

.tmu: A Low-Entropy Tree-Structured Representation for LLM-Assisted Scientific Writing

Tianyou Liu^{†,1}, Ziqiang Li^{†,2}, Xurui Liu³, Yu Wu⁴, Yansong Li^{*,5}

¹Southern University of Science and Technology, ²Alibaba,

³Tsinghua University, ⁴Rutgers University, ⁵Eastern Institute of Technology, Ningbo

Abstract

As large language models (LLMs) increasingly assist scientific writing, the limitations and token costs of generating $\text{\LaTeX}$ become increasingly visible. This paper analyzes $\text{\LaTeX}$'s architectural mismatch with LLM workflows, stemming from its lack of an explicit structural representation, to illustrate its limitations on generated semantics and error localization. As an alternative, we introduce .tmu, a low-entropy tree-structured representation. With its efficient data structure and clear contextual boundaries, .tmu outperforms .tex in the above aspects. Experiments across four LLMs provide evidence for this claim in most evaluated settings. Furthermore, we show that due to its lower information entropy, fine-tuning LLMs on .tmu achieves approximately 43% lower final training loss than on .tex. Our work provides a more scalable and LLM-friendly data representation for LLM-assisted scientific writing.

1 Introduction

LLM-assisted scientific writing is moving from isolated chat interactions to production workflows: agents inspect manuscripts, draft equations and proofs, merge revisions, and repair build failures in editors, CI systems, and conversion pipelines. Most such workflows still operate over $\text{\LaTeX}$ source. $\text{\LaTeX}$ remains excellent for typography, mathematical notation, and long-term compatibility, but its source format is optimized for typesetting rather than machine action. For LLM systems, the bottleneck is therefore representational: tools must recover document structure from a macro program that was not designed to expose it.

Three practical failure modes follow. First, $\text{\LaTeX}$ exposes no explicit document tree, so agents spend tokens rediscovering sections, labels, equations,

and scope. Second, structure and rendering instructions are coupled, making static analysis unreliable without executing the document. Third, errors are often non-local: a small syntactic defect can surface many lines later and send an agent into repeated compile-and-repair loops. In industry settings, these failures translate directly into latency, tool calls, compute cost, and lower reliability (we detail the mechanisms in Section 2).

We study .tmu, a low-entropy tree-structured representation from the GNU $\text{\TeX}$ macs lineage (van der Hoeven, 2001), as an alternative substrate for LLM-assisted scientific writing. In .tmu, formulas, citations, sections, and layout objects are represented as addressable tree nodes. This makes local edits and reference updates explicit and reduces source-level ambiguity: many equivalent $\text{\LaTeX}$ encodings collapse into a more canonical tree. The same representation also supports WYSIWYG authoring for humans, but our claim here is machine-facing: structured, low-entropy source should be easier for LLM systems to locate, merge, repair, and model. In practice, .tmu already underlies an existing scientific-writing system supporting document import, structured editing, node-level LLM assistance, syntax checking, and local repair.

Our contributions are as follows:

- We diagnose three representational frictions of $\text{\LaTeX}$ for LLM authoring—no explicit AST, structure–typesetting coupling, and non-local error reporting—and tie each to a concrete LLM failure mode.
- We present a low-entropy, tree-structured .tmu representation (on the $\text{\TeX}$ macs lineage) with explicit semantic nodes and clear contextual boundaries.
- We empirically evaluate .tmu on structure location, style-aware document merging, debugging of ill-formed documents, and formula-completion fine-tuning, showing gains in most

[†] Equal Contribution.

^{*} Corresponding author. ysli@eitech.edu.cn

evaluated settings (Section 4).

2 Background: Why TeX Limits LLM Workflows

This section details the frictions summarized in Section 1.

2.1 Introduction of TeX

For decades, L^AT_EX has been the standard for scientific writing (Lamport, 1994). As an unstructured macro language (van der Hoeven, 2020), however, it presents structural limitations for modern LLM-driven workflows along the axis of machine-tractable document structure (*we defer a brief history of TeX to Appendix A*).

2.2 Principles of TeX Compilation

TeX uses a batch-processing model: it reads source code from beginning to end, expanding macros step by step without building an explicit document tree (Abstract Syntax Tree) in memory (van der Hoeven, 2001). Lacking internal state, L^AT_EX must resolve cross-references and bibliographies through multiple compilation passes, writing temporary data into intermediate files (.aux, .toc, .bbl) and reading them back on the next run (*detailed in Appendix B*). This multi-pass design is inherently slow and opaque, precluding the sub-second, real-time feedback required by modern editors and AI agents (Mädje, 2022).

2.3 Architectural Defects in TeX

TeX processes documents as a linear sequence of commands in which structural semantics and type-setting decisions are deeply coupled within a single execution path. Consequently, AI models and static analysis tools cannot reliably extract the document’s structure without executing the code (*see Appendix C for a deep analysis*). This separation problem is fundamental, not incidental: Frank Mittelbach, the technical lead for L^AT_EX, candidly conceded in an interview (de Souza, 2021) that

the separation between local and global in L^AT_EX seems to be hard. [...] First of all, it is right now hard in TeX...

The same coupling makes error feedback delayed and ambiguous: the reported location of an error is rarely where the mistake actually happened, and a subtle local defect can irreversibly abort the entire compilation. Listing 1 shows a typical case—the second argument of `\frac` is missing its closing

brace, yet TeX keeps consuming subsequent tokens and only reports a Runaway argument? error at `\end{equation}`, far from the true cause.

Listing 1: Unclosed brace: error reported at `\end{equation}`, not at its source

```
1 \begin{equation}
2 x=\frac{-b\pm \sqrt{b^2-4ac}}{2a}
3 \end{equation}
```

Because such logs provide little structural guidance, models are prone to prolonged, often non-terminating debugging loops and hallucinations (Jain et al., 2026). This mechanism constrains LLM-driven writing.

