FlexiTEX: LATEX Collaboration Without Giving Up Personal Project Structure

WOUTER TEN BRINKE, University of Twente, The Netherlands

LATEX gives users a lot of freedom in how they structure their projects, but this becomes a problem when working with others. Popular collaboration tools assume that all users follow the same project structure, which does not reflect how people actually prefer to organize their work. This thesis introduces FlexiTEX, a system that allows each user to keep their own project structure while still collaborating on the same content. The system works by flattening a LATEX project, parsing it into an abstract tree that captures the logical structure. It then applies transformation rules to rebuild the project structure based on a configuration file. The transformation is designed to be reversible, idempotent and preserve the ability to compile the document. A proof of concept shows how this approach can be used in a collaborative setup where each user works in a personal branch and changes are synced through a shared internal version. An evaluation on real-world projects shows that the system preserves content in most cases, although some limitations remain due to parser behavior. Overall, FlexiTEX makes it possible to collaborate on LATEX projects without forcing everyone to adopt the same structure.

Additional Key Words and Phrases: LaTeX, document transformation, collaborative editing, abstract syntax tree, configuration-based layout

1 MOTIVATION

While LATEX's flexibility is a strength for individual users, it does not enforce any standards for file layout, naming conventions, or project organization. This can become a problem in collaborative environments where individual file management structures and document styles cannot be merged and lead to inefficient workflows and miscommunication. Currently, there is no adopted framework for structuring projects. As a result, each user organizes content differently, making it harder to share or reuse LATEX when collaborating.

1.1 Research Questions

This research project investigates how LATEX project structures can be managed locally according to individual user preferences, while still maintaining a consistent shared structure to support collaboration. The research focuses first on understanding the variety of existing project structuring styles and then on exploring how principles from software engineering could help in designing and developing a transformation tool. The goal of this work is to make LATEX collaboration easier without requiring users to give up their personal file structures. To guide the project, the following research questions were formulated:

- **RQ1:** How can the logical structure of a LATEX project be represented in a way that is independent of its physical structure, while preserving all document content?
- **RQ2:** How can user-defined LATEX project structures be expressed through a configuration format?
- **RQ3:** How can transformations between personal LATEX structure be performed reliably using the shared abstract representation?

TScIT 43, July 4, 2025, Enschede, The Netherlands

2 BACKGROUND

LATEX is a widely used typesetting system, particularly popular in academic and scientific communities. It was originally developed by Lamport [9] as a set of macros on top of Donald Knuth's TEX system [8]. LATEX allows authors to create professional-looking documents with relatively little effort. It provides fine-grained control over document structure and appearance, making it especially well-suited for content that includes mathematical notation, bibliographies, cross-references and complex figures [7].

Because of these strengths, LATEX has become the preferred tool in academic fields such as mathematics, physics and computer science. A 2009 questionnaire by Brischoux and Legagneux [4] asked editors of scientific journals about typesetting practices. The results showed that 97% of mathematics papers, 89% of statistics papers, 74% of physics papers and 46% of computer science papers were typeset using LATEX. In contrast, Microsoft Word remained the dominant tool in most other disciplines. So while Word is still the most widely used editor overall, LATEX has established itself in more technically oriented fields.

To support collaboration on LATEX documents, various editing tools and workflows have emerged. Overleaf is a popular cloud-based editor that allows multiple users to edit the same project in real time [15], similar to how Google Docs enables shared editing of word processing documents [20]. These platforms make collaboration easier, especially for beginners, by removing the need to set up a local environment.

Other tools like Live Share for VSCode [11] offer real-time editing across machines without relying on the cloud. These are commonly used for code collaboration and can also support LATEX, but they assume that all collaborators work with the same underlying file and project structure.

Git-based workflows offer a more flexible alternative by allowing users to work on their own branches and merge changes over time. This model gives collaborators control over their individual environments and supports version control, but it lacks real-time feedback and does not account for structural differences between users' project.

Despite their differences, all of these approaches share a common limitation: they require collaborators to work within a single, shared project structure. Whether editing happens in real time or through version control, the underlying assumption is that all users organize their documents in the same way. This limits flexibility and makes it harder to support personalized project structure within collaborative workflows.

To understand later sections of this research, it is useful to introduce several fundamental concepts from compiler construction.

Parsing is the process of analyzing a source text according to a formal grammar to determine its syntactic structure, organizing the input into a hierarchy of elements such as expressions and variables [1]. In this project, parsing makes it possible to create a hierarchical view of a LaTeX file by distinguishing elements like sections, environments and commands. An abstract syntax tree (AST) is built from the parse tree to model the logical structure of the input as a nested hierarchy, simplifying the representation by omitting unnecessary details [1]. Code generation traverses the AST to produce output in a pre-defined target format using the information captured in the tree [1]. In this project, code generation allows reconstructing LaTeX source files from the AST.

