SwiftLaTeX- Exploring Web-based True WYSIWYG Editing for Digital Publishing

Elliott Wen
The University of Auckland
jq.elliott.wen@gmail.com

Gerald Weber
The University of Auckland
g.weber@auckland.ac.nz

ABSTRACT

The text processing tool LaTeX has prevailed as a standard in many fields of exact sciences; it is evident that LaTeX is likely to be here to stay. From that perspective, it is important to explore what are the best possible ways to support the author in efficiently editing documents. There have been several approaches that provide graphical editing support for LaTeX. We argue that a true WYSIWYG (What You See Is What You Get) approach is a justified requirement for future systems and we present here the first cloud-based true WYSIWYG editor. This allows the author to edit the document in its print form directly in a web-based PDF viewer. Building such a system creates unique challenges compared to existing approaches. We identify these challenges and name workable solutions. We also provide a usability evaluation of the new system. In short our finding is that editing LaTeX directly in the PDF view is possible for a wide range of edits and valuable for many major user groups and use cases; hence it is a fair requirement for future top-of-the-line LaTeX editors.

CCS CONCEPTS

• **Human-centered computing** → **Graphical user interfaces**; **Text input**;

1 INTRODUCTION

The TeX typesetting system developed by Donald Knuth ushered in an era of high-quality open-source electronic document publishing. Knuth embedded deep knowledge of the traditional art and craft of typesetting into TeX.Over the intervening decades, various TeX derivatives have established themselves. Among them, LaTeX is particularly widespread and maintained by a large community.

However, the TeX family of tools exhibits a major usability limitation due to their nature as batch processing systems which results in an edit-compile-review cycle. This cycle is generally acknowledged to possess certain usability disadvantages; the user is forced to perform difficult mental mappings back and forth between the one-dimensional textual TeX source codes and the two-dimensional graphical print output, displayed on the screen only after a slow compilation process [7]. This takes longer than the time which users are known to find acceptable in interactive systems. It renders various editing tasks fairly inefficient such as proofreading, which often involves a large share of single-character edits.

A number of partial solutions to this problem have been developed. Some LaTeX editors (e.g., ShareLatex [25], now merged with Overleaf [27]) provide asynchronous and regular refresh of the print output, which is placed side by side with the code editor. Optionally, the editor also features formatted text input, where a fixed number of LaTeX text style commands are displayed in different colors and fonts accordingly, e.g., section headers are shown in a large and bold font. However in these approaches there is still a large discrepancy between the edit view and the print layout view and the user has to alternate and switch focus between them in many work cycles.

A thorough analysis of these existing solutions (see Section 2) leads us to the research hypothesis that enabling WYSIWYG editing of LaTeX documents directly in the print (PDF) layout will bring about a significant improvement in usability for various user groups due to the removal of the notorious edit-compile-review cycle. To match this aspiration, and to keep the usability advantages gained with current cloud services, we introduce SwiftLaTeX, the first cloud-based LaTeX true WYSIWYG system for.

However, implementing a responsive WYSIWYG editor entails three challenges. First, allowing direct editing of the PDF document requires that the product of the LaTeX compilation must be combined with the new user input as quickly as possible. This is challenging since traditional recompiling is too slow for character-wise quick response. Secondly, the editor is expected to intercept the modification on the PDF document and to apply the alternation synchronously in the corresponding position of the source code. To achieve this, the editor requires the ability to infer the source code position of each element in the PDF with character-wise accuracy. This is a stretch target since the highest level of accuracy in existing open-source tools is merely line-level. Finally, for a cloud-based system with a massive number of potential users, maintaining high scalability, reliability and compatibility remains a challenging issue, particularly since as long as we rely on repeated compilations, there is an obvious tradeoff between computing resource consumption and responsiveness.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DocEng '18, August 28–31, 2018, Halifax, NS, Canada
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5769-2/18/08…$15.00
https://doi.org/10.1145/3209280.3209522

In this paper, we present practical solutions to cope with the above challenges. Specifically, to correctly combine the product of the LaTeX compilation and the user's editing, SwiftLaTeX proposes an asynchronous merging mechanism. To infer the source code positions, SwiftLaTeX explores an advanced text-matching algorithm and dynamically patches the LaTeX engine with a bookkeeping mechanism in pursuit of character-level accuracy. To achieve high scalability, reliability and compatibility, SwiftLaTeX embraces a distributed system back-end architecture and features a responsive front-end design.

We consolidated the above techniques and implemented a prototype of SwiftLaTeX on cloud computing platforms, based on which we conducted a preliminary user study involving six LaTeX users to evaluate the usability aspects of the system. The evaluation results show that participants reacted positively to the innovative web-based editable PDF viewer and praised its efficiency for certain editing tasks such as proofreading.

To sum up, the main contributions of this paper are as follows.

(1) We provide a review of existing LaTeX editors in Section 2 and derive requirements for our system as a cloud-based WYSIWYG editor in a narrower sense in Section 3.
(2) We propose a high-level algorithmic standard architecture for asynchronous WYSIWYG editors working on a batch-style text processing system in Section 4.
(3) We implement and integrate a fine-grained source code positioning system in LaTeX system in Section 5.
(4) We present a working cloud-based prototype which is scalable, reliable and compatible with mainstream browsers on different kinds of devices in Section 6.
(5) We conduct a preliminary pilot study evaluating usability aspects of the system in Section 7.

2 BACKGROUND OF THE LaTeX FAMILY OF SYSTEMS

2.1 Factors influencing the future of the TeX family

Donald Knuth's program TeX is one of the few notable programs developed back in the 1970s that are still in use today. There are several factors that make the continued use and significance of TeX and its derivative LaTeX very plausible in the foreseeable future:

Excellent Output Quality: LaTeX with its built-in typesetting expertise, manages to generate output considered by many to be unsurpassed in quality.

Technical Stability: The stability of LaTeX has been proven by its intensive use by millions of users with demanding input.

Format Stability: Since the minor switch to LaTeX 2e in 1994, the central component of the LaTeX engine has not undergone any incompatible version changes, which ensures that old source packages continue to work.

Strong Maintenance Support from Community: A large and stable community is maintaining the system in a self-organized fashion, which is nurturing the technical innovation for LaTeX.

Community Investment: Many communities using LaTeX are heavily invested in it in the sense that many of the workflows and assets in the shape of LaTeX stylesheets are locked-in to LaTeX.

2.2 Systematics of LaTeX Editing Systems

Over the past decade, various editing systems have been developed to make LaTeX editing more convenient. They can mainly be divided into two categories, standalone versus web-based, or cloud-based.