2.4 UX & Ecosystem Constraints

As detailed in Appendix D, L^AT_EX’s ecosystem is too heavy and fragmented: a full TeX Live distribution exceeds 6GB, hindering deployment in cloud environments or sandboxes, while different engines (pdfLaTeX, XeLaTeX, LuaLaTeX) handle fonts and Unicode differently, yielding inconsistent results. While L^AT_EX retains clear technical strengths in typographic quality and mathematical expressivity, its continued dominance is also sustained by established user habits and publisher requirements (van der Hoeven, 2020); for LLM-driven workflows specifically, these factors leave its structural limitations unaddressed. This motivates a more scalable, LLM-friendly data representation.

Related Work The friction between L^AT_EX and LLM-driven workflows is well-documented across three critical dimensions.

First, **generation and execution failures**. Benchmarks reveal that LLMs severely struggle with L^AT_EX’s verbose syntax and long-range dependencies, showing low accuracy (Kale and Nadadur, 2025) and high token inefficiency in multimodal tasks (Ling et al., 2025; Xia et al., 2024). This causes systemic bottlenecks in automated science pipelines (Lu et al., 2024) and “execution illusions” of uncompileable code (Lyn and Graham, 2025). Thus, augmenting compiler logs with live ASTs is necessary for reliable AI assistance (Jain et al., 2026).

Second, **training and representation overhead**. Upstream, processing heterogeneous L^AT_EX incurs massive engineering overhead (Paster et al., 2023) and prohibitive token costs (Lin et al., 2024). L^AT_EX’s syntactic redundancy reduces predictability and inflates training costs (Taylor et al., 2022).

Conversely, explicitly modeling markup hierarchies drastically improves document understanding (Li et al., 2022) and AST-guided code generation (Zeng et al., 2025).

Third, **system architecture limitations**. To address batch compilation delays, existing solutions propose incremental compilers (Mädje, 2022; Haug, 2022) or GUI overlays (Gobert and Beaudouin-Lafon, 2022), but still separate authoring from visual output. Meanwhile, studies highlight $\text{\LaTeX}$ ’s real-world fragility (Tan and Rigger, 2024), user inefficiency (Knauff and Nejas-mic, 2014), and extreme runtime costs when used as a computing substrate (Gardner et al., 2025). We build on the tree-based WYSIWYG lineage of $\text{\TeX}$ macs (van der Hoeven, 2001), but our contribution is distinct: whereas $\text{\TeX}$ macs is prior art designed for human editing, we treat its `.tmu` format as a low-entropy, tree-structured *representation for LLMs* and contribute the first evaluation of such a structured representation on LLM generation and fine-tuning. In doing so, we provide evidence that a tree-structured format offers advantages over $\text{\LaTeX}$ along both axes the literature above identifies as broken—structural transparency for generation and predictability for training. (*A comprehensive review is deferred to Appendix E.*)

3 The `.tmu` Representation: A Low-Entropy Tree Structure

While $\text{\TeX}$ represents formulas through layout-oriented markup in the form of typesetting instructions, the `.tmu` format adopts a tree-based, functional representation for both mathematical formulas and document structure. This tree-structured representation shows advantages over the formats underlying other editors in Table 7. (*A detailed introduction to the `.tmu` representation and a comparison are provided in Appendix F.*) The following discussion evaluates this representation with specific examples.

The central property of our structured representation is that it is *low-entropy*, by which we mean reduced source-encoding multiplicity. In $\text{\LaTeX}$, a single rendered output can be produced by many equivalent source strings: x^2 and $x^{\{2\}}$ render identically, $\frac{a}{b}$ and $\{a \text{ over } b\}$ denote the same fraction, and arbitrary whitespace can be inserted almost anywhere without changing the result. In the `.tmu` tree, by contrast, each rendered structure corresponds to an essentially

canonical source, so this one-to-many redundancy is largely eliminated. Fewer equivalent encodings mean lower next-token uncertainty and therefore lower information entropy. This low-entropy property is the property that the remainder of this section instantiates through concrete examples.

The same low-entropy representation benefits both human authors and LLMs. Because the tree directly captures the rendered document, it supports a What You See Is What You Get (WYSIWYG) editing interface in which humans edit the rendered document in place, rather than following $\text{\LaTeX}$ ’s edit-source-then-compile loop. In this paper, however, we make no quantitative claim about human input speed; our measured contribution is the machine-facing payoff, where the lower-entropy structure is easier for LLMs to locate, merge, repair, and model.

3.1 Tree-Structured Formulas and Document Structure

Unlike $\text{\LaTeX}$ ’s compilation model, which centers on linear text and macro expansion, $\text{\TeX}$ macs features an explicit tree-based document structure designed from the outset. Document components—such as chapters, formulas, citations, and typesetting elements—exist as structured nodes, rather than being implicitly embedded within macro calls or token streams. This architectural distinction directly impacts citation updates, compilation efficiency, and the interactive user experience.

A fraction serves as a primary example of this difference. In $\text{\TeX}$, a fraction is represented as `\frac{1}{2}`; internally, the compiler processes `\frac`, `{1}`, and `{2}` serially. In the `.tmu` representation, however, the fraction is represented as the tree (`frac "1" "2"`), written here in a compact S-expression notation (in the style of Lisp/Scheme). This forms a tree structure, as illustrated in Figure 1.

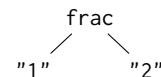


Figure 1: The Tree Structure of `.tmu` Formulas

As a further example, consider the mathematical expression in Equation (1):

$$\int_a^b f(x) dx = [F(x)] \Big|_a^b \quad (1)$$

Its `.tmu` tree¹ is shown in Listing 2:

¹On disk, a `.tmu` file stores this tree in an XML-like serial-

Listing 2: The .tmu tree of the integral, in S-expression notation

```
1 > `(math (concat (big "int") (rsub "a"
  ") (rsup "b") "f" (around* "(" "x"
  ")") "<mathd>x=" (around* "<
  nobracket>" (around* "[" (concat "
  F" (around* "(" "x" ")") "])")
  "|") (rsub "a") (rsup "b")))
```

Note that the segment $[F(x)]$ is managed via the tree structure illustrated in Figure 2. Consequently, even if a user omits a parenthesis or two, this omission does not negatively affect the rendering of the entire mathematical expression under the .tmu representation.