3 DEFINING LATEX FILE ORGANIZATION

As mentioned before, LaTeX gives authors significant flexibility in how they structure documents and projects. While this flexibility is powerful, it makes it difficult to standardize document file organization. Templates provide some guidance, but in practice, authors define custom commands, organize content differently and adopt varied file layouts.

A tool that aims to transform these projects must first abstract away from the physical structure and instead focus on the document's logical structure. This requires a general model that can represent any document, regardless of how it is written or organized. But to design such a model, it is first necessary to understand how users typically structure their documents in practice.

To get a better idea of this, 30 real-world LaTeX projects were manually analyzed. This number was chosen to cover a wide variety of document types and structures while keeping the manual inspection feasible within the time available for this research. The selected projects included short student summaries, academic articles and full master's theses. Only English, Dutch and German documents were considered, to allow for manual inspection and projects had to compile successfully without major modifications. The goal of this analysis was to capture the diversity of styles, project sizes and structural conventions used in practice and from that extract a basic abstraction that includes all relevant information from these projects.

To identify what makes up the structure of a LaTeX document, each project was read and compared at both the code and content level. The focus was not on formatting or styling details, but on how content was grouped, which commands contributed to the document's hierarchy and how information was logically arranged. This included looking at how authors split content across files, how they used environments and which macros they defined or relied on. The rest of this section is about the findings from that analysis.

3.1 Observing file organization patterns

The analysis revealed that the logical structure of a document often guides how authors organize their files. During the examination of the projects two recurring strategies for splitting content into separate files appeared.

In the first strategy, documents that became too long or too wide to edit comfortably in a single file were divided by moving large sections such as complete chapters or appendices into their own files. What counted as too long or wide varied between users. Some split files after just a few pages or a section with many figures, while others kept larger files but splitted them eventually.

In the second strategy, the file organization more directly mirrored the logical hierarchy of the document. Authors placed each chapter in its own file and sometimes split content further at the level of individual sections or figures. It was also common to group related section files inside a folder named after the chapter, creating a directory structure that matched the logical structure of the document. This method resulted in a physical project structure that closely reflected the document's internal hierarchy.

Many authors used systematic naming schemes for their files and folders. Common practices included directory names like `chapters/` or `sections/` and file names indicating content order, such as `1-introduction.tex`. These conventions supported the chosen organization and made larger projects easier to navigate and maintain.

3.2 Modeling document hierarchy

It stood out that the logical structure of a LaTeX document can be modelled as a tree. The observations in the previous subsection showed that authors often organize their files in a way that mirrors the document's hierarchy, grouping related sections under folders named after their parent chapters. This pattern reflects a nested structure where each element, such as a chapter or section, can contain other elements. Since these relationships form a natural hierarchy, representing the document as a tree makes it possible to capture both the order and the parent-child connections between parts of the text. This tree model ignores how content is split across files and instead shows the document's logical structure, providing a consistent way to analyze and process projects with different project structures.

After examining all the projects, five types of elements were identified that consistently contributed to the document's logical organization, with any custom-defined commands also fitting into one of these categories.

- **Hierarchical macros:** Commands such as `\chapter` and `\section` define the document's structural hierarchy. These built-in macros follow a predefined order [14]. Users can also define custom structural macros, in such cases their relative order must be specified manually to establish the intended hierarchy.
- **Environments:** Defined using a `\begin{}` and `\end{}` pair, environments act as containers that group content. In the structural tree, all content within an environment is treated as a child of that environment.
- **Regular macros:** Commands like `\title` or `\usepackage` serve functional purposes rather than contributing to the document's structure. These are a children to the nearest structural parent node, which may be a hierarchical macro or an environment.

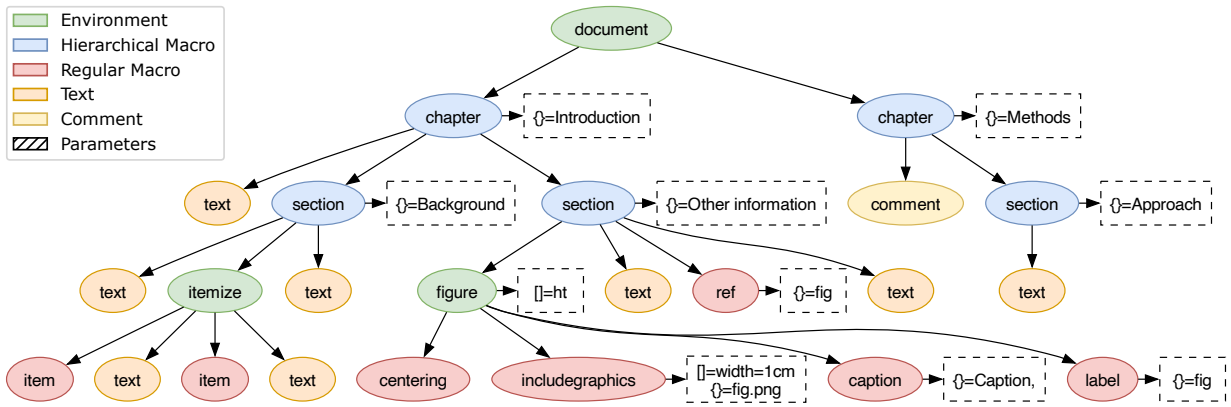

Fig. 1. Tree representation based on the LATEX project defined in fig. 2 (generated by FlexiTEX).