Standalone: The most rudimentary standalone systems are command line based applications (e.g., Latexmk [9] and LatexRun [2]), which commonly serve as automated compilation tools. Given a set of source files, these tools issue the appropriate sequence of commands to communicate with the typesetting engines and generate the resulting document. Generally, the command line programs are rarely invoked directly due to debugging difficulty. Instead, they are commonly utilized as the backend for the graphical editors.

Integrated Development Environments (IDEs): Graphical editor applications (e.g., TeXstudio [32], TeXworks, TeXnicCenter and TeXshop [14]) aim to facilitate document editing by providing a graphical user interface (GUI) with various features such as syntax highlighting and automated error correction. These features significantly reduce the difficulty of debugging and increase the editing productivity. Nevertheless, setting up such an editor may involve difficult and laborious configurations (e.g., installing LaTeX packages and maintaining software dependency), which is not user-friendly for novice users. This gap opens opportunities for the emergence of web-based editing systems.

Desktop WYSIWYG editors: Desktop WYSIWYG editors work in a very similar way to office tools: The open-source tool LyX [13] acknowledges that the edit view does not reflect the print output and calls the approach WYSIWYM (what you see is what you Mean). The Bakoma TeX tool [29] is a WYSIWYG editor in the narrower sense.

Web-based editors: Compared with the standalone editors, the web-based editing systems possess a distinct advantage, namely zero set-up time. Specifically, these web-based editing systems run instantly on at least some mainstream browsers and require no software installation on the users' devices. In other words, web-based systems make LaTeX rapidly and easily accessible in the user's web browsers. Another merit of the web-based system is cloud storage, which enables project files to be accessed anywhere in the world and greatly facilitates collaboration. These advantages make web-based systems fairly prominent; the most popular platform, Overleaf now has over 6 million users and hosts over 10 million projects [26]. It is therefore not far-fetched to assume continuing growth of web-based LaTeX editing and argue

that any innovation on web-based, cloud-based LaTeX editing systems is making a substantial contribution to digital publishing in the 21st century.

2.3 Limitations of Current Editing Systems

However, despite the invaluable features of the TeX family, they possess major limitations, namely a steep learning curve, and the batch-style text processing as well as the lack of structure in LaTeX scripts. Specifically, it is generally accepted that the script language of LaTeX poses a substantial technical hurdle for novice users [12]. The batch processing nature further aggravates this problem; the necessary edit-compile-review cycle is generally acknowledged to possess certain usability disadvantages. First, it further steepens the learning curve for beginners who have to wait for the time-consuming compilation process to finish before they can examine whether the resulting document is correct. If not, they have to go back to the editor and pinpoint which line of the source code is causing the problem. The demanding debugging-like process tends to cause a sense of confusion and frustration. More importantly, users are now forced to carry out intensive mental mappings back and forth between the essentially one-dimensional textual TeX source codes and the two-dimensional graphical print output [7]. It can make certain editing tasks such as single-character edits very inefficient.

A number of partial solutions have been developed to solve this problem. The most widely-used one utilized by web-based systems is to provide asynchronous and regular refresh of the resulting document, which is placed side by side with the code editor. The editor in Overleaf is referred to as the 'Rich Text' feature, where the source code is styled in different colors and fonts accordingly to the categories of terms. Still, there exists a strong visual disagreement between the source and compiled document and the user has to alternate and switch focus between them in many cases.

Accordingly we aim to take the term WYSIWYG literally, which naturally raises the research question: how much improvement of usability would true WYSIWYG (i.e., enabling direct editing in the print form) bring about due to the removal of the edit-compile-review cycle. We will first clarify the definition of the term in a narrower sense in Section 3. We then present our research findings during the implementation of SwiftLaTeX, which is the first web-based, cloud-based WYSIWYG LaTeX editing system. We report the challenges we encountered and provide the corresponding open-source solutions.

3 USABILITY REQUIREMENTS ANALYSIS FOR A WYSIWYG EDITOR

The core usability question that we are focusing on in this work is the notion of a WYSIWYG editor. The term is often used in a loose manner. However there seems widespread consensus that "Rich Text" views, as for example used currently in Overleaf, should be kept in a separate category. In this section, we provide a usability requirement analysis of WYSIWYG editing for LaTeX in a narrower sense. Note that since we focus on this question here, we consider here other important usability questions such as collaboration/project management to be out of scope.

3.1 WYSIWYG in the narrower sense

For the following discussions we introduce an important definition, the notion of *editing quiescence*. Editing quiescence is a point in time during use of the editor when the editor has processed all previous edits of the user, and there are hence no pending edits that would further change the output. With this definition we can in turn provide the following definition: A WYSIWYG editor in the narrower sense is an editor fulfilling the following requirements:

(1) At editing quiescence, the editor shows the print layout of the document, i.e., acts as a faithful print viewer.
(2) At editing quiescence, the user can position the cursor anywhere in the document with the mouse, arrow keys or a combination thereof.
(3) The user can perform edits at the cursor position by simply typing the keys or backspace and get immediate feedback in the sense that the editor shows a *preview version* of the print view.
(4) If the user input pauses, the editor reaches editing quiescence automatically in a reasonable amount of time (which can, however, be as long as a LaTeX compile cycle).

Requirement 4 together with 1 means that the editor replaces the possible preview version with the faithful print view. The requirement 1 is the one characterizing WYSIWYG in the narrower sense; we argue that this is the actual literal translation of the acronym. A surprising fact is that the editor created in this project is apparently one of the few cloud-based WYSIWYG editors that is available at all in the narrower sense, not only in the LaTeX domain. Requirement 3 is asking for the following replacement capability: all text that is visible in the resulting PDF document (including the basic text in formulae) is editable. For instance, a formula $\frac{4}{cx}$ in the PDF document can be modified to $\frac{4}{ct}$ instantaneously and in place in the PDF.

In contrast, many WYSIWYG editors that are part of office suites do not fully meet Requirement 1. They offer faithful print views only in a non-editable form and rather offer editable WYSIWYG views in a weaker sense: The editable view might represent all text styles, including font changes and tabulation, but the edit view is notably different from the print view and can include incorrect word wraps or even page breaks. The community of the system LyX makes the case that this can also be seen as a feature and has introduced the term WYSIWYM (what you see is what you mean) to characterize such an editor. We argue that there is a clear need for WYSIWYG editing: It is commonly acknowledged that in particular final edits in a close-to-print document

take a disproportionate amount of time in a non-WYSIWYG editor due to the turnaround between edit view and print view. The Bakoma TEX system offers WYSIWYG for a non-cloud based setting.