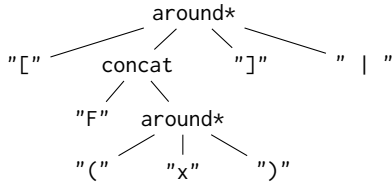


Figure 2: Tree representation of $[F(x)]$

Unlike T_EX, this tree structure exists at the rendering level, not the syntax level. This rendering-level tree structure offers two distinct advantages:

1. Local Scoping of Input Errors:

- Errors do not cause the entire document to fail rendering (a major drawback of L^AT_EX).
- Local edits do not trigger global layout instability (a major drawback of MS Word).

2. Independent Subtree Layout:

Because the document is represented as a tree, independent subtrees can be laid out separately rather than strictly serially, which can reduce the work required to re-render after a local edit.

The same locality extends beyond formulas: each line, image block, and structural element maps to an independent tree node, so local edits such as resizing an image need not disrupt surrounding layout.

3.2 Functional Symbol Representation

In T_EX, all mathematical symbols and fonts are represented as strings. However, in the .tmu representation, certain special symbols are defined via functions. For instance, the inner product $\langle \cdot \rangle$, as shown

ization; throughout this paper we display the same tree in the equivalent S-expression notation for readability, the two being interconvertible without information loss.

in Equation (2), is an anonymous function that automatically scales according to the symbols it contains. While this is similar to `\langle \rangle` in L^AT_EX, the difference lies in the fact that, as a complete function, the left and right brackets always render a unified pair rather than isolated characters. To obtain a single bracket, a structured editor operating on .tmu lets the user delete the undesired character from the rendered pair.

$$\left\langle \int \right\rangle \langle f \rangle \quad (2)$$

3.3 Dynamic System Optimization

The .tmu representation encodes citation relationships as structural links directly embedded in the document tree. This mechanism avoids a full reparsing of the document when updating. Furthermore, because complexity is dictated by what the document tree actually contains, an editor operating on .tmu can load functionality on demand rather than relying on a pre-configured feature set, so that system complexity is dictated by document content. This mechanism significantly reduces the burden for deploying cloud environments or sandboxes. (*Further details on reference rendering and plugin loading are available in Appendix G and Appendix H.*)

4 Numerical experiments

We compare .tmu with .tex on compiling/rendering speed, LLM task performance, and fine-tuning.

Setup. All experiments derive from a 120-page arXiv document (arXiv:2502.17655) and run on an Intel Ultra 9 285H CPU, GeForce RTX 5080 GPU, and 32GB RAM. We evaluate four LLMs—Deepseek-v3.2, Claude-Sonnet-4.5, Gemini-3-pro, and ChatGPT-5.2—on structure locating (20 questions), style-aware file merging (5 tasks), and debugging (20 samples). Scores reward correctness while penalizing total token usage T (input, thinking, output, and MCP tools), and timing benchmarks report the average of three trials per document. We refer to these scores as Cost-aware Task Utility. Incorrect outputs receive zero, while token cost discounts successful completions; a correct but over-budget response may also score zero, without being considered semantically equivalent to a hallucination.

4.1 Benchmark on compiling/rendering time

We benchmark compilation and rendering on 6 arXiv documents. `.tmu` outperforms `.tex` on both full compilation and incremental updates; the latter is most decisive (Figure 3), because `.tmu` updates local tree nodes without I/O overhead ($t_{IO} = 0$, activated only when opening the document). These low feedback delays directly reduce agent time cost in real-time LLM writing. (*Full-compilation results and the t_{IO} cost model are in Appendix I.*)

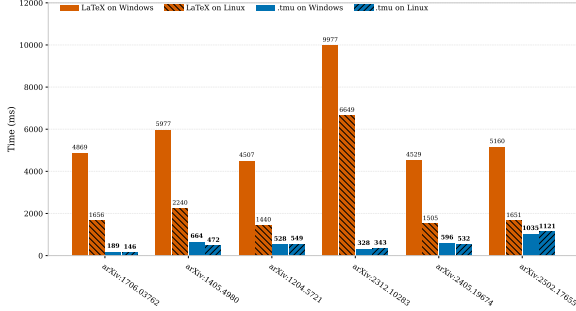


Figure 3: Benchmark on incremental update. Each bar represents the average of three trials.

4.2 Performance in LLM tasks

The standardized grammar and explicit tree-node tags in `.tmu` help models locate targets, merge contexts, and debug ill-formed structures more efficiently than `.tex`. Figure 4 summarizes the three LLM task results.

4.2.1 Locating document structure

To evaluate structural understanding, each LLM answers 20 document-structure questions (Appendix J). For each answer, we take

$$u_s = \max \left(0, \begin{cases} 5 - \left\lfloor \frac{T}{1 \times 10^4} \right\rfloor & \text{right answer,} \\ 0 & \text{wrong answer,} \end{cases} \right)$$

where $\sum u_s \in [0, 100]$. The 10k-token scale separates confident structure location from longer reasoning/tool-use loops, which indicate lower efficiency and higher cost (Ling et al., 2025).

Despite LLMs’ stronger pretraining exposure to $\text{\LaTeX}$, `.tmu` leads for most models (Figure 4a). Its reference metadata is updated and stored directly in the file (e.g., `<associate|sec:tree-struct-on-mogan|<tuple|5.1|13>>`), whereas `.tex` requires scanning and resolving the compiled numbering context.

Gemini is the only exception, scoring 90 on `.tex` versus 87 on `.tmu`. Thus, we claim gains in

most, rather than all, settings and interpret this as a comparison of the complete native representations rather than tree syntax alone.

4.2.2 Merging files with distinct doc-styles

We define *doc-style* as macro naming and command usage. Documents may use different theorem aliases, redefine the same macro differently, or give the same command incompatible meanings.

We ask LLMs to complete two assignments.

Assignment 1 generates two independently compilable files: `theorems.tex`, a theorem collection with many new macros and packages, and `proofs.tex`, a disordered proof collection with a distinct doc-style and conflicting packages. The files are connected by theorem and equation cross-references.

