for samples with different resolutions/ratios, but, as one would expect, they tend to show better results when trained for one specific resolution/ratio.
- *Output* – pure noise. After enough noise-addition steps, the forward process should turn the input image into an unrecognizable pixel mess.

To illustrate the process, I created a Python script where I iteratively add Gaussian Noise to a random input image. Figure 17 shows the result of that experiment. The forward process repeatedly adds noise to the input (called **steps**) until the sample is pure noise. In this example, the number of steps (**T**) is 4. The noise added at each step is regulated by a *variance schedule*, $\beta_1, \ldots, \beta_T \in (0, 1)$.

In Figure 17, a back-propagation system iteratively adds noise to the previous output, obtaining a gradually noisier image. In the noisy images of the example, the Gaussian Noise is applied in a 2-dimensional space, while, in practice, this is to be done in a multi-dimensional

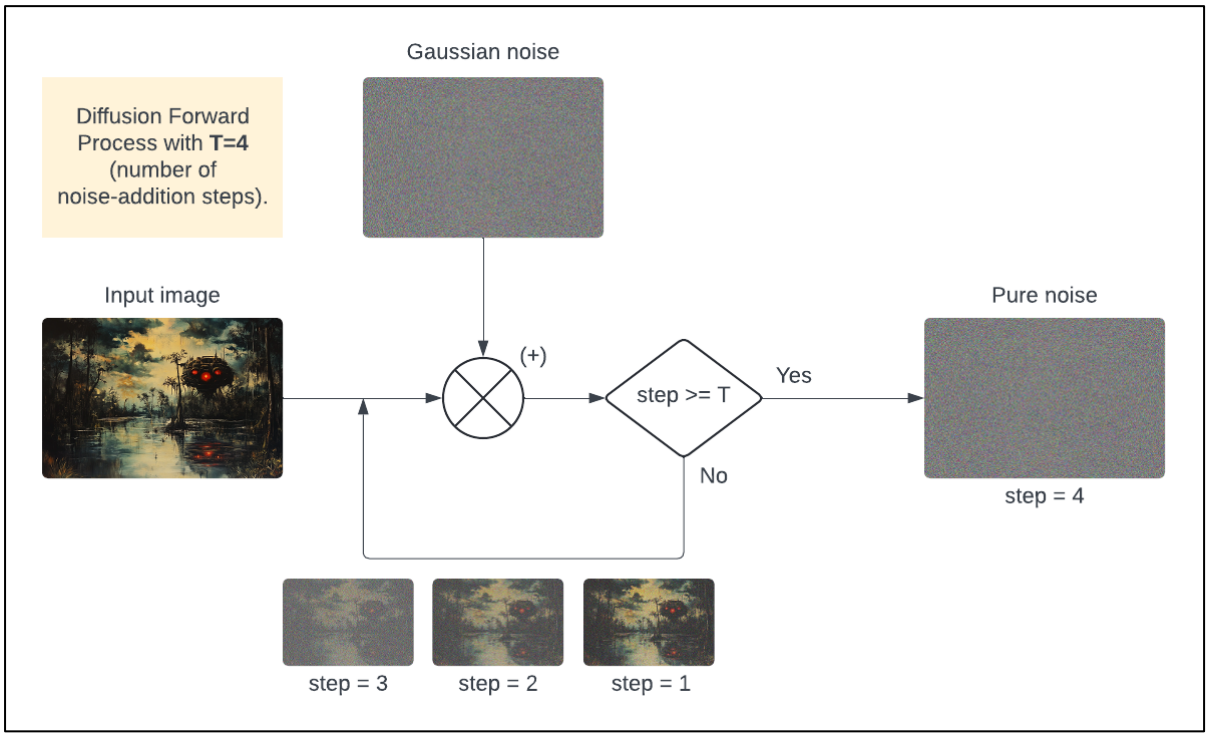


Figure 17 - Diffusion forward process with T=4

latent space.

The process of adding noise (represented by the summing junction in Figure 17) is given by the following expression:

$$q(x_t|x_{t-1}) = \mathcal{N}(x_t;\ \sqrt{1-\beta_t}x_{t-1},\ \beta_t I)$$

- $\boldsymbol{t}$ is the current step;
- $\boldsymbol{x_t}$ is the output of the forward process at step $\boldsymbol{t}$;
- $\boldsymbol{x_{t-1}}$ is the output of the forward process at step $\boldsymbol{t-1}$;
- $\boldsymbol{\mathcal{N}(\mu, \sigma^2)}$ is a normal distribution, where:
  - $\boldsymbol{\mu}$ is the mean;
  - $\boldsymbol{\sigma^2}$ is the variance;
- $\boldsymbol{\beta_t}$ defines the variance at step $\boldsymbol{t}$, according to the variance schedule;
- $\boldsymbol{I}$ is the identity matrix for whatever multi-dimensional space the scenario represents (latent space);
- $\boldsymbol{\beta_t I}$ is always a diagonal matrix of variances;
- $\boldsymbol{\sqrt{1-\beta_t}x_{t-1}}$ is the mean at step $\boldsymbol{t}$.

At each step $\boldsymbol{t}$, a new latent variable $\boldsymbol{x_t}$ is produced, with the distribution $\boldsymbol{q(x_t|x_{t-1})}$.

Considering that this process is applied throughout multiple steps (**T** steps), a tractable way to write the previous expression, from the input image $\boldsymbol{x_0}$ to $\boldsymbol{x_T}$, is:

$$q(x_{0:T}) = \prod_{t=0}^{T} q(x_t|x_{t-1})$$

However, this approach presents a serious problem. If this process is to be efficient and quick, it's not ideal to have to perform the noise addition **300** times, just to get to $\boldsymbol{x_{300}}$, assuming **T > 300**. Note that Figure 17 is an unrealistic illustration, and achieves pure noise after only 4 steps.

I'm not going to delve into the specifics of it, but one solution to this problem is the **Reparameterization Trick**, explained by the University of California, Berkeley, in their *"Denoising Diffusion Probabilistic Models"* paper, published in July 2020 (Denoising Diffusion Probabilistic Models, 2020). Essentially, this simple Reparameterization Trick can be applied to get the desired output at any $\boldsymbol{x_t}$ by simply precomputing the variances and other parameters.

2.6.2 The Reverse Process

Also called the "Denoising Process", the Reverse Process intends to transform the noisy sample (output by the Forward Process during training, or randomly generated during inference[4]) into a sample as close to the original sample image as possible, by predicting the noise added at each timestep.

- *Input* – the image that resulted from the noise-adding process, with a specific resolution (number of pixels). Also, the variance schedule is the exact same as the Forward Process, and that is a very important piece of pre-determined information.
- *Output* – a reconstruction of the original image that first entered the Forward Process. After enough denoising (noise-removal) steps, the Reverse Process should turn the noisy input data into an approximation of the earliest sample.

[4] Inference refers to the process of deriving conclusions from brand-new data samples. It is the phase in the AI system's lifetime where it simply applies the knowledge contained in the already trained model.

In the Forward Process, as **T** tends to ∞, the system approaches an isotropic gaussian distribution, in other words, a diagonal variance matrix. Now, if we learn the reverse distribution $\mathbf{q}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ we can iteratively sample a novel datapoint $\boldsymbol{x_{t-1}}$ with a starting prior datapoint $\boldsymbol{x_t}$.

However, in practical terms, $\mathbf{q}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ is unknown. What Diffusion Models often do, is approximate it with a neural network $\boldsymbol{p_\theta}$. Since $\mathbf{q}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ will also be Gaussian, for small enough , we can choose $\boldsymbol{p_\theta}$ to be Gaussian and just parameterize the mean and variance.

$$p_\theta(x_{t-1} \mid x_t) = \mathcal{N}(x_{t-1};\, \mu_\theta(x_t, t),\, \beta_\theta(x_t, t)I)$$

If $\boldsymbol{p_\theta}$ is now applied from $\boldsymbol{x_T}$ to $\boldsymbol{x_0}$:

$$p_\theta(x_{0:T}) = p_\theta(x_T) \prod_{t=0}^{T} p_\theta(x_{t-1}|x_t)$$

The neural network $\boldsymbol{p_\theta}$ will learn to predict the Gaussian parameters, the mean ($\boldsymbol{\mu_\theta(x_t, t)}$) and the variance ($\boldsymbol{\beta_\theta(x_t, t)I}$) for each timestep.

Mathematical expressions aside, the Denoising Process system is illustrated in Figure 18.

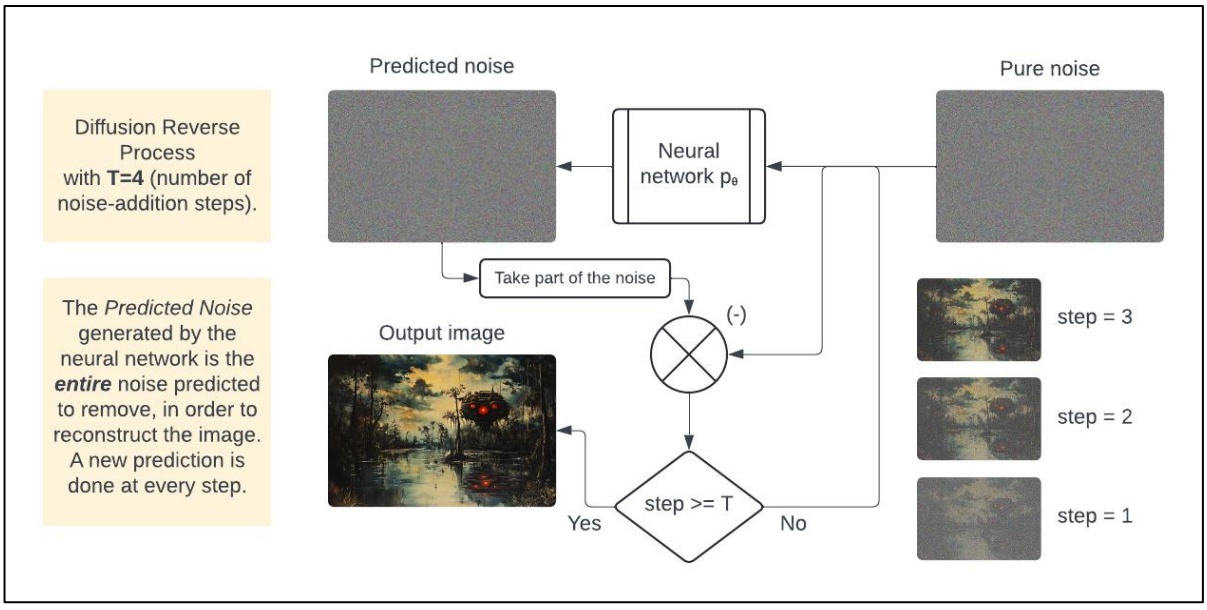


Figure 18 - Diffusion reverse process with T=4

In Figure 18, the input is the pure noise obtained in the Forward Process, and for each step:

- The neural network $\boldsymbol{p_\theta}$ predicts the total noise that must be removed from the input sample, to fully restore it to the original sample;
- Take part of the predicted noise ($\boldsymbol{n_t}$), according to the variance schedule and the current timestep;
- Subtract the taken noise $\boldsymbol{n_t}$ from the sample $\boldsymbol{x_t}$ that was fed to the neural network;
- Back-propagate the resulting data to the next step ($\boldsymbol{x_{t-1}}$ in the next step).

I will not explore the depths of how this neural network is trained, since it is not the point of this document. However, because I mentioned “Gaussian noise” so many times, I will briefly explain what it is, and why it is used.

2.6.3 Gaussian Noise

I have brought up Gaussian noise multiple times now, and how it is often used in Diffusion Models. But why Gaussian noise, specifically? Unlike any other sort of noise generation, Gaussian noise is used a lot, for example, in communication systems testing and modelling. That is due to two important reasons, that I will detail in upcoming chapters:

1. **The Central Limit Theorem –** Provides a strong theoretical basis for the diffusion process.
2. **Not That Noisy an Equation –** Gaussian noise follows the normal distribution, which makes math quite simple.

2.6.3.1 The Central Limit Theorem

Take, for example, a scenario where a dice with 6 faces is rolled *n* times. Drawing the frequency histogram (Figure 19) of this experiment, gives me how many times a number (from 1 to 6) was rolled.

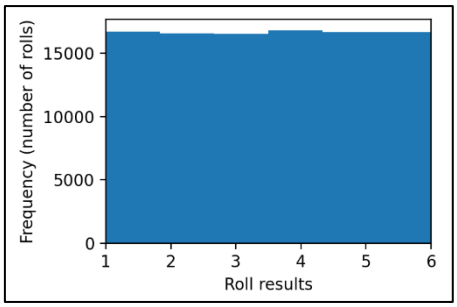


Figure 19 - Histogram for 100 000 dice rolls

What I've just considered is an example of a *uniform distribution*, one for which each outcome is equally likely. That’s why the histogram looks nearly flat. Of course, I can think of many scenarios for which this isn't the case. If I roll two dice and add the numbers, then I have now 11 possible outcomes (the numbers 2 up to 12) and, this time, not all are equally likely to be

rolled. A lot of different combinations can give you a result of 7 (1 6, 2 5, 3 4, 4 3, 5 2 or 6 1). You're much more likely to end up with a 7 than you are to end up with a 2 or 12.

This reminds of a boardgame called Catan (1995), where players roll 2 dices to determine which terrain tiles will produce resources, as seen in Figure 20.

Figure 20 - The boardgame Catan published by KOSMOS (1995)

Obviously, certain numbers are much more likely to be rolled, which means certain terrains will produce resources way more often. Players need to make strategic decisions as to which terrains to occupy, according to the rareness of each resource and the likelihood of its production.

Let's think of a scenario where the dices now have 20 faces, and we group rolls into samples, each with 5 rolls. Note that I'm using the mean, and not the sum of rolled numbers.

The histogram on the left (Figure 21) shows the frequency of each dice outcome, with no regard for samples or mean calculation, just like Figure 19. The histogram on the right (Figure 21), however, graphs the mean of each sample of 5 rolls. The edge mean values are much less likely to be rolled. The scenario has developed into an approximation of a normal distribution.

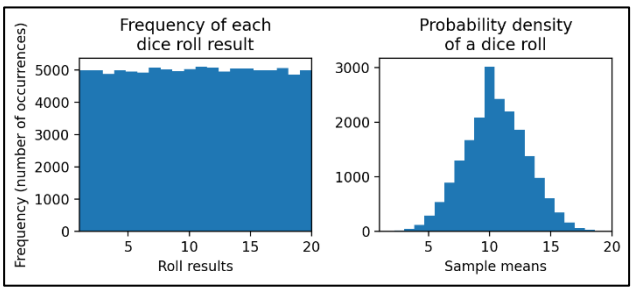


Figure 21 - Comparison between roll result occurrences and sample mean occurrences

As I increase the number of rolls, the approximation gets closer and closer. But let us not forget that we are working with discrete values. If I took, as another example, the height of the

population (which can range from the smallest person in the world to the tallest), I would be dealing with a real, continuous and infinite set of possible outcomes. That's why probability density functions are so important. They allow predictions to be made in such an environment.

There's one inherent, but not inhibiting limitation that comes to mind. Let us imagine I need to know the probability of someone being 173 centimetres tall. At first glance, it seems like an approachable challenge. But the more I think about it, the more my answer closes in on zero. The problem is that the person needs to be **exactly** 173 centimetres tall. Not one millimetre above. Not one micrometre above. And so on... It tends infinitely to zero.

What probability density functions can do, is work with ranges of values, which is perfect for deep learning. Asking, for example, the probability of a person being less than 173 centimetres tall can be dealt with by simply calculating the integral from 0 to 173, for the probability density function.

Essentially, the Central Limit Theorem provides Diffusion Models with the assumption that the data at intermediate steps follows a Gaussian distribution, providing a stable and robust foundation for the training process, making it easier to learn the reverse process. It also provides a solid theoretical basis for the iterative noise addition and removal processes during inference, ensuring that the overall diffusion process converges to a predictable and well-defined result.

This miraculous theorem is, however, not the only reason why Gaussian noise is chosen over other noise generation types.

2.6.3.2 Not That Noisy an Equation

Gaussian noise is preferred by Generative AI because it follows a normal distribution, a mathematically tractable and well-understood distribution.

The entire distribution is described by only two values – the mean and the variance. The standard deviation (σ) (which will be mentioned a lot more from now on) is the square root of the variance (σ^2).

Looking at a normal distribution (Figure 22), the mean (μ) determines the location of the curve. If the value of the mean decreases, the curve will shift to the left, and if it increases, the curve will shift to the right.

The standard deviation determines the width of the curve. A larger value for the standard deviation means the curve will be shorter and wider. A smaller value for the standard deviation means the curve will be taller and narrower.

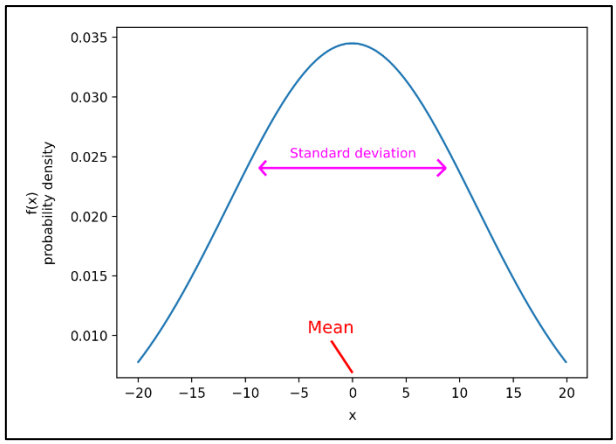


Figure 22 - Components of a normal distribution

These two parameters are very important in the determination of the distribution's MLEs (Maximum Likelihood Estimations), crucial for ensuring the accuracy of the predictions made by the neural network from the Reverse Process. My project report will not delve into the specifications of these estimations or the math involved. But it is important to understand that the use of Gaussian noise (instead of any other type of noise) makes the calculations relatively easy.

2.6.4 Prompt-Embedding

Now that I have a general idea on how Diffusion Models' Reverse Process works, the final important concept to understand is Prompt-embedding.

As mentioned on a previous chapter, what makes Stable Diffusion so popular, is its capability to process text input and generate an image based on that. This input is called the **Prompt**. During inference, the goal is to condition the Denoising Process into generating an image coherent with the desired text description.

For the sake of simplicity, I've explained the Reverse Process as if the only information the neural network needs were the distorted image sample. In practice, each step in the Reverse Process adds an embedding with information about the current timestep and Prompt. The goal is to condition the Denoising Process.

The first step is to convert the text information to what is often called the “text embeddings”, using Transformers. The encoder transforms the Prompt into high-dimensional series of vectors, capturing the semantic meaning of the text. The text embeddings are then used to condition the diffusion process, in a **Conditioning Mechanism**. There are several different approaches to integrating the text embeddings into the model.

- **Concatenation** simply concatenates the text embeddings with the noise vectors at each denoising step. This is the most straight-forward, but less effective approach, and is often used in try-on experiments.
- **Cross-Attention** is a more advanced method that involves cross-attention mechanisms. The model attends to the text embeddings while processing the image (or while processing the noise) at each denoising step. This the method adopted by famous models like Stable Diffusion. There’s a comprehensive paper by the South China University of Technology, called *“Towards Understanding Cross and Self-Attention in Stable Diffusion for Text-Guided Image Editing”,* published in March 2024.
- Techniques like **Classifier-Free Guidance** can also be used, where the model is conditioned on both the text and an unconditional signal, allowing for more control over the output. There is another paper by University of California, Berkeley, called *“Classifier-Free Diffusion Guidance”*, published in July 2022 (Classifier-Free Diffusion Guidance, 2022) that elaborates on this topic.

This project report does not aim at implementing a custom Diffusion Model, but it is important have a general notion of how these systems work and how to interact with them, to be able to select a good state-of-the-art solution to fulfil TEIA’s requirements.

2.7 Leading Generative AI Services

As previously referred, I will not implement generative AI models from scratch, particularly for the following two reasons:

- High level of complexity;
- It would take months of training and terabytes of graphical memory to be able to even come close to state-of-the-art models.

What I did was create an API that utilizes Stable Diffusion models (already trained), to experiment with Diffusion models and have an API ready to test. However, even a simple API for inference would require a considerable financial investment, if it were to be hosted

consistently. What I did for testing purposes was host it momentarily with Google Colab (Google Colab, Founded in 2014), which is a tool created by Google that provides Jupyter (Jupyter, Founded in 2014) Python scripts hosted in very powerful machines.

In any case, **Stable Diffusion** is the leading open-source text-to-image Diffusion Model (unfortunately, MidJourney is closed-source), and that is the service TEIA will use for image artificial generation.

For an idea-giving tool that will assist writers in completing their pages, I will use the LPU (Language Processing Unit) provided by **Groq** (Groq API, 2016), that serves as a chat-completion model. Groq is a recent solution that has displayed impressive inference speed. Groq's LPUs are designed to accelerate various machine learning models, including transformers, optimizing for high-performance and low-latency workloads.

Chat-GPT was the first alternative that came to mind. However, **Groq** has many advantages, some of them being:

- It is a cheaper service;
- Its inference times are quicker than ChatGPT;
- It is an LPU modelled for many possible applications, while ChatGPT is optimized to be a chat-bot.

3 Design and Architecture

Having explored the state-of-the-art technology and mechanisms that will power TEIA's core features, I'm equipped to start its development. This chapter will precise TEIA's requirements, explore options or the user interface and the user experience (from both writer and reader perspectives), and detail the architecture components present in Figure 23.

I will follow the MVC (model-view-controller) architectural design pattern[5], which means the architecture will be split into, four different layers – a Presentation Layer, that handles the **Views** (entities that provide the necessary UI to interact with the application), a Functionality Layer, that handles the **Controllers** (entities that enables the communication between the **Models** and the **Views**), a Business Layer, that manages the **Models** (entities that represent useful data), and a Database Layer, that stores the data remotely and provides read and write

[5] An architectural design pattern organizes an application's logic into distinct layers, each of which carries out a specific set of tasks.

access. Refer to the paper published by the Romanian-American University (Designing an MVC Model for Rapid Web Application Development, 2014), for a more detailed discussion of this architectural pattern, in the context of web applications.

In an attempt to highlight the most important components of this project, Figure 23 provides an overview of TEIA's architecture, serving as a foundational reference for understanding the

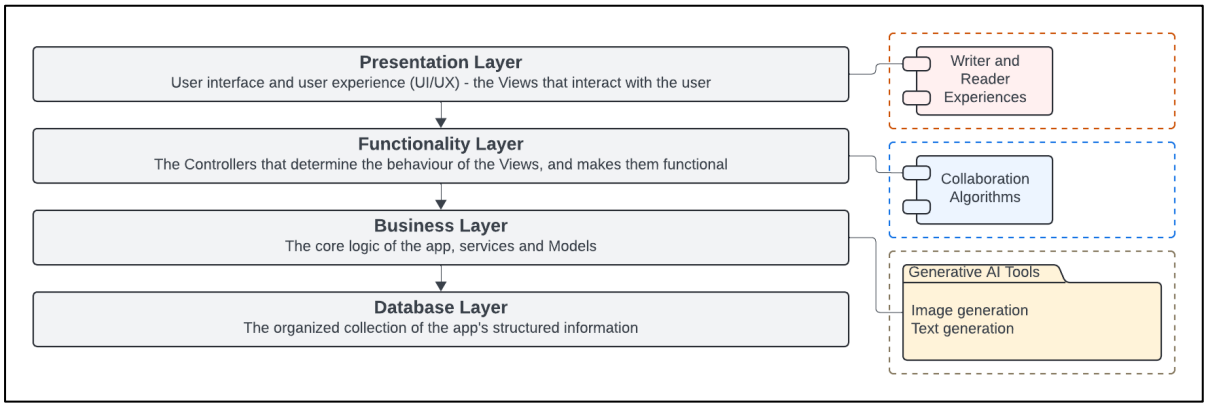


Figure 23 - Layered architecture diagram of TEIA

subsequent technical details.

Firstly, in the Presentation Layer, the layer responsible for interaction with the user, the most relevant aspect to consider is **the experience of both the writer and the reader**. TEIA must provide users with an enjoyable environment for both content creators and content consumers. In the case of writers, when working on a new chapter, they should have ease of access to all the chapter content, a way to manage interactions and an access point to the Generative AI Tools. In the case of readers, it is TEIA's object to create a seamless reading experience, as well as clear interactions with the text. From an architectural perspective, this component will consist mainly of views – refer to (Designing an MVC Model for Rapid Web Application Development, 2014).

As for the Functionality Layer, the most complex side of the application is the writer side. Since multiple writers will be collaboratively working on a chapter, TEIA needs to support powerful synchronization mechanisms, to avoid data conflicts. Specifically, when editing text (the most conflict-susceptible operation), **collaboration algorithms** are needed to preserve the convergence and integrity of the document (more on this later). Mainly comprised of controllers, this layer allows the user interface to communicate with the data models. Data models contain useful information that the Functionality Layer prepares to be shown to the user by the Presentation Layer.

Inside the Business Layer, there are some important services TEIA will need to comprise. Among them, the **Generative AI Tools service**, that will interact with external third-party

services. Two essential tools are **image generation** and **text generation**. The former used to illustrate a scenario (or anything else) the writer wants the reader to see, and the latter for inspiration or even narrative completion. The most obvious third-party services to use are, respectively, *Stable Diffusion* and *Chat-GPT*. This layer handles the structuring of useful data, so that it can be accessed by the controllers and shown by views. It also allows the controllers to edit and manage data models.

Lastly, the Database Layer should provide TEIA with a trusty way to organize all the data remotely.

> ***Note:*** *The design and architecture of a web application play a critical role in determining its performance, scalability and maintainability, in enhancing the user experience and in ensuring ease of use. For that, one must first ascertain the requirements of the system.*

Before delving into the technicalities of the design architecture, I will first establish the requirements of TEIA, as a product.

3.1 Requirements

The requirements for TEIA are subdivided into to two categories, non-functional and functional. Non-functional requirements describe the constraints or qualities needed to ensure a smooth and reliable user experience, while functional requirements describe the specific tasks the application must accomplish (mostly from the end-user perspective).

3.1.1 Non-Functional Requirements

TEIA's core feature rely on collaborative systems and synchronization mechanisms. Knowing that, the most important measurable quality about TEIA is the database response time. TEIA needs to employ a database provider that allows for almost-instant write/read interactions with the data, so that users visualize other user's changes to their common environment, with swiftness, accuracy and reliability.

Usability is another quality that TEIA must excel at. It must provide an easy to use, self-explanatory, and welcoming user interface and deliver an enjoyable user experience.