- **Comments:** Comments are essential for collaborative editing, as they may contain author notes or context. Although they do not affect the compiled output, they are preserved in the model by attaching them to the nearest structural parent to maintain correct document order during in-order traversal.
- **Text:** Any plain content not part of a command or environment. Text is attached to the nearest structural parent and represents the main body of the document.

Commands like `\input` and `\include` are excluded from this model. They are only used to split content across files and do not affect the logical structure of the document. Instead the content of the underlying files is placed at the location of the `\input` and `\include` commands.

An example of how a tree structure can represent the logical organization of a project is shown in fig. 1, which illustrates the abstract structure of a small LATEX project. The tree is based on the minimal LATEX document shown in fig. 2. In the tree, hierarchical macros such as chapters and sections are in blue, environments are shown in green and regular macros are red. The parameters of the commands and environments are included so that each node can be traced back precisely to the corresponding command in the source document.

```
\begin{document}
\chapter{Introduction}
Some text.
\section{Background}
Some text.
\begin{itemize}
    \item First
    \item Second
\end{itemize}
Some text.
\section{Other information}
\begin{figure}[ht]
    \centering
    \includegraphics[width=1cm]{fig.png}
    \caption{Caption,}
    \label{fig}
\end{figure}
Referencing~\ref{fig}.
\chapter{Methods}
% Second chapter
\section{Approach}
Some text.
\end{document}
```

Fig. 2. Example of a small LATEX document.

4 RELATED WORK

Several tools exist that convert LATEX documents into other formats. Pandoc [10] supports many input and output formats and can turn LATEX into HTML, Markdown, or DOCX. LaTeXML [12], which is used by arXiv, focuses on preserving semantic structure when rendering documents as HTML [12]. plasTeX [6] parses and interprets macros to produce detailed transformations into formats like HTML. pylatexenc [16] provides utilities for parsing LATEX code and converting it to Unicode. Although these tools effectively extract structured information, they are designed for converting documents into other formats rather than reorganizing or reformatting LATEX while staying in the same format. They also do not preserve all commands or macros, since many are unnecessary when targeting non-LATEX outputs.

Existing works discuss how to design domain-specific languages (DSLs), with a focus on making them expressive and aligned with user needs. The DYOL toolkit introduced by Zaytsev [21] encourages deliberate language design based on user intent and provides ideas for creating languages that match how people actually use them. Building on these principles, the xBib language by Visser [19] applies concepts from DYOL to target BibTEX documents specifically. Since BibTEX is often used alongside LATEX, xBib shows how a specialized DSL can simplify transformations in a familiar domain. In contrast, Bravenboer et al. [3] describe Stratego/XT as an example of a general-purpose framework for program transformation, which uses rewrite rules and strategies as its core abstraction. While both xBib and Stratego/XT demonstrate how DSLs can express complex transformations, neither focuses on transformations of complete LATEX projects, which are central to the problem addressed in this research.

5 SYSTEM DESIGN

The core idea behind the system is to ignore how a LaTeX project is physically structured and instead work with its logical structure. As explained in section 3.2, this structure can be represented as a tree, where each node corresponds to an environment, macro, comment, or text block. By applying transformation rules to this abstract view of the document, different project structures can be generated based on personal or collaborative preferences.

5.1 Design Properties

To support correct transformation and collaborative possibilities, the system is designed with three main properties in mind.

First, the transformation must be reversible, as this is important for collaboration. If each user can transform the shared structure into their personal structure and also convert it back to the shared version, then it becomes possible to switch between personal structures by passing through the shared structure. Without reversibility, it cannot be reliably guaranteed that transformations between different structures will always be possible or consistent.

Second, the transformation must be idempotent. When a rule is applied, such as splitting a chapter into its own file, applying the same rule again should have no effect. This confirms that the rule has already been applied and that the structure is in the expected state. If a rule continues to make changes each time it is applied, it suggests that the structure never fully stabilizes and the transformation cannot be trusted to produce consistent results.

Lastly, the system must preserve compilability. If the input project compiles successfully, the transformed version should also compile and produce the same result. Since most users care about the final PDF or output file, any transformation that changes that would be a problem. The goal is to change structure, not meaning.

5.2 System Architecture

The system uses a phase-based architecture inspired by the phases a compiler passes through [1]. Like a compiler, it transforms the input into an intermediate representation, applies transformations and generates output. By giving each phase a clear and limited responsibility, this design keeps the system modular and makes it easier to adapt or extend. The complete pipeline, including all phases, is illustrated in fig. 3 and the following sections explain each phase in detail.