Assignment 2 merges the generated files into the doc-style of `theorems.tex`, while preserving compilability and placing each proof below the referenced theorem.

We obtain `theorems.tmu` and `proofs.tmu` from their `.tex` versions; rendered content is identical, as guaranteed by a $\text{\LaTeX}$ importing engine.

Assignment 1 is 1 generation task; Assignment 2 is 4 merge tasks. For each task, we take

$$u_m = \max \left(0, \begin{cases} 20 - 2E_{\text{ref}} - \left\lfloor \frac{T}{10^4} \right\rfloor - E_{\text{sty}} & \text{success on the first try,} \\ 10 - 2E_{\text{ref}} - \left\lfloor \frac{T}{10^4} \right\rfloor - E_{\text{sty}} & \text{success on the second try,} \\ 0 & \text{fail within two tries,} \end{cases} \right)$$

where E_{ref} is the number of reference failures (i.e., “??” appears but compilation succeeds), E_{sty} is the number of cases where the merged file has the doc-style of `proofs.tex` (we required LLMs to write the merged file in the doc-style of `theorems.tex`, so other macro aliases or redefined macros are not accepted), $\sum u_m \in [0, 100]$.

As shown in Figure 4b, `.tmu` scores higher for all LLMs. Its grammatical consistency removes many macro-conflict and style-unification choices that make complex $\text{\LaTeX}$ generation brittle (Kale and Nadadur, 2025); merging structured contexts becomes closer to copying compatible subtrees.

4.2.3 Debugging ill-formed documents using error messages

Debugging ill-formed documents is a common LLM co-writing use case. We construct 20 samples from the source document (Table 1), feed error messages to LLMs, and ask them to repair the document.

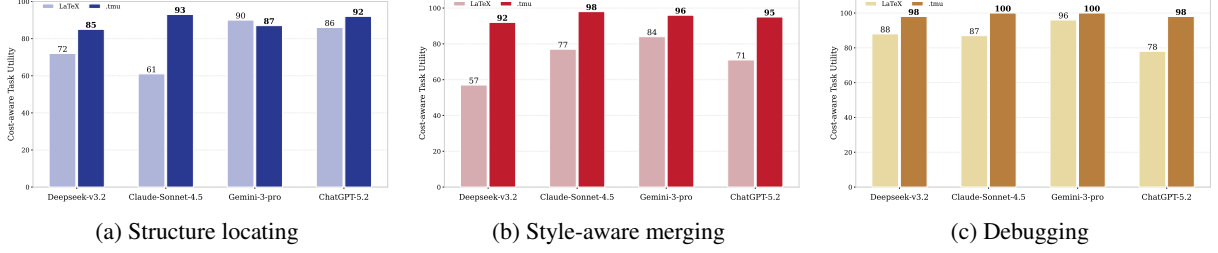


Figure 4: LLM task performance for `.tmu` and `.tex`.

Table 1: Distribution of error types.

Error Type	Count
Unclosed bracket	4
Unclosed environment	5
Wrong command usage	4
Undefined cross-reference	3
Conflicting packages	2
Self-recursive macros	2

For each error, we take

$$u_d = \max \left(0, \begin{cases} 5 - \left\lfloor \frac{T}{1 \times 10^4} \right\rfloor & \text{right answer,} \\ 0 & \text{wrong answer,} \end{cases} \right)$$

where $\sum u_d \in [0, 100]$.

We treat this as an end-to-end evaluation under each format’s native validation and diagnostic workflow, rather than a controlled comparison using identical plain-text error messages.

The `.tex` group requires substantially more reasoning on most samples, while the `.tmu` group solves all samples and only two exceed 10k tokens (Figure 4c). A defect in `.tmu` affects only the closest tree-tag ancestor (Section 3.1), yielding localized messages instead of long logs detached from root causes.

Well-formed `.tmu` also precludes classes such as unclosed environments or self-recursive macros; when a malformed tree exists, a structured editor can surface the affected node directly while the rest of the document remains renderable.

4.3 Efficiency in fine-tuning

Because `.tmu` is tree-structured while `.tex` follows a linear macro flow, next-token prediction should be less ambiguous in `.tmu`.

We fine-tune Qwen2.5-7B-Instruct (Qwen Team, 2025) with LoRA on 1000 random formulas, converting each $\text{\LaTeX}$ formula to an equivalent rendered `.tmu` S-expression. The formulas cover common mathematical constructs; each is split into a prefix prompt and completion target.

Following Dong et al. (2025), both runs use identical LoRA settings and differ only in training format. Figure 5 shows that `.tmu` converges to around 0.4 loss, while `.tex` converges to around 0.7. This supports the low-entropy claim: equivalent $\text{\LaTeX}$ strings such as `\frac{a}{b}` and `{a \over b}`, or `x^{2}` and `x^2`, create extra prediction uncertainty absent from canonicalized `.tmu` trees.

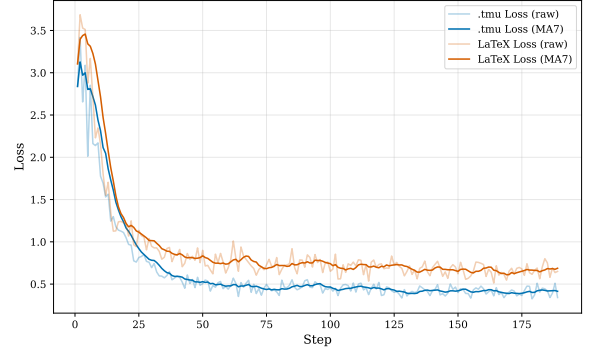


Figure 5: Experiment of LoRA based on Qwen2.5-7B-Instruct.

This advantage matters especially for large documents with multiple doc-styles, where $\text{\LaTeX}$ models must infer commands from preamble-local macro definitions (Lin et al., 2024).

Moreover, discussions of the `.tmu` format versus Markdown are attached in Limitations.

Limitations

Pretraining exposure and evaluation scope.