Table 3 presents the primary non-functional requirements, a description, and how they are important to TEIA.

Table 3 - Non-functional requirements and relevance to TEIA

Requirement	Description	Relevance to TEIA
Performance	The quality of being responsive and reactive to the users' actions. The users require constant machine feedback for a good and lucid experience.	The most import requirement to fulfil, as users expect to see the layout adapting to their, and other users' actions.
Usability	The quality of being intuitive and easy to use, of providing a pleasant experience. This allows for enhanced user engagement.	Another very important consideration for TEIA, given that the users, especially writers, will engage with the application for long periods of time.
Reliability	The quality of always producing accurate results. For multiple-user platforms, the quality of keeping data synchronized.	TEIA must ensure a reliable collaborative experience. The page content must be the same for all writers, always.
Safety	The quality of keeping the user experience secure and private. This may refer to user accounts, user-generated content or anything that requires safety/privacy.	To preserve the users' privacy, TEIA must prevent random users from joining private sessions, and confine groups to invited users.
Error Handling	The quality of clearly stating or handling any errors or problems that may expectedly or unexpectedly occur.	As a full stack application, comprised of multiple components, TEIA is susceptible to, for example, network errors. Error handling is essential to

		ensure a crash-free experience.
Scalability	The quality of maintaining a good performance, regardless of an increase in database population or the growing number of users.	TEIA must ascertain a good performance even for large groups and complex chapters.

As mentioned, **Performance** and **Usability** are the priority qualities to ensure for the application, but the other requirements are also very relevant for the reasons mentioned in Table 3.

3.1.2 Functional Requirements

The functionality of TEIA is central to meeting its objectives of collaboration a user interaction. This chapter outlines the key functional requirements that the system must fulfil to ensure that it meets both the user needs. As mentioned, functional requirements define the specific behaviours and actions the system must perform, detailing the interaction between the users and the application.

Figure 1 briefly introduced the fundamental use cases of TEIA, and this chapter will expand on it. Figure 24 presents all the use cases that TEIA should provide.

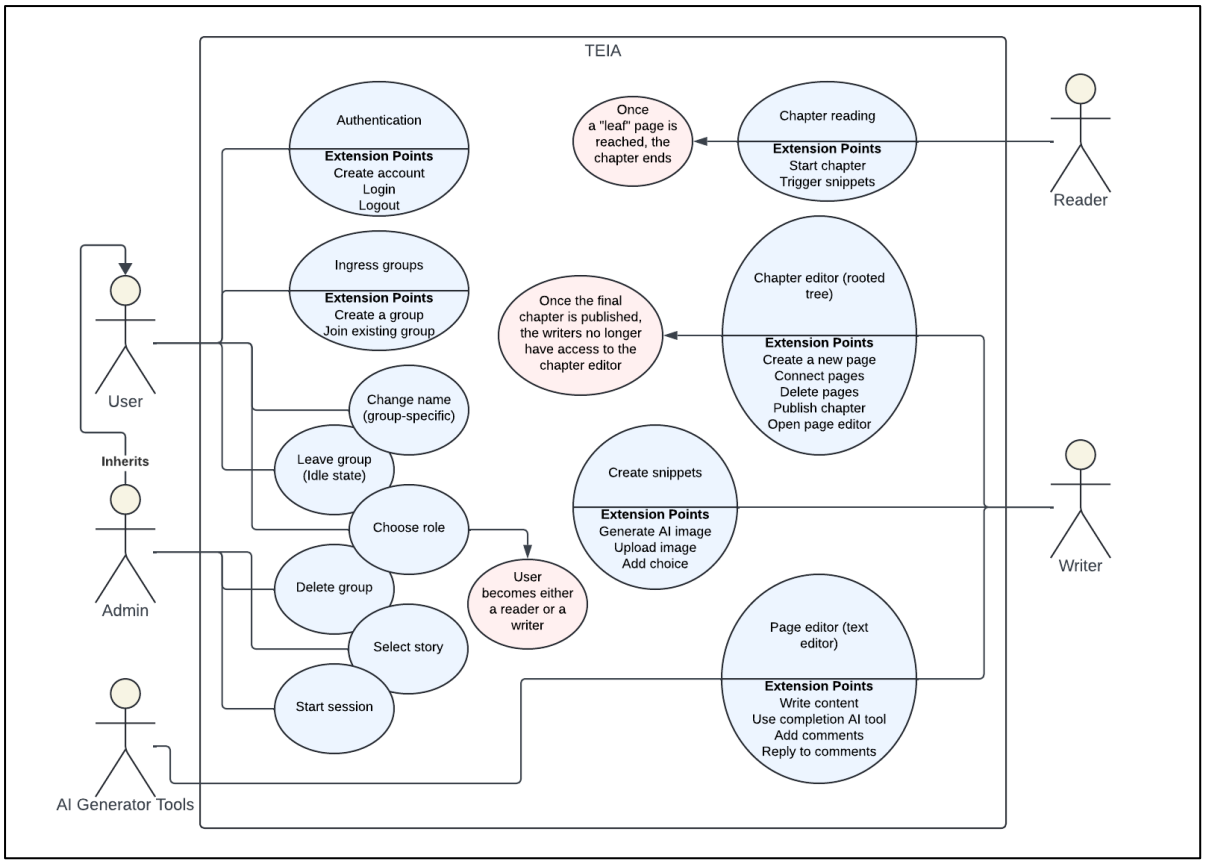


Figure 24 – Use case diagram for TEIA

There are five agents in TEIA's environment, one of which being the AI tools that the writers will have access to. The other four agents are regular users, or users with special permissions.

> ***Note:*** *In use case diagrams, external tools and third-party systems are considered agents that interact with the app. That's why "AI Generator Tools" is considered as an agent in Figure 24.*

A **user** is an agent that represents a human interacting with the app, but not yet in the context of any group or any story. Once authenticated, this is the agent that can create groups and join existing ones. When creating a group, the user becomes the **admin** of this group. Summarizing, the user can:

- Authenticate:
  - Create account, with an email and password;
  - Login, with an email and password;
  - Logout;
- Ingress in a group (by joining or creating one);
- Inside an idle group:

    - Change username (this should be saved for the group specifically);
    - Leave the group;
    - Select a role.

The **admin** only exists in the context of a group, and has extended permissions when compared to a regular user. The actions the admin can take that a regular user cannot are:

- Inside an idle group:
    - Delete the group;
    - Select story (by creating one, or selecting one that already exists);
    - Start group session.

Once inside a group, a user (admin or not) can choose a role – **writer** or **reader**. This choice, once the group session begins, will assign different permissions to the user.

A **reader** has access to chapter reading features:

- Start a chapter, if the group is in reading state (refer to Figure 2);
- Inside the chapter, interact with the text to trigger snippets.

A **writer** has access chapter/page editing features. As previously mentioned, to enhance the writers' experience, the chapter editor should display both the rooted tree of pages and the page text editor at the same time. The rooted tree of pages is a manageable tree-like diagram where the nodes are the pages in the chapter. The page text editor is the collaborative text document for the currently opened page. More on this in an upcoming chapter, regarding the design of this screen. To summarize, the writer should be able to:

- Create a new page in the rooted tree, connected to a parent page;
- Delete a page in the rooted tree;
- Click on a tree node (on a page), to open the page editor;
- In the page editor:
    - Edit the page text content;
    - Use the completion AI tool;
    - Create and reply to comments;
    - Create snippets:
        - Create image snippet (by AI generating or by uploading);
        - Create a choice snippet (by selecting the child page, the link)
- Publish the current chapter, to make it available to readers;

Figure 24 lists all the specific use cases for each agent in each context. Also, note that the "Generative AI Tools" agent play its part in the snippet creation context, for the writer agent.

The functional requirements ensure that TEIA meets user needs and supports the collaborative and social storytelling process in an efficient, engaging way.

3.2 Principles of TEIA's Storytelling

It's true that TEIA is all about storytelling. But, in the context of a digital adaptation of CYOA books, what exactly is a story? Where does it start and where does it end? In this chapter, I will determine exactly how TEIA will handle the storytelling flow and how stories will be structured by the writers.

3.2.1 Storytelling Flow

Storytelling will happen, as mentioned, inside groups. Groups are a safe and private place for users to gather and either contribute to the story as content creators (writers) or content consumers (readers).

The sequence diagram in Figure 25 accurately describes the storytelling flow that will happen inside a group. This flow was already briefly introduced by the state machine in Figure 2. Knowing which states a system can transition to is very helpful, especially in understanding a sequence diagram.

Figure 25 precises how the different agents interact throughout time, from the moment the group is created, to the moment the final chapter is read by the final reader.

> ***Note:*** *In this example, there is only one reader, but in practice, the group only transitions from Reading State to Writing State when all readers are done with the chapter.*

Reiner starts by creating a group, becoming the admin. This means only he can start the session. The group requires credentials (a name and a password) to enter, so Reiner needs to find a way to share these credentials with the users he intends to invite. This can be done through a third-party platform, like chat services (*WhatsApp*, *Discord*, etc.), or in real life.

Invitees (in this case, Annie) can only join the group while it is in Idle State. Users may select a role only while it is in Idle State. That is the next step both Reiner and Annie take, in Figure 25. Reiner becomes a writer and Annie becomes a reader.

Next, the admin (Reiner) may create a story. He needs to specify a title. After that, all the requirements to start a session are met:

1. A story is selected;
2. There is at least one writer and one reader.

The admin starts a session and the group transitions into Writing State, which means the readers are waiting for a new chapter. When one is published by the writers, the group transitions into Reading State. In this state, as pictured by Figure 25, two things happen simultaneously:

- The readers gain access to the newly available chapter;
- The writers are free to start working on the next chapter.

The group re-enters the Idle state when the story is finished. When the writers set the current chapter to be final, the end of the story is triggered, and when all the readers are done with the chapter, the session ends.

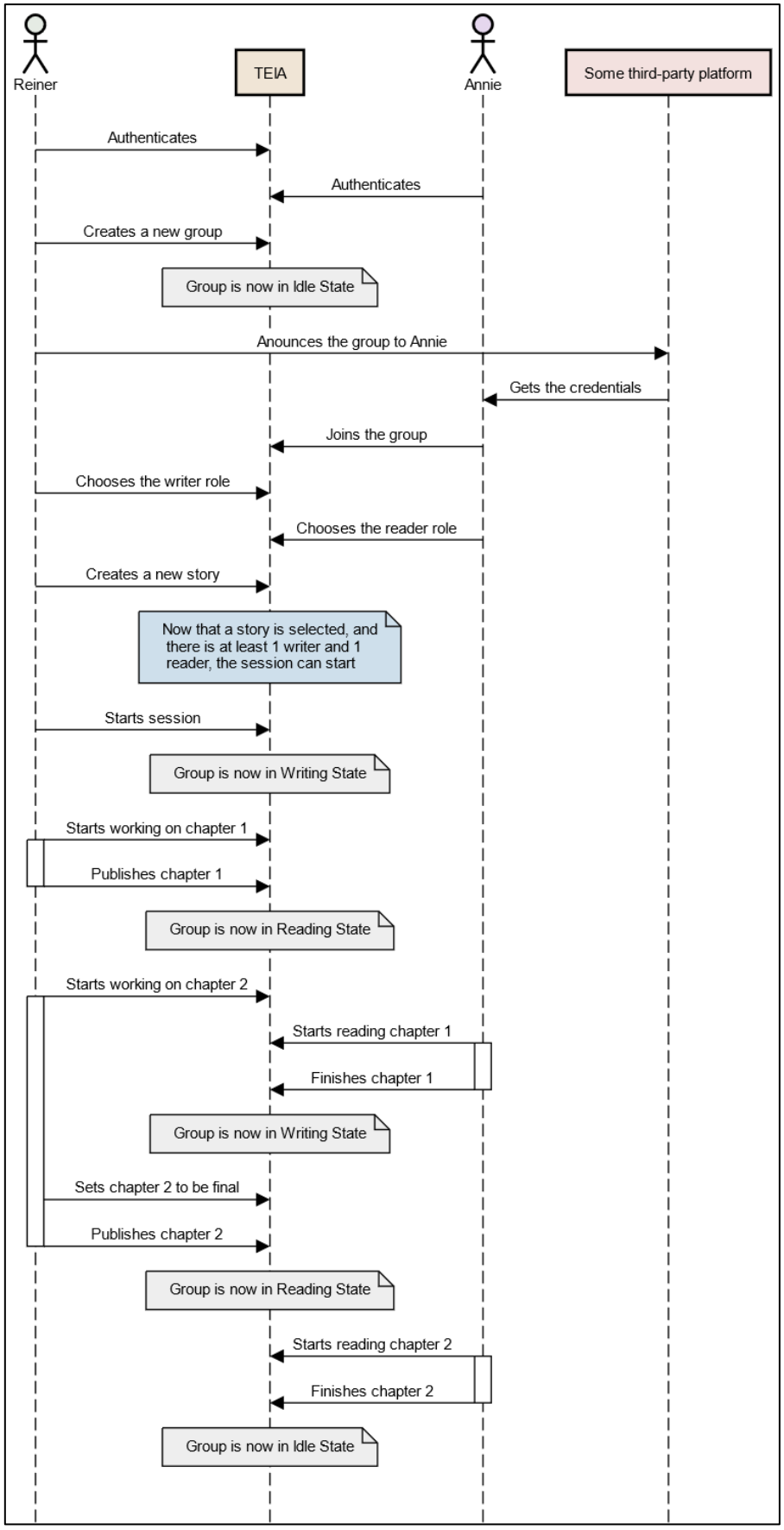


Figure 25 - Sequence diagram for the full storytelling experience

3.2.2 Story Structure

Throughout the story and its chapters, the readers will make decisions that will branch out from other possibilities. This means that different readers may finish the same chapter at different pages, considering the choices they made.

For example, in Figure 26, Chapter 1, consider that user Reiner makes a decision, in Page 1, that takes him to Page 3. In the same chapter, Bertholdt makes different decisions, and ends up in Page 6. The two users experience a different line of events, and as such, finished the chapter in a different situation.

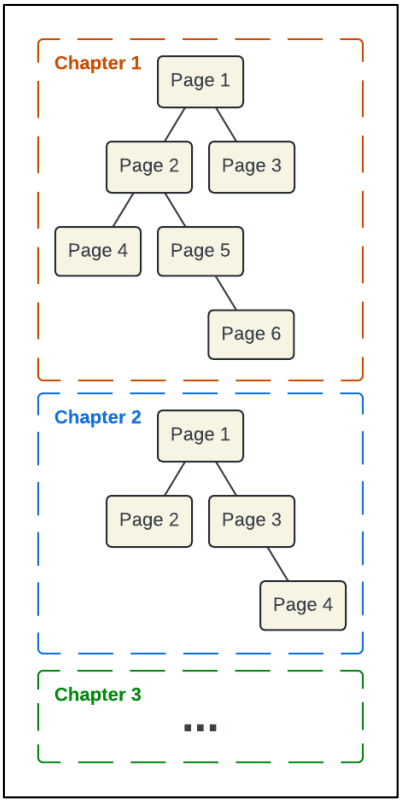


Figure 26 - Story structure of TEIA

However, regardless of where the user ends the previous chapter, the next one will have all users start at the exactly same page (in Figure 26, for example, Page 1 of Chapter 2). I made this structural decision for two reasons:

1. The writers will be tasked with less work, since they do not have to consider inter-chapter branches.
2. Because writers will comprise branches and choices to the current chapter only, there is no point in creating long chapters. This means they will be shorter and published with more frequency, which is an advantage to the readers, as well.

In summary, all the readers will start each chapter at the same page, independently from what happened in the previous chapter. This means the writers need to consider chapters as independent experiences for readers. Much like a scene-changes on stage plays.

Another great example of this kind of non-linear storytelling, are Quantic Dream's videogames (Quantic Dreams, Founded in 1997). Specifically, games like *Heavy Rain*, *Beyond: Two Souls* or *Detroit: Became Human*, where players make decisions within chapters that influence primarily the chapter in which they were made. They also may slightly impact upcoming chapters, depending on the importance of the decision.

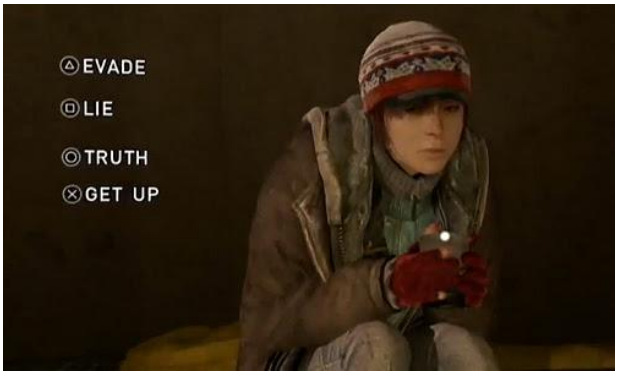


Figure 27 - Screenshot of a gameplay decision in Beyond: Two Souls (Quantic Dream)

Figure 27 is a screenshot of an example of these gameplay decisions. Ellie (the protagonist) finds herself living with a group of homeless people, due to the latest events in her life. The screenshot captures the moment in which she must decide how much information she is willing to share with them. Although this decision may affect how the chapter's outcome, it will have no impact in further chapters.

TEIA will adopt a similar approach to storytelling, as it will rely on chapter-affecting decisions, not story-affecting decisions.

3.3 Design and Screens

This chapter details the design principles, user interface (UI) decisions that will be made during the development of TEIA, and the screens that are required plus their respective layout.

The design of TEIA was driven by a need to provide users with an intuitive yet powerful interface that supports both individual creativity and collaborative storytelling. Each screen has been crafted to align with the app's core goals, ensuring that users can navigate easily.

3.3.1 The Navigation Flow

Before designing the interfaces, I need to determine the flow of the web application, to understand which screens are needed. Figure 28 presents the navigation diagram that outlines the navigation between the different required screens.

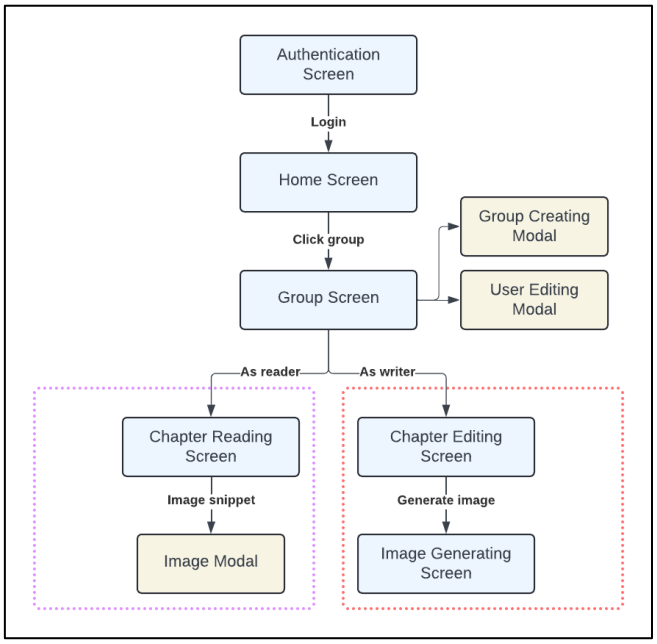


Figure 28 - Navigation diagram for TEIA

Note that the diagram includes not only screens, but modals as well. A modal is a visual element that displays on top of the current page and requires the user to interact with it.

Additionally, an important idea to retain from Figure 28 is that the writer will work in just one screen alone. The Chapter Editing Screen, although just one, is meant to contain everything the writer needs to edit chapter content. The only circumstance the writer leaves this screen is to generate an image with the Generative AI tools. Hence the Image Generating Modal in Figure 28. In an upcoming chapter, I will explore its layout in detail and explain how Teia will meet the requirements that were established for the writer's experience.

3.3.2 Authentication

User authentication is a critical aspect of most web applications, especially when managing persistent user data and asynchronous interactions. TEIA requires users to create and account to be able to join groups. This decision is driven by the asynchronous nature of TEIA's

sessions, where users can contribute to multiple stories at different times and read chapters at their own pace, as opposed to synchronous, real-time interactions.

In contrast, platforms like Gartic (Gartic, Founded in 2008) operate on a synchronous format, where users can simply choose a username and participate in live sessions immediately. This difference in session management creates a key distinction in how the two platforms handle user engagement and data persistence. In this chapter, we will explore the reasoning behind TEIA's approach to user authentication and how it facilitates a unique user experience tailored to an asynchronous format.

> ***Note:*** *The terms "synchronous" and "asynchronous" were mentioned before in this document, but in the context of collaborative text editing. TEIA must support* ***synchronous*** *collaborative text editing, in the sense that writers should be able to see each other's changes live. However, group CYOA sessions are* ***asynchronous****, in the sense that users don't have to gather in a live session and go through all the narrative in one sit. Writers deploy chapters when they can, and readers don't have to read chapters together. Each reader does it at their own pace, and when they have time.*

3.3.2.1 Synchronous vs Asynchronous Sessions

Gartic is a real-time drawing and guessing game that operates with synchronous sessions. Users do not need to create accounts or provide personal information. Instead, they can choose a temporary username and jump straight into a game. This simplicity is ideal for short, real-time interactions where user data does not need to be persisted beyond the session. Each session in Gartic has a clear beginning and end, with all players interacting synchronously, making long-term data storage or user tracking unnecessary.

On the other hand, TEIA is designed to support asynchronous sessions, allowing users to contribute to/read stories over arbitrary periods of time. In such a format, user contributions must be saved and associated with specific users, ensuring that stories can be accessed, modified, or resumed at any time.

This is why TEIA requires authentication – to identify each unique user.

3.3.2.2 Authentication Screen

Users will authenticate with an email and password. The authentication screen should have two text fields for both those parameters, as pictured by Figure 29.

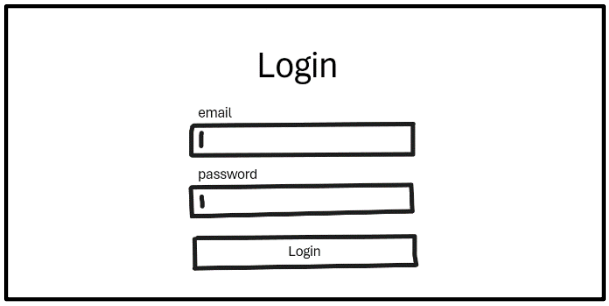


Figure 29 - Layout draft of TEIA's authentication screen

As one of the less important screens, TEIA will keep the authentication screen minimal and simple. This screen should also provide a way to perform a registration, instead of a login.

3.3.3 Home

The Home Screen serves as the central hub for authenticated TEIA users. Its design should be crafted with simplicity and usability in mind, providing users with quick access to ongoing groups, and to options to create and join new ones.

To create a welcoming environment where users can pick up where they left off, it is important to clearly display a list of ongoing groups that the user is part of. Each element of this list should contain succinct information about the group and its current state. I will opt for a bi-directional card list, as shown in Figure 30.

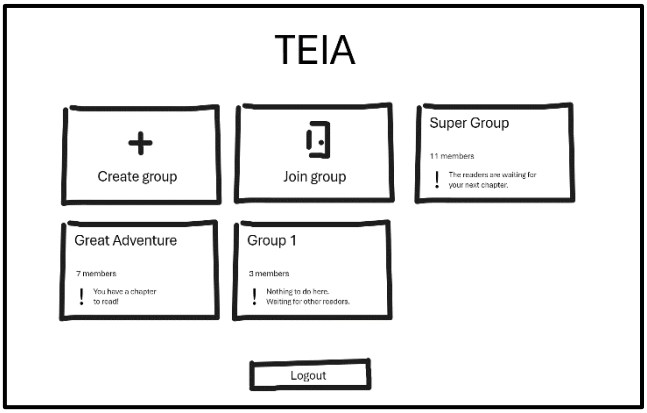


Figure 30 - Layout draft of TEIA's home screen

The first two elements of the list will be the options to create and join groups. The rest of the elements will be ongoing groups the user already joined or created. As such, when the user first joins the app, the list will contain nothing but the two first elements.

Additionally, the home screen should provide an option to log the user out, as seen in Figure 30.

The group card must not be too complex and chaotic, but, at the same time, contain information that is useful for the user to check from the home screen, without needing to navigate into the group screen. A fundamental field to add to the card is the title, so that the user can identify the group. Other fields are not as important, but can make the experience better. For example, the card could present a brief sentence explaining what is next expected of the user, in that specific group. In Figure 30, the third element of the bi-directional list contains the following sentence:

> *"The readers are waiting for your next chapter!"*

This allows the users to recognize immediately that they have work to do in that group.

3.3.4 Group

To be able to provide a dynamic experience, depending on the current state of the group's session, the group screen itself must be dynamic. Refer to Figure 2 for a state machine detailing a group's lifecycle. This screen should automatically change its layout, based on:

- The state of the group. The group can be in Idle, Writing or Reading state.
- The role of the user – writer or reader.

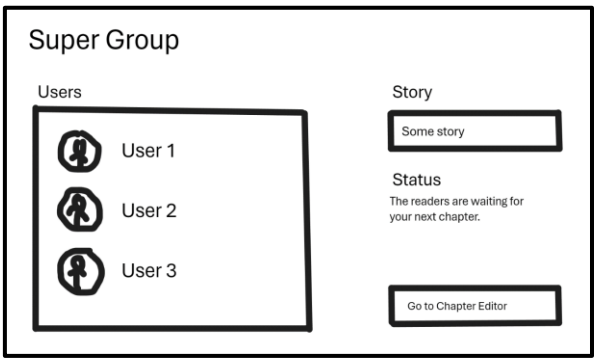


Figure 31 - Layout draft of TEIA's group screen in Writing state, from the writer's perspective

Based on these two variables, the group screen can present one of five layouts:

1. The Idle group screen should show the story currently selected, and a button to start the session (available only to the admin user – the one who created the group);
2. The Writing group screen should show the writer a button to navigate to the Chapter Editing screen. From the writer's perspective, the Writing state means readers are already waiting for the next chapter in the story. This is the example pictured by Figure 31;
3. The Writing group screen should show the reader a message stating that they are now waiting for the next chapter;
4. The Reading group screen should show the writer a button to navigate to the Chapter Editing screen, to work on the next chapter. From the writer's perspective, the Reading state means readers are still reading the previous chapter in the story;
5. The Reading group screen should show the reader a message stating that they are expected to read the current chapter. The screen should provide a button to navigate to the Chapter Reading screen.

All five Group screen outcomes should also should the list of users in the group, and what they are expected to do, given the state of the chapter and their role. Figure 28 shows an example view for this list, titled as "Users".