An even weaker form of a formatted editor should not be included in the WYSIWYG paradigm and instead should be referred to as Rich Text View. In a rich text view for LATEX only a fixed number of text style commands are translated into formatted representation. The classically used text styles include headers, bold, italic, typewriter, bullet-lists and enumerations. In a LATEX rich text viewer, only the commands corresponding to these features are hidden and the editable text is formatted accordingly. The difference from a weak WYSIWYG editor is that the formatted representation does not change with stylesheet changes, i.e., if a different font family is chosen, or a subsection is rendered with small-caps. In a classic rich text view, subsections are always rendered with the same header style.

4 CHALLENGES IN ASYNCHRONOUS COMPILATION FOR A WYSIWYG EDITOR

The aim of allowing direct editing of the PDF document creates completely novel system challenges not faced in current asynchronous cloud-based systems, namely that the product of the LATEX compilation process must be combined with the current user edit.

SwiftLaTeX offers two different editing views. The first one is the source view. This is inherited from the conventional cloud-based systems, which allow advanced users to directly manipulate LATEX source code in a classical ASCII editor and to preview the PDF on a separate viewer. The source view needs to be preserved because only in source code can arbitrary LATEX scripting be done.

Nevertheless, the main contribution of SwiftLaTeX is the second view, the WYSIWYG view. This is an editable PDF viewer that allows the user to directly edit a document in its print form, but with effect on the source. To start an edit, the user simply clicks on any visible text that should be modified. A text cursor will appear indicating that the viewer is now ready for the user's input as shown in Fig 1 (a). Afterwards, all editing operations such as character appending or removal will instantly be rendered on the viewer. As mentioned in the requirement 3 in Section 3, the instant rendering is a *preview version* of the PDF. Meanwhile, to retain the modification, the user's editing operations will also be applied at the corresponding position of the LATEX source code. After the re-compilation process, the user will automatically be presented with the new *revised version* of PDF.

To provide an authentic experience of WYSIWYG editing, SwiftLaTeX puts considerable effort into generating a preview version that possesses minimal difference from the corresponding revised version as depicted in Fig 1 (b) and Fig 1 (c). This is mainly achieved by mimicking the typesetting behavior used in the LATEX engine. For instance, when a character is being appended, it will automatically inherit the

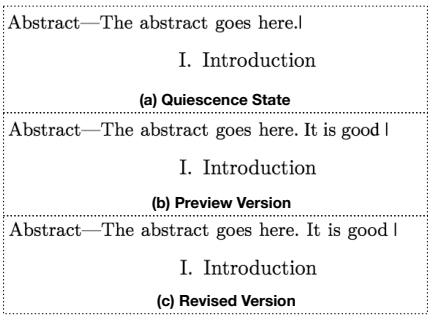

Figure 1: Three Editing States in the WYSIWYG View.

font settings from the previous characters to make itself visually agreeable. Though the typesetting imitation approach is feasible for minor editing, it possesses certain limits when dealing with lengthy edits. As a consequence, without any further precautions, the difference between the previewing version and the revised version would gradually accumulate along with the user's input.

To address this issue, one natural idea is to compile the source periodically, and base the preview version on the newly-obtained revised version, such that the accumulated difference is cut back and remains unnoticeable most of the time. However, implementing such a mechanism entails a challenge, which is mainly attributed to the nature of asynchronous communication, i.e., the fact that the user can keep on editing while a new version is already in the process of being compiled. We depict the predicament in Fig 2 (a) using a timing chart. The chart contains two separate timelines. The topmost timeline indicates the period which the user spends on editing. Likewise, the other timeline shows the timespan that the background compilation process consumes. It can be observed that at the moment t_1, a background compilation request is initiated. Meanwhile, since SwiftLaTeX is asynchronous, the user continues his editing regardless of the status of the background compilation process. At the moment t_2, the compilation is finished and the revised PDF is available for display. However, it should be noted that this revised PDF is an output of the source code at the moment t_1. It does not reflect the user's inputs within the timespan between t_1 and t_2, which are referred to as the *uncommitted edit*. As a consequence, directly displaying the newly-obtained revised version would erase the user's uncommitted edit from the screen, and that would obviously be inappropriate because the user's editing process would be severely disrupted.

One naive way to bypass this issue would be to take advantage of the user's pauses in editing as shown in Fig 2 (b). Specifically, it is not uncommon that a user occasionally takes a break from typing. If such a break is detected, a background compilation would then be initiated. If the compilation is finished before the user restarts his editing, the resulting PDF then can be safely displayed on the screen. This approach essentially uses the aforementioned concept

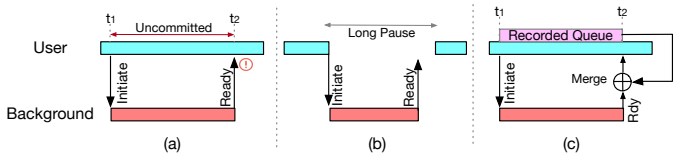

Figure 2: Timing Charts in the WYSIWYG View.

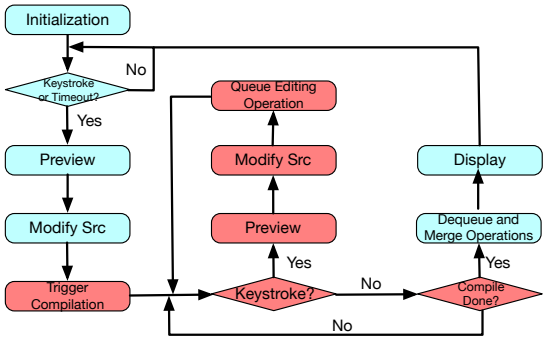

Figure 3: Workflow for the WYSIWYG view.

of editing quiescence as a way to achieve consecutive safe updates of the PDF that do not interfere with the user input. Nevertheless, this method is prone to starvation (i.e., long delay of the compilation process) if the user is typing swiftly and continuously for a long time.

Instead, SwiftLaTeX adopts a more sophisticated approach which does not rely on the user's pauses. This is depicted in Fig 2 (c); the revised PDF, before being displayed in the viewer, is 'merged' with the uncommitted edit so that the user's editing process will not be disrupted. This can be achieved as follows. First, when the background request is initiated at t_1, our system starts recording all the editing operations in a queue data structure. When the revised PDF is available at t_2, all the operations are dequeued and replayed on the revised the PDF sequentially. This generates a new preview PDF, which derives from the just-received revised PDF by adding the uncommitted edit. Thus it is ready to be displayed in the viewer. The cursor is repositioned at the correct logical position in this new preview PDF and the user can continue editing without interruption.