The LLMs we evaluate were pretrained on very large $\text{\LaTeX}$ corpora and have seen essentially no `.tmu` during pretraining, placing `.tmu` at a substantial model-familiarity disadvantage. Evaluating models with better-matched exposure is future work. This scopes our evaluation, not our thesis: the architectural frictions $\text{\LaTeX}$ poses for LLMs (one-off batch compilation, weak structural semantics, error messages detached from their root cause; Appendices C–D) are properties of the format, not

artifacts of model familiarity, and remain our core motivation.

Evaluation scale and statistics. The LLM-task evaluation is focused rather than broad: ~ 20 items each for structure-locating and debugging, only a handful of merging tasks, drawn largely from a single source document (arXiv:2502.17655). We report point estimates without significance testing or variance, and do not yet claim domain generality. A larger, multi-domain benchmark with repeated trials and significance testing is future work.

Scope of the fine-tuning evidence. The fine-tuning result uses a single LoRA run on one model family (Qwen2.5-7B-Instruct), a single seed, and formula-level rather than full-document data. Lower training loss is consistent with but does not by itself establish better downstream generation; validation loss, exact-match and rendered-equivalence metrics, and multi-seed runs across model sizes are future work.

Utility metric. Our utility scores (u_s , u_m , u_d) combine task correctness with token cost under a hand-chosen 10k-token scale; absolute scores are not comparable across task types, and the metric entangles accuracy with token efficiency. Reporting these two axes separately is future work.

Baselines and the tool-vs-representation confound. Our baselines are limited to $\text{\LaTeX}$; comparison against Typst, Markdown-with-math, and MathML/XML and other structured editors is future work. We note Markdown specifically because it reuses $\text{\LaTeX}$ math syntax and is the natural lightweight baseline, but a fair comparison is non-trivial given the gap in intended use, so we do not treat it as central. A further confound is that some gains may stem from the editor’s tooling (incremental update, structured editing, the $\text{\LaTeX}$ import engine) rather than the representation alone; disentangling these requires controlled ablations we have not yet run.

On low entropy. We currently support the low-entropy claim with an operational proxy (reduced source-encoding multiplicity for a fixed rendered output, e.g. x^2 vs. $x^{\{2\}}$) together with indirect evidence from fine-tuning convergence. A full information-theoretic treatment is future work.

Future work. A concrete agenda: (i) a fair controlled comparison with an ablation separating the contribution of tree structure from that

of pre-expanded reference numbers, isolating the structure-locating input asymmetry (in .tmu, numbering is stored inline after incremental update, whereas $\text{\LaTeX}$ requires a full scan); (ii) a larger multi-domain evaluation with repeated trials, significance testing, and additional baselines (Typst, Markdown-with-math, MathML); (iii) fine-tuning with validation loss, exact-match, and multi-seed runs across model sizes, plus a Markdown comparison and a larger, standardized end-to-end LLM-repair benchmark.

References

- Paulo Ney de Souza. 2021. Interview with Frank Mittelbach, leader of the $\text{\LaTeX}$ project. *TUGboat*, 42(2):114–120.
- Haoyu Dong, Pengkun Zhang, Mingzhe Lu, Yanzhen Shen, and Guolin Ke. 2025. [MachineLearningLM: Continued pretraining language models on millions of synthetic tabular prediction tasks scales in-context ml](#). *arXiv preprint arXiv:2509.06806*.
- James AD Gardner, Will Rowan, and William AP Smith. 2025. NeuRa $\text{\LaTeX}$: A machine learning library written in pure $\text{\LaTeX}$. *arXiv preprint arXiv:2503.24187*.
- Camille Gobert and Michel Beaudouin-Lafon. 2022. i- $\text{\LaTeX}$: Manipulating transitional representations between $\text{\LaTeX}$ code and generated documents. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, pages 1–16.
- Martin E Haug. 2022. Fast typesetting with incremental compilation. *Master’s thesis, Technical University of Berlin, Germany*.
- Laura Elizabeth Jackson and Herbert Voß. 2001. Lyx—an open source document processor. *TUGboat*, 22(1/2):32–41.
- Nilesh Jain, Rohit Yadav, and Andrej Karpathy. 2026. [Bibby AI: AI \$\text{\LaTeX}\$ editor writing assistant for researchers vs. Overleaf alternative vs. OpenAI Prism](#). *Preprint*, arXiv:2602.16432.
- Sahil Kale and Vijaykant Nadadur. 2025. [TeXpert: A multi-level benchmark for evaluating \$\text{\LaTeX}\$ code generation by LLMs](#). In *Proceedings of the Fifth Workshop on Scholarly Document Processing*, pages 7–16, Vienna, Austria. Association for Computational Linguistics.
- Markus Knauff and Jelica Nejasmic. 2014. An efficiency comparison of document preparation systems used in academic research and development. *PLOS ONE*, 9(12):e115069.
- Donald E. Knuth. 1984. *The TeXbook*. Addison-Wesley, Reading, Massachusetts.