3.3.5 Chapter Editing

One of the key features of TEIA is the support for collaborative story writing, making the Chapter Editing screen the most important in the entire web application. This screen must be easy to use, but also provide access to all the tools intended for the writer.

As mentioned previously, an important requirement is that the writer has simultaneous access to the chapter structure and the currently opened page. Figure 32 illustrates an example of such an approach to this interface.

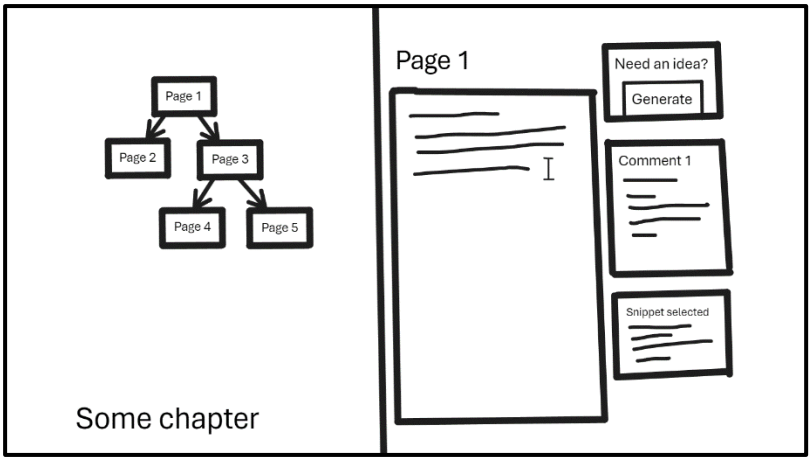


Figure 32 - Layout draft of TEIA's chapter editing screen

I decided to use a split-view layout, meaning I will divide the screen in two parts. On the left, the chapter rooted tree, representing the pages and the links, and on the right, the text editor for the page that is currently active. When the user clicks on page X in the rooted tree, the text editor opens page X to be edited.

3.3.5.1 Additional features

Figure 32 also includes some extra views that will provide the remaining features, to the right of the text editor.

- The “Need an idea?” box will connect TEIA to the Groq API and request AI assistance for this page’s completion or even total generation.
- The “Comment 1” box illustrates an example comment thread. Writers can leave comments in pages for other writers to see and reply to.
- The “Snippet selected” box shows the currently selected snippet. When the user places the cursor inside a snippet, this box appears and displays the snippet’s details.

3.3.5.2 Creating Snippets

The snippet box is very useful, but how do writers include snippets in the text in the first place? The first solution that I contemplated was adding a toolbar to the text editor that allowed snippets to be added, much like traditional editors provide text decorations like bold, italic, etc.

However, there is an even more efficient way for writers to create snippets.

When the writers want to add, for example, a choice snippet to the text, they need to specify the portion of the text that will be assigned to the snippet. In other words, the segment of text that the readers will need to click on, to trigger the snippet. Knowing this, the perfect timing to display options to create snippets, is when the user selects text, by dragging (or double/triple clicking) the cursor. When there's an active text selection, the Chapter Editing screen will fade in additional options to add snippets.

In the zoomed-in draft of the chapter editing screen illustrated by Figure 33, there is an active text selection in yellow. Because of that, the snippet options appeared above the text editor. There's an option to add a choice snippet, and an option to add an image snippet. When clicking the choice snippet option, the writers will then choose which page they wish to connect the snippet to. When clicking the image snippet, they will navigate into the Image Generating screen.

Figure 33 – Layout draft for text selection options in TEIA's chapter editing screen

3.3.5.3 Rooted Tree and Links

An important design decision I should mention is that there is a slight difference between the links between pages, and the effective rooted tree structure. This difference exists because I want users to be able to connect pages to each other, without linking them together through choice snippets.

Let us imagine that a writer creates Page 1, and writes its content without creating any choice snippets. At this point, the writer does not yet have an idea on where to add a choice snippet in Page 1, but wants to start working on Page 2. TEIA should allow the creation of Page 2, connected to Page 1, and warn the user that the two pages are not yet linked through a choice snippet.

In Figure 34, Page 2 is connected to Page 1 in the rooted tree, but not yet linked, hence the broken chain red icon. Once the writer creates a choice snippet in Page 1 that links to Page 2, this icon should disappear.

Figure 34 also illustrates the button (+) that allows for page creation.

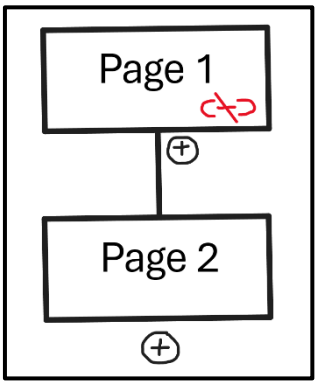


Figure 34 – Layout draft for UI detail of an unlinked page

3.3.6 Image Generating

This is the screen where writers can generate images with Stable Diffusion, and assign them to snippets. This screen is use only for generation, so I will keep the interface minimal and very simple.

As evidenced by Figure 35, this screen must, at least, provide:

- An image view, to see the generated media;
- A text field, to write the prompt that will guide Stable Diffusion in the generation process;
- A “Generate” button to generate a new image;
- A “Use” button to assign the current image to the snippet in question;

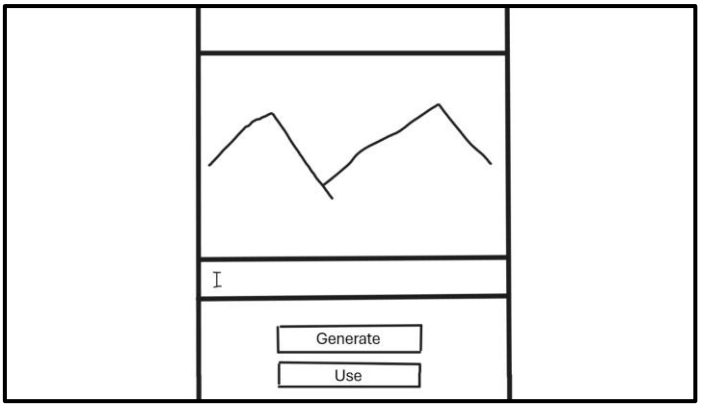


Figure 35 – Layout draft for TEIA's image generating screen

When the user clicks “Generate” after an image was already generated, the previous image is discarded permanently. Only the new image prevails, and only the new image can be used.

> ***Note:*** *Keep in mind the writer only navigates to this screen upon creation of a new image snippet. That is why the “Use” button is required – essentially to assign the image to the snippet and allow the user to get back to Chapter Editing screen.*

3.3.7 Chapter Reading

The Chapter Reading screen is only accessible by readers, and should provide the CYOA experience. If there is a chapter available for the reader, the Group screen should provide a button to navigate to this screen.

To ensure the readers’ immersion, this screen must keep the layout minimalistic, and provide almost only the text of the page currently active, as shown in the draft of Figure 36. Every chapter always starts at the first page, and as the user progresses in the chapters by triggering choice snippets, the currently active page will change.

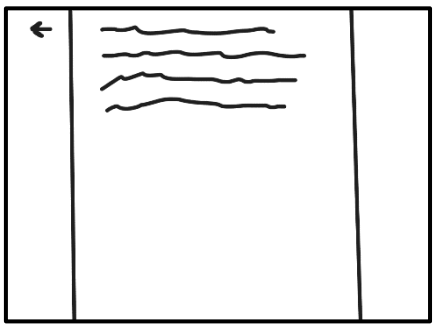


Figure 36 – Layout draft for TEIA's reading chapter screen

The action of triggering a choice snippet is unreversible. That means once the user jumps to page X in a specific chapter, TEIA does not provide a way to go back. This contributes to a more immersive experience, in which every decision is meaningful. The other interactions, like image snippets, are facultative, meaning they do not affect the narrative course.

Although not present in Figure 36, this page should also contain information about the chapter. For example, the chapter title and the number of pages read by the reader.

> ***Note:*** *The "number of pages read by the reader" is the total number of pages the user has been through, not the index of the pages themselves. For example, if the reader, throughout their chapter adventure, reads page 1, page 4 and page 7 (because of specific choices), this number will show "3", because they read a total of three pages.*

3.4 Architecture and Data Modelling

The architecture of TEIA is designed to support asynchronous collaborative storytelling, providing a robust and scalable foundation for users to create, edit, and explore dynamic CYOA narratives. This chapter will determine the system's infrastructure, architecture and data models that will form the core of the application, enabling multiple users to interact with stories in a fluid and intuitive way.

3.4.1 Infrastructure

At its heart, TEIA will follow a modular, client-server architecture, with a front-end component and a back-end component:

- **Back-end:** TEIA requires real-time collaboration. That means, not only it necessitates a responsive front-end, but also a back-end that is capable of updating and streaming data instantly. Firebase is a great candidate, due to its real-time capabilities and unlimited features. It does not rely on the conventional SQL table data structuring, but its No-SQL (non-relational) collection-document data structuring is very appealing.
- **Front-end:** As previously mentioned, the client-side software will be developed on Flutter's framework (Flutter, Founded in 2017). It relies on Dart (Dart, Founded in 2011), a programming language developed by Google, very similar to *Java* and with some features from *JavaScript*. Flutter supports deployment to multiple platforms (Android, iOS, Web, Desktop, etc.) from one single codebase. Like Firebase, the

biggest advantage of Flutter, when compared to other frameworks, is the easiness and speed of development and deployment. The focus, for now, is to deploy a web version of TEIA, and that is the platform in which it will be tested.

> ***Note:*** *Referencing to Figure 23, front-end is the component that will handle the Presentation, Functionality and Business layer, and back-end is the component that will handle the Database Layer.*

Figure 37 summarizes TEIA's infrastructure, and how each component relates to one another.

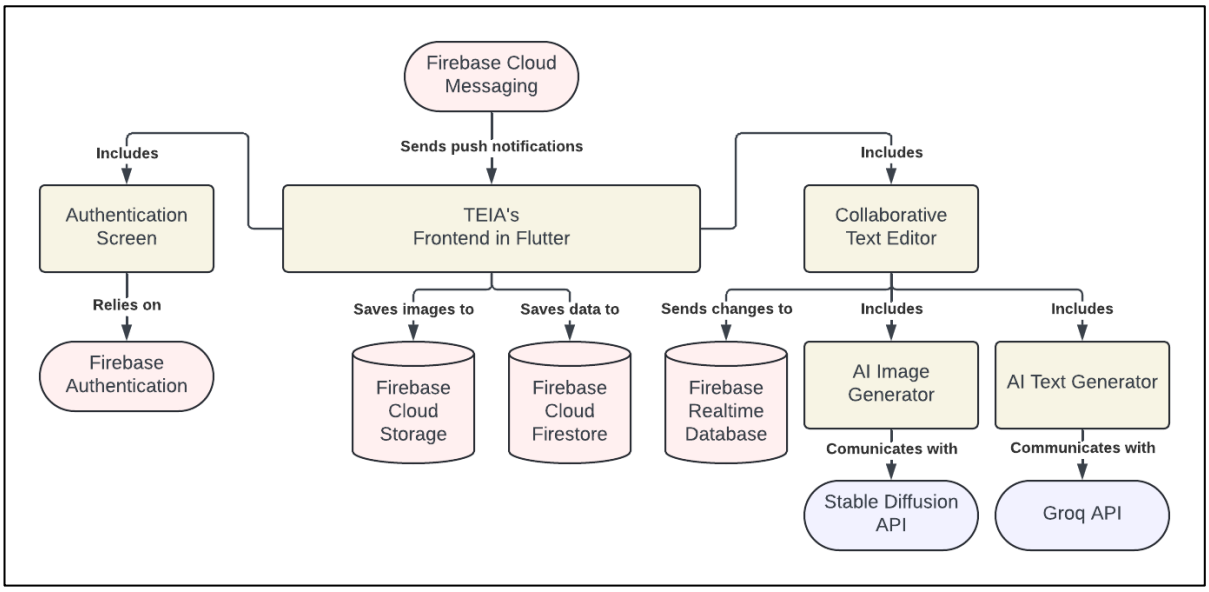


Figure 37 - TEIA's infrastructure diagram

Figure 37 references the multiple Firebase services that TEIA will use. All these services will play an important role in the system, so it is important to clearly list them and further explain why TEIA needs each one.

- **Cloud Firestore:** TEIA's main database. This where almost all structured data will be stored in the Collection-Document format (Cloud Firestore, Founded in 2017). Cloud Firestore is the most recent and scalable database service Firebase provides, with better querying and data structuring than its predecessor (Realtime Database).
- **Realtime Database:** This is also a database service, exclusively used for the collaborative text editor changes (Realtime Database, Foudned in 2012). This is a database service optimized for frequent updates, and offers a lower latency than Cloud Firestore.
- **Authentication:** Firebase's secure authentication service (Authentication, Founded in 2014).
- **Cloud Messaging:** A cross-platform messaging solution that reliably send messages to clients. It will be used to implement TEIA's push notifications (Cloud Messaging, Founded in 2014).

- **Cloud Storage:** Used to store and serve user-generated content, such as images. It will be used to store the images generated or uploaded by writers for their stories (Cloud Storage, Founded in 2012).

In the following chapters, I will discuss the architecture decisions and solutions for the components in Figure 37.

3.4.2 The Collection-Document Structure

Most of TEIA's data and user-generated content will be stored in Cloud Firestore, one of Firebase database providers. This is the database option that allows data to be structured in a more complex, flexible and scalable format – the Collection-Document format.

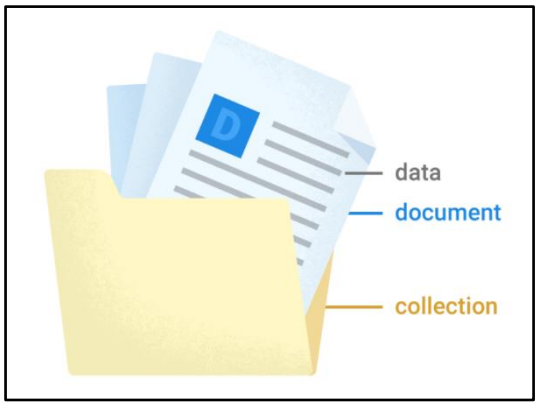


Figure 38 - Collection-Document database (image taken from the Firebase documentation)

The Collection-Document format organizes data in documents that belong to collections (check Figure 38), unlike relational databases that organize data in tables and rows. Collections do not store any actual information - they can be perceived as groups of documents that follow a specific JSON[6] structure, with key-value pairs. Every document in any collection has a unique identifier (ID) that is either automatically generated or defined by TEIA.

Documents cannot directly contain other documents but can reference other documents through subcollections. Subcollections allow for a nested data structure, where documents in a collection can have their own child collections. This creates a hierarchical, tree-like structure in which data is organized in layers. For example, in Figure 39, the **"stories"** collection

[6] **JSON** (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is also easy for machines to parse and generate.

contains documents that represent a story, and each one references a **“chapters”** subcollection, which contains the chapters for that story.

3.4.2.1 Redundancy in No-SQL modelling

Also, it is important to note that No-SQL database modelling presents some differences from conventional SQL modelling. One fundamental difference is, since reads are far quicker and more efficient than writes, redundancy is a good practice.

For example, in the context of SQL, if I wanted to show a list of the users that belong to a group, I would create a many-to-many relationship that would result in a separate table relating the table **“users”** to the tables **“groups”**.

In a No-SQL context, the approach is contrastingly different. The best practice is to repeat the needed user information in the group document. In other words, the fields under the **“users”** collection would be copied to some other field in the **“groups”** collection.

The disadvantage is, when updating user information, the application would need to update that information in all groups as well. But from a No-SQL point of view, that would rarely happen, and would happen far fewer times than reads, at a ratio that makes this approach worth it.

3.4.3 Database Model

With the database modelling rules well established, I can now design the database model. In this chapter, I will display the full database model for TEIA, discuss the decisions that were made regarding the structure and study each individual collection in detail.

Figure 39 contains the full Collection-Document database model for TEIA.

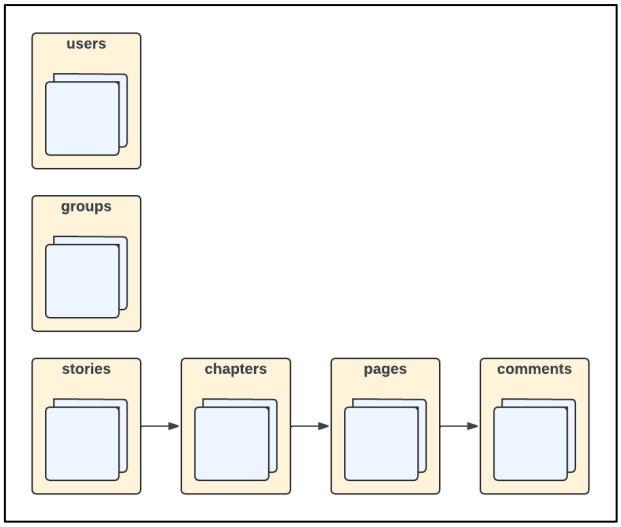


Figure 39 - Collection-Document database model for TEIA

Each document contains JSON data of a specific format, depending on the collection it belongs to. The only top-level collections in Figure 39 are **“users”**, **“groups”** and **“stories”**, given that these are the fundamental entities in TEIA’s environment. The rest are sub-collections. The arrows indicate which documents reference which sub-collections.

lot of decisions were made while developing the diagram in Figure 39. I will explain each one in detail, while I study each collection in detail, in the upcoming chapters.

3.4.3.1 Users Collection

The **“users”** collection is the simplest collection in the whole database model (Figure 40). Its humble purpose is to keep track of the users that register in the app. Upon the creation of a user account, Cloud Firestore generates a unique ID automatically, for the document that will host the new user’s information. The only information that is important to store, is the email and username.

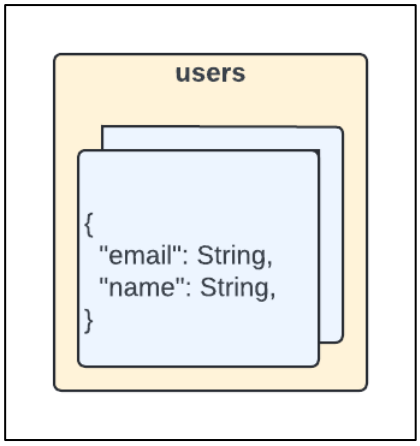


Figure 40 - Users collection

A description of each field:

- **“email”:** The email the user used to register into TEIA.
- **“name”:** The temporary name of the user. It is, by default, set to the section of the email address that comes before “@”. This is the name that will be attributed to the user whenever they join a new group. They can change it for each group.

3.4.3.2 Groups Collection

The **“groups”** collection keeps track of all the groups that have been created in TEIA (Figure 41). When users create a new group, they must specify its name and password, so that other users can use these credentials to join. Apart from that, the other fields are automatically generated, and TEIA will manage them according to the users’ actions and the state of the group.

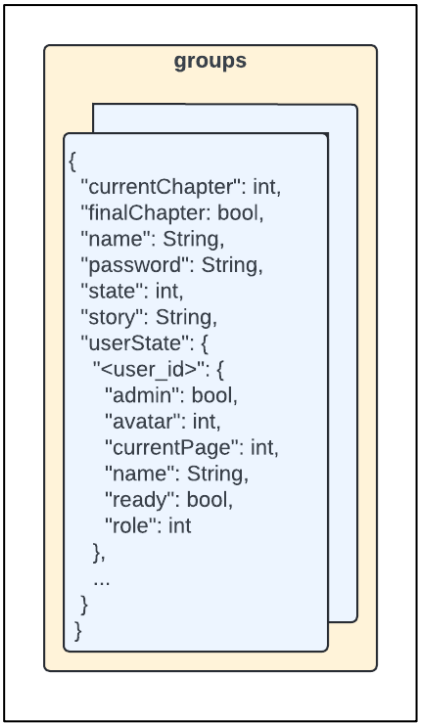


Figure 41 - Groups collection

A description of each field:

- **“currentChapter”:** The currently active chapter index. This is, always and independently of the group’s state, the chapter that the writers are working on. Starts, by default, at 1.
- **“finalChapter”:** A Boolean field that indicates if the currently active chapter is the final one. This field can be changed by the writers, before publishing a new chapter. If the field is set to true, there will not be any further chapters after the current one.
- **“name”:** The name of the group, set by the user who created it.
- **“password”:** The password required to join the group, set by the user who created it.
- **“state”:** The current state of the group. As previously mentioned, can be one of three – Idle State (0), Writing State (1) or Reading State (2).
- **“story”:** A reference to the ID of the Story that this group’s session is building upon. Can be *null*, if the group was just created and is still in Idle State.
- **“userState”:** A key-value dictionary containing information about which users have joined, and their respective state and variables. The keys in this dictionary correspond to the users’ ID, and the values correspond to yet another dictionary, containing the variables.
    - **“admin”:** A Boolean indicating whether this user was the one who created the group.
    - **“avatar”:** The avatar of the user. This represents a profile picture.

- **“currentPage”:** The current page the user is at, in the currently active chapter, if they are a reader.
- **“name”:** The name of the user (this can be changed for this group specifically).
- **“ready”:** A Boolean indicating whether the user is ready or not. This can have different meaning, depending on the group’s state and user’s role. For example, when the user is a ready and the group is in Reading State, if “ready” is set to *true*, it means the user has already read the latest chapter.
- **“role”:** The role of the user. Can be one of two – Writer (0) or Reader (1).

> ***Note:*** *The “avatar” field is mentioned in multiple collections. This is the picture of the user. The reason it is an Integer is because they will only be able to choose from a pre-defined set of possible profile pictures, each one assigned to an Integer value.*

3.4.3.3 Stories Collection

he **“stories”** collection keeps track of the stories that are created and used in groups (Figure 42). The documents in this collection reference a sub-collection named **“chapters”**. This means that a story may contain multiple chapters.

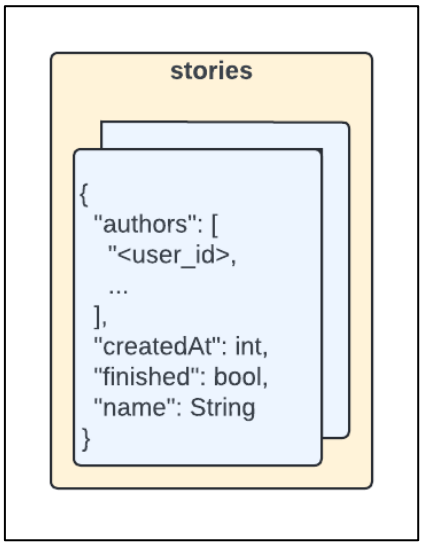


Figure 42 - Stories collection

A description of each field:

- **“authors”**: A list of the IDs of the users who worked on this story as writers.
- **“createdAt”**: The point in time (in milliseconds) when this story was created.

- **“finished”**: A Boolean indicating whether this Story has been finished or not. If the group in which this story took place already completed its session, this field is set to *true*.
- **“name”**: The name of the Story, determined by the user who created it.

3.4.3.4 Chapters Collection

The **“chapters”** collection keeps track of the chapters inside a story. The documents in this collection reference a sub-collection called **“pages”**. This means that a chapter may contain multiple pages.

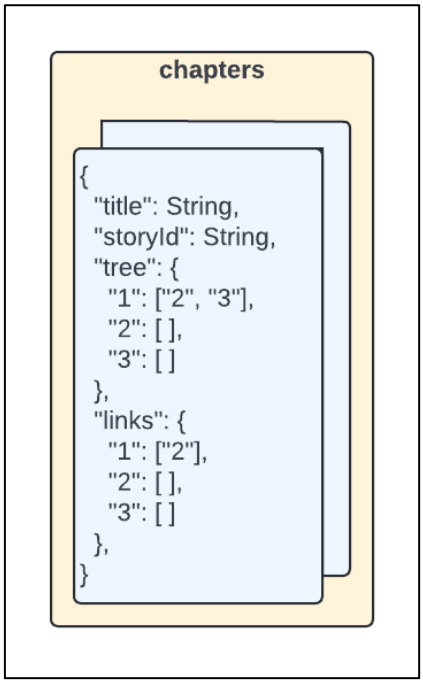


Figure 43 - Chapters collection

A description of each field:

- **“title”**: A title for the chapter.
- **“storyId”**: The ID of the story this chapter belongs to.
- **“tree”**: A dictionary containing information about the rooted tree of pages. This field holds information on how each page connects to one another. The keys are the ID of each page that exists under this chapter, and the values are a list of IDs of the pages that are connected to the key.
- **“links”**: A dictionary with the exact same structure as “tree”, except it only lists the pages that are not only connected, but also linked to the key through a choice snippet

(check chapter *3.1.5.3 Rooted Tree and Links* for a complete explanation). For example, in Figure 43, page “3” is connected to page “1”, but not linked.

3.4.3.5 Pages Collection

The **“pages”** collection keeps track of the pages inside a chapter (Figure 44). The documents in this collection reference a sub-collection called **“comments”**. This means that a page may contain multiple comments.

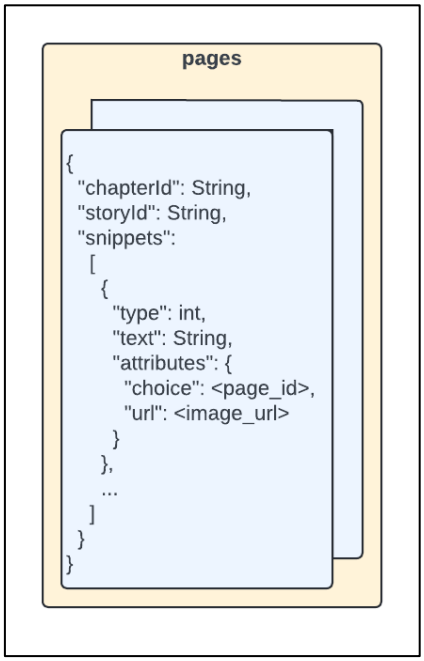


Figure 44 - Pages collection

A description of each field:

- **“chapterId”**: The ID of the chapter this page belongs to.
- **“storyId”**: The ID of the story this page belongs to.
- **“snippets”**: A list of dictionaries, each containing information about a snippet for this page. This list includes snippets of regular text. This list isn’t used in the writer’s context, and is only generated when the chapter is published. These are the snippets that will be rendered in the Chapter Reading screen.
    - **“type”**: The type of snippet. Can be one of three – text (0), image (1) or choice (2).
    - **“text”**: The text portion corresponding to the snippet.
    - **“attributes”**: A dictionary containing optional fields that relate to specific snippet types.

- **“choice”**: The ID of the page this snippet should redirect to, in case of a choice snippet.
- **“url”**: The URL of the image for this snippet, in case of an image snippet.