We summarize the aforementioned design with a flowchart shown in Fig 3, where the blocks with ongoing compilation process are colored in red. A new compilation process will be triggered whenever either a keystroke or a timeout event (i.e., periodically trigger) is detected. This ensures that the users can obtain the latest revised version as soon as possible.

5 ACHIEVING FINE-GRAINED SOURCE CODE POSITIONING

A WYSIWYG editor that synchronously changes the source code requires a very finegrained ability to reconstruct the source code position of each element in the PDF.

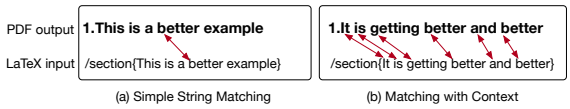

Figure 4: Two typical examples for source code position inference.

5.1 Inferring Source Code Positions

As a WYSIWYG editor, SwiftLaTeX intercepts the modification on PDF elements (e.g., words or equations) and directly applies the corresponding alternation on the source code. To achieve this, SwiftLaTeX is required to infer the source code position of the modified PDF elements. Such an ability is commonly referred to as LaTeX input-output synchronization, and this has been explored by a LaTeX plugin named SyncTeX.

SyncTeX is designed for conventional typesetting editors, which usually incorporate two views; one for entering LaTeX source code, the other for viewing the resulting PDF. Generally, the source code and PDF are too long to fit a window and thus only certain parts of the contents are visible to a user. To facilitate the user's editing process, the two windows are required to be synchronized such that they are displaying roughly the same part of the document. Solely taking this use case into account, the author of SyncTeX deems it sufficient to provide line-level accuracy, which allows mapping from a PDF element to a whole line in input files [15]. This coarse-grained accuracy is unable to meet the requirement of SwiftLaTeX. To address this issue and to boost the synchronization accuracy from line-level to at least word-level, we have developed and tested two different approaches: first, we have developed an advanced text-matching approach. Secondly, we have enhanced the TeXengine to achieve on demand even character-wise synchronization.

5.1.1 Text-Matching Approach. For a better understanding of the text matching approach, Figure 4 depicts two typical cases, each of which contains one line of LaTeX source code and its corresponding PDF output obtained by SyncTeX.

Consider the first case in Fig 4 (a) where we wish to infer the source code position of the word 'better' in PDF. A straightforward solution is to run string matching algorithms on the line of source code, which reveals the correct position. However, this approach may cause ambiguity if the word 'better' occurs more than once in a sentence as shown in Fig 4 (b). This exception naturally motivates us to take the surrounding words (i.e., context) of the target into consideration, more specifically, matching a sequence of words instead of an individual target. This serves as the intuition of our approach to solve the source code position inference problem formulated as follows.

Given two character arrays L and P, which represent a line of LaTeX source codes and its corresponding PDF output string respectively. Besides, L_j denotes the j-th character in the array L and $1 \leq j \leq len(L)$. Likewise, P_k represents

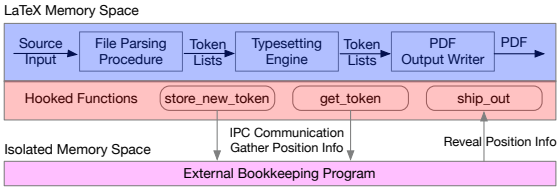

Figure 5: Internal Mechanism of the Engine Patch.

the k-th character in the array P and $1 \leq k \leq len(P)$. A matching f between L and P is defined as a function that satisfies

$$P_k = L_{f(k)} \qquad \forall k \text{ in } [1, len(P)]. \tag{1}$$

Here we safely assume every character in P has its matching on L. If there exists no matching for a specific character (e.g., the numbering characters in section header), we simply mark it as uneditable and remove it before proceeding.

Meanwhile, we say a matching is valid if it is monotonic, which can be illustrated as follows.

$$f(i) < f(j) \quad \text{iff } 1 \leq i < j \leq len(P). \tag{2}$$

This property guarantees that the order of the characters in P is also preserved in L.

Multiple matching may occur and we denote the fittest matching f_b as the one which satisfies the following requirement:

$$f_b = \arg\min \sum_{2}^{len(L)} (f(k) - f(k-1)), \tag{3}$$

where f represents a valid matching. This requirement contains a simple intuition; for two consecutive characters in P, the distance between their matching characters in L is also minimized under the fittest matching f_b.

It can now be easily seen that the answer of the source code position inference problem happens to be the fittest matching f_b.

It is not difficult to obtain this mapping even in a brute-force manner considering the search space has already been greatly limited in a line of source codes. Moreover, the search space can be further reduced by certain optimizations. For instance, the string matching algorithm can be first utilized on words that only occur once to immediately confirm their matching.

Nevertheless, such an approach is still prone to matching failure. The main culprit derives from the precision of the SyncTEX, which is ±1 line [15] in most cases. The precision significantly degrades when the element is generated by a LaTeX macro (e.g., math equations or section headers). Clearly, the incorrect line number negatively impacts the reliability of our approach.

The unstable technique now forces us to develop a more sophisticated approach, which increases the synchronization accuracy from the internal perspective of LaTeX engines.

5.1.2 Enhancing TEX Engine. Existing open-source LaTeX engines possess no mechanism for tracking down the source code positions for each PDF element. In this work, we make the first attempt to enhance the engines with a bookkeeping mechanism that provides character-level synchronization accuracy.

The main challenge of implementing such a functionality lies in how to minimize the potential interference which our modification creates to the engines. Specifically, the original implementation of the TEX engine is generally considered very complicated code that is difficult to modify in a safe way. Directly patching this source code to alter the internal data structures or execution logic is considered not advisable from a standpoint of software reliability engineering, because an under-optimized patch is likely to result in unheeded bugs. Therefore, to avoid undermining the reliability of LaTeX engines, we adopt a neutral observer design in our patch as depicted in Fig 5.

It can be seen that the patch is partitioned into two essential parts. The first one is an external bookkeeping program. This runs in its own memory space and passively receives the position-related information from the LaTeX engines in file parsing and typesetting phrases. The information is obtained by the second part of our patch, which is a dynamic hooking library. In short, hooking is the process of intercepting a program's execution at a specific point, typically entries of functions, in order to augment the program's behavior. The binaries of the augmented functions are placed on a dynamic hooking library, which can be injected into LaTeX engines' memory space via special instructions to program linkers on the run-time. The merit of the hooking technique is its unintrusiveness; neither modification nor re-compilation of the original source codes is ever required. Moreover, the augment functions in our patch does not alter the memory content of the LaTeX process. The neutral observer design ensures the reliability is untarnished.