- Leslie Lamport. 1994. *LaTeX: A Document Preparation System*. Addison-Wesley.
- Junlong Li, Yiheng Xu, Lei Cui, and Furu Wei. 2022. MarkupLM: Pre-training of text and markup language for visually rich document understanding. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6078–6087.
- Liii Network. 2025a. What is Liii STEM? <https://liii.pro>. Accessed: 2025-03-01.
- Liii Network. 2025b. What is Mogan STEM. <https://mogan.app>. Accessed: 2025-03-01.
- Yiming Lin, Madelon Hulsebos, Ruiying Ma, Shreya Shankar, Sepanta Zeigham, Aditya G. Parameswaran, and Eugene Wu. 2024. [Towards accurate and efficient document analytics with large language models](#). *arXiv preprint arXiv:2405.04674*.
- Jun Ling, Yao Qi, Tao Huang, Shibo Zhou, Yanqin Huang, Jiang Yang, Ziqi Song, Ying Zhou, Yang Yang, Heng Tao Shen, and Peng Wang. 2025. [Table2LaTeX-RL: High-fidelity LaTeX code generation from table images via reinforced multimodal language models](#). *Preprint*, arXiv:2509.17589.
- Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. 2024. The AI scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*.
- Jiawen Lyn and Yvette Graham. 2025. TransLaTeX: Exposing the last-mile execution gap in LLM-agent for scientific formatting. In *Proceedings of The First Workshop on Human-LLM Collaboration for Ethical and Responsible Science Production (SciProdLLM)*, pages 19–24.
- Laurenz Mädje. 2022. Typst: A programmable markup language for typesetting. *Master’s thesis, Technical University of Berlin, Germany*.
- Overleaf Team. 2023. Overleaf documentation: Learn LaTeX. <https://www.overleaf.com/learn>.
- Keiran Paster, Marco Dos Santos, Zhangir Azerbayev, and Jimmy Ba. 2023. [OpenWebMath: An open dataset of high-quality mathematical web text](#). *arXiv preprint arXiv:2310.06786*.
- Qwen Team. 2025. [Qwen2.5 technical report](#). Technical report, Alibaba Cloud.
- Jovyn Tan and Manuel Rigger. 2024. Inconsistencies in TeX-produced documents. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1415–1427.
- Ross Taylor, Marcin Kardas, Guillem Cucurull, Thomas Scialom, Anthony Hartshorn, Elvis Saravia, Andrew Poulton, Viktor Kerkez, and Robert Stojnic. 2022. [Galactica: A large language model for science](#). *Preprint*, arXiv:2211.09085.
- Joris van der Hoeven. 2001. GNU TeXmacs, a free, structured, WYSIWYG and technical text editor. *Cah. GUT*, 39–40:39–50.
- Joris van der Hoeven. 2020. *The Jolly Writer*. Scypress, Chevreuse, France.
- Renqiu Xia, Song Mao, Xiangchao Yan, Hongbin Zhou, Bo Zhang, Haoyang Peng, Jiahao Pi, Daocheng Fu, Wenjie Wu, Hancheng Ye, Shiyang Feng, Bin Wang, Chao Xu, Conghui He, Pinlong Cai, Min Dou, Botian Shi, Sheng Zhou, Yongwei Wang, and 4 others. 2024. [DocGenome: An open large-scale scientific document benchmark for training and testing multi-modal large language models](#). *arXiv preprint arXiv:2406.11633*.
- Yiming Zeng, Jinghan Cao, Zexin Li, Yiming Chen, Tao Ren, Zhuochun Li, Dawei Xiang, Xidong Wu, Shangqian Gao, and Tingting Yu. 2025. [TreeDiff: AST-guided code generation with diffusion LLMs](#). *Preprint*, arXiv:2508.01473.

A A Brief History of T_EX

Numerous derivatives of T_EX have emerged in the past decades, with L^AT_EX being the most prominent. Built as a macro language on top of T_EX, L^AT_EX significantly simplifies its usage, allowing users to leverage T_EX’s powerful typesetting capabilities without needing a deep understanding of intricate commands. By defining commands and templates that align with standard typesetting practices, L^AT_EX has made the production of scientific literature and books far more efficient and accessible, eventually becoming the de facto standard for scientific document preparation.

In the academic field, the T_EX system and L^AT_EX in particular have become the standard of the scientific community thanks to their exceptional mathematical typesetting capabilities. The core design of T_EX originates from the pioneering work of Knuth (1984). The American Mathematical Society (AMS) strongly encourages mathematicians to submit manuscripts using T_EX, and widespread adoption by world-class publishers such as Wiley and IEEE has made it a staple for books and journals. Consequently, T_EX occupies a pivotal position in the production of academic papers and monographs, serving as a vital tool for scholarly communication and knowledge dissemination.

However, T_EX’s original design and its subsequent development trajectory have resulted in numerous legacy issues. This article primarily examines the underlying architecture of T_EX and explains why certain design choices have led to significant problems.

On another note, it was long believed that “What You See Is What You Get” (WYSIWYG) was incompatible with structured editing. The emergence of T_EXmacs, however, proved this assumption fundamentally incorrect. Professor Joris from École Polytechnique wrote a critique on this subject (see van der Hoeven, 2001; Liii Network, 2025a). The design philosophy of L^AT_EX has inspired a series of similar editing software; beyond the aforementioned T_EXmacs, these include LyX and the recently popular Typst.

Notably, the formula rendering in Typst and T_EXmacs is completely independent of the T_EX system, whereas LyX serves as a front-end for T_EX. While this article contains significant criticism of the T_EX system, we wish to clarify our stance: we are by no means denying T_EX’s historical status, nor do we suggest it was outdated for its time. We

simply argue that today—especially in an era of rapidly advancing AI tools—T_EX’s underlying design presents many issues.

B An Introduction to T_EX Compilation Principles

Listing 3: Minimal example of L^AT_EX code

```
1 \documentclass{article}
2 \usepackage{siunitx}
3 \begin{document}
4 Hello, World! from \LaTeX.
5 The speed of light is \SI
   {299792458}{\meter\per\second}.
6 \end{document}
```

Listing 3 shows a minimal example of L^AT_EX code. The source code begins with the `\documentclass` command. Lamport’s L^AT_EX framework simplifies the use of T_EX (Lamport, 1994) by defining the document class used for the file. Immediately following this, we can use `\usepackage` to import packages. We then use `\begin{document}` and `\end{document}` to mark the start and end of the body content, placing the text between them.

The area between `\documentclass` and `\begin{document}` is called the *preamble*. In addition to importing packages via `\usepackage`, *preamble* is used for global macro definitions and document configuration, which can also be left empty.

Finally, we can compile the code. Running `pdflatex example.tex` generates the `example.pdf` file. Simultaneously, several auxiliary files are generated in the same directory. For instance, the `.aux` file saves intermediate information such as cross-references, tables of contents, and numbering; consequently, documents often require multiple compilations to yield correct results. The `.log` file records the complete compilation process and is the primary resource for locating warnings and errors. The `.dvi` file stands for *DeVice Independent file*, an intermediate format unrelated to the specific output device.