5.2.1 *Pipeline Phases.*

1. Input Flattening. The process begins by flattening the input project, resolving commands like `\input{}` and `\include{}` so that all content is combined into a single continuous stream. Because the system builds its own model of the document's logical structure, it does not depend on the original project structure.

2. Tokenization. After flattening, the document is parsed into a list of tokens that represent LaTeX elements such as commands, environments, comments and text blocks. The order of this list matches the original order of the content in the source files. Environments are already grouped as nested structures, since their content lies between `\begin` and `\end` pairs.

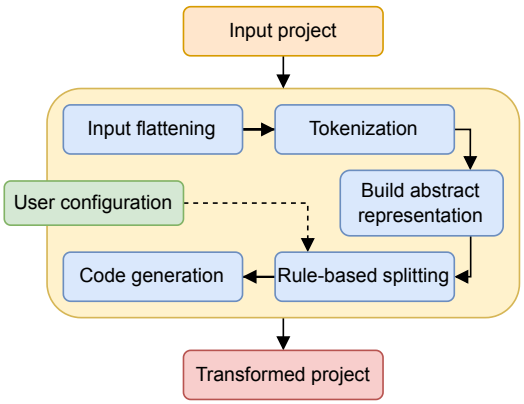

Fig. 3. Overview of the transformation pipeline (yellow) phases.

The parser preserves tokens that follow command, such as spaces and newlines. For example, an `\item` macro often has a space before the item text and a `\maketitle` command is usually followed by a newline. Retaining this information ensures that the original text of the project can be reconstructed exactly after transformation.

3. Abstract Representation Construction. The system converts the token list into a tree structure that represents the document's logical hierarchy by adding every item from the token list to the tree according to the construction algorithm. Construction starts from a root node to ensure the document forms a valid tree. Environments are inserted together with their already grouped children, preserving their nested structure. Structural macros are placed at the correct level by comparing their structural depth with existing nodes, while regular macros are simply attached to the current parent. Text and comments are added to the most recent structural parent. This organization ensures the logical relationships in the document are maintained. The construction logic is shown in algorithm 1, with the placement of structural macros handled by the FindParent function detailed in algorithm 2. Because all elements are nested in their surrounding environments, traversing the tree in pre-order can reconstruct the project's original structure.

4. Rule-Based Spliting. Once the abstract representation is constructed, transformation rules determine how the tree should be split into separate files following the findings in section 3.1. Each rule targets a specific macro or environment type, may include a condition for when it should be applied and contains a template for naming the output file. Rules are applied in post order, processing child nodes before their parents, which is necessary when a rule depends on properties of a subtree such as its length. For example, if a rule splits a node based on its length, the length must include the content of its children but if a child has already been split, its content no longer contributes to the parent's length. This ensures that rules evaluate each node based on the final state of its subtree. When a node matches a rule it is detached from its parent and turned into a separate tree named after the filename specified in the rule. A command like `\input` or `\include` is then inserted at the original location pointing to the new file.

Algorithm 1 Abstract representation construction

```
function BuildAST(token_list)
    root ← Root()
    current ← root
    for node in token_list do
        if IsEnvironment(node) then
            env ← Environment(node.name, node.args)
            children ← BuildAST(node.content)
            for child in children do
                AddChild(env, child)
            end for
            AddChild(current, env)
        else if IsMacro(node) then
            macro ← Macro(node.name, node.args)
            if IsStructural(node.name) then
                parent ← FindParent(current)
                AddChild(parent, macro)
                current ← macro
            else
                AddChild(current, macro)
            end if
        else if IsText(node) then
            text ← Text(node.text)
            AddChild(current, text)
        else if IsComment(node) then
            comment ← Comment(node.comment)
            AddChild(current, comment)
        end if
    end for
    return root
end function
```

Algorithm 2 Finding structural parents

```
function FindParent(node)
    level ← GetLevel(node)
    while NotRoot(node) do
        if IsMacro(node) and IsStructural(node) then
            if GetLevel(node) < level then
                return node
            end if
        else if IsEnvironment(node) then
            return node
        end if
        node ← GetParent(node)
    end while
    return node
end function
```

5. Code Generation. The output consists of a list of trees, each representing the contents of a file. Each tree is converted back into LATEX code through a post-order traversal: the code for each node is generated after recursively generating the output of its children. The abstract representation contains the necessary information to reconstruct macros, their arguments and the spacing around them as they originally appeared. For environments, the correct opening and closing syntax is added and their content is assembled from their nested elements. Since each tree corresponds to a complete file, the text collected at the root node forms the full content of that file. The final result has the desired structure while preserving the original content.

5.3 Collaborative Design

The system can be used for collaboration in two ways, depending on whether only FlexiTEX is used or whether it is combined with a tool like git diff.