For better understanding of how the bookkeeping patch keeps track of the source code positions, it is reasonable to first introduce the procedure that LaTeX engines utilize to parse input files. Briefly speaking, LaTeX digests the input files character by character. These characters are generally stored and organized in a data structure named *token lists*, which are essentially linked lists. A token list can represent a character array to be printed or a parameter for a macro call (e.g., 'Paper' in \title{Paper}). Depending on the implementation of the macros, a parameter token lists can be moved around in the memory by being duplicated and removed several times before they are finally formatted and printed on PDF.

Therefore, to establish a mapping between the source code positions in input files and their output elements in PDFs, two essential steps are involved. The first step is to build and attach a position record for every constructed token list when processing input files. The immediate step is to ensure the position record can be propagated to its newly duplicated token lists.

To implement the two steps, at first glance it would appear unavoidable to modify the data structure of the token list such that it can store the source code position. However, our patch is able to avoid this by adopting a significantly

different approach. Specifically, the patch hooks two functions including 1) *store_new_token()* and 2) *get_token()*. The former function is used to dynamically allocate the memory space for a new token being appended to a token list. The latter function, on the other hand, may be used to retrieve a token from memory for duplication. These functions are now augmented such that the memory addresses and contents of the newly allocated token or the recently retrieved token will be delivered to the external bookkeeping program via inter-process communication channels. Meanwhile, if a token is constructed when parsing input files, the corresponding source code position is also delivered.

By monitoring the memory data received, the external program now can keep track of every token list and its associated position record. The maintained information will then be utilized in the hooked *ship_out* function at the PDF output stage. Specifically, if a token list is being printed on PDF, the external program will simultaneously reveal the associated position information, which may be stored in the following approaches.

5.2 Storing Position Information

With the enhanced LaTeX engine, every PDF element is now associated with its source code position. The remaining question lies on how to store the position information such that it can be easily retrieved by a viewer.

A first solution is to follow SyncTeX, which takes advantage of an auxiliary file. Specifically, the file serves as a dictionary which maps the coordination of PDF elements to their line numbers; whenever a user clicks a specific PDF element on the viewer, the mouse coordinates can be checked against the auxiliary file to obtain the corresponding line number. Clearly, the principle can be easily adapted in our work. However, one drawback of this solution stems from the specifically-designed auxiliary file, which can only be parsed by a unique external program [16]. It negatively impacts the portability of PDF files.

Alternatively, we propose a novel approach which directly embeds the position meta information to the PDF files. The merit of this approach is that it allows viewers to retrieve the positions without the auxiliary file and external program. Nevertheless, the implementation entails an inevitable challenge that the PDF specifications do not take synchronization into consideration and there exists no generally-acknowledged approach for storing the position information on a PDF file.

To tackle this issue, we explore an idea that piggybacks the position information on undocumented PDF metadata. Specifically, PDF metadata is utilized to describe properties such as the font, shape and position of a PDF element. Among them, a property named *flatness* is barely documented by the PDF specification [3]. We investigate how this property is handled via cross-checking various open-source PDF viewers such as Poppler[10] and Evince [11]. The results indicate that all of the viewers simply ignore this property when rendering a PDF on screen. In other words, the flatness property can be stored as any value without impeding the functionalities of

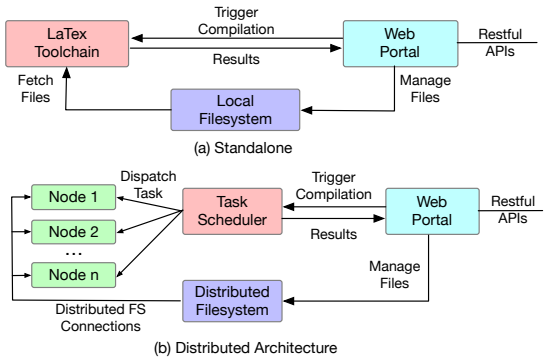

Figure 6: Backend Architectures.

the viewers. Therefore, we piggyback the position information on the flatness property. In this manner, whenever a user clicks a PDF element, its flatness property can be retrieved and utilized by a viewer to infer the source code position. Note that since almost every PDF viewer possesses graphical or programming interfaces for directly accessing the metadata, neither external program nor patch for viewers is required.

Despite its promising feature, we have to admit that the piggyback approach indeed contains a minor drawback; the size of the resulting PDF is slightly bloated due to the extra flatness property. Nonetheless, this side effect is negligible thanks to the PDF stream compression mechanism [3], which greatly reduces the redundancy of the position information. Moreover, the piggyback option can be switched off easily when the user decides to publish the final version of the PDF.

6 SYSTEM IMPLEMENTATION

In this section, we present the implementation details of SwiftLaTeX and showcase its main user interfaces.

6.1 Achieving High Scalability and Reliability

As a cloud service designed for a large number of potential users, SwiftLaTeX aims to meet demanding requirements for scalability and reliability.

To understand the challenges, let's discuss a prototype running on a single server. As shown in Fig 6 (a), the prototype contains two components, a web application and a LaTeX toolchain. The web application offers a set of restful endpoints enabling users to upload their LaTeX source code to the server. These files can be processed by the LaTeX engine to generate the resulting PDFs.

Despite the simplicity, an obvious drawback of this single-server prototype lies in limited system capacity. More specifically, the performance bottleneck can be mainly attributed to the time-consuming compilation process, and the hence limited number of compilation tasks that can be run on a single server concurrently. Another major limitation of this single-server prototype is its questionable reliability. If the prototype stores all users' files solely in its local hard disk,

then the failure of the hard disk can create a single point of failure, resulting in data loss or system unavailability.

To address these issues, SwiftLaTeX replaces this prototype with a sophisticated distributed architecture as shown in Fig 6 (b). One major enhancement is providing support for scale-out. Specifically, instead of executing all the compilation tasks on a single host, the distributed architecture now is able to dispatch different tasks to multiple hosts named *compilation nodes*. Such a design not only ensures the scalability by incorporating additional nodes, but also enables optimization of the system latency by applying scheduling algorithms on different types of tasks. Currently, SwiftLaTeX schedules the compilation tasks by adopting a Multilevel Queue approach inspired by the process scheduling in modern operating systems [28] . Briefly speaking, compilation tasks are first assigned with a priority (e.g., high, medium and low) based on the length of the source code. Afterwards, the tasks with an identical priority will be dispatched to the same compilation nodes in a first-come-first-served manner. Compared with a randomized scheduling mechanism, this scheduling algorithm is highly likely to reduce the average latency, since it alleviates the convoy effect that a time-consuming task holds other simpler tasks for a long time [17]. This also mitigates the problem that none of the existing LaTeX engines takes advantage of multi-core processing.