3.4.3.6 Comments Collection

The **“comments”** collection keeps track of the comments inside a chapter (Figure 45). These are the comments writers can leave each other in a page’s context.

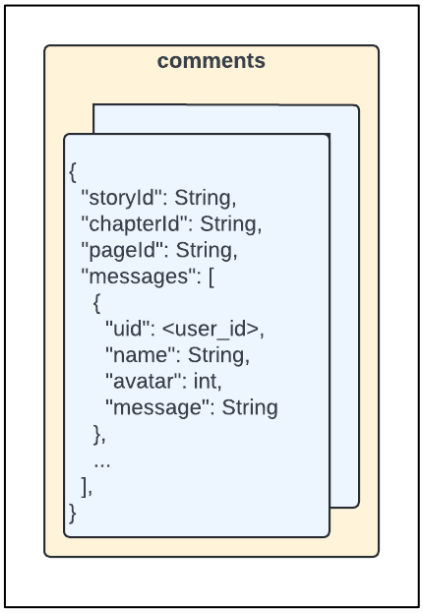


Figure 45 - Comments Collection

A description of each field:

- **“storyId”**: The ID of the story this comment belongs to.
- **“chapterId”**: The ID of the chapter this comment belongs to.
- **“pageId”**: The ID of the page this comment belongs to.
- **“messages”**: A list of dictionaries, each containing information about a message in this comment thread.
    - **“uid”**: The ID of the user who wrote this message.
    - **“name”**: The name of the user.
    - **“avatar”**: The avatar of the user.
    - **“message”**: The message text.

3.4.4 Collaborative Text Editing Queue

The Realtime Database service provided by Firebase structures data in JSON format. TEIA will rely on this database to keep track of changes made to collaborative text editors, inside a page. This database was designated for this purpose due to its efficiency when handling many data writes per unit of time. It is also a good solution for systems that require low latency, which is the case of text collaboration.

The JSON content in Figure 46, illustrates the queue of changes in an example page. Each change is identified by its ID.

```
{
  “stories”: {
    “abc123”: {
      “chapters”: {
        “1”: {
          “pages”: {
            “1”: {
              “changes”: {
                “<change ID>”: {
                  <change content>
                }
              }
            }
          }
        }
      }
    }
}
```

Figure 46 - JSON database for text editor changes

For each change the writer performs in the collaborative text editor from Chapter Editing screen, a new entry in this queue will be created.

At this point, I cannot yet determine the “change content”. In an upcoming chapter, I will describe, in detail, my custom approach to collaborative text editing. Only then, will I be able to define this structure.

3.4.5 Storing Images in the Cloud

An essential service for TEIA that Firebase provides, is Cloud Storage. This service allows the application to upload files of various formats to a cloud space. It also allows the creation of directories (folders) in said space.

This feature is important because writers will upload images to the back-end in two ways:

- Uploading an image from the device's storage to use as an image snippet.
- Generate an image with the Stable Diffusion tool, that is then uploaded all the same.

Cloud storage additionally provides a URL in which an uploaded file is hosted. In the case of TEIA, this facilitates the image rendering, as the UI can simply display an image coming directly from the web.

3.5 Front-end Architecture and Data Modelling

It is also important to establish some ideas about TEIA's user-facing layer, before starting the implementation. This chapter details the front-end architecture, including the navigation system, state management, and the data models that drive the application's core functionality.

3.5.1 Navigation and Routing

The navigation system is a fundamental aspect of any application, especially complex web applications, in which its dynamic URL also changes with the screens.

The requirements for navigation are as follows:

- Each screen should have a different URL.
- Screen URLs should be parametrized, when applicable. For example, the URL for the Group screen for group 1 should be different for group 2.
- Popups don't affect the URL.

In Flutter, there's a package that, among other things, handles the navigation efficiently and facilitates the job of the developer, called GetX (GetX, 2023). GetX is the most famous package in the Flutter ecosystem, and is useful for many things other than navigation.

3.5.2 State Management

Given the asynchronous nature of TEIA, state management is crucial to a good user experience. State management is how an application handles and tracks the state of visual components, data, and user interactions, over time. In the context of front-end development, **state** refers to the status of variables at a given point in time.

Take, for example, the Group screen. It is a dynamic screen, in the sense that its layout changes with the state of the group and other factors. When a user is looking at the screen and the group's state changes, the screen layout should adapt right in front of them.

The Group screen is a complex example. State management refers also to the simplest user interactions. The reason a letter appears in a text field after a user presses a key in the keyboard, is due to state management.

This is where GetX comes in, once more. This powerful package also provides a great way to manage state, by introducing Rx variables. These variables follow the concept of **two-way data binding** from *Angular* and other similar frameworks. Two-way data binding has the following characteristics:

- If the variable changes in the code, the UI is automatically updated to reflect the new value.
- If the user updates the value through the UI, the variable in the code is updated as well.

This a very useful concept that will help TEIA make the UI responsive to changes in the data.

3.5.3 Data Streaming

To enhance the user experience, users should be able to see changes made by other users as quick as possible. Or even changes made by themselves.

When the changes are made client-side only for the local user to see, that is not a problem, since asynchronous requests to the backend are not involved. Client-side state management can provide updates to the interface by itself.

However, when the changes are made to the database, and then broadcasted to all users (including the one who made the change), each client-side application should be able to pick up this change and show it to the user, as soon as possible.

To implement this mechanism, I will integrate Firebase's data streaming services. Firebase provides a Pub/Sub (Publish/Subscribe) service for the collections (or even specific documents) in the database. Pub/Sub is essentially a messaging architectural pattern, much used in distributed systems. It enables the movement of messages between different components of the system, without the components being aware of each other's identity. These systems are comprised of **publishers** and **subscribers**. The **subscribers** subscribe to specific topics and open channel for messages regarding such topic, and **publishers** publish messages to chosen topics. This system is briefly introduced in Figure 47.

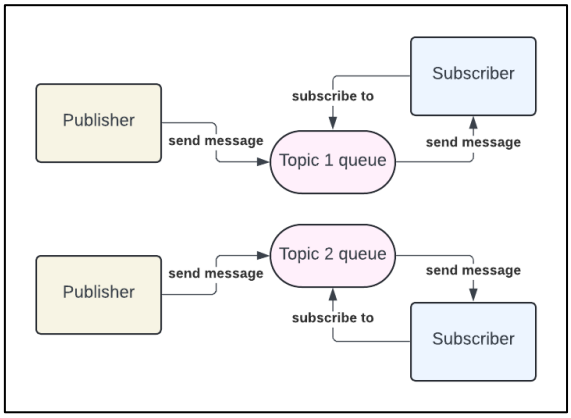


Figure 47 - Pub/Sub messaging architectural pattern

In Firebase's context, the **subscribers** are the client-side application, and will open channels to receive messages containing changes to selected documents or collections. The curious thing, is that the **publishers** are also the client-side applications (the users). They will trigger messages in the backend whenever they make changes to database documents. These messages will then be broadcasted to all **subscribers** that have subscribed to those document, or respective collections. Consequently, the topics are the documents/collections to which the client-side applications subscribe.

Figure 48 shows an example of this interaction, between two users (client-side applications), one document and the Firebase access layer.

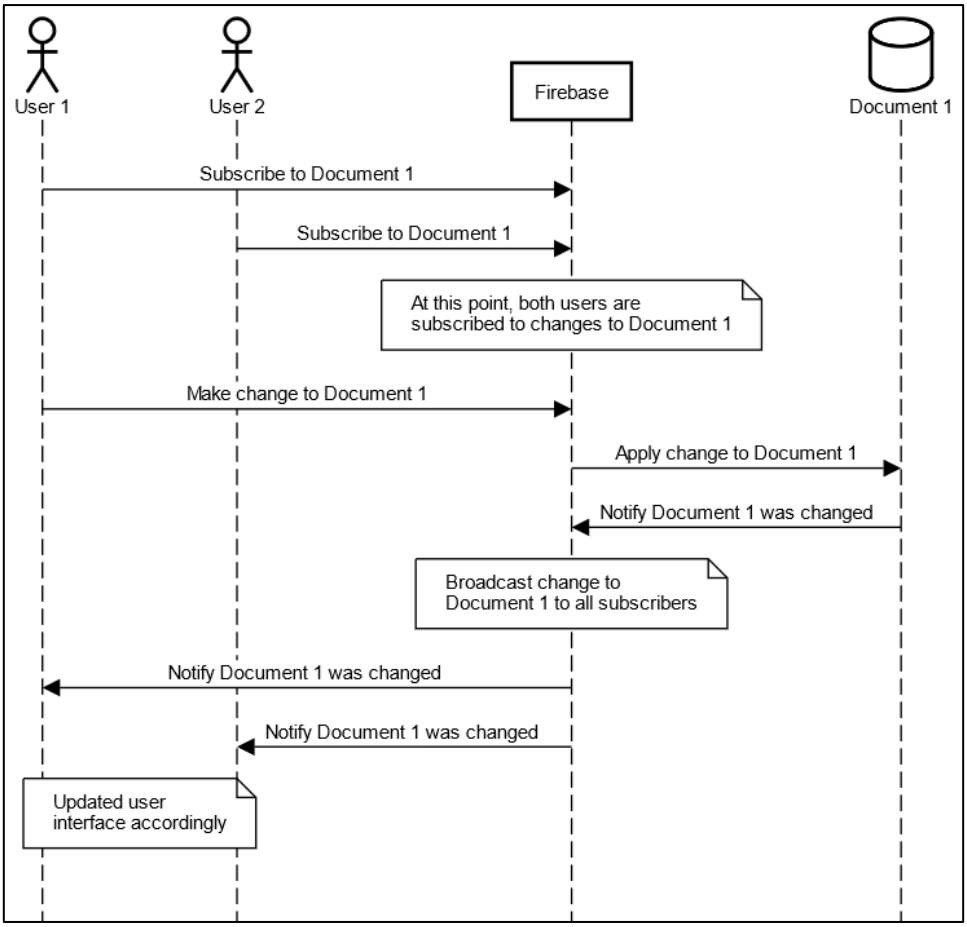


Figure 48 - Sequence diagram for Firebase's data streaming

As seen in Figure 48, both users subscribe to changes to Document 1. When one of them makes a change to it, both will receive the update and update the screen accordingly.

This mechanism allows the application to, for example, subscribe to a group document and keep the group screen updated. When another user (**publisher**) makes a change to this group document (topic), the application (**subscriber**) will receive these changes and update the screen accordingly.

3.5.4 Data Models

The data models in the front-end will be derived from the collections in the back-end, discussed in a previous chapter. This chapter will study the class diagram in Figure 49 and study the decisions that were made while designing it.

The differences between the front-end models and the back-end models are mostly related to how the data will be fetched and parsed, and to the fact that some dictionaries inside some of the back-end collections take place in dedicated classes.

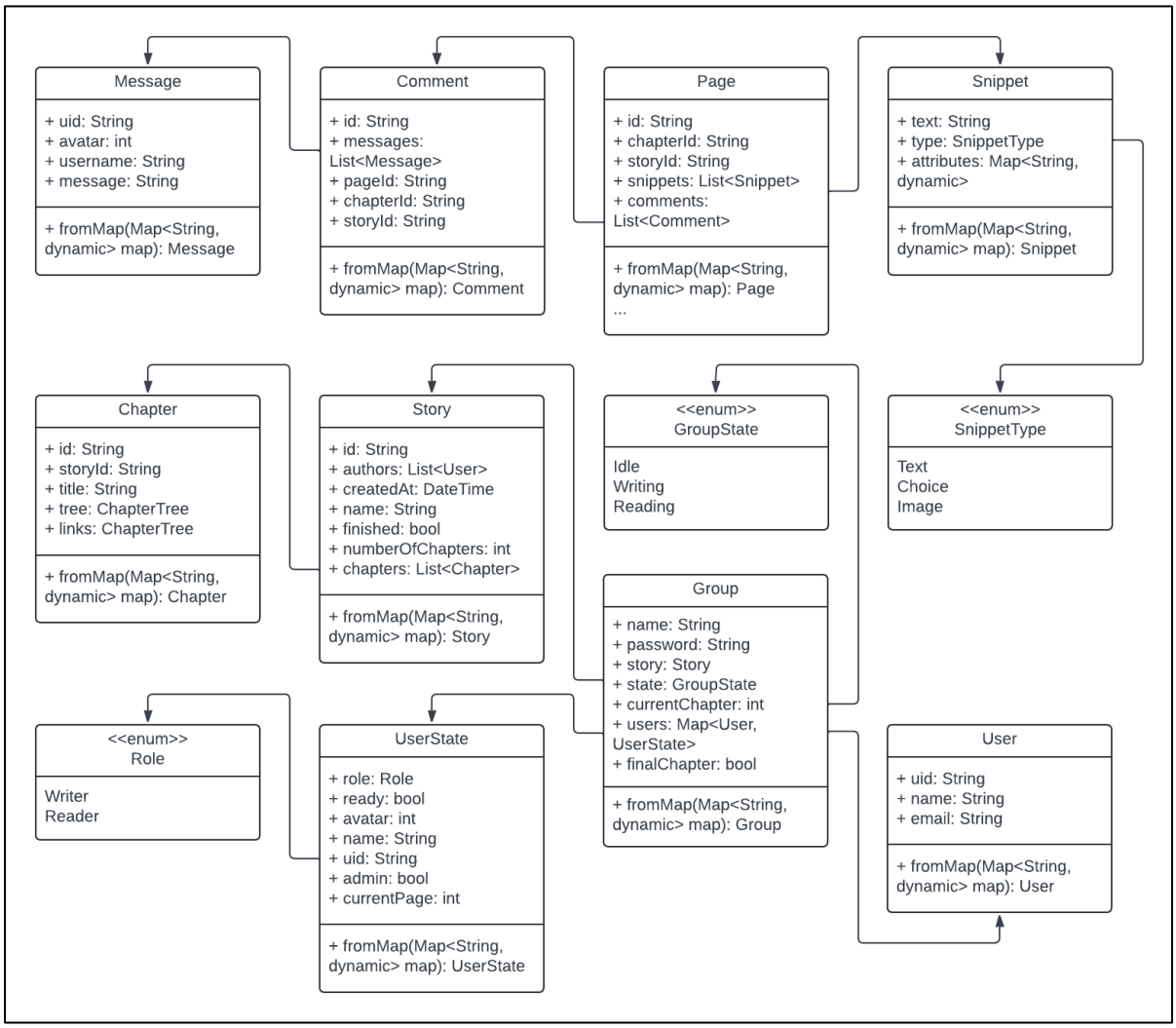


Figure 49 - Class diagram for TEIA

For example, as seen in Figure 49, what was simply a list of dictionaries inside page documents, is now a list of *Snippet* objects. In the back-end data modelling, **"snippets"** inside Page is a JSON list, while in the front-end a custom class was created for *Snippet*.

This happened for *UserState* and *Message* as well. This modulation facilitates data management in the front-end.

Another difference is the concept of Enums. In the database model, enumerations are treated as simple Integers starting at zero. In the frontend, a developer-friendly Enum is used instead, with custom, easy to identify names.

3.6 Design and Architecture Overview

To consolidate the key components regarding requirements, design and architecture, this overview will attempt to summarize the discussion in the previous chapters, and highlight its conclusions and resulting decisions.

Firstly, it was concluded that TEIA must prioritize the delivery of performance and usability. It must prioritize both its visual responsiveness and ease of use. A good user experience is the most relevant quality to aim for.

Then, I ascertained on the concepts of storytelling flow and story structure, which are two core concepts of the app. I decided that decisions the user make in chapter X, will have no consequence in chapters other chapter X itself. An upcoming chapter Y is completely independent, mechanically.

With these concepts in mind, I moved on to the layouts of the screens. All screens turned out relatively simple, except for Chapter Editing Screen. This is the screen where users in general will spend the most time, as it is the content creation screen. It should provide an enjoyable experience. The main requirement for this screen, is that it must display a split-view containing:

- On one side, the rooted-tree diagram, comprised of the connected pages that form the current chapter;
- On the other, a text editor for the currently selected page out of the diagram.

Regarding the infrastructure decisions, as a back-end solution, I'll use Firebase to:

- Store and recover data from a database (No-SQL Collection-Document database);
- Manage cloud files (mostly image files):
- Handle authentication (email and password);
- Send push notifications.

Additionally, I will use Flutter as a front-end development framework. Two very important elements of front-end development to attend to, are navigation and state management. To complement the state management component, I will take advantage of Firebase's Pub/Sub functionality to integrate remote data streaming.

I also designed state machine diagrams, sequence diagrams, UML class diagrams, and other diagrams to assist the implementation phase.

4 Implementation

In the previous chapter, I dissected TEIA's general architecture in detail, and explored some specific important components its infrastructure. Now, it is essential to describe the implementation process, including the creation of the screens and visual components, and the discussion of specific algorithms and mechanisms that are important for the proper functioning of TEIA's prototype.

For the most part, the implementation of the user interface will follow the hand-made layout draft of the screens that was mentioned in the previous chapter. This chapter will exhibit the definite visual aspect of each screen, and detail its respective functionality.

4.1 Managing Groups

The first major screen (after the authentication process) the users will see first, is the home screen. This screen should be welcoming and serve as the gateway to the rest of the app. The home screen is the place where users can:

- See the groups they belong to;
- Create a new group;
- Join a new group;
- Logout;
- Enter a group.

As seen in Figure 50 - Home screen, the screen initially contains only two cards. One to create a new group, and another to join an existing one.

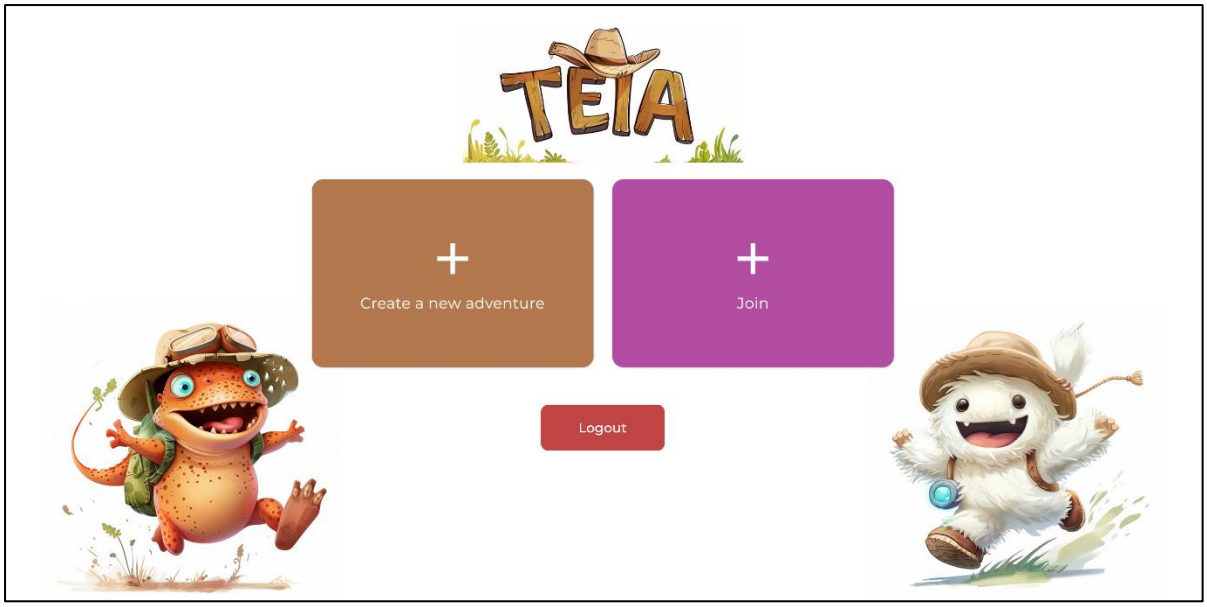


Figure 50 - Home screen

When the user clicks on the “Create a new adventure” card in Figure 50,

State management and data streaming are very important in this screen – they play a fundamental role in keeping the interface updated with database changes.

When the users click logout, TEIA sends them back to the authentication screens. When they click join adventure, it shows the modal in Figure 51.

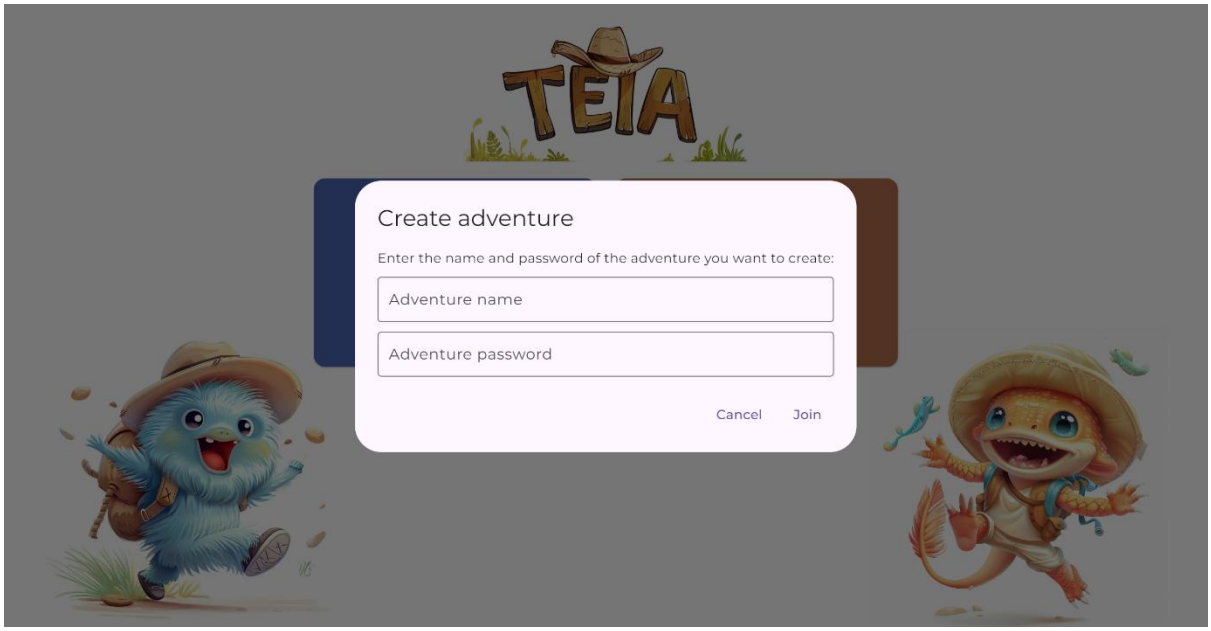


Figure 51 - Group creation modal

Once the users fill the credentials needed to create a group (presented in the modal from Figure 51), it is imperative that the home screen is immediately updated with the new group that they belong to.

Clicking “Join”, TEIA sends a request to create a new Firestore document, under the **“groups”** collection containing the new group’s information. Because the users are subscribed to changes to this collection (this subscription is created in the home screen), Firebase will send the newly created group to the them, even if they were the creator. This allows TEIA to update the list of groups accordingly, and immediately. The state of home screen after this update is as shown in Figure 52.

Figure 52 - Home screen creating a group

Another aspect to note about the screen implementation, is that I opted for a cartoonish and joyful approach to the art. In Figure 50, some art is rendered into the home screen, including TEIA’s logo. A reoccurring visual component is the explorer hat, remembering the users that this is an adventure game – a reading game, with some exploration features.

> ***Note:*** *All the art for TEIA’s screens was generated with Midjourney, an AI image generator.*

4.2 Group State

Once the users enter the group from the home page, TEIA will show them the group screen, drafted in Figure 31. This screen must:

- Display the group name;
- Display the list of users in the group (independently of which state the group is in);
- Describe the state of the group, and what is currently required from each user;
- Allow user to take actions related to what is expected of them, based on the state of the group (mainly through buttons that take them to other screens).

In Figure 53, the group screen (in idle state) shows, on the left, the users list. In this case only one user has joined – the creator of the group. On the right, the user can assign a story to this group, by creating one, or reading one that already exists. The feature that enables assigning existing stories to the group is not implemented yet, as of the time this document is being written.

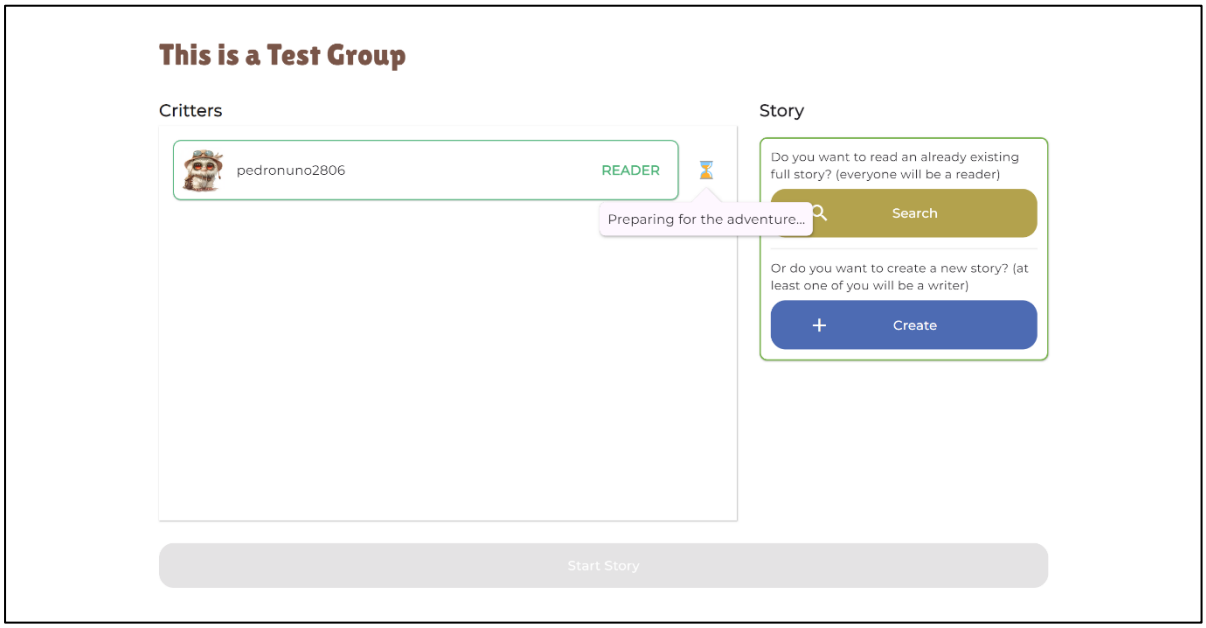


Figure 53 - Group screen for the idle state

Inside the list of users, each row (representing a user) contains an avatar, a name, the role (writer or reader) and the status. As seen in Figure 53, when hovering the status icon, TEIA will show a succinct description of what the user is currently doing, in the form of a tool tip[7].