In the first case, with only FlexiTEX, collaboration works in a turn-based manner where users take turns modifying the project. The design follows a hub-and-spoke model with a shared version of the document at the center, which always reflects the most recent state. When a user begins editing, they lock this central version and transform it into their personal structure using their own configuration. They then make their changes in that personalized structure. Once finished, the user transforms their version back into the shared structure and updates the central copy, releasing the lock. Any user can then repeat the process. This ensures that only one person edits the shared state at a time, avoiding conflicts and preserving consistency. Because transformations are reversible, users can work in their own preferred structure without needing to agree on a single common structure.

In the second case, if the system is combined with a tool like git diff, it supports mergeable collaboration. In this workflow each user pulls the latest changes from the remote branch, transforms the project using their own configuration, makes their edits, pulls again to update with new remote changes, merges any conflicts locally, transforms their updated version back into the shared structure and then pushes the result to the main branch. This allows multiple users to work in parallel while resolving merge conflicts in their own personalized structure.

6 IMPLEMENTATION

The system described in the section 5 was implemented as a proof of concept command-line tool written in Python. The full source code is available as open source [17] and consists of roughly one thousand lines of code. The implementation is structured into four modules and the transformation pipeline lives in the `main` file.

6.1 Configuration Format

The tool uses a configuration file to define how a LATEX project should be split into separate files. The configuration is expressed in YAML for easy editing and human readability. Each rule describes a structural element by its name, type (macro or environment), optional conditions and a file naming pattern. Placeholders inside angle brackets like `<length>` always refer to the current node, while square brackets like `[chapter]` can refer to either the current node or a parent node depending on the type match. If there is no match, the system searches upwards to the nearest parent of the specified type. An example of the format is fig. 4, this configuration defines how the system splits the document based on section length. If a section has more than five lines, it is split using the first rule. Shorter sections are matched to the second rule. Chapters are then split using the third and last rule. File names can include placeholders such as `[chapter]`, which represents the chapter index, or `[name:section]`, which uses the name of the section. If a placeholder does not

```yaml
structure:
    - name: "section"
      type: "macro"
      condition: "<length> > 5"
      file_name: "chapters/ch[chapter]-[name:chapter]/
          sec[section]/[name:section].tex"

    - name: "section"
      type: "macro"
      file_name: "chapters/ch[chapter]-[name:chapter]/
          sec[section].tex"

    - name: "chapter"
      type: "macro"
      file_name: "chapters/ch[chapter]-[name:chapter].
          tex"

input:
    folder: "input"
    main_file: "main.tex"

output:
    folder: "output"
    main_file: "thesis.tex"
    figure_folder: "figs"
```

Fig. 4. FlexiTEX configuration example

match the current node's type, it is resolved by searching upwards to the nearest parent of the corresponding type. Rules are applied using a fall-through approach similar to a switch case statement: the system checks each rule in order and stops at the first matching rule for a node. If no rule matches, the node remains unsplit.

6.2 Usage and Internals

The command-line tool transforms a LATEX project according to the configuration file. It accepts several flags to customize its behavior. One of these flags generates a diagram of the abstract document tree, as shown in fig. 1. Additional options are described in the repository's README. After installing the module according to the repository instructions, users can run the `flexitex` command in the terminal. When executed, the tool searches for the configuration file in the current working directory and applies the transformation as specified.

Internally, the system builds on pylatexenc [16] for tokenization and uses an implementation ofalgorithm 1 to build the abstract tree. Some adjustments were necessary because pylatexenc categorizes certain elements in more detail than required. For example, verbatim blocks are treated as distinct environment types and must be mapped back to the general environment category during tree construction. These distinctions do not change the overall logic but require handling a few additional cases.

6.3 Collaboration

Collaboration features following the first mode described in section 5.3 are demonstrated in a GitHub repository [18]. In this setup each collaborator works in a personal branch using their own configuration style. These branches stay in sync through a shared internal branch, which contains the project in a flattened structure. This setup allows anyone to write using their own configuration while still working on the same content. In principle, any number of people can collaborate in this way, though not all at once. The system supports turn-based collaboration at scale, as long as pushes are made one at a time.

When a collaborator commits and pushes changes to their branch, a GitHub Actions workflow runs FlexiTEX with the internal configuration file to convert the content into the internal format and updates the internal branch. Another workflow then reprocesses the internal content with each collaborator's configuration and pushes the outputs back to the respective branches. Since FlexiTEX is idempotent, this leads to two outcomes on the author's branch. If the branch is already correctly structured, no changes are made and no commit is produced, which prevents infinite commit loops. If there are changes, a new commit is created, but when this version is transformed back into the internal structure, the content remains unchanged so the cycle ends naturally without triggering repeated updates.