Also the architecture significantly increases the system reliability by utilizing distributed file systems. Specifically, files are now replicated and scattered across multiple hosts, such that despite the failure of a small number of hosts, the data would remain intact and available. Another advantage of distributed file systems is the intrinsic support for file version control. It greatly facilitates the implementation of Version Control Systems (VCS) in SwiftLaTeX, which allow users to straightforwardly manage the change of their LaTeX source codes.

SwiftLaTeX implements the above mechanisms with the help of open-source components including Redis Queue [8], GridFS [4] and Docker [19]. Specifically, Redis Queue is used to implement the task queues for each compilation node. Meanwhile, their hard drives are all connected to form the distributed file storage using GridFS. To facilitate fast deployment, the entire SwiftLaTeX is packaged into a Docker image format, which is a de facto standard used by various cloud computing platforms.

6.2 Embracing Responsive Frontend Design

To ensure the compatibility (i.e., being responsive) to mainstream browsers running on various kinds of devices such as desktops and tablets, SwiftLaTeX is solely built atop the standard front-end technologies including HTML5, CSS3 and Javascript. Nonetheless, these technologies are supported to a different extent by each browser and different browser version. The differences pose a main challenge to the frontend design, notably, rendering PDF correctly in the WYSIWYG view across various browsers.

Specifically, each browser has its own setting to control how PDFs open from a webpage. Browsers such as Internet Explorer or Opera display the PDF by invoking an external PDF reader (e.g., Adobe Reader), which is not always available. Though a small number of browsers like Chrome possess built-in PDF readers, their functionalities are commonly limited (e.g., previewing PDF in an independent browser window). It is generally difficult for SwiftLaTeX to integrate and extend them due to the lack of application programming interfaces (APIs). To tackle this issue, SwiftLaTeX explores two different approaches including a Javascript-based PDF viewer and a plugin for the LaTeX engine to generate HTMLs instead of PDFs.

The first method takes advantage of *PDF.js* [20], a Javascript library for parsing and rendering PDF atop the canvas of a browser window. Despite the straightforward setup, the library is likely to experience performance issues (e.g., slow rendering) when processing PDFs with substantial pages. It is mainly because Javascript, as an interpreting language, is not performance-oriented especially on the resource-constrained mobile devices. Moreover, PDF.js is known to have compatibility issues with certain browsers such as Internet Explorer and WebKit used in earlier versions of Android [21].

The second approach is inspired by a library named *Pdf2htmlEX* [33] that converts PDFs to HTML5 pages. The conversion is carried out in an manner that all the texts, figures, mathematical formulas and page layouts are preserved with precise fonts and locations. It ensures that the visual difference between PDF and HTML is barely noticeable. SwiftLaTeX adopts the core idea and serves HTMLs rather than PDFs to users. The essential merit lies in its high compatibility and tiny resource consumption. In particular, the HTML pages can be directly displayed by most browsers and the process only consumes a tiny amount of CPU and memory resource. It makes SwiftLaTeX more friendly to resource-constrained devices.

SwiftLaTeX consolidates the technologies mentioned above and implements a user interface as shown in Fig 7. We reuse a set of responsive open-source components, for example, Bootstrap [30] to provide a responsive layout, jQuery [5] to manage restful connections with the backend, and Ace.js [1] to provide syntax highlighting in the source view.

7 PROTOTYPE EVALUATION

To evaluate the usability of SwiftLaTeX, we invited 6 participants from the academic domain to test our system. The evaluation has been organized as a Discount Usability Test [22], which involves a small number of participants with a focus on qualitative studies on prototype design. Previous studies [24][23] have suggested that a discount usability test, even with only 5 participants, provides robust evaluation results and may discover up to 85% of the usability problems. In this section, we demonstrate the settings and outcomes of our evaluation. The preliminary test offered us many invaluable insights and feedbacks. In the near future, we plan to

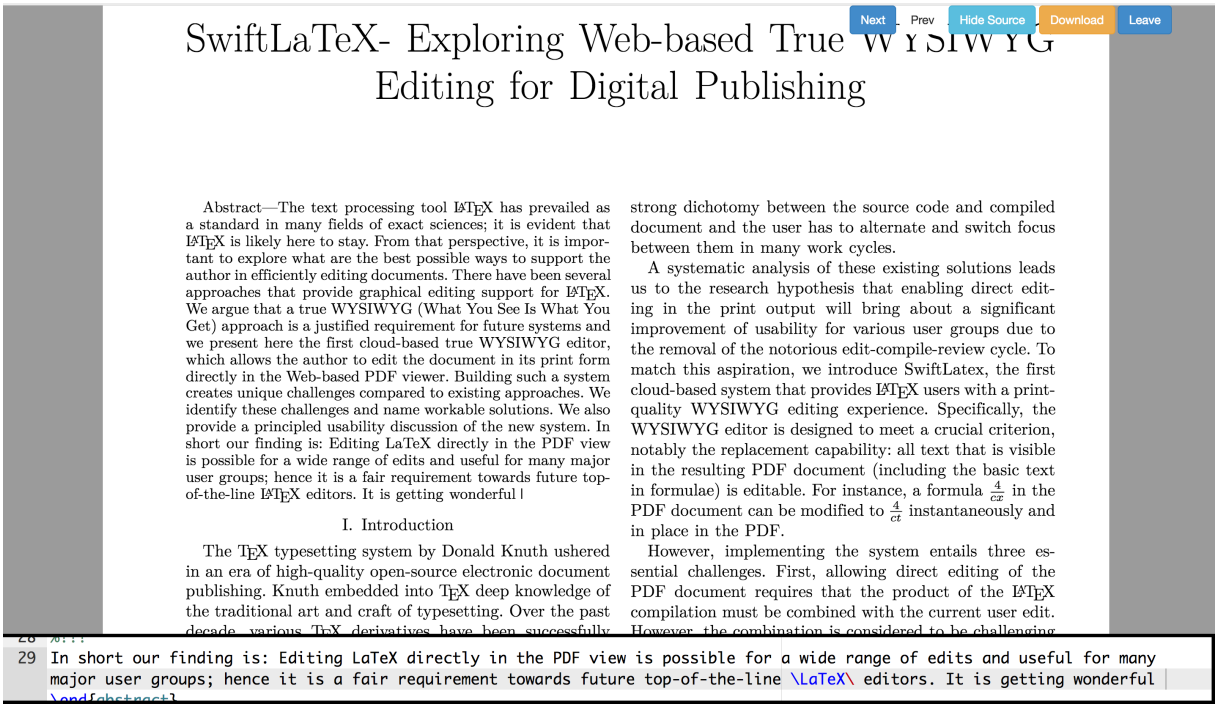

Figure 7: WYSIWYG Editor of SwiftLaTeX

perform a comprehensive usability study, which covers more participants from different domains.