[7] A tool tip is a user interface element that appears when hovering over a screen element. It is a text box that displays information about the element.

When clicking on the row that displays their own user details, a modal (Figure 54) appears and the user may edit:

- The avatar (select one of 25 available ones);
- The name that other users in this group will visualize;
- The role.

Of course, each user can only modify details regarding their own information. Clicking other users' rows will not open the modal shown in Figure 54.

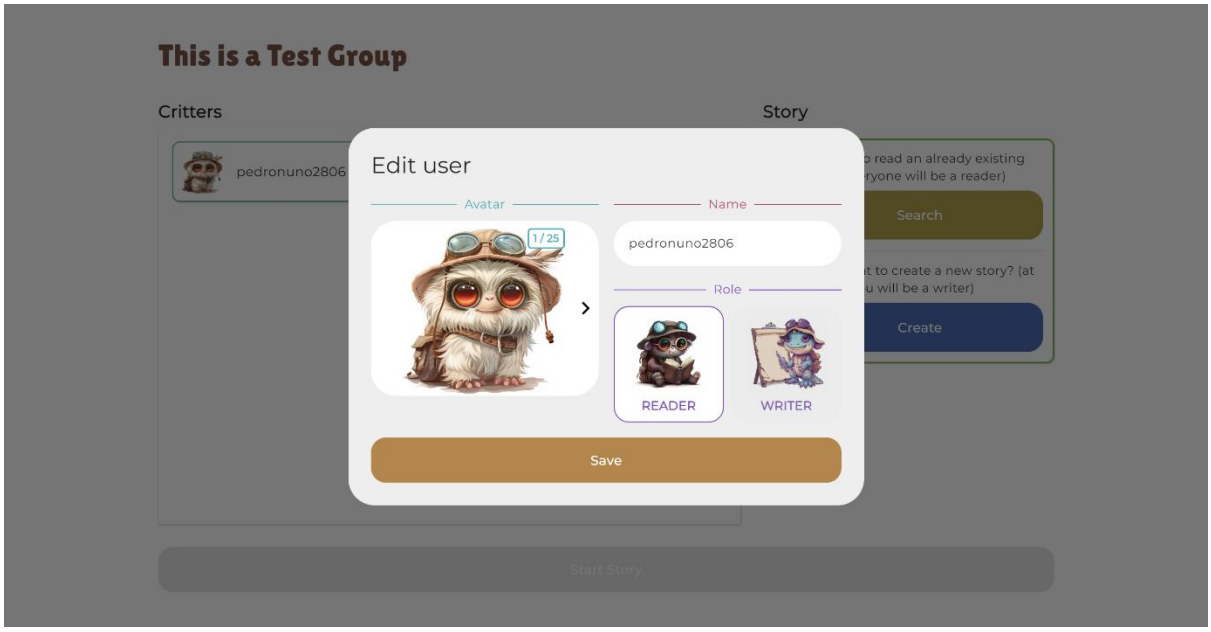


Figure 54 – User edition modal

Once the user clicks "Save", their new information is saved, and instantly displayed in the group screen, for all users that are currently viewing this screen.

As soon as there is at least one writer and one reader, and there is a story selected, the button "Start story" unlocks, and the admin may start the session .

When the admin starts the session, the layout of the group screen changes to all users, according to the role of the user. For writers, the option to edit the first chapter becomes available – a button to navigate to the chapter editor screen appears on the right, as shown by Figure 56.

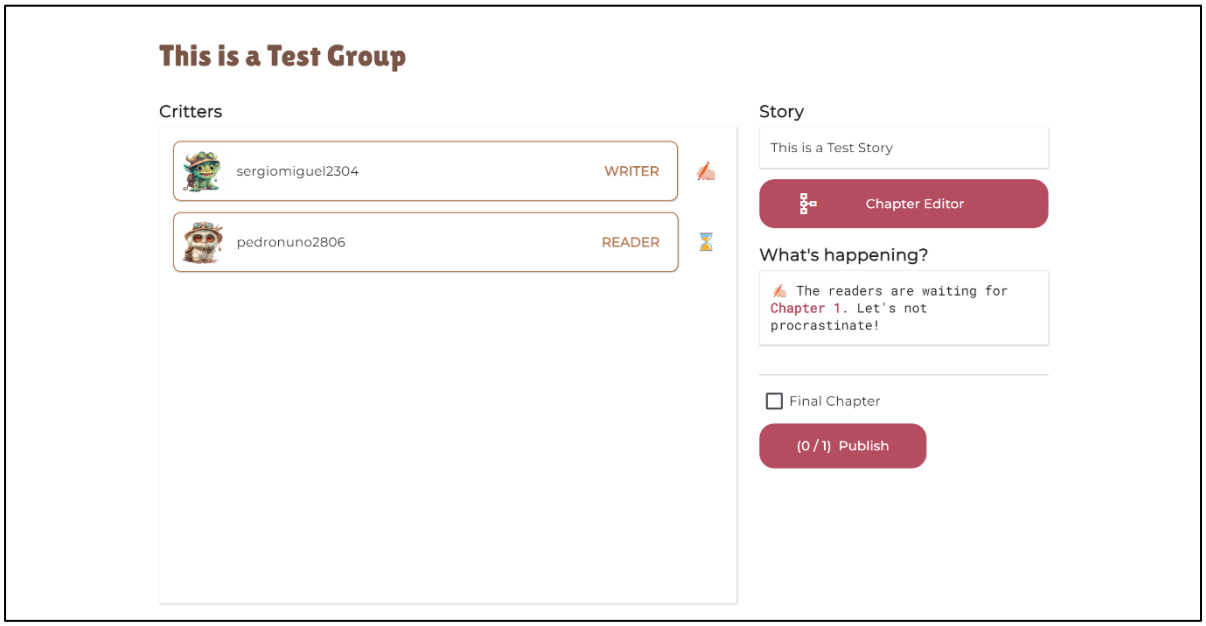


Figure 55 - Group screen in writing state, for the writer

Also, note in Figure 56 that the layout also includes a check box labelled as “Final Chapter”, to let writers select the current chapter as the final chapter of the story, before publishing. This would mean no further chapters would succeed this one. It also includes a button to publish the chapter. All writers need to press it to make the chapter public, and make the group transition to the next state.

Meanwhile, the reader must wait for the first chapter to be written by the writers. Only then will the option to read the chapter become available.

> ***Note:*** *The icon (and its tool tip) following each row in the users list changes with the actions of the respective user and with the state of the group. It is always updated with useful information about each user.*

4.3 Chapter Editor

As the most important screen in the application, the chapter editing screen contains two fundamental components for the content creation side of TEIA. The chapter editor – the tree-like interactive structure of pages, and the page editor – the collaborative text editor for a

Figure 56 - Chapter editing screen with page editor closed

specific page. By default, page editor is closed, as seen in Figure 57.

To open the page editor, the writer must click on one of the pages in the chapter editor. In the case of Figure 57, only page 1 is created, and cannot be deleted. The page editor will be covered in the next chapter.

As seen in Figure 57, other than the rooted-tree structure, the remaining visual components are:

- The index of the chapter – in this case, chapter 1;
- The title of the chapter, editable – in this case, “New chapter”.

> ***Note:*** *The term “rooted-tree” comes from the fact that the chapter structure always has a root page in the beginning. Only one, not less, not more.*

If the writer clicks the button to add a child page to page 1, the layout resolves what Figure 58 illustrates. Two new pages are created and page one now displays a warning icon indicating unlinked pages.

As previously explained, unlinked pages are pages that are connected in the rooted-tree, but are not connected via snippets in the text. In this case, it means that page 1 does not contain a choice snippet that would send the reader to page 2 or page 3.

Figure 57 - Chapter editing screen with 3 pages and with the page editor closed

The writers may create as many pages as they want, but they must be aware of two things:

- For all pages, all the respective direct children pages must be linked through choice snippets;
- The more branches the chapter includes, the harder it will be to start the next chapter. The reason being that all readers, regardless of what decisions they made, will start on chapter 1 (the root) of the next chapter.

The chapter editor is very easy to use, and provides the writers with a good amount of freedom to create interesting chapter structures.

4.4 Page Editor

In the previous chapter, I addressed the chapter editor component of TEIA. The page editor, together with the chapter editor, constitute the chapter editing screen – the screen that allows writers to create full chapters for readers to enjoy.

When clicking a page in the rooted-tree chapter editor, the page editor opens on the right, as seen in Figure 59. The writer can then close this view by clicking on the arrow that is positioned on the left end.

The page editor Is comprised of three visual components – the text editor on the left, the title section on top, and the extras column on the right. The extras column provides some important features, that are not all displayed in Figure 59 (more on this later). The two features displayed are:

- **Comment creation –** for writers to leave comment to each other, and discuss ideas or changes to the page's content.
- **AI text generation tool –** an upcoming chapter will discuss this tool in further detail;

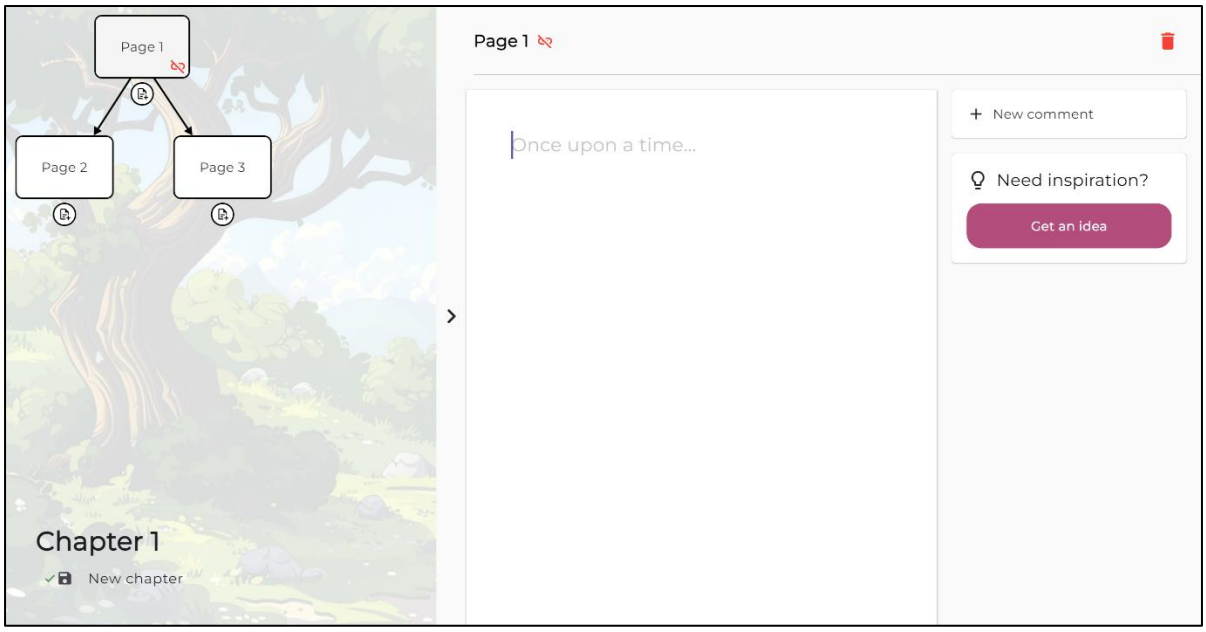


Figure 58 - Chapter editing screen with the page editor open

The snippet addition features only become available when the text editor effectively contains text. If the writer selects a portion of text, the options appear between the text editor and the extras column, as shown by Figure 60.

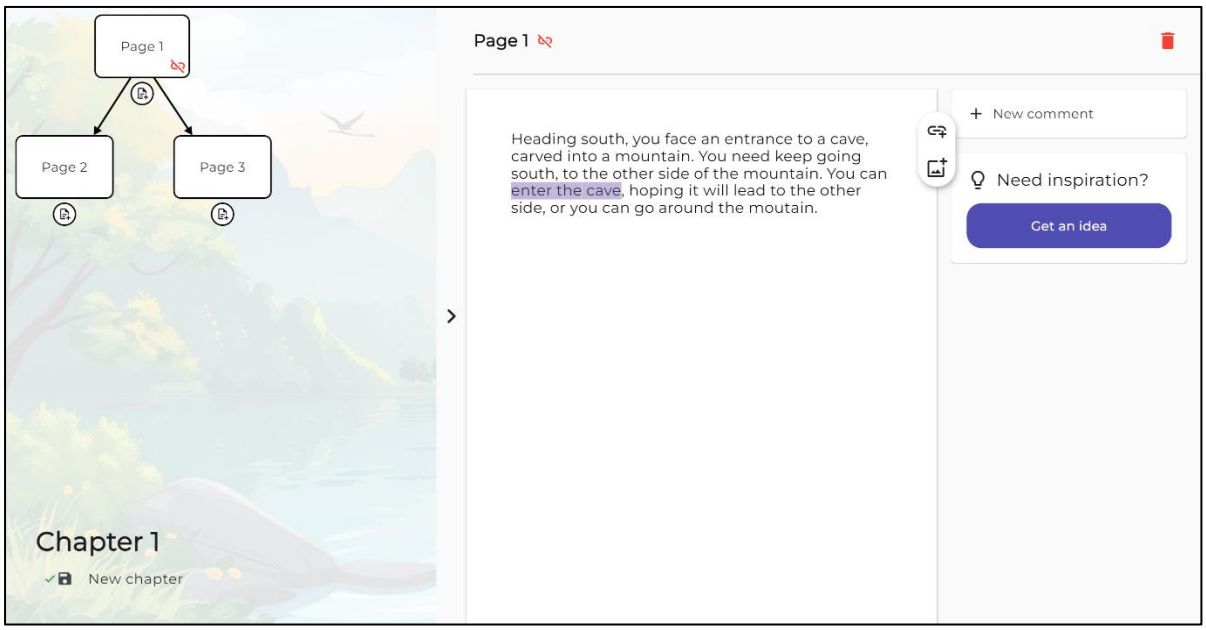


Figure 59 - Chapter editing page when selecting text

The snippet options contain two buttons – one to create a choice snippet and another to create an image snippet.

To create a choice snippet (Figure 61), the writer has two options:

- Link a currently connected page (**"1"** or **"2"**, because those are the children pages of page 1) to the selected portion of text. The numbers in red symbolize the pages that are currently unlinked and must, therefore, be linked. If one of the numbers is green, it means that page is already linked;
- Create a new connection in the chapter editor (**"+"**), and link the selected portion of text to the newly created page.

Figure 60 - Choice snippet options

To create an image snippet (Figure 62), the writer, again, has two options:

- Upload an image from the device, and link it to the selected portion of text;
- Use the AI image generation tool (more on this in an upcoming chapter), and link the resulting image to the selected portion of text.

Figure 61 - Image snippet options

After a snippet is created, when placing the cursor inside the portion that was linked to the snippet, the writer may check the snippet information, as seen in Figure 63.

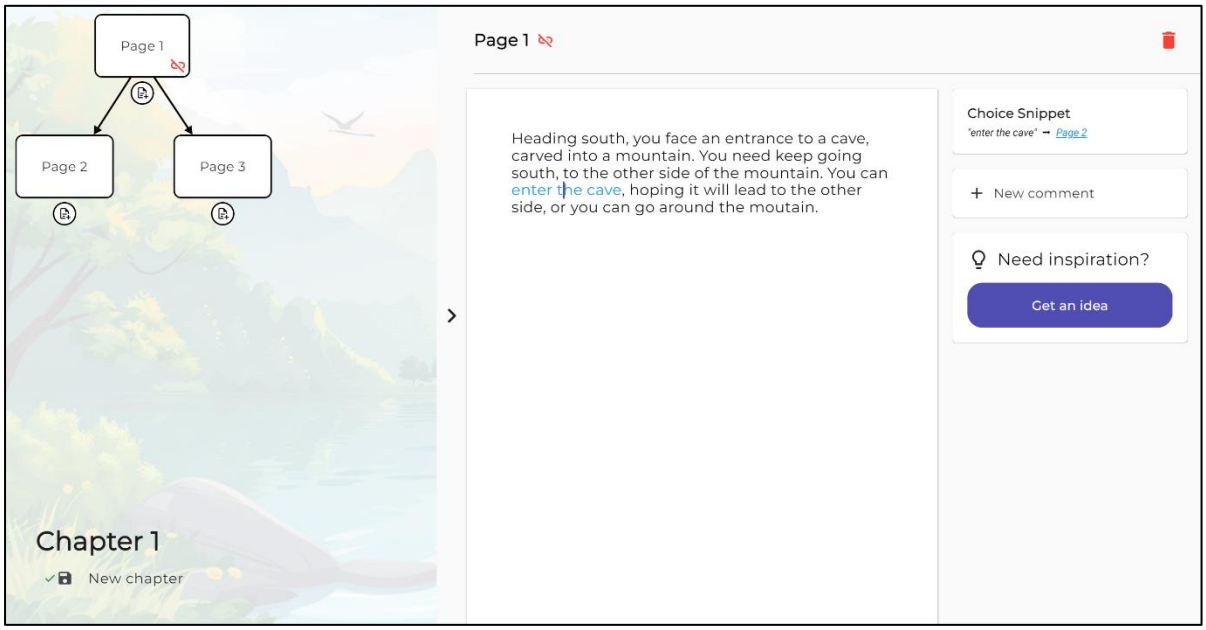


Figure 62 - Chapter editing screen when cursor is place inside snippet

The page editor is a crucial component of TEIA, for the many reasons mentioned. However, the requirements stated that multiple writers could work on the same page, simultaneously. For that, TEIA must ensure a conflict-free, intent-preserving, document-converging collaborative experience, when it comes to the text editor. The next chapter will address the solutions adopted to address this problem.

4.5 Text Editing Collaboration

By **text editing collaboration**, I mean the ability for writers to collaborate in a user-friendly environment, writing and contributing to the story they are working on. In this chapter, I will focus on the writing element alone, and as such, the inevitable conflicts and challenges that come with an otherwise simple real-time collaborative text editor.

Real-time collaborative text editors are, in essence, text editors that can be edited by multiple users, simultaneously, and, in this case, remotely.

In practice, each user will have a local copy of the document (the editor content) in their device, but a collaborative text editor must ensure the users are always in sync. That is, ensure all users are always looking at the same document, no matter the changes they make, and in what order. In other words, each local copy of the document should converge to the same state.

A collaborative text editor must also preserve the intention of the user. If a user intends to delete a specific character, but a different one is deleted, the text editor is not reliable.

The Conclave Team wrote a great article about their Conclave case study, a collaborative text editor in JavaScript (Conclave, A Private and Secure Real-time Collaborative Text Editor, 2023). This article lists most of the challenges any real-time collaborative text editor inherently faces and explains them very well.

To ensure document convergence and preservation of intent, whatever solution I set out to use will have to abide by the following rules, regarding the operations performed by users:

- Commutativity
- Idempotency

Commutativity occurs when operations applied in different orders produce the same result. For example, addition is commutative because:

$$A + B = B + A$$

Subtraction, however, does not share the same feature:

$$A - B \neq B - A$$

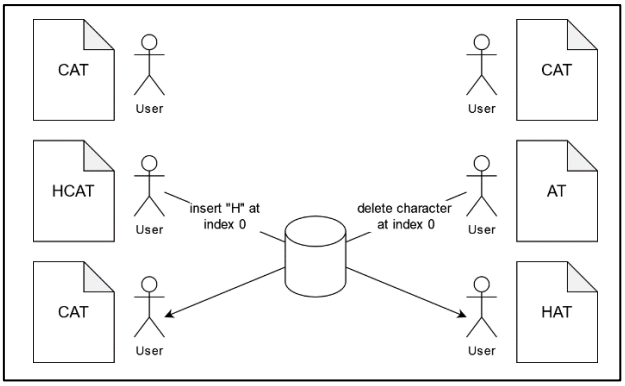


Figure 63 - Sequence diagram representing the absence of commutativity

The operations performed by users must be commutative so that, no matter what order they get processed, the result will be the very same.

In the example of Figure 64, the user on the left ends up with a different document when compared to the user on the right. That happens because the operations are not commutative.

Idempotency, in turn, occurs when repeated operations produce the same result. As an example, multiplying by 1 is an idempotent operation. Multiplying a number by 1 multiple times, will always result in that very same number. A collaborative text editor must ensure all the users perform idempotent operations, so that repeated operations may happen, without causing conflicts. Figure 65 shows an example where a non-idempotent solution generates a conflict, when two different users perform a delete operation on the same index. While it is true the document converged to the same state, the user's intent was not preserved. Both users intended to delete character **"C"**, but character **"A"** was also deleted.

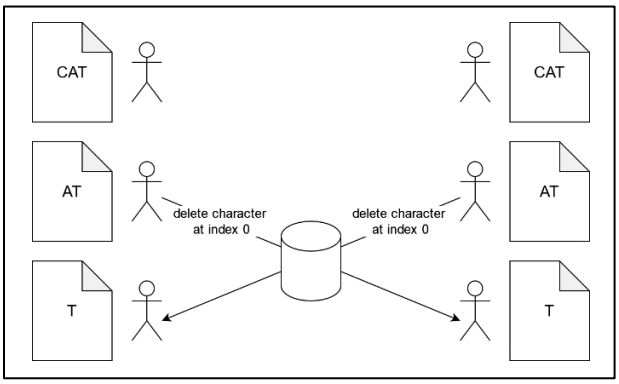


Figure 64 - Sequence diagram representing the absence of idempotency

4.5.1 The Firebase-only Approach

For the first approach, I will set the Deltas and Quill document aside, and work exclusively with pure text data (Strings) and a general text editing Flutter widget (*TextField*). Firebase provides a data streaming feature, that enables an application to listen to collection/document changes, triggering custom client actions.

In this case, it would make sense to have one single copy of the document stored in Firebase, and have it streamed across all users. The users would still have their local copy, but upon any operations, Firebase notifies all listeners that a change has been made. Virtually, there is only on copy of the document. Firebase ensures each user has the updated version.

But does that mean this is a reliable approach? Not at all. Even though commutativity is not needed (because all users always have the same document, ensuring convergence), their intention is not always preserved.

Due to the nature of Firebase's no-code backend, when a user performs a change, the whole document (with the change already made) will be uploaded to the Firebase database, overriding the one already there. Figure 66 shows that procedure.

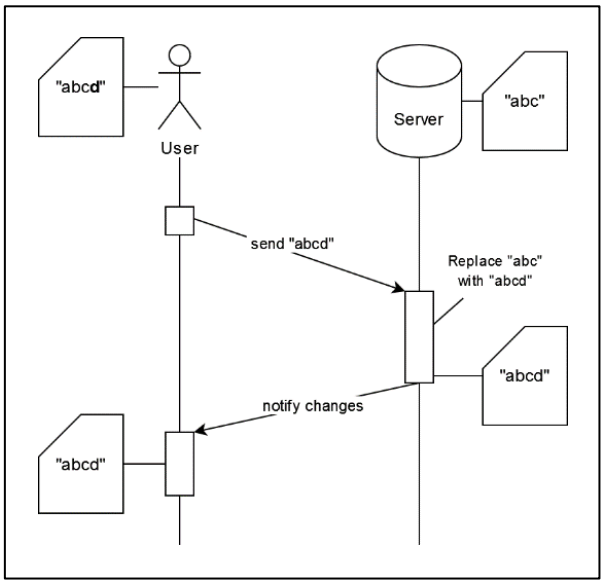


Figure 65 - Sequence diagram describing the Firebase-only approach to collaboration

The user first performs the change on his local copy of the document. Then sends it to Firebase, so that other users are notified of the change. The problem here is that operations are not idempotent.

Figure 66 illustrates this mechanism, where a user interacts with Firebase's service.

4.5.1.1 Problem 1 – Self-Override

The first problem with this approach did not even require a second user interacting with the system. I noticed this when I tried to write incredibly fast. Figure 67 shows how two operations performed right after one another, by the same user, can cause what I call **"self-override"**.

In Figure 67, the user inserts "e" at the end of the document, and right after that, removes it. Those two updates are sent to Firebase that, in turn, broadcasts them to all users.

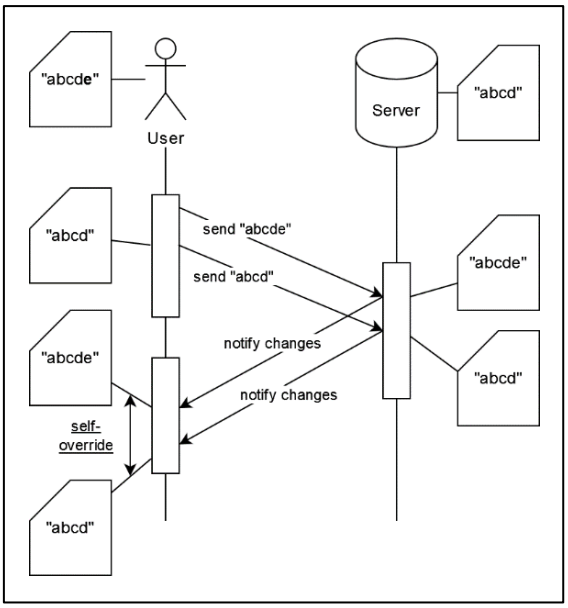


Figure 66 - Sequence diagram illustrating the self-override problem of the Firebase-only approach

Because the change is performed locally first (on the *TextField* widget, client-side), the user sees the changes instantly. However, he will also see the updates coming from Firebase. This means, because the operations happened so close together, the local copy of the document will assume incoherent states, even though its final state is correct.

In extreme cases, not only would it show inconsistent states of the document, but it also wouldn't preserve the intention of the user.

To solve this, each user should ignore all updates that come from themselves, avoiding these incoherent document states. The changes are performed locally, so they don't need to be performed again.

4.5.1.2 Problem 2 – Remote-Override

Convergence could be assured because of Firebase streaming. But now, because of the solution to the previous problem, even that is no longer granted. If we take the previous problem, and transcribe it to a scenario with two users, we obtain the problem described by Figure 68.