6.4 Open Issues

One issue of the implementation is figure handling. Figures are only moved during the code generation phase. However, because the system flattens the project during preprocessing, it loses the original context of where each relative path pointed. As a result, many LATEX projects using relative paths in `\includegraphics` commands end up with broken references after transformation, since the new folder structure changes the meaning of those paths. The system also does not handle cases where the image file extension is omitted (for example, using `\includegraphics{figure1}` instead of `{figure1.png}`), which prevents locating and copying the file correctly. While figure files are handled specially, all other files in the input project, including style files and bibliography data, are simply copied to the output directory without modification.

Another issue is related to pylatexenc, which has a limited database of known macros and environments. Unknown commands are treated as plain text, disabling splitting for those commands. If the parser recognizes parameters inside unknown commands, it may still parse them as separate structured elements, which can create a tree that no longer matches the original document. Additionally, if pylatexenc recognizes a command but does not know it accepts parameters, it silently discards those parameters. These issues often occur with custom macros, especially when they have the same name as a known macro but different behavior.

7 EVALUATION

To evaluate FlexiTEX, a representative dataset of real-world projects was collected. A Python script downloaded repositories from GitHub using search queries that exclude typical sample projects by filtering out repositories containing keywords (eg. example, sample and template). This avoids polished or minimal examples and focuses the selection on actual user-written documents. To ensure the repositories are substantial enough for testing, only projects between 500KB and 25MB in size were included. The lower limit excludes nearly empty repositories, while the upper limit helps manage disk usage during repeated transformations. While this filtering is necessary

to make the evaluation practical, it may bias the dataset toward structurally simpler or better-formed projects.

Due to restrictions of the GitHub API, each search query returns at most 1000 repositories. From this set, 479 unique repositories met the initial conditions. Of these, 121 were discarded because they use relative figure paths involving the `graphicspath` macro, which cannot be resolved by the system(section 6.4). Another 34 were excluded because they either fail to compile or use image paths without file extensions (section 6.4). After filtering, 324 repositories remain and form the final test set. For each of these, the main file was identified by searching for a document containing the `\documentclass` command. This file later serves as the entry point for the transformation process.

Since the most important task of FlexiTEX is to preserve the content of the document, the tool is evaluated based on its ability to maintain that content across transformations. This is done by applying a chain of n transformation steps and then comparing the similarity between the original input and the final output. Each step uses a different configuration and the output of one becomes the input for the next. fig. 5 illustrates this process. The test runs on every repository with five different transformation chain lengths: 1, 3, 5, 10 and 20 steps. These lengths were chosen to represent a range from minimal to extensive use of the tool, allowing evaluation of both short transformation sequences and longer chains to stress-test the system's stability.

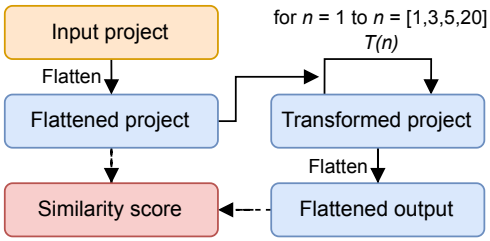

Fig. 5. Evaluation setup: the input project is flattened, transformed n times using transformation ($T(n)$) and flattened again. The similarity is then computed based on the flattened input and output.

A similarity score is then computed using Python's `SequenceMatcher` from the `difflib` module. The two files are loaded as plain text and passed to the matcher, which identifies matching blocks and measures how much of the content aligns. The `.ratio()` method returns a value between 0 and 1, with 1 meaning the files are identical. This score provides a simple way to determine the accuracy of the transformation process. In addition to the similarity score, the test also checks whether the final output contains the `\begin{document}` macro. This serves as a basic check to confirm that the transformed result still includes the document body and did not fail entirely due to a command in the preamble.

The results suggest that the transformation process tends to either succeed fully or fail entirely, with little middle ground. As shown in fig. 6, most projects either achieve very high similarity or drop to the lower end of the scale. In total, 39.2% of outputs achieved a similarity score of at least 95%, while 66.7% scored above 80%. Only 1.2% resulted is a perfect match, this is mainly because figures always get moved to the figure folder which then results in a different input parameter for the `includegraphics` command, so when a project contains any figures it cannot be a perfect match. The lower-scoring outputs often fail to include the document body, which points to issues in the preamble. This pattern suggests that certain commands are not handled correctly and break the transformation completely, leading to an unusable result. When those issues are absent, the content is typically preserved quite correctly.

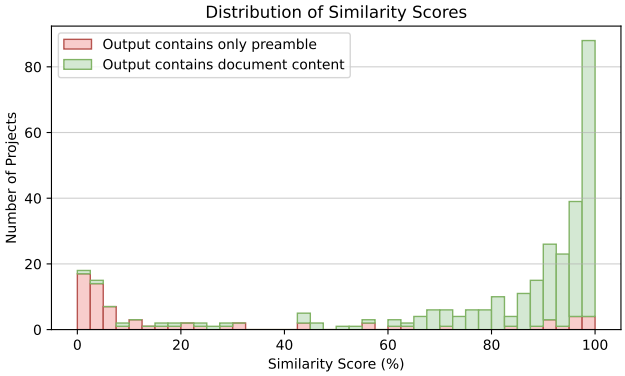

Fig. 6. Histogram of similarity scores for 20-step transformation chain.