7.1 Evaluation Settings

We invited 6 participants, who are Ph.D. candidates and Postdoctoral researchers from various disciplines such as Computer Science and Geo-Informatics, to attend our evaluation session.

Before undertaking the test, each subject was requested to complete a short questionnaire about their background knowledge and skills in using LaTeX. Specifically, we required each participant to answer the following two categories of questions on a five-point scale (strongly disagree, disagree, neutral, agree, strongly agree):

- Knowledge about LaTeX script language.
- Experience in using LaTeX editors (either online or standalone).

Afterwards, the participants were instructed to perform the following tasks in an unsupervised manner; the subjects were required to finish the tasks on their own without any prior knowledge of SwiftLaTeX.

(1) We instructed each participant to compose a paragraph with approximately 200 words to describe their current research interests.
(2) We offered each participant with one English paragraph on a topic of general interest. In each line of the paragraph, there exists an obvious typo and the participants were required to correct it.
(3) We provided each subject with a set of simple mathematical equations with incorrect calculation (e.g., $1 - \frac{1}{4} = \frac{2}{4}$). The participants were required to correct the final results of the equations.

After completing the aforementioned tasks, we asked the participants to fill in a System Usability Scale (SUS) questionnaire, which is a widely-used reliable tool for perceived usability evaluation even with a small sample size [6]. We also requested participants to report their experience of using SwiftLaTeX.

7.2 Evaluation Outcomes LaTeX

SUS Score. The mean SUS score of SwiftLaTeX is 78 (in a range spanning from 0 to 100). It exceeds the threshold score of 68 which indicates a decent level of usability [6]. In addition, as suggested by the research [18], we also compute the mean values for the usability sub-scale and learnability sub-scale, which are 74 and 91 respectively.

Meanwhile, by considering participants' answers given in the background questionnaire (i.e., Knowledge of LaTeX language and Experience in LaTeX editors), we obtain two more background sub-scores. They are compared with the SUS score and sub-scale scores to obtain the Pearson correlation coefficients. We notice a weak negative correlation between the two background sub-scores and the learnability score, which may imply that a novice LaTeX user may find it easier to master our system. However, due to the small sample size,

Table 1: Axial coding of the participants' comments on SwiftLaTeX

Categories	Positive	Negative
WYSIWYG Editing	6	2
In place Math Editing	6	0
Efficiency	6	0
Web-based	3	0
Hotkey	0	4
Synchronization	1	0
Learning Curve	1	0

each correlation measure appears to be not statistically significant. To reach a more solid conclusion, a more comprehensive dataset is required.

Feedback. All the participants' comments are summarized in Table 1 using Axial coding [31]. On one hand, the positive feedback is mainly attributed to 1) innovative WYSIWYG editing for basic text and mathematical formulas, 2) the web-based interfaces, and 3) high efficiency for various kinds of editing tasks such as proofreading. A participant also provides valuable insight into the benefit of fine-grained synchronization. Specifically, existing LaTeX editors behave poorly in proofreading tasks mainly because of the coarse-grained synchronization functionality. A word in PDF is commonly mapped to a lengthy line in the source code. To modify the word, users are required to skim through the line to get the exact position of the word, which is fairly time-consuming. SwiftLaTeX successfully overcomes this issue by achieving character-level synchronization accuracy. Another participant who has little experience in LaTeX applauded the smooth learning curve of SwiftLaTeX, which possesses high familiarity with the existing word processors but produces much more high-quality typesetting output. In addition, SwiftLaTeX allows users to focus on document composition without concerning about the complex script language and the time-consuming compilation and debugging process.

On the other hand, Table 1 also reveals some widely perceived issues. Specifically, one important issue highlighted was the rudimentary functionality to conduct structural changes (e.g., adding a new section or list structure). Currently, these structural changes could be applied via clicking the corresponding options in the context menu (i.e., pop-up menu). However, one subject suggested that it would be more helpful if SwiftLaTeX could mimic the autoformat features (e.g., automatic bullets or numbering) from the word processor. Alternatively, SwiftLaTeX could support Markdown language such that a simple snippet could be used to generate structural changes. Meanwhile, the majorities of subjects also complained about the invalidity of commonly used hotkeys (e.g., control + c). Another report also reveals that the delete key is not working properly.

8 FURTHER DISCUSSIONS

SwiftLaTeX shows the potential for web-based WYSIWYG editing for LaTeX. Still, it bears several limitations that need further improvement.

8.1 Enabling Hotkeys and Clipboard

The implementation of clipboard features such as cut/paste was only postponed due to project management reasons and is one of the next straightforward development steps. With the architecture of SwiftLaTeX it is easy to achieve cut/paste even of structural elements such as section headers. This offers a further option for users to perform structural edits.

8.2 Facilitating Structural Edits

Finding a natural interface for structural edits is a challenge to WYSIWYG tools. Many tools support structural edits primarily as inserts with a large number of menus or analogs of often hard-coded menus. Our design is a menu showing small pre-defined LaTeXsnippets that are examples of the needed elements. For example for a header the snippet could be "`\subsection{Sectiontitle}`" and in the menu this is shown in its compiled version. If the user selects the snippet, essentially a paste (insert) of the snippet at the current cursor position is performed, which can be subsequently adapted by the user. This is a natural interface in that it reduces the structural change to a simpler concept, the paste concept. It also offers a natural visual appearance (show the PDF output of the example) and is in principle extendable and end-user-programmable. For parameterized elements such as LaTeX tables, our design uses an extension of this idea that allows users to specify the row/column number of tables and creates a table snippet accordingly.

8.3 Enhancing Third-party Storage Integration

SwiftLaTeX offers two different options for users to manage their project files. The simplest option is through the web interface where the users can create, remove, and upload their files. The second option is through third-party storage services. Currently, SwiftLaTeX provides basic support for syncing files to Github, Dropbox and Google Drive. We aim to support more storage services and bi-direction synchronization (from SwiftLaTeX to third-party or vice versa) in the next version. SwiftLaTeX provides intrinsic support for real-time collaboration in a style common to many web-based LaTeX editors: SwiftLaTeX generates a shared link for each project and a project's owner can invite other users to work on the project by sending the link to them.