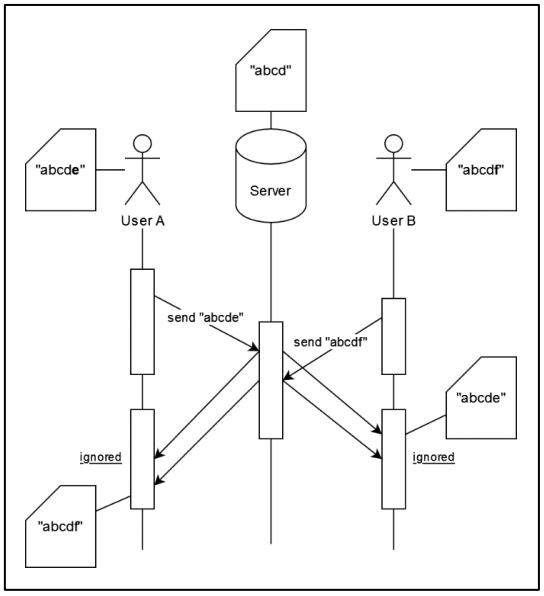


Figure 67 - Sequence diagram illustrating the remote-override problem with the Firebase-only approach

Both users start with the document **"abcd"**, as well as Firebase. User A appends a **"e"**, and User B appends a **"f"**.

The problem happens because these operations happen so close together, that each user had not yet received the update about one another. So, when User B sends the document **"abcdf"**, he didn't know of the upcoming **"abcde"**, that was on its way.

This scenario ends up with the first update to reach Firebase (in this case, the update from User A) being overridden.

In Figure 68, because I'm using the solution to the **"self-override"** problem, not only does the document not preserve the users' intent, it doesn't even converge as well.

One might think: "These are very specific corner-cases that have a very slight chance of happening". The reason for such thought is understandable - when compared to the time taken by an operation to reach Firebase, the time between human interactions is much longer. But in a scenario like this, where multiple users will be constantly, rapidly writing at the same time

(and in different parts of the document), these types of conflicts are bound to happen, and are highly inconvenient.

4.5.2 The CRDT Approach

When addressing the state-of-the-art collaborative text editing solutions, I mentioned CRDT as a possible solution. Unfortunately, I cannot fix the previous Firebase-only Approach by just throwing CRDT at it. Let us look at what is happening in Figure 69. Three users start a session with a document containing **"CAT"**.

- **"C"** is the index 0, **"A"** index 1, and **"T"** index 2.
- The *delete* operation is symbolized as *remove(index, length)*.
- The *insert* operation is symbolized as *insert(text, index)*.

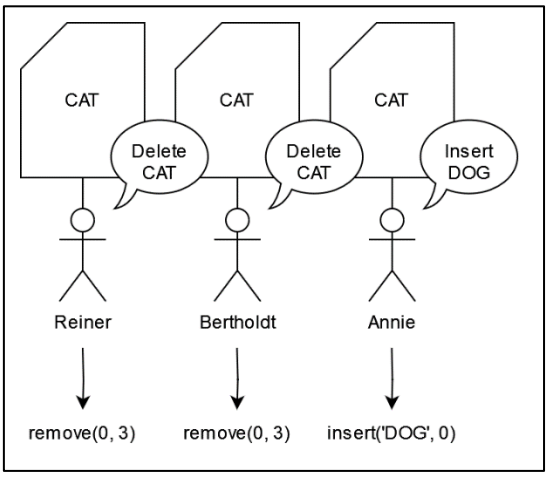


Figure 68 – First iteration of a scenario where 3 users make simultaneous changes to a document

Reiner wants to delete from index 0 to 2, as well as Bertholdt. Annie wants to insert another **"DOG"** at the beginning. Because the three operations are happening concurrently, the order at which the operations will arrive each user is volatile, and not predictable.

One potential outcome is illustrated in Figure 70. Reiner receives Bertholdt's operation first than Annie's, and the resulting document is **"DOG"**. However, Bertholdt receives Annie's operation first than Reiner's, and the resulting document is empty.

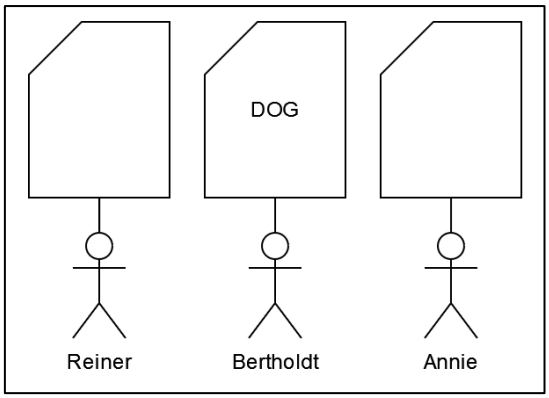


Figure 69 - Example outcome to the scenario in Figure 69

This happens because the operations are not idempotent, nor commutative. The way CRDTs solve these conflicts is by attributing an identifier to each character of the document. If each character has a unique identifier:

- A *delete* operation deletes the exact characters intended because they are uniquely identified.
- An *insert* operation inserts text at the exact index intended, because the character before and after are uniquely identified.

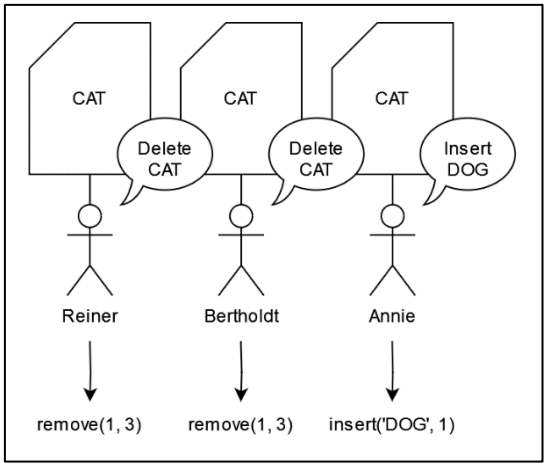


Figure 70 – Second iteration of a scenario where 3 users make simultaneous changes to a document

For that, I need to rethink the operations' parameters. Instead of using indexes, I must use unique identifiers:

- **"C"** has id 1, **"A"** id 2, and **"T"** id 3.
- The *delete* operation is symbolized as *remove(id, length)*.
- The *insert* operation is symbolized as *insert(text, id)*. The text is inserted after the existent character with *id*.

With this new configuration, I re-simulated the same scenario in Figure 71. The users perform the exact same operations, but as seen in Figure 72, the outcome is different. The documents converged and the users' intent was preserved.

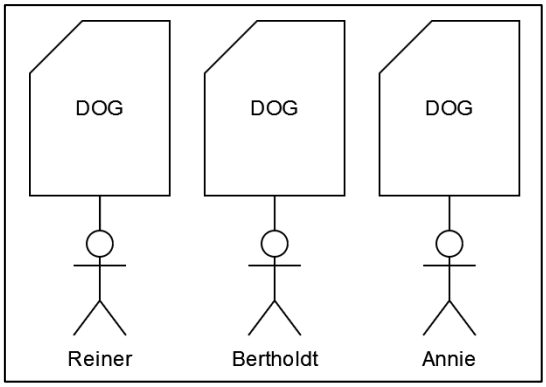


Figure 71 - Outcome to the scenario in Figure 71

CRDT made the operations commutative and idempotent, just by changing the way the document is structured.

4.5.3 The CRDT-Delta Approach

The remote collaboration in the previous approach works, but only if we can gather the information about *insert* operations and *delete* operations, which is not possible with a regular *TextField*. With the Flutter widget *TextField* that I am using now, I only have access to an **"onChanged"** callback, that gives me the entirety of the text in the editor, every time it is changed. However, I want to get the changes only, not the full text. For that, I need a different, more powerful text editor.

TEIA will serve a text editor in the front-end client-side application. I will use a community-made text editor that supports Quill Documents and Deltas (Delta, Rich Text Editor), called FlutterQuill (FlutterQuill, Published in 2021). The Quill Editor supports Delta Quill and has many handy features. By managing the editor content in a Delta fashion, not only it allows for CRDT integration, but it allows text formatting, which will be very helpful for snippets.

4.5.3.1 CRDT-Deltas

The next step is to find a way to convert Deltas to what I am going to call **CRDT-Deltas**. This needs to happen when the user performs a local change, on the local change callback of Quill Editor, as indicated by Figure 73. The opposite will need to happen on the remote change callback, when the user receives a CRDT Delta from the other users.

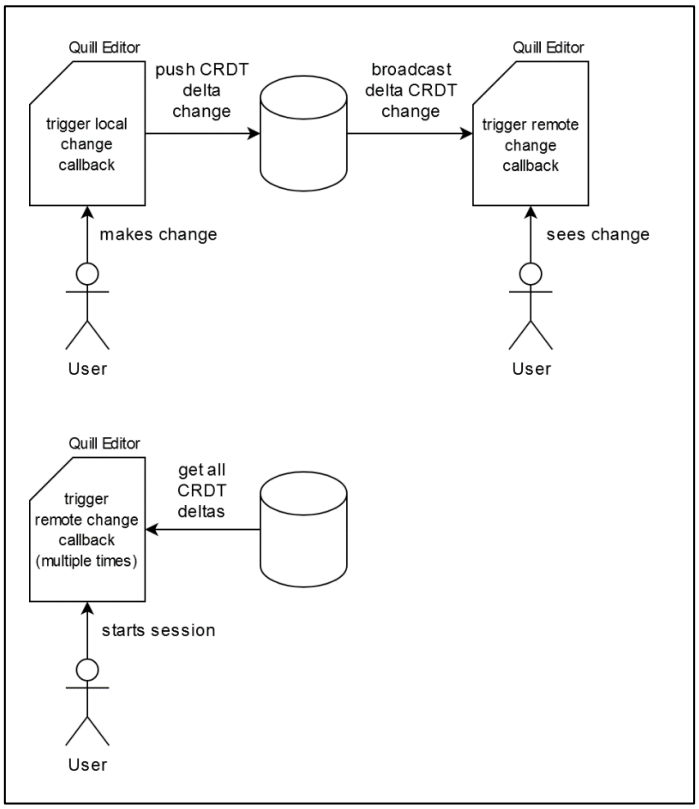


Figure 72 - Diagram describing the Delta-CRDT architecture

The way this is done, is simply by replacing indexes with CRDT unique identifiers. That means each user will have a CRDT structured document hidden behind the scenes, so that these callbacks can keep track of the identifiers.

However, I stand before yet another problem. How do I generate the unique identifiers upon insertion?

4.5.3.2 Generation of Unique Identifiers

The CRDT solution is based on the guarantee that each character of the document has a unique identifier, and that they are always sorted. However, the generation of identifiers can quickly become a very complex task. Let us assume our identifiers are integers, and that the document starts with the content: **"ABD"**. The identifiers are, respectively, 1, 2 and 3. When a user inserts a **"C"** after **"B"**, a new identifier between 2 and 3 needs to be generated. In the set of integers that is not possible.

A simple, short-term solution is to use a set where there are infinite possible numbers between any two numbers. Take the set of real numbers, for example. If we need to generate an identifier between 2 and 3, 2.5 is a possibility. And if we need to generate an identifier between 2.5 and 3, 2.75 is a possibility. And so on, infinitely.

However, while this solution is viable in reality, the virtual representation of real number (a *double*) is limited by its size in bits. There are only so many possible decimal places. Not only that, but the double precision is also limited. Depending on the programming language, after a certain number of decimal places, the calculations are not fully precise (due to rounding).

For a more reliable long-term solution, I will use the method explained in Figure 74. By using a list of integers instead of a double, we overcome the problem of double imprecision, as well as the problem of having a limit of decimal places. Now, each array position represents a decimal place.

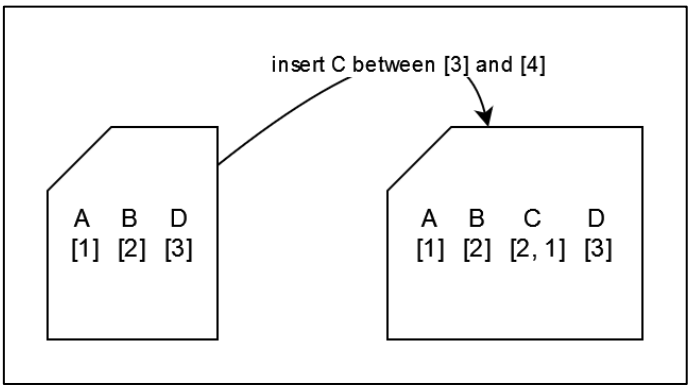


Figure 73 - Method for generating unique sequential character identifiers

The trade-off is that the size of each character identifier (the size of each list) will increase in size, depending on where in the character is being inserted. In other words, it's not just a double with a fixed bit-length anymore, it's a dynamically sized array. For the context of Teia, however, this is not a big problem since the documents won't be very long.

There are some published papers describing mechanisms that optimize the generation of identifiers for specific contexts. I will address the possible improvements in an upcoming chapter.

4.5.4 Delta-CRDT Change Model

For the Delta-CRDT approach to work, I'll need to determine a data structure in which changes will take form. Figure 75 is the class diagram for the models involved in this approach.

Snippet is already defined in Figure 49. **Letter** is the representation of each individual characters or succession of characters in the document. A letter points (or not) to a snippet. **LetterId** is the unique identifier of a character in the document – it is essentially a list of integers, as discussed in the previous chapter.

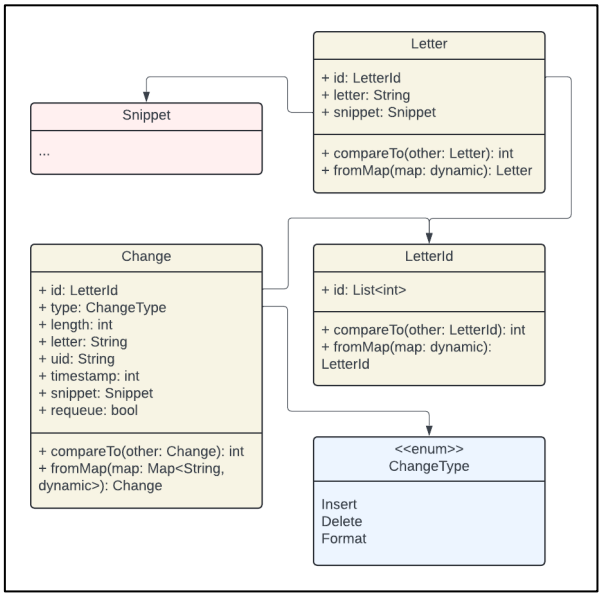


Figure 74 - Change data model (class diagram)

The **Change** model contains a *timestamp* and a *uid* field. These indicate which user made this change and when (milliseconds since EPOCH[8]). It contains many other fields, but only some are relevant, depending on the change type (defined by *type*):

- For the **Insert** type, *id* refers to the character after which the insert takes place, and *letter* is the text to insert. Most of the time, a single character, but it can contain more than one (for example in the case of pasting from the clipboard).

[8] Unix EPOCH, 00:00:00 UTC on 1 January 1970

- For the **Delete** type, *id* refers to the character after which the deletion will begin, and the *length* indicates how many letters should be deleted.
- For the **Format** type, *id* refers to the character after which the formatting will begin, the *length* indicates how many letters to format, and the *snippet* indicates which snippet this segment should be formatted with.

> ***Note:*** *The format change is how the writers will integrate snippets into the chapter pages.*

Only one field remains – *requeue*. This field is part of the solution to one of the problems that the Delta-CRDT approach struggles with, called the requeue mechanism. More on this on the next chapter.

4.5.5 Problems With the Delta-CRDT Approach

Although the Delta-CRDT solution works very well, it still presents some two major challenges, both related to the CRDT nature of this approach.

4.5.5.1 Incremental Change Queue

As I was trialling the approach, I quickly realized how populated the change queue would become. As the writers create content inside a single page, each single character insert and character delete will be represented by a single change instance. I will explain why this is such a serious issue.

Firstly, let us picture the following scenario:

1) The writer creates an empty page;
2) Then, writes a sentence of **50** characters;
3) Then, closes the page editor;
4) Then, reopens the page editor.

At stage 4), TEIA will have to fetch from Realtime Database **50** or more changes instances, to get the current state of the document. As the document evolves, this number will only increment, contributing to a proportionally increase of page editor loading time. The more changes TEIA needs to fetch, the more time it will take the page editor to load the document.

If the writer adds 10 more sentences like the previous, the queue escalates to **500** change instances, not counting any changes of type Delete (which is unrealistic – this number would be even larger in reality).

A possible solution to this considerable problem is to keep the queue change organized. As mentioned, the field *letter* in the Change model, may contain text, not just a single character. If I could minimize the amount of change instances in the queue by merging them together, the problem would be severely attenuated. I will call this solution the **requeuing process**.

The best place to trigger this algorithm is when the writer closes the page editor – TEIA will gather all the changes in the queue and merge them to the least total amount possible, replacing everything in the queue. This is why the requeue field is important. When this algorithm is triggered, it is important to notify other users that these new change instances are the result of a requeuing process, not regular change instances.

Figure shows the example of a **requeuing process** and its result.

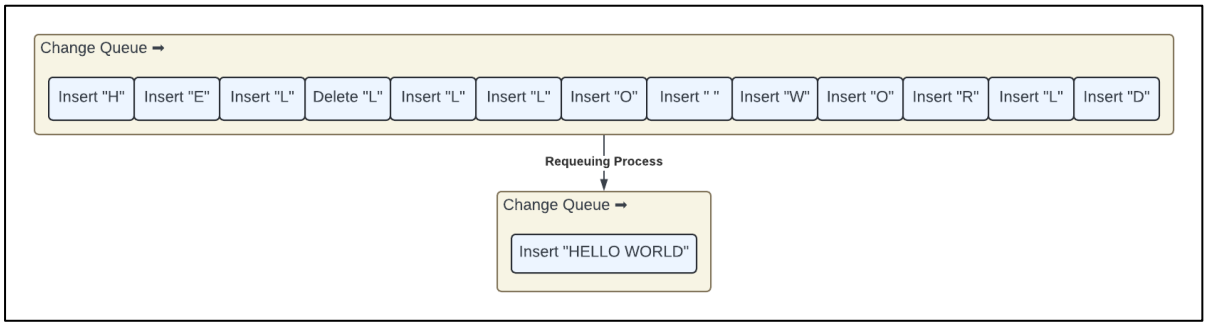


Figure 75 - Requeuing process

Even for such a simple sentence like “HELLO WORLD”, the requeuing process already does a great job at simplifying the queue. Because of Deltas’ optimization and simplicity, the result of the requeuing process is always one single change instance, plus format-type change instances.

This requeuing process effectively addresses the issue of an overloaded change queue by significantly reducing the number of change instances. As a result, it ensures that TEIA can handle large change queues without a proportional increase in load times, fully solving the performance problem described.

> ***Note:*** *This solution fixes the change queue overhead, but this is not the only problem causing extended load times. In the next chapter, I introduce the other problem.*

4.5.5.2 Text Blocks Overload

At first glance, the Delta-CRDT architecture should not struggle with pasting large blocks of text, since it would just create a single change instance containing all the text. However, the problem actually lies in the local tracking of each *Letter* individually, to allow TEIA to properly handle remote changes.

Even though the change instance sent to Firebase Realtime Messaging is apparently simple, once it arrives at the client-side application to be rendered by the Quill text editor, each and every single character will be processed one at a time, causing an excessively large loading time for large documents.

This problem requires an architecture redesign – there is no easy solution. However, in the context of TEIA, where documents are not intended to be too large, its impact is limited. While present, the problem is not critical and has a minimal effect on the user experience – unless writers dare creating excessively large documents.

4.6 Generative AI Tools

Now that TEIA features a content creation screen, with an interactive chapter editor and a functioning collaborative text editor, the only components missing in the writers' environment are the **Generative AI Tools**. These tools are an added-value to the content creation experience, as they provide ways to stimulate the writers' creativity. The two types of tools I will include in TEIA are:

- **Text generation tool –** a narrative-completion feature that will provide writers with ideas on how to finish (or even write from scratch) a specific page, based on the full context of the pages before.
- **Image generation tool** – a *text-to-image* feature that converts the image snippet text to an image. This image can then be triggered by readers during their reading session, by clicking on the respective text portion of the page.

4.6.1 Text Generation

This is the AI tool that is the most useful to the writers. It is a powerful, fast LPU (Language Processing Unit) provided by the Groq (Groq API, 2016).

While it is true that this tool can generate complete chapters, this approach to content creation (with full dependence of the AI tool) would result in boring, generic narratives. The tool relies on the writers' creativity, ideas and world-building.

This tool works especially well when the chapter already has some context and some pages.

In the example of Figure 77, the chapter currently contains 8 pages. In **Page 8**, the writers intend to use the text generation tool. To gather the context needed for such a task, TEIA will trace back the path that links the root page (**Page 1**) to the current page (**Page 8**), and fetch the page content of exclusively the pages in the path.

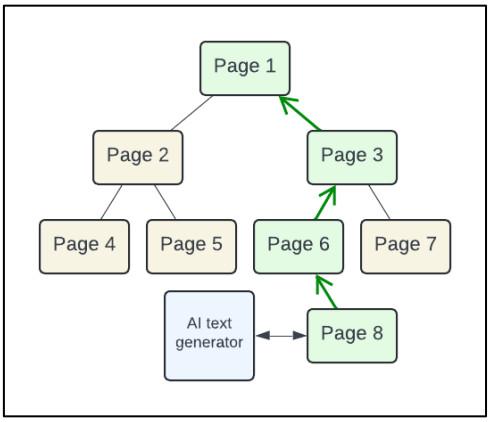


Figure 76 - Back-propagation when generating text

In Figure 77, the pages in the path are – **Page 1**, **Page 3**, **Page 6** and **Page 8**. TEIA will feed Groq's API with the content of all 4 pages, to get a relevant, contextualized prediction on how to complete Page 8.

This process is essential to ensure that the tool is useful, not to generate lazy full chapters, but to stimulate the writers with new and refreshing ideas.

4.6.2 Image Generation

TEIA has the potential to include a large amount of snippet types, to make the readers' experience richer and more interesting. The only snippet that will be included for now, as mentioned, is the image snippet. This means the writers can hide images in text segments inside pages, so that readers can then trigger the image to display, when interacting with the correct text segment.

The writers have two options, when it comes to integrating images into the text. They can either upload an existing image in their device, or generate a new artificial one, with the AI generation tool.

Figure 78 shows the layout that TEIA provides to generate AI images. As mentioned, the application will use the Stable Diffusion (Stable Diffusion, Founded in 2022).

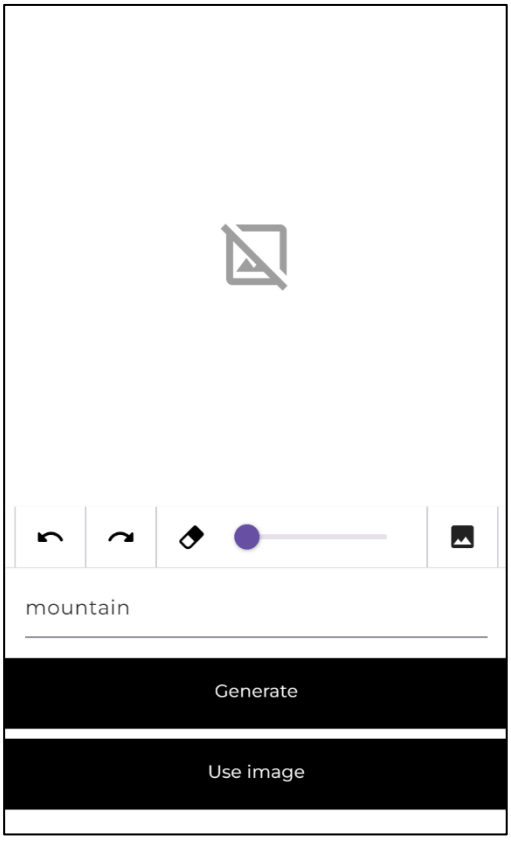


Figure 77 - Screen for generating AI images

This screen contains:

- A tool bar for in-painting capabilities, also provided by Stable Diffusion. Unfortunately, these are not yet implemented as of the time this document is being written;
- An image frame, for the generated image;
- A text field, for the text prompt that will guide the image generation;
- Two buttons, one to generate an image (once the writer is satisfied with the prompt), and another to use the image (once the writer is satisfied with the image). When clicking “Use image”, the image will be associated to the text segment the user selected.

4.7 Reading a Chapter

The last screen to implement is the reading screen. This is the screen readers will see while reading chapters. As shown in Figure 79, the focus is to provide the text content of the current page.

Figure 78 - Chapter reading screen

As mentioned in the analyses to Table 1, the segments of text containing snippets are hidden, which means they are rendered the same as other text. The reader does not know which type of snippets are hiding in the page they are reading. They must make pondered decisions on which ones to trigger, because some can be good (may contain useful information, or interesting narrative directions), but others can be bad (unfortunate outcomes for the protagonist). When the user clicks on a text segment that contains an image snippet, the image appears in the form of a dialog, as Figure 80 illustrates.

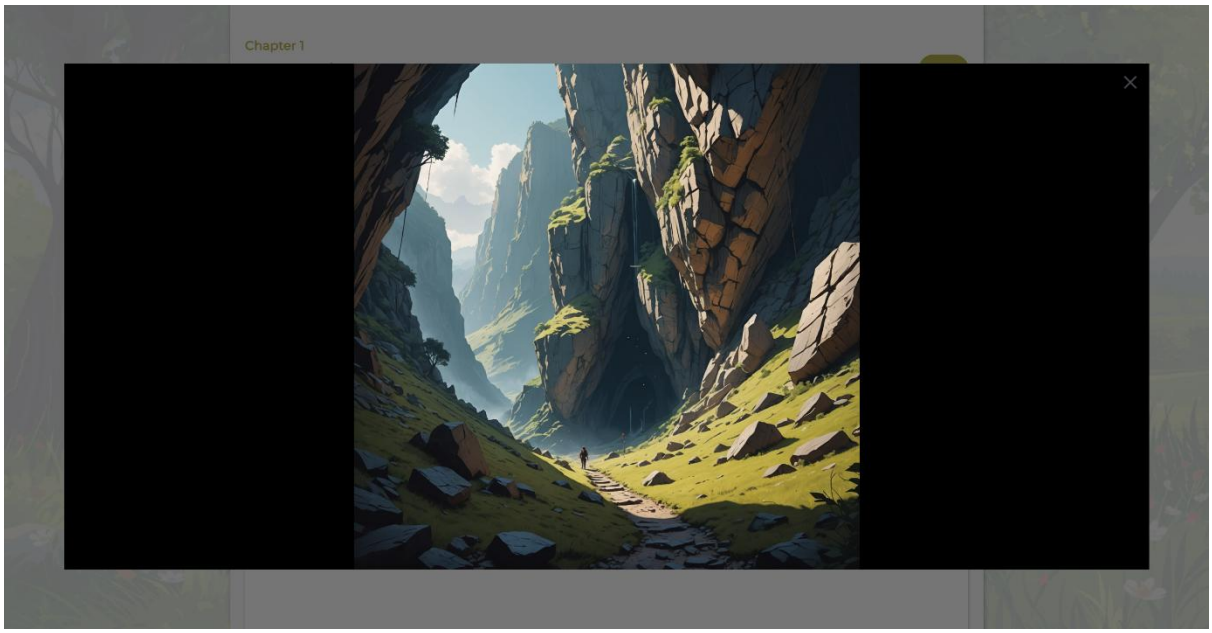


Figure 79 - Image snippet dialog

Another important aspect of this screen is that readers cannot go back to a previous page, once they click on a choice snippet to go to another page. This is important because it blocks readers from experiencing multiple paths in the same chapter – what is intended for TEIA is that each reader experiences their own side of the narrative (their own path) based on their decisions, and then share between each other.