To understand whether document length influences these outcomes, fig. 7 plots the similarity scores from the longest chain (20 configurations) against the number of lines in the original input file. There is a slight downward trend, but the number of data points in the higher line ranges is limited, so it's difficult to draw strong conclusions. In practice, most LATEX documents don't accumulate thousands of lines in a single file since paragraphs are often only one or two lines long. Overall, failures are more closely tied to structural or formatting difficulties that are often in the preamble and not directly to the raw size of the input.

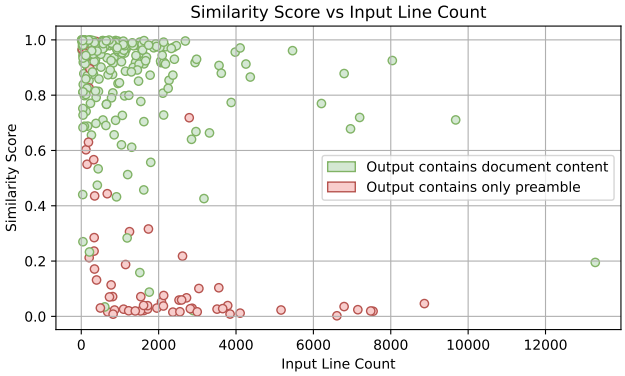

Fig. 7. Similarity score vs input file line count for 20-step transformation chains.

Finally, fig. 8 tracks how the median similarity score changes with the number of transformations. The score remains high across

all chain lengths, dropping only slightly as the number of steps increases. This gradual decline indicates that even repeated restructuring does not significantly degrade the final output. However, the observed drop is primarily due to cases where an error in the first transformation causes a command to be omitted or misrepresented, leading to incorrect code that then propagates through subsequent transformations. This can result in the gradual intensification of the problem, eventually causing entire code blocks to be removed or treated as plain text.

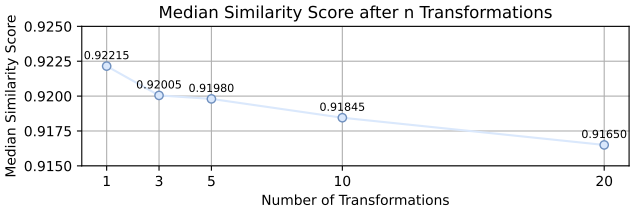

Fig. 8. Median similarity score per transformation chain length. Slight decline with longer chains suggests minor cumulative effects.

These results show that FlexiTEX performs well in many cases, especially when documents use common macros and follow typical structural conventions. However, the evaluation also reveals a significant limitation: the system does not consistently preserve enough of the original content to be relied on in real-world workflows. Projects that fail tend to do so completely, often producing broken output or omitting the document body altogether. These failures are usually caused by parser limitations, such as unrecognized or misclassified macros in the preamble.

As a result, the system can currently only be considered as proof of concept. While the underlying approach is promising, further improvements to the parsing logice are necessary before it can reliably support collaboration at scale. The core architecture appears robust, but its success depends on extending the system's ability to accurately handle the diversity of real-world LATEX usage.

8 CONCLUSION

This research project set out to make LATEX collaboration easier without forcing users to give up their personal project structure. Instead of requiring a common structure it explored how structural information could be separated from physical organization so that each user could work in their own way while still contributing to a shared result.

Research Question 1 asked how the logical structure of a LATEX project could be represented independently of its project structure The answer lies in modelling the document as a tree that captures the hierarchy of macros environments comments and text blocks. This abstract structure reflects how content is organized logically and does not depend on how the project is split into files. It makes it possible to treat every project in a uniform way regardless of structure or naming conventions.

Then, Research Question 2 considered how user-defined project structures could be expressed through configuration. This was addressed by designing a rule format that targets specific structural elements and describes when and how to split them into files. File names can include placeholders that reflect the position of each element in the tree. Conditions can also be attached to rules so that splitting decisions can depend on content length or width.

Finally, Research Question 3 focused on how reliable transformations could be performed between project structures using the shared abstract structure. The system solves this through a multistage pipeline that parses the input project builds the abstract representation applies the configured rules and reconstructs the output files. By ensuring that the transformation is reversible and idempotent and that compilability is preserved the system supports back-and-forth conversion between user styles without loss of content.

Together these answers show that collaborative editing on LATEX projects can be made more flexible by separating structure from physical organization. The implementation demonstrates that this approach works in practice for many real-world projects although it also reveals technical limitations that need to be addressed. Still the result is a system that allows each user to work in their own style while contributing to the same shared content.