Beyond the `.tex` source files we write, every package and document class consists of files with specific extensions. Table 2 lists the files that frequently appear in L^AT_EX templates:

L^AT_EX generates numerous auxiliary files and logs during the compilation process. Features such as cross-references, bibliographies, tables of contents, and indices require an initial compilation to generate auxiliary files, followed by a subsequent

Table 2: L^AT_EX file extensions and their descriptions

Ext.	File Type	Description
.sty	Package file	The package name corresponds to the filename without the extension.
.cls	Document class file	The document class name corresponds to the filename without the extension.
.bib	BibTeX database file	A database file storing bibliographic information.
.bst	BibTeX style file	A template file defining the formatting style for bibliographies.

compilation to read these files to produce the correct result. Therefore, complex L^AT_EX source code often requires multiple compilation passes as summarized in Table 3:

Table 3: L^AT_EX auxiliary file extensions, tools, and descriptions

Ext.	Related Source	Tool / Description
.log	Typesetting engine	Records the compilation process; used for error debugging.
.aux	latex	Main auxiliary file; records cross-references, TOC, citations, etc.
.toc	latex	Records content for the Table of Contents.
.lof	latex	Records content for the List of Figures.
.lot	latex	Records content for the List of Tables.
.bbl	bibtex	The generated bibliography record file.
.blg	bibtex	Log file for the BibTeX compilation process.
.idx	latex	Raw index record file to be processed by makeindex.
.ind	makeindex	Formatted index file generated after processing .idx.
.ilg	makeindex	Log file for makeindex.
.out	hyperref package	Record file for generating PDF bookmarks.

To intuitively understand why L^AT_EX documents often require multiple compilations, Listing 4 shows a minimal example containing cross-references and a table of contents:

Listing 4: Cross-reference example

```

1 \documentclass{article}
2 \usepackage{hyperref}
3
4 \begin{document}
5 \tableofcontents
6
7 \section{Introduction}\label{sec:
  intro}

```

```

8 See Section~\ref{sec:intro}.
9 \end{document}

```

When compiling this code with pdf_latex for the first time, L^AT_EX cannot yet determine section numbers or the content of the table of contents (TOC). The compiler records this information into auxiliary files during the typesetting process: section titles are written to the .toc file, and numbering information for \label{sec:intro} is written to the .aux file. Meanwhile, the \ref{sec:intro} in the body text is temporarily output as a placeholder. Consequently, the PDF generated from the first pass has an empty TOC, and cross-references appear as "??".

During the second pass, L^AT_EX reads the .aux and .toc files generated previously, obtaining complete numbering and directory information to typeset them correctly into the document. Thus, the cross-references appear correctly, and the TOC is populated. This process demonstrates that the L^AT_EX typesetting process is essentially an iterative workflow that relies on intermediate results to converge. Any feature involving global information or forward/backward dependencies almost inevitably requires multiple compilation passes to produce the final correct output.

In practice, the typesetting result is determined by the underlying engine. Common engines include:

- **pdfLaTeX:** The most traditional and compatible engine. It generates PDFs directly but has limited support for Unicode and system fonts, relying mostly on T_EX's own font system.
- **XeLaTeX:** Renowned for excellent Unicode support and the ability to call operating system fonts directly. It is highly suitable for multi-language typesetting (such as CJK), though its compilation speed and package compatibility are often more complex than pdfLaTeX.
- **LuaLaTeX:** Builds upon XeLaTeX by introducing Lua scripting as a programmable extension layer. This mechanism allows for dynamic customization of typesetting logic, offering the most power but also the highest complexity, with a strong dependency on the quality of templates and packages.

Differences in font handling, package support, and compilation behavior among these engines mean that the same L^AT_EX source code frequently

yields different results—or fails to run entirely—in different environments. The coexistence of these diverse engines is one of the root causes of the long-standing complexity within the L^AT_EX ecosystem.

C Fundamental Defects in T_EX's Compilation Design

The batch processing compilation model adopted by the T_EX system was rational within the computing environment of its inception in the 1970s and 80s. However, in the context of contemporary academic writing, which emphasizes interactivity, instant feedback, and multi-platform publishing, this model exposes profound systemic tension. Its core issues lie in its *one-off processing flow*, *weak semantic representation*, and *delayed error feedback mechanisms*. These problems not only directly impair user experience but fundamentally constrain the evolution of the tool ecosystem.

C.1 Batch Model Limitations: Unidirectionality, Weak Semantics, and Delayed Feedback

T_EX's compilation architecture is rooted in the batch processing paradigm. While capable of efficiently handling static documents, it struggles to adapt to the dynamic, iterative needs of modern writing and LLM training (Paster et al., 2023).

C.1.1 Unidirectional and One-off Processing Flow

The working mechanism of T_EX follows a rigid sequence: Linear Input Reading → Macro Expansion → Typesetting Calculation → Output Generation.

This flow possesses strong unidirectionality and irreversibility. The system does not maintain an intermediate representation amenable to incremental re-computation; instead, it treats input as a continuous stream of instructions. This design leads to:

Delayed Manifestation of Global State: The overall typesetting state is only evaluable after the full compilation cycle concludes. T_EX's typesetting decisions (pagination, float positioning, citation numbering, etc.) rely on global information, which is often only determined after the document has been fully read and executed. Consequently, users cannot verify whether the document is "typeset correctly" before compilation is complete. Listing 5 shows a typical example:

Listing 5: Cross-reference before definition

```
1 \documentclass{ctexart}
2 \begin{document}
3
4 As shown in Equation~\ref{eq:test}.
5
6 \newpage
7
8 \begin{equation}
9   E = mc^2
10 \label{eq:test}
11 \end{equation}
12
13 \end{document}
```

When this code is compiled with xelatex for the first time, `\ref{eq:test}` displays as `??`. This discrepancy occurs because the equation's numbering information remains ungenerated; T_EX cannot know the correct reference number during the current compilation pass. Only after the first pass concludes and the numbering information is written to the `.aux` file can T_EX read this information during the second compilation, enabling `\ref{eq:test}` to display the correct number.

This requirement for multiple passes demonstrates that T_EX's typesetting decisions are global and latent: the typesetting engine cannot accurately infer cross-references or numbering within the document before the first compilation finishes. Multiple passes are strictly required to obtain the final result.