Once the reader reaches a page that has no children pages, the screen shows a new button – the **“Finish Chapter”** button, as seen in Figure 81.

Finishing a chapter means the reader is effectively ready for the next one. The group will only transition to another state when all readers are ready – in other words, when all readers have clicked the **“Finish Chapter”** button.

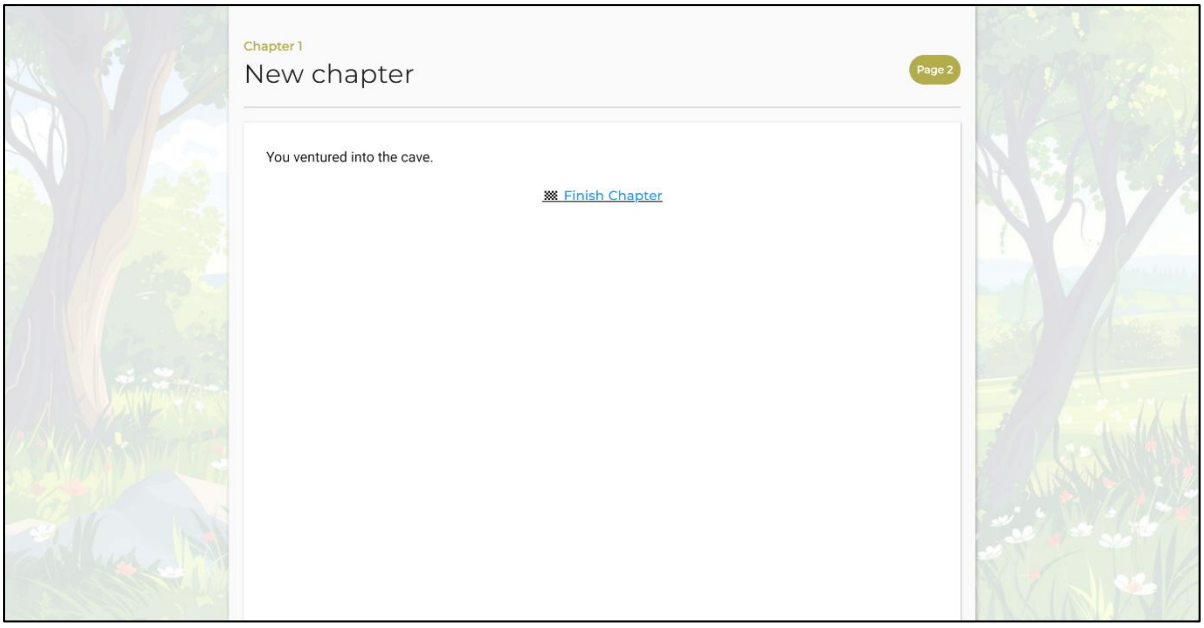


Figure 80 - Chapter reading screen for the last page

4.8 Implementation Overview

During the implementation of TEIA, not only it became clear that a few features would have to be left out due to time constraints, but new ideas and improvements continuously came to mind.

As expected, the component that raised the most challenges was the chapter editing screen – in other words, the content creation mechanisms. The view that contains the rooted-tree structure of the chapter and the collaborative real-time text editor were both elements that took a long time to implement. Especially the collaborative algorithm (the Delta-CRDT approach), since it constantly presented different issues, throughout the implementation phase.

The result of this implementation is a usable product, although it was crafted as a proof of concept. The next chapter will address the immediate improvements and ways to enhance the user experience.

5 Results and Future Work

This chapter will present the outcomes of the development and implementation of TEIA, focusing on the key results that demonstrate the efficiency of the system in facilitating collaborative storytelling. The findings will be discussed in relation to the objectives set out in the early chapters of this document, highlighting both the successes and the challenges encountered.

Following the presentation of results, this chapter will also explore potential directions for future work. These suggestions aim to address the current limitations, improve user experience, and expand the application's functionality to cater to a broader audience. The goal is to provide a roadmap for the continued evolution of TEIA.

5.1 Evaluation of Results and Achievements

The most important achievements of this implementation are related to the collaborative storytelling features, which were the main objectives of TEIA. Other important results to evaluate are the text and image generation tools, and how useful and easy to use they are. Finally, the reader experience is also an essential aspect of the application that warrants significant attention, despite not being the primary focus in the current implementation.

5.1.1 Collaboration Results

To evaluate how the collaborative real-time text editor reacts to multi-user interactions, I will analyse the scenario in Figure 82, where 2 users edit the same document, simultaneously:

- User 1 attempts to write "once upon a time" in line 1;
- User 2 attempts to write "in a galaxy far away" in line 3.

Both users' intent was preserved and the document converged – meaning both users now are in possession of exactly the same document (text content).

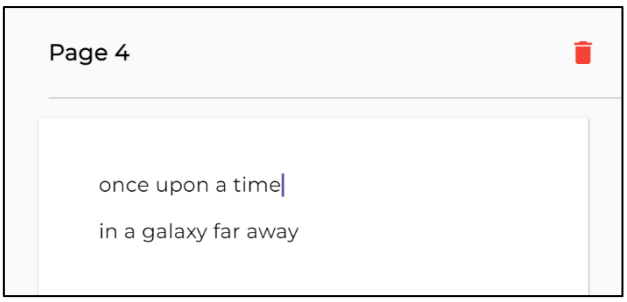
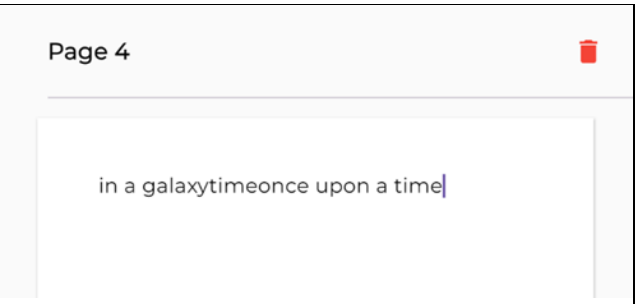
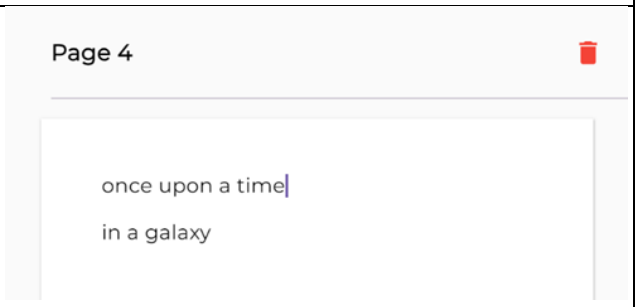
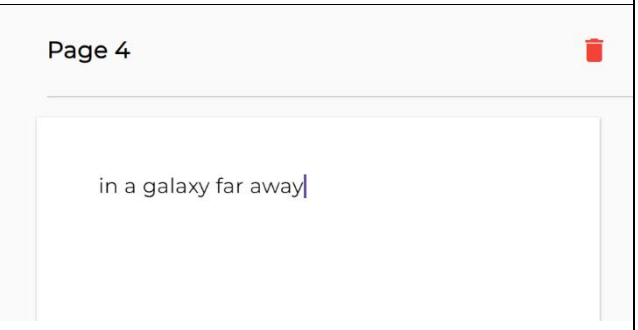


Figure 81 - Text collaboration result

The text editor is working quite well, especially when compared to previous iterations of the latter CRDT-Delta solution. For example, the Firebase-only approach, in the same scenario, would often result in one of the possibilities in Table 4.

Table 4 - Results of previous text collaboration solutions

Overlapping text: Input from User 2 overlaps input from User 1, in the same line.	
Lost characters: Input from User 2 is in the correct line, but missing characters.	
Total override: Input from User 1 was completely overridden by input from User 2.	

The simpler approaches originate different types of conflicts in the document, when users collaborate in different lines. The CRDT-Delta solution avoids all the natures of conflicts mentioned in Table 4, and I am yet to discover any others, upon further testing. There is, although, room for improvement, especially when it comes to performance. For example, pasting an extraordinarily large block of text takes an obnoxiously long time. This will be discussed in an upcoming chapter, as a potential improvement.

Another achievement in TEIA's implementation is the snippet integration. Snippets were envisioned as a tool for writers, to attribute media and interactions to text segments in a page. This translated to the final product very well, as snippets are very easy to use and functioning very well. As seen in Figure 83, writers can select an excerpt from the text and choose one of the snippet options that appear on the right.

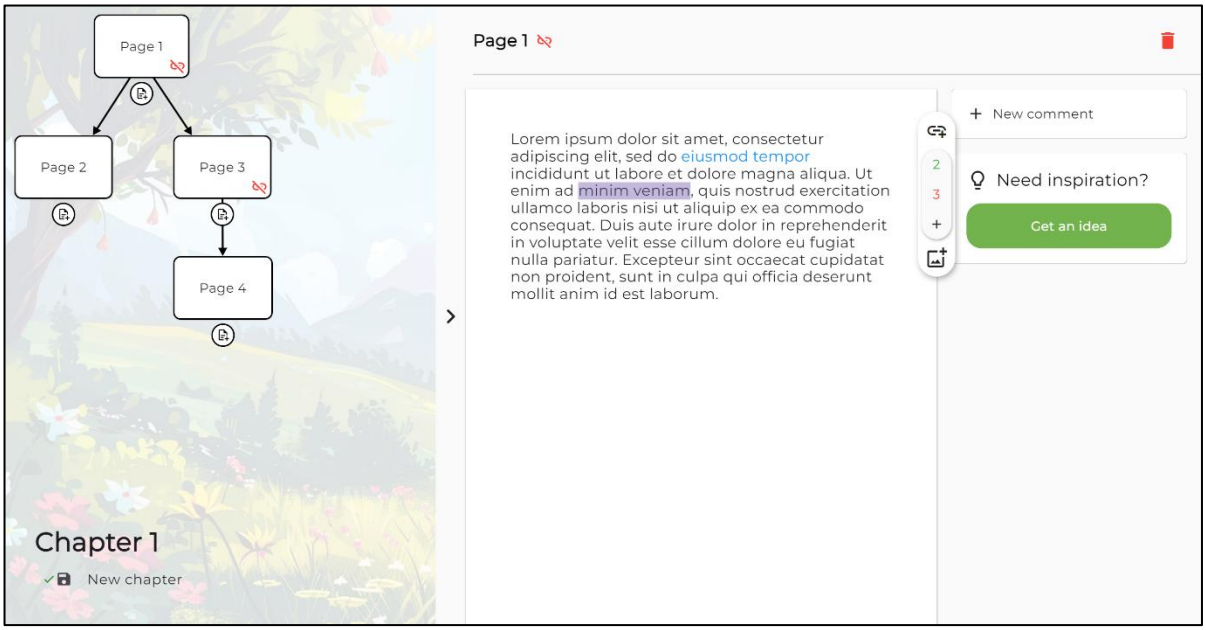


Figure 82 - Text selection in the chapter editing screen

Once a snippet is confirmed, the text formatting for the excerpt selecting changes to the colour blue, to clearly indicate where the snippet is located.

Another important feature that is also illustrated in Figure 83 is the connection-link tracking feature TEIA provides. The page opened in the page editor is **Page 1**. In the chapter editor (the rooted-tree diagram), **Page 1** is marked with the "unlinked" red symbol. This means that one of the connected pages (**Page 2**, **Page 3** or both) is not linked with a snippet, inside **Page 1**. Figure 83 also captures the choice snippet options, and lists the available pages to link.

Page 2 is listed as green, meaning it is already linked – in this case, through the snippet in *"eiusmod tempor".*

> ***Note:*** *The term "connection-link" refers to the duality of pages' relationship. Two pages are connected when they are parent and child in the rooted-tree structure. Two pages are linked when one of them is referenced through a choice snippet by the other.*

Another collaborative aspect of TEIA's storytelling are the comments that writers can leave to each other, in specific pages, as seen in Figure 84. They can also reply, and resolve the comment once it is no longer relevant.

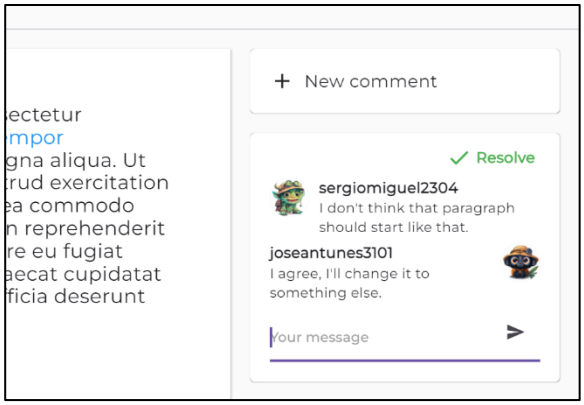


Figure 83 - Comment thread for a specific page

They work as intended, but they would be even more helpful if they could refer to a specific part of the document, like Google Docs' comments. This improvement will be covered in an upcoming chapter.

5.1.2 Text Generation Results

Intended as a tool to stimulate the writers' creativity when their imagination is running low, the text generation tool should provide narrative ideas for a specific page, considering all the context from the previous pages in the chapter. As mentioned, the context must come exclusively from the pages belonging to the path (narrative line) that leads to the current page, from the root page.

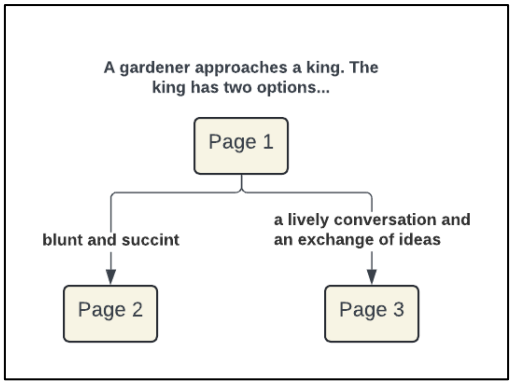


Figure 84 - Chapter structure for king-gardener scenario

TEIA can now provide meaningful narrative options to writers. Consider the chapter structure in Figure 85. The body of Page 1 is represented in both Figure 86 and Figure 88.

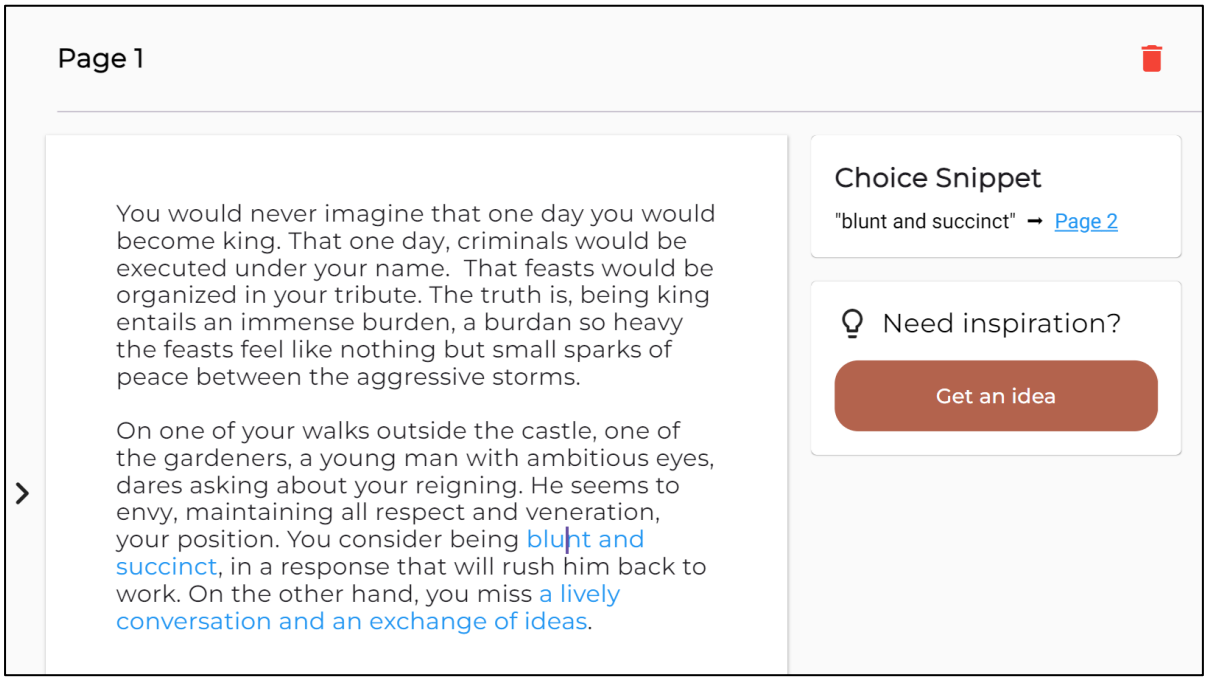


Figure 85 - First snippet in page 1 of king-gardener scenario

The story is about a king that is approached by his gardener outside the castle, asking about the burdens of his duty. The king must now decide if he will spend time with the servant, or swiftly dismiss him.

In Figure 86, the excerpt “blunt and succinct” is linked to **Page 2**. In Figure 88, the excerpt “a lively conversation and an exchange of ideas” is linked to **Page 3**.

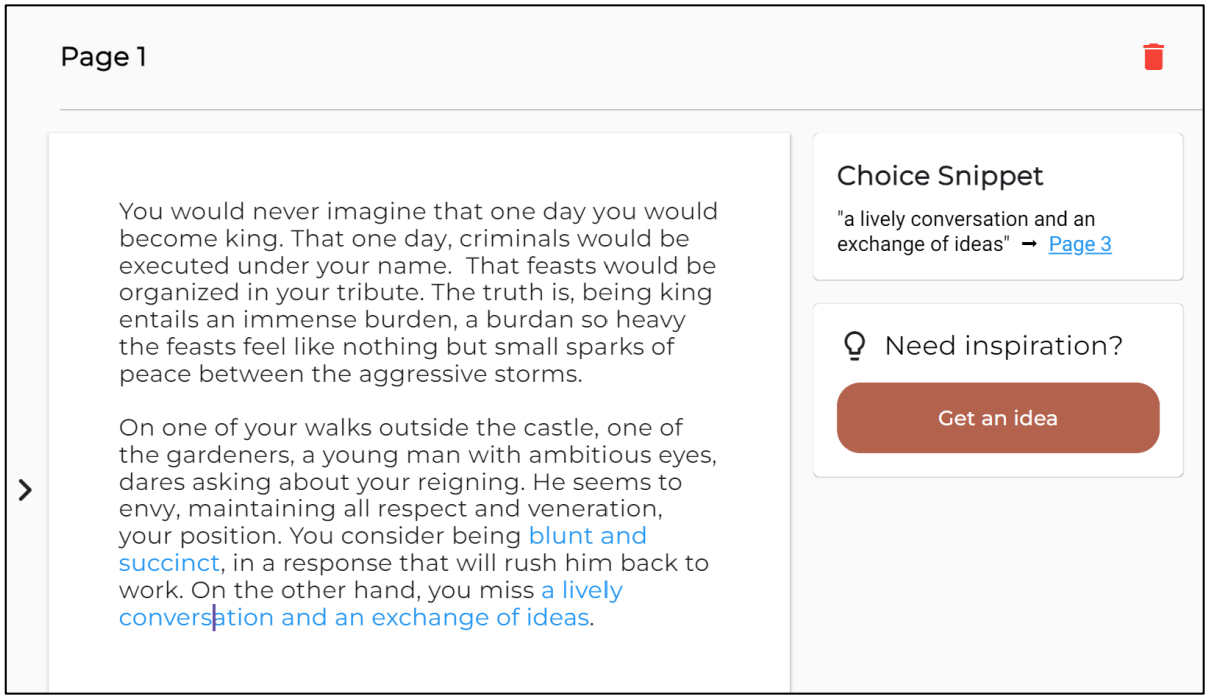


Figure 87 - Second snippet in page 1 of king-gardener scenario

If the writers navigate to **Page 2**, and try to generate text in an empty page, TEIA will consider the page that came before that. In Figure 87, the text generated by this tool is presented, for **Page 2** (the blunt and succinct choice).

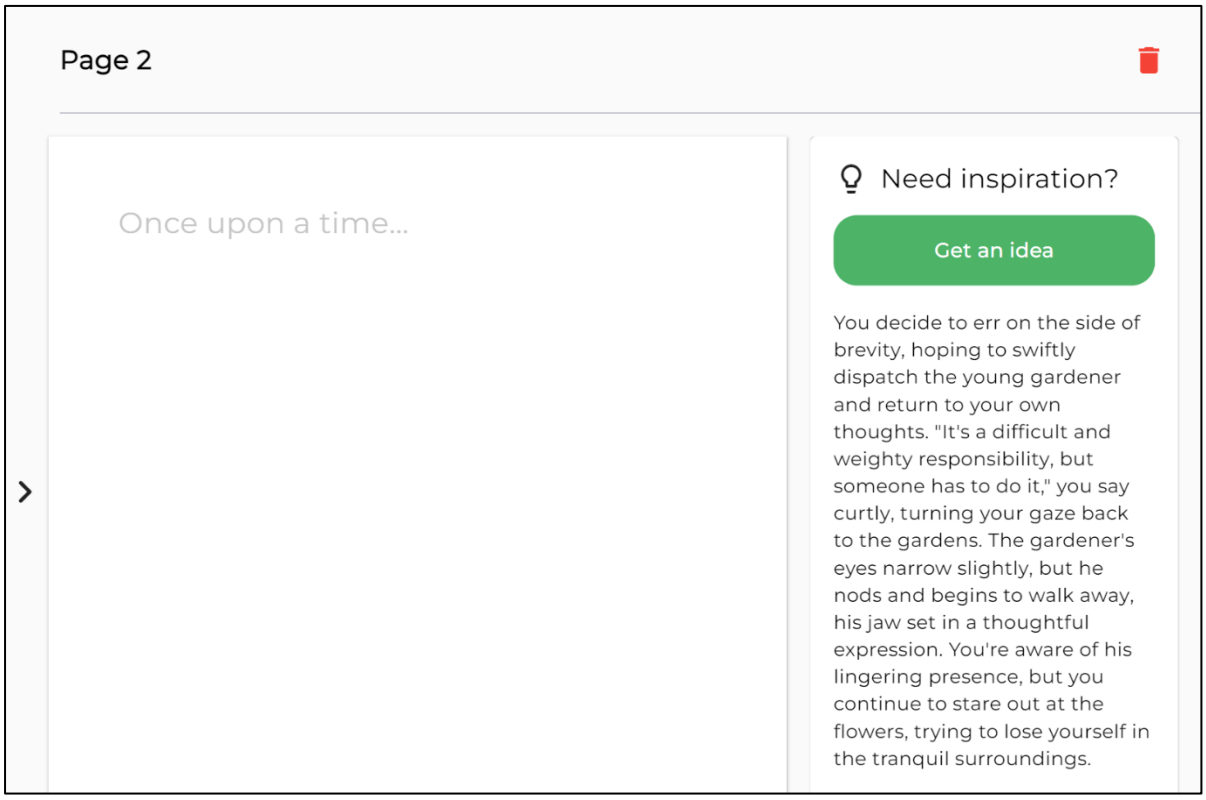


Figure 86 - Text generation for page 2 in king-gardener scenario

In Figure 87, the text generating tool generates an abrupt option, as expected, starting with:

> *"You decide to err on the side of brevity (…)"*

This means the king was blunt and succinct to the gardener, as intended. In Page 3, the opposite should happen – the text generation tool should generate a narrative option in which the king stops to talk to the gardener, patiently. Figure 89 shows that option.

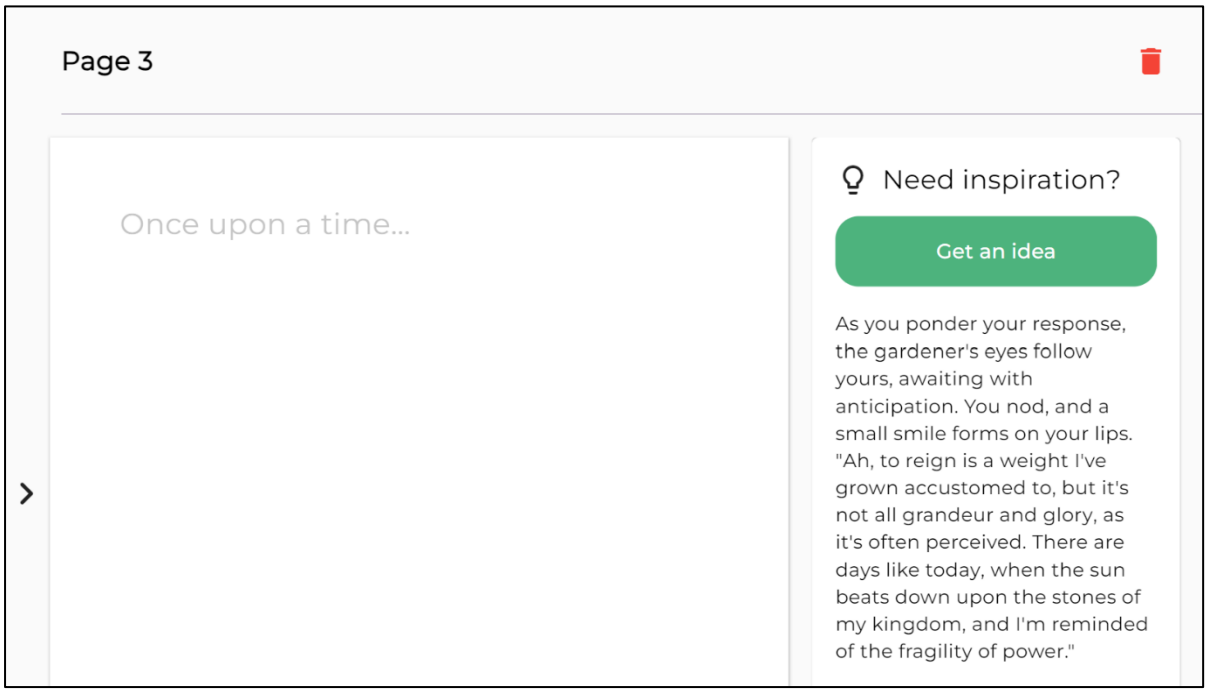


Figure 88 - Text generation for page 3 in king-gardener scenario

> *"You nod, and a small smile forms on your lips."*

In this variation of the narrative, the king was interested in the gardener's question, and the generative text AI tool took that into consideration.