8.4 Tuning Performance

We are close to achieving high-throughput compilation in SwiftLaTeX, but have not yet met our low-latency (i.e., the timespan of each compilation request) target. We are investigating the possibility to replace restful APIs with websockets in order to reduce communication overhead between browsers

and the backend. Meanwhile, we are exploring the potential to enhance the single-threaded LaTeX engine to leverage thread-level parallelism.

8.5 Embracing E-paper Devices

The advent of full size A4 e-paper readers now offers a further argument that the classic close-to A4 sized scientific publication has a secure future. We have been testing our system on various kinds of E-paper devices. Their low-power reflective display leads to excellent readability and significantly reduces eye strain. These merits make E-paper devices a very interesting potential platform for document composition. However, despite the promising features, these devices possess two major drawbacks, namely, low refresh rate and ghost effects. To overcome these issues, we are currently experimenting with a user interface especially optimized for e-paper devices and planning to conduct further related research.

9 CONCLUSION

In this paper, we presented SwiftLaTeX, the first cloud-based true WYSIWYG editor that allows the user to edit the document in its print form directly in the web-based PDF viewer. We have identified fundamental architectural challenges in providing WYSIWYG for LaTeX and identified resuable solutions. We have defined WYSIWYG in the narrower sense; the SwiftLaTeX system shows that a cloud-based WYSIWYG sytem for LaTeX is achievable. Although the batch-character of LaTeX poses challenges for such a system, we could show that these challenges can be overcome with a remarkably clean and lightweight architecture. In a first user study we have found positive feedback that the system has great potential and high efficiency for important editing tasks. Given that in the end a WYSIWYG editor in the narrower sense is also a PDF view, the user can only win from the provision of such a new feature. A member of the community, Jérôme Laurens, once wrote "TeX really deserves a good human interface, not just a user interface". With this work we have shown that a WYSIWYG editor should be integral part of such an interface of the future, also on the cloud. sad

REFERENCES

[1] ACE. 2018. Ace - The High Performance Code Editor for the Web. (2018). https://ace.c9.io/

[2] Aclements. 2015. A 21st century LaTeX wrapper. (2015). https://github.com/aclements/latexrun/

[3] Adobe. 2008. Portable Document Format. (2008). https://www.adobe.com/content/dam/acom/en/devnet/pdf/pdfs/PDF32000_2008.pdf

[4] Kyle Banker. 2011. *MongoDB in action*. Manning Publications Co.

[5] Bear Bibeault and Yehuda Kats. 2008. *jQuery in Action*. Dreamtech Press.

[6] John Brooke et al. 1996. SUS-A quick and dirty usability scale. *Usability evaluation in industry* 189, 194 (1996), 4–7.

[7] Alan W Brown. 2013. *Integrated project support environments: the aspect project*. Vol. 33. Elsevier.

[8] Josiah L Carlson. 2013. *Redis in action*. Manning Publications Co.

[9] J Collins. 2015. Latexmk 4.43 a: Fully automated LaTeX document generation. *Penn State Department of Physics, Pennsylvania State University* (2015).

[10] FreeDesktop. 2018. Poppler Pdf Rendering library. (2018). http://poppler.freedesktop.org/

[11] GNOME. 2018. Document Viewer. (2018). https://github.com/GNOME/evince

[12] G Grätzer. 2008. A gentle learning curve for LATEX. (2008). https://tug.org/pracjourn/2008-3/gratzer/gratzer.pdf

[13] David Kastrup. 2002. Revisiting WYSIWYG paradigms for authoring LATEX. *COMMUNICATIONS OF THE TEX USERS GROUP TUGBOAT EDITOR BARBARA BEETON PROCEEDINGS EDITORS KAJA CHRISTIANSEN* 23, 1 (2002), 57.

[14] Richard Koch, Max Horn, Gerben Wierda, and Various Contributros. 2006. TEXshop. *See website at http://www. uoregon. edu/koch/texshop/texshop. html* (2006).

[15] Jerôme Laurens. 2008. Direct and reverse synchronization with SyncTEX. *TUGBoat* 29 (2008), 365–371.

[16] Jerôme Laurens. 2011. Library for Parsing SyncTeX files. (2011). https://packages.debian.org/sid/libsynctex-dev

[17] Yun-Han Lee, Seiven Leu, and Ruay-Shiung Chang. 2011. Improving job scheduling algorithms in a grid environment. *Future generation computer systems* 27, 8 (2011), 991–998.

[18] James R Lewis and Jeff Sauro. 2009. The factor structure of the system usability scale. In *International conference on human centered design*. Springer, 94–103.

[19] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* 2014, 239 (2014), 2.

[20] Mozilla. 2018. A general-purpose, web standards-based platform for parsing and rendering PDFs. (2018). https://mozilla.github.io/pdf.js/

[21] Mozilla. 2018. What browsers are supported by PDF.js? (2018). https://github.com/mozilla/pdf.js/wiki/Frequently-Asked-Questions

[22] Jakob Nielsen. 1989. Usability engineering at a discount. In *Proceedings of the third international conference on human-computer interaction on Designing and using human-computer interfaces and knowledge based systems (2nd ed.)*. Elsevier Science Inc., 394–401.

[23] Jakob Nielsen. 2009. Discount usability: 20 years. *Jakob Nielsen's Alertbox Available at http://www. useit. com/alertbox/discount-usability. html [Accessed 23 January 2012]* (2009).

[24] Jakob Nielsen. 2015. Why you only need to test with 5 users, March 19, 2000. *Useit.com Alertbox* 27 (2015).

[25] Henry Oswald, James Allen, and Brian Gough. 2018. ShareLaTeX, the Online LaTeX Editor. (2018).

[26] Overleaf. 2017. 600,000 users on Overleaf make over 2 billion edits! (2017). https://www.overleaf.com/blog/449-600-000-users-on-overleaf-make-over-2-billion-edits

[27] Overleaf. 2018. Overleaf, Collaborative Writing and Publishing. (2018). https://www.overleaf.com

[28] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. 2014. *Operating system concepts essentials*. John Wiley & Sons, Inc.

[29] BaKoMa Soft. 2011. BaKoMa TEX 9.77. (2011). http://www.bakoma.com

[30] Jake Spurlock. 2013. *Bootstrap: Responsive Web Development*. " O'Reilly Media, Inc.".

[31] Anselm Strauss and Juliet M Corbin. 1990. *Basics of qualitative research: Grounded theory procedures and techniques*. Sage Publications, Inc.

[32] B van der Zander, J Sundermeyer, D Braun, et al. 2018. TeXstudio. https://www.texstudio.org/. (2018).

[33] Lu WANG and Wanmin LIU. 2013. Online publish via pdf2htmlEX. (2013).