9 FUTURE WORK

As shown in the evaluation the current system does not handle all documents reliably. Most failures can be traced back to limitations in the parser which struggles with unknown or user-defined macros. Future work could explore more dynamic parsing strategies that use structural heuristics to guess missing commands or recover from unknown syntax. Another direction is to replace the current parser with a parser that better understands LATEX structure and can track user-defined macros during processing. These improvements would help the system handle a broader range of documents and make the transformation more reliable.

Another promising direction builds on the abstract representation already used in the system. Since the system constructs an abstract syntax tree (AST), it could be extended with a differencing step that compares structural versions of the document. This would enable more robust parallel editing by detecting whether users have modified the same part of the tree or worked in separate subtrees. When conflicts do occur, the system could provide more informative merge output by highlighting changes within the document structure rather than only at the line level.

Similar ideas have been explored in the context of source code. GumTree, introduced by Falleri et al. [5], computes differences between ASTs instead of lines of text, enabling precise change detection. RefactoringMiner builds on this approach by extracting detailed information about structural changes in Git histories, detecting edits and classifying them based on the type of refactoring performed [2]. Another tool, RefDetect, demonstrates techniques for detecting refactorings in multiple programming languages [13], which could inspire similar strategies for structural differencing in LATEX. Incorporating these ideas into FlexiTEX would make it possible to support parallel editing more effectively, reduce merge conflicts and provide users with meaningful structural insights into how their documents change over time.

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA.

[2] Pouria Alikhanifard and Nikolaos Tsantalis. 2025. A Novel Refactoring and Semantic Aware Abstract Syntax Tree Differencing Tool and a Benchmark for Evaluating the Accuracy of Diff Tools. *ACM Transactions on Software Engineering and Methodology* 34, 2 (Jan. 2025), 1–63. https://doi.org/10.1145/3696002

[3] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2008. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Science of Computer Programming* 72, 1 (June 2008), 52–70. https://doi.org/10.1016/j.scico.2007.11.003

[4] François Brischoux and Pierre Legagneux. 2009. Don't Format Manuscripts. *The Scientist* 23, 7 (2009), 24. https://www.the-scientist.com/dont-format-manuscripts-44040

[5] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-Grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (Automated Software Engineering)*, Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher (Eds.). ACM, New York, NY, USA, 313–324. https://doi.org/10.1145/2642937.2642982

[6] Kevin Smith. 2007. *PlasTeX.* PlasTeX. https://github.com/plastex/plastex

[7] Markus Knauff and Jelica Nejasmic. 2014. An Efficiency Comparison of Document Preparation Systems Used in Academic Research and Development. *PLOS ONE* 9, 12 (Dec. 2014), 1–12. https://doi.org/10.1371/journal.pone.0115069

[8] Donald E. Knuth and Duane Bibby. 1986. *The Computers & Typesetting, Vol. A: The Texbook.* Addison-Wesley Longman Publishing Co., Inc., USA.

[9] Leslie Lamport. 2024. My Writings. https://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.pdf

[10] John MacFarlane, Albert Krewinkel, and Jesse Rosenthal. 2006. *Pandoc.* Pandoc. https://github.com/jgm/pandoc

[11] Microsoft. 2018. *Microsoft/Live-Share.* Microsoft. https://github.com/microsoft/live-share

[12] Bruce R. Miller and Deyan Ginev. 2004. *LaTeXML.* National Institute of Standards and Technology. https://math.nist.gov/~BMiller/LaTeXML/

[13] Iman Hemati Moghadam, Mel Ó Cinnéide, Faezeh Zarepour, and Mohamad Aref Jahanmir. 2021. RefDetect: A Multi-Language Refactoring Detection Tool Based on String Alignment. *IEEE Access* 9 (2021), 86698–86727. https://doi.org/10.1109/ACCESS.2021.3086689

[14] Overleaf. 2019. Sections and Chapters. https://www.overleaf.com/learn/latex/Sections_and_chapters

[15] Overleaf. 2025. Can Multiple Authors Edit the Same File at the Same Time? https://www.overleaf.com/learn/how-to/Can_multiple_authors_edit_the_same_file_at_the_same_time%3f

[16] Philippe Faist. 2015. *Pylatexenc.* pylatexenc. https://github.com/phfaist/pylatexenc

[17] Wouter ten Brinke. 2025. *FlexiTeX.* https://github.com/wtb04/FlexiTeX

[18] Wouter ten Brinke. 2025. *FlexiTeX Example.* https://github.com/wtb04/FlexiTeX-Example

[19] Pepijn Visser. 2023. xBib : The Language Design and Implementation of a Transformation Language. https://essay.utwente.nl/94379/

[20] Google Workspace. 2025. Collaborate With Real-Time Editing. https://workspace.google.com/resources/real-time-editing/

[21] Vadim Zaytsev. 2017. Language Design with Intent. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS).* IEEE, 45–52. https://doi.org/10.1109/MODELS.2017.16