Overall, the text generation tool performs effectively, and according to the requirements.

5.1.3 Image Generation Results

Although there is not much to address in this chapter, image generation plays a meaningful part in TEIA's storytelling. The ability to generate high-quality images, shaped as per their own requirements, without having to possess any artistic knowledge or skill, allows writers to add a visual flavour to their stories.

TEIA utilizes Stable Diffusion to generate AI compositions. Even though writers have total freedom on guiding this tool to generate the images they need, TEIA enforces a specific style. All prompts originated from TEIA are appended by:

> *"cinematic illustration"*

These extra two words condition the generation to create cinematic compositions that are not photo-realistic. This is the style I envisioned for TEIA.

Figure 90 illustrates 3 examples of images, with their respective prompts.

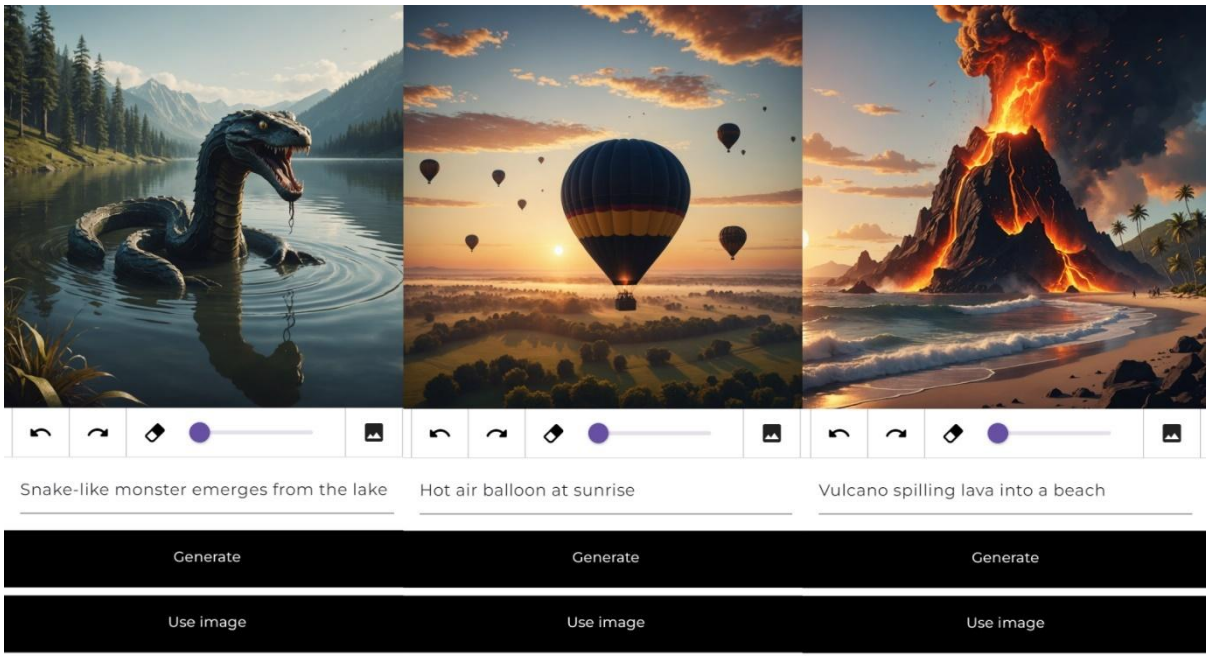


Figure 89 - Example images generated from TEIA, with Stable Diffusion

The guiding text prompts are very ambiguous in terms of style, angle and feel. Thanks to TEIA's prompt-appending styling, the images are cinematic and their style are not very diverging from each other. Very fitting for storytelling visual aids.

5.1.4 Reading Results

Due to time constraints, the reader experience did not achieve the level of engagement initially anticipated, and it also fell short of meeting the ambitious requirements for social interaction. Social interaction was achieved in part by having the writers collaborate in storytelling, but it could prove interesting to also have readers interact with each other in some way. These possibilities will be addressed in an upcoming chapter.

Making the reader experience more interesting, would also mean making the writers' task more complex and time-consuming. For example, by adding more diversity of snippets. TEIA must aim for a compromise in which the readers experience gets richer, but the writers still find their storytelling task compelling and enjoyable.

Even so, TEIA is currently able to provide a complete reading experience, that reflects the work of the writers.

5.2 Next Steps and Future Improvements

This chapter outlines planned next steps for the TEIA and suggests potential areas for future improvement. While the current implementation has successfully achieved its primary objectives, there are several opportunities for enhancing the application's functionality, user experience, and scalability. The following sections will detail these prospects, providing a roadmap for continued development that addresses existing limitations and explores new possibilities for the system's evolution.

5.2.1 Implement Notifications

The first feature that would immensely improve on the user experience, for both reader and writer, and was initially planned to be included in this prototype, is a push notification system.

Since TEIA is a web application, the best way to notify users, is through email messages. Table 1 lists example scenarios in which a notification would be triggered and why.

Table 5 - Notifications and their intent

Notification	Target users	Description
Adventure started	All users	Triggers when the admin start the group session to warn the users that the adventure started.
Adventure ended	All users	Triggers when the last reader reads the last chapter in the story, to warn the users that the adventure ended.
New chapter available	All readers	Triggers when the writers publish a new writer, and warns readers of its availability.

All readers are done	All writers	Triggers when the last reader finishes the current chapter, to warn the writers that they can now publish a new chapter.
Co-writer is done	Writers that have not yet voted to publish the chapter	Triggers when a writer votes to publish the currently editing chapter, and warns the remaining writers.
Chapter writing reminder	A specific writer	Triggers when a writer has not logged in for 24 hours, and the readers are waiting for a new chapter.
Chapter reading reminder	A specific reader	Triggers when a reader has not logged in for 24 hours, and there is a new chapter to read.

These notifications are very useful to alert the users of group advancements, so that they do not have to constantly open the app to check, for example, if a new chapter is available (third row in Table 5) - they can simply wait for the notification stating so.

Overall, notifications would enhance the user experience, especially outside the application. They help smoothen the experience and promote social interaction.

5.2.2 Improving the Delta-CRDT Algorithm

While the Delta-CRDT algorithm reached a state of which I'm quite proud of, there is plenty of room to improve, as mentioned in the results. This chapter will address each aspect that can be enhanced in detail. Some of these aspects were already briefly mentioned during the implementation or during the results analysis.

5.2.2.1 Handling Larger Documents

An issue that can prove quite significant as document increase in size, is TEIA's overhead when parsing large blocks of text. This happens in two situations:

- When pasting large blocks of text into the document;
- When parsing large documents from the backend (when the writer opens the page editor).

First, it is important to understand why this happens.

The only information that is stored in Firebase Realtime Database is the change queue. Thankfully, TEIA relies on an algorithm that collapses the change queue into as little changes as possible, regularly. However, in this case, the change queue size is not the cause of the problem, but the total amount of text characters in the document. Because the application keeps track of all character identifiers locally (so that local insertions and deletions are possible), the first fetch from the backend can be computationally expensive, since it needs to generate all identifiers. This means - the time this serialization takes is proportional to the length of the document.

This necessity of tracking each character and respective identifier locally comes from the fact that *FlutterQuill* (FlutterQuill, Published in 2021) does not support my custom CRDT-Delta architecture.

Possible solutions to this issue are:

- Find a text editor that supports custom data structures.
- Fork from *FlutterQuill*'s repository, and add support to the CRDT-Delta data structure. In other words, implement my own text editor widget.

Both the previous solutions preserve the CRDT-Delta architecture as it was designed in, but other approaches might involve changing or even redesigning the entire architecture.

The paper (Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks, Published in 2009) published to the French comprehensive open-access repository Hal Open Science, inspired my approach to collaborative text editing. The paper elaborates on the essence of CRDTs, and covers details overlooked by my solution - details that are not too relevant for short documents as Teia is intended to host.

Fortunately, TEIA's documents are not intended to be very large, which means this issue will be present only in extreme cases.

5.2.2.2 The Generation of Unique Identifiers

The generation of unique identifiers for the document's characters can also be improved. Currently, the identifiers consist of an array of integers, not at all optimized for human interaction. When users are editing a text document, they can place the cursor in whatever position they desire, to edit a specific part of the document. When that happens, the series of characters inserted will contain one more element than the characters that came before.

Figure 91 illustrates a scenario in which the writer places the cursor between **"E"** and **"D"** (identifier **[2]** and **[3]**, respectively), in a document that consists of the word **"RED"**. Because this word written from left to right, the identifiers are sequential (1 to 3).

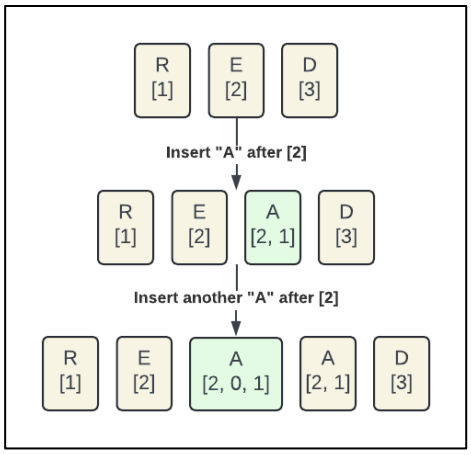


Figure 90 - Cursor repositioning identifier issue

When the writer inserts **"A"**, TEIA needs to calculate an identifier between **[2]** and **[3]**. Since there is no integer between 2 and 3, the only possibility is to add a new element to the array. So, the new identifier becomes **[2, 1]**.

After that, the writer places the cursor, again, after **"E"** (identified by **[2]**), and inserts another **"A"**. This time, the new character finds itself between **[2]** (which is equivalent to **[2, 0]**) and **[2,1]**. Since there is no integer between 0 and 1, a new element is needed once more. The new identifier becomes **[2, 0, 1]**, as we see in the result of Figure 91.

If the writer keeps placing the cursor after the very same character, before inserting text, the identifiers' array length will keep growing indefinitely. While it is true that this example is extreme, this kind of cursor placements will happen with considerable frequency, and the length of identifiers might escalate to undesirable values.

There is some research done in this area, that resulted in complex, optimized algorithms. The paper (LSEQ: an Adaptive Structure for Sequences in Distributed Collaborative Editing, Published in 2013), published to Hal Open Science, exposes intricate research on how the generation of unique identifiers can be optimized for both efficiency and storage, using a more complex tree-based structure.

This unoptimized approach to identifier generation is not a big problem for TEIA, however. As I pointed out in the previous chapter, the documents are meant to be kept short. This will prevent identifiers' length to escalate to problematic values, under normal conditions.

5.2.3 Enhance the Writer Experience

Improving the writer experience is crucial for fostering creativity and efficiency within TEIA's storytelling core. This section discusses the potential enhancements aimed at streamlining the writing process, while also offering a more diverse set of tools. By focusing on these improvements, the platform can better serve the needs of writers, ultimately leading to more engaging and cohesive storytelling experiences.

5.2.3.1 Chapter History

One of the critical features to enhance the writing experience within the TEIA platform is the implementation of a **chapter history** functionality. This feature is designed to provide writers with easy access to the content of previous chapters while they are working on the current one. The inclusion of a chapter history allows writers to reference earlier narrative points, ensuring continuity and coherence in the storytelling process.

In collaborative and non-linear storytelling, maintaining consistency across the narrative is challenging, especially when multiple authors contribute to different parts of the story. Writers may need to revisit earlier sections to recall plot details or thematic elements, essential to the narrative's progression. Without a convenient way to access this information, the risk of narrative inconsistencies increases. The chapter history feature will directly address this challenge.

Other than revisiting previous chapters, some useful features the chapter history could provide are:

- Search for keywords, to more efficiently find plot points;
- Re-reference media used in previous chapters (like images).

This feature is essential for a good writing experience, and to help writers ensure narrative consistency.

5.2.3.2 User Text Cursors

When thinking of collaborative text editing experiences, I immediately refer to Google Docs. Google Docs, among other things, has a remote cursor functionality, that allows users to see each other's cursors inside the document, as Figure 92.

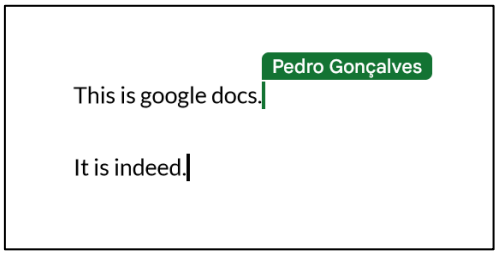


Figure 91 - Remote cursors in Google Docs

A similar feature for TEIA would positively improve the writers' experience, because they can locate their co-workers in the document, and avoid interfering with their work.

It is a simple, but helpful feature, that I was not able to achieve with the FlutterQuill widget, after many attempts.

5.2.3.3 Grammar Check Feature

To further assist the writers, TEIA could also help them identify grammar/spelling errors in the document. This is another feature inspired by Google Docs.

Google Docs not only points out the grammar/spelling mistakes, but it also suggests fixes, as seen in Figure 93.

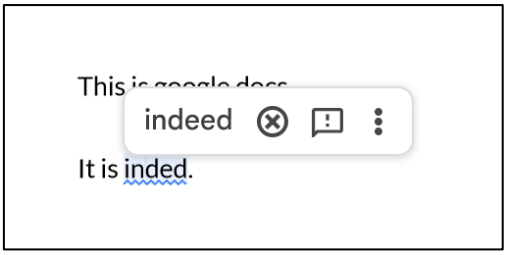


Figure 92 - Grammar check feature by Google Docs

In the pursuit of creating a more polished and professional writing experience within TEIA, the integration of a **grammar check feature** stands out as a key enhancement.

The primary purpose of the grammar check feature is to, as mentioned, assist writers in producing grammatically correct text, which is essential for maintaining the credibility of the narrative. In a collaborative writing environment like TEIA provides, where multiple authors contribute to a single story, maintaining consistent language quality is vital. This feature ensures that all contributors can produce text that adheres to standard grammar rules, reducing the likelihood of errors that could detract from the reader's experience.

5.2.3.4 Image Inpainting

TEIA supports the generation of AI images, a very useful resource for a more visual storytelling approach. However, Stable Diffusion does not always generate images consistent with what the writers require. Inpainting is a feature also provided by Stable Diffusion that allows users to reiterate a selected part of the image.

This is particularly useful when the generated image is close to what the users envisioned, but contains an element that they would rather have removed or even replaced by something else.

As an example, I attempted to inpaint a dragon into a medieval village, generated by MidJourney. The results are shown in Figure 94 and Figure 95.

Figure 93 - "Medieval village" generated by MidJourney

Figure 94 - Dragon inpainted into Figure 93

As seen in Figure 95, MidJourney seamlessly inpainted a dragon into the skies above the medieval village. If ever implemented into TEIA, this feature should:

- Allow users to select portions of an AI generated image by either:
  - Square-selecting a quadrilateral section of the image;
  - Drawing a mask over the image.
- Generate content for the portion selected, while considering the parts of the image that were preserved;
- Options to undo/redo.

Image AI Inpainting would serve as a powerful tool for writers who wish to incorporate or modify images as part of their storytelling. By using AI to intelligently fill in missing parts of an image or to alter specific areas, this feature allows users to create custom visuals that align closely with their narrative.

5.2.3.5 Independent Story Editor

Another problem that TEIA currently carries, that should be addressed in the future, is the lack of freedom in story creation. This application was idealized as a platform that supports asynchronous sessions of storytelling, where the stories are told chapter by chapter, while readers traverse the narrative as the chapters become available. However, what if a user logs

into TEIA, and wants to write a story from start to finish, without involving any readers in the writing process?

Currently, TEIA does not provide a way to work on stories, independently from group sessions. Having an independent story editor as an option in TEIA, would improve the writers' experience, by providing an option for a writer-only experience.

This independent story editor would offer helpful features, like:

- Create complete stories, without having to wait for readers to read each chapter, and without belonging to any group;
- Edit already existing stories.

The independent story editor is not an urgent feature, but it would offer writers this useful writer-only experience, to privately edit or create stories.

> ***Note:*** *The independent story editor is not intended to be a solo experience, although it can be. Multiple writers can still collaborate inside the independent story editor. This experience can essentially be viewed as group consisting of only writers - a group that always skips the reading state of its lifecycle (refer to Figure 2).*

5.2.4 Enhance the Reader Experience

While the TEIA has been primarily designed to facilitate collaborative storytelling, the reader experience is equally crucial in ensuring the appeal of the application. This chapter focuses on enhancing the reader's journey throughout the narrative, with the goal of making the experience more immersive, intuitive, and engaging. By refining the reader interface and introducing features that support deeper interaction with the story, TEIA will offer a richer and more satisfying experience to its audience.

5.2.4.1 Retracing Previous Decisions

One of the key enhancements proposed to improve the reader experience on the TEIA platform is a feature that allows readers to retrace previous decisions in the story. This functionality allows readers to revisit and review the choices they have made throughout their journey. By enabling readers to track their previous decisions, this feature allows them to recall important details that may impact future choices.

> ***Note:*** *This feature won't allow readers to see alternate paths that differ from their previous decisions. They will be able to revisit exclusively pages that they have already read. This feature is intended to revive the user's memory about details in previous chapters*

Currently, there is no way for readers to, for example, revisit the pages from chapter 1, when they are already in chapter 2. This is a very urgent feature to implement, because readers should be able make decisions based on past occurrences. Also, the time that passes between chapters, depending on how quickly writers deploy new chapters, might be very long. It's fundamental that readers have the option to retrace their steps at any time.

5.2.4.2 Read Existing Complete Stories

This feature was planned in the beginning of the project, but due to time constraints, it was left out of TEIA's prototype. Once a group's session finishes, the story is complete – it now has a beginning and an end. This means that other users (readers) would be able to read it, if they have access to it.

Figure 95 - Search for existing stories option

When selecting a story for a group (Figure 96), it should be possible to select an already existing one, instead of creating a new one. In the group screen (for the idle state), when the group has not started a session yet, there should be a button to “Search” for existing stories. In this case only readers are allowed in the group, as no writers are required.

5.2.4.3 More Diversity of Snippets

As TEIA continues to evolve, expanding the diversity of snippets offers an exciting opportunity to enrich the digital storytelling experience. Snippets – interactive portions of text that trigger media or other types of events upon interaction – currently include:

- **The choice snippet**, that sends the user to the corresponding page, when interacted with. This is the core mechanism that allows readers to traverse the non-linear narrative.
- **The image snippet**, which displays an image when interacted with.

By introducing additional types of snippets, TEIA can offer a more dynamic and immersive experience for both writers and readers, enabling more creative ways to engage with the narrative.

However, as previously mentioned, I need to ponder how harder the writers’ task would become with the addition of specific snippets. While it is true that diversity enhances the reader experience, it is also true that the story writing process becomes more complex.

Several new snippet types could be introduced to TEIA, each offering unique ways to interact with the story:

- **Audio Snippet:** This snippet would allow writers to embed audio files directly into the text, enabling readers to listen to music, sound effects, or voice narration as they progress through the story.
- **Interactive Map Snippet:** For stories with complex worlds or multiple locations, an interactive map snippet could allow readers to explore the setting more deeply. Clicking on different parts of the map could reveal additional information, trigger events, or even transport the reader to different parts of the narrative.
- **Poll Snippet:** The poll snippet could allow writers to engage readers in collaborative decision-making, and gather feedback on story elements. Readers could vote on a

decision that affects the story's direction or simply share their preferences, creating a more participatory reading experience.

- **Text Expansion Snippet:** This snippet would enable portions of the text to expand or reveal hidden content when clicked. This could be used to provide additional background information, character thoughts, or alternative perspectives without interrupting the flow of the main narrative.

All previous four suggestions are interesting ways of making the reading experience more compelling, and all of them are easy to integrate into the text, from a writer's perspective. All, except for one.

The Interactive Map Snippet requires an extra investment from the writer. Although it is a fantastic way of visually aiding readers with a physical representation for the world they are journeying through, the writers would have to dispend time to design a coherent world, with enough detail to even be interesting to the reader.

Introducing more diverse snippets requires careful planning to ensure that they are seamlessly integrated into TEIA. Each snippet type must prove compelling for both writers and readers. The interface should allow writers to easily embed and configure snippets within their text, while readers should find the snippets intuitive and non-disruptive to their reading experience.

5.3 Results and Future Work Overview

In this chapter, I have examined the significant results achieved through the ongoing development of TEIA and discussed various proposed enhancements aimed at improving both writer and reader experiences. The successful integration of features such as snippets and collaborative mechanisms has laid a strong foundation for a more interactive storytelling environment.

I explored the importance of diversifying snippets, introducing various types such as audio, polls, and even interactive map snippets. These additions will enable writers to craft more engaging and multi-dimensional narratives, promoting a more immersive and diverse reading experience. Furthermore, the proposed enhancements, including features like retracing previous decisions and re-utilization of existing stories, aim to create a completer and more robust platform.

Through ongoing development and innovation, TEIA will continue to enhance its offerings, ensuring that it remains a valuable resource for writers and readers alike.

6 Conclusion

The development of TEIA represents a significant step forward in the evolution of digital storytelling, particularly in the context of non-linear narratives. TEIA innovates by transitioning the CYOA (Choose Your Own Adventure) format to a collaborative, digital fashion. Throughout this thesis, I have explored the various components and features that make TEIA a unique and versatile tool for both writers and readers.

By fostering an environment that encourages creativity and collaboration, TEIA aims to support a wide array of narratives and empower users to explore new storytelling possibilities. Moreover, the integration of AI (Artificial Intelligence) tools capabilities, including text and image generation, equips writers with powerful tools to enhance their creative process.

There have been challenges and limitations, particularly in balancing the development of advanced features within the constraints of time and resources. However, these challenges have also presented opportunities to refine the platform and focus on what truly enhances the user experience. Additionally, the proposed future improvements offer promising directions for the continued growth of TEIA.

In conclusion, TEIA contributes in a meaningful way to the field of collaborative storytelling, where the lines between writer and reader are increasingly intertwined, allowing for stories to be shaped collectively rather than individually.

Bibliography

(2022). *A Dummy's Guide to Word2Vec.* https://medium.com/@manansuri/a-dummys-guide-to-word2vec-456444f3c673: Manan Suri.

(2017). *A simple approach to building a real-time collaborative text editor.* https://digitalfreepen.com/2017/10/06/simple-real-time-collaborative-text-editor.html : Rudi Chen.

(2023). *Attention Is All You Need.* https://arxiv.org/abs/1706.03762: Google.

Authentication. (Founded in 2014). https://firebase.google.com/docs/auth: Firebase.

(2018). *BERT Explained: State of the art language model for NLP.* https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270: Rani Horev.

(2016). *Byte-Pair Encoding tokenization.* https://huggingface.co/learn/nlp-course/en/chapter6/5.

(Founded in 2022). *ChatGPT.* https://openai.com/index/chatgpt/: OpenAI.

(Founded in 2010). *Choice of Games LLC.* https://www.choiceofgames.com/.

(2022). *Classifier-Free Diffusion Guidance.* https://arxiv.org/abs/2207.12598: University of California, Berkeley.

Cloud Firestore. (Founded in 2017). https://firebase.google.com/docs/firestore: Firebase.

Cloud Messaging. (Founded in 2014). https://firebase.google.com/docs/cloud-messaging: Firebase.

Cloud Storage. (Founded in 2012). https://firebase.google.com/docs/storage: Firebase.

Conclave, A Private and Secure Real-time Collaborative Text Editor. (2023). https://conclave-team.github.io/conclave-site/: Conclave Team.

Dart. (Founded in 2011). https://dart.dev/: Google.

Delta. (Rich Text Editor). https://quilljs.com/docs/delta/: Quill.

(2020). *Denoising Diffusion Probabilistic Models.* https://arxiv.org/abs/2006.11239: University of California, Berkeley.

Designing an MVC Model for Rapid Web Application Development. (2014). https://www.researchgate.net/publication/275540078_Designing_an_MVC_Model_for_Rapid_Web_Application_Development: Romanian-American University.

(Founded in 2011). *Firebase.* https://firebase.google.com/: Google.

(Founded in 2013). *FirePad.* https://firepad.io/: Google.

Flutter. (Founded in 2017). https://flutter.dev/: Google.

FlutterQuill. (Published in 2021). https://pub.dev/packages/flutter_quill: Bullet Journal.

Gartic. (Founded in 2008). https://gartic.io/: Onrizon.

(2023). *Generative AI exists because of the transformer.* https://ig.ft.com/generative-ai/: Finantial Times.

GetX. (2023). https://pub.dev/packages/get.

(Founded in 2014). *Google Colab.* https://colab.google/: Google.

(Founded in 2006). *Google Docs.* Microsoft: https://www.google.com/docs/about/.

(2016). *Groq API.* https://groq.com/.

(2016). *Hugging Faces.* https://huggingface.co/.

(Founded in 2012). *Inklewriter.* https://www.inklestudios.com/inklewriter/: Inkle.

(Founded in 2014). *Jupyter.* https://jupyter.org/.

Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks. (Published in 2009). https://inria.hal.science/inria-00432368/document: Stéphane Weiss, Pascal Urso, Pascal Molli.

LSEQ: an Adaptive Structure for Sequences in Distributed Collaborative Editing. (Published in 2013). https://hal.science/hal-00921633/document: Brice Nédelec, Pascal Molli, Achour Mostefaoui, Emmanuel Desmontils.

Quantic Dreams. (Founded in 1997). https://www.quanticdream.com/en: David Cage.

(2019). *Real Differences between OT and CRDT in Building Co-.* https://arxiv.org/abs/1905.01517: Nanyang Technological University.

(2019). *Real Differences between OT and CRDT in Correctness and Complexity for Consistency Maintenance in Co-Editors.* https://arxiv.org/abs/1905.01302: Nanyang Technological University.

(2019). *Real Differences between OT and CRDT under a General Transformation Framework for Consistency Maintenance in Co-Editors.* https://arxiv.org/abs/1905.01518: Nanyang Technological University.

Realtime Database. (Foudned in 2012). https://firebase.google.com/docs/database: Firebase.

(Founded in 2022). *Stable Diffusion.* https://stability.ai/: StabilityAI.

(2019). *To OT or CRDT, that is the question.* https://www.tiny.cloud/blog/real-time-collaboration-ot-vs-crdt/: Andrew Herron.

(Founded in 2009). *Twine.* https://twinery.org/: Chris Klimas.