Inability to Implement Incremental Updates:

Local modifications cannot trigger efficient, localized re-computation; instead, the entire compilation process requires repetition. Even if only a single local element in the document is modified, T_EX must re-execute the entire input stream from the beginning, re-expanding macros, re-paginating, and re-calculating all layouts. Listing 6 illustrates this issue:

Listing 6: Incremental update example

```
1 \documentclass{ctexart}
2 \usepackage{lipsum}
3 \begin{document}
4
5 \section{Section 1}
6 Some text, some text, some text.
7 \lipsum[2-8]
8 \section{Section 2}
9 More text, more text, more text.
10
11 \end{document}
```

Suppose we delete line 7, `\lipsum[2-8]`, which originally generated multiple paragraphs occupying significant vertical space. Keeping the rest of the code unchanged, semantically, this is merely

a local modification to the internal content of the first section; it does not touch the structure of the second section or anything following it.

However, during actual compilation with `xelatex`, this modification triggers a cascading reaction: the vertical space previously occupied by `\lipsum[2-8]` vanishes, drastically reducing the height of the first section. This alters the pagination results for the entire document. The title of the second section, which might have been on the next page, moves up to the previous page; consequently, page numbers, headers, footers, and the positions of any floating bodies must be recalculated. Despite the modification occurring in just a single line at the beginning of the document, `TEX` is compelled to re-read the input stream from the start, re-expand all macros, and re-execute the complete pagination algorithm. It is unable to perform a local reflow solely for "Section 1" or the "affected pages."

This example clearly demonstrates that within `TEX`'s execution model, there is no stable, reusable intermediate typesetting state. Any seemingly local change can alter all subsequent typesetting decisions. Therefore, the system has no choice but to perform a full re-compilation, making the efficient incremental updates expected of modern editors impossible.

Lack of Visualization Upon Interruption: Once compilation is interrupted by an error, users cannot obtain reliable partial results for preview. `TEX` typically aborts the output process immediately, preventing users from seeing "what has been typeset so far." Listing 7 demonstrates this scenario:

Listing 7: Missing argument error

```

1 \documentclass{ctexart}
2 \newcommand{\mycmd}[1]{\textbf{#1}}
3 \begin{document}
4 This page contains perfectly correct
   content.
5
6 \mycmd % Forgot to provide argument
7
8 The content following this
   theoretically does not
9 depend on this command.
10 \end{document}
```

When this document is compiled with `xelatex`, the process fails to complete, generating only `.aux` and `.log` files. The error prompt is `Runaway argument?`. This occurs because `TEX` detects an incomplete or missing argument during macro processing (in this case, `\mycmd` lacks a necessary ar-

gument), causing macro expansion to exceed its expected scope. Consequently, subsequent content cannot be typeset, and the entire PDF output fails. Users are forced to analyze the log to locate the error, unable to verify the pages that were already successfully typeset.

This characteristic stands in sharp contrast to the incremental update and continuous feedback mechanisms prevalent in modern editors, significantly constraining user iteration efficiency.

C.1.2 Tight Coupling Between Compilation and Semantic Phases

In the `TEX` system, syntax parsing, macro expansion, semantic interpretation, and typesetting decisions do not form a clear stratified structure; instead, they are deeply coupled within a single execution path. Macro expansion assumes the role of "semantic modeling" while simultaneously triggering specific typesetting behaviors, thereby conflating abstract structure with layout details. While this design offered high flexibility in the early days, it has exposed serious deficiencies in the context of modern document engineering.

First, macro commands often bear the dual responsibility of structural semantics and layout control. Taking `\section` as an example, this command should logically represent only the structural semantic of "section hierarchy." However, its implementation dictates the font size, spacing, numbering method, and whether a table of contents entry is generated as shown in Listing 8:

Listing 8: Section command with dual responsibility

```
1 \section{Introduction}
```

This single line of code not only declares a new structural node but immediately triggers a series of typesetting side effects, including incrementing counters, formatting the title, writing to the TOC (`.toc`), and generating bookmarks (if `hyperref` is loaded). Because these actions occur simultaneously during the macro expansion phase, downstream tools cannot distinguish whether a command is a "structure declaration" or a "concrete typesetting instruction," making it difficult to understand the document structure without executing the code.

Second, the macro definition mechanism itself cannot distinguish between abstract semantics and concrete implementation. This results in the same syntactic form potentially representing either high-level semantics or

merely a typesetting shortcut. For example: `\def\theorem#1{\textbf{Theorem.} #1}`. Superficially, `\theorem` appears to be a semantic structure (a theorem). In reality, it is a simple wrapper for bold text, lacking numbering, cross-referencing, or hierarchical information. In contrast: `\newtheorem{theorem}{Theorem}` introduces true structural semantics, including automatic numbering, scoping, and citability. However, from the perspective of the $\text{\TeX}$ engine, there is no essential difference between these two definitions during the macro expansion phase—both are simply "executable text substitution rules." This semantic indistinguishability prevents static analysis tools from determining whether a specific macro represents document structure or merely affects visual appearance.

Furthermore, the conditional expansion mechanism directly influences the final structure of the document during the parsing phase, rather than merely acting as a late-stage typesetting choice. Listing 9 shows an example:

Listing 9: Conditional structure

```
1 \ifdefined\includeappendix
2   \section{Appendix}
3 \fi
```

Whether the structural node "Appendix" exists depends entirely on the truth value of a condition at the moment of macro expansion. In other words, the document's logical structure is not a stable, enumerable abstract tree, but a result dynamically generated during execution. Any tool attempting to analyze the document structure without running $\text{\TeX}$ must completely simulate macro expansion and conditional logic, which is virtually unfeasible in practice.

The aforementioned high degree of coupling leads to multifaceted negative impacts. First, static analysis tools find it difficult to intervene. Due to the lack of clear semantic boundaries, tools cannot reliably construct an Abstract Syntax Tree (AST) or structural model, making advanced features like code refactoring, semantic checking, and consistency verification arduous to implement.

Since $\text{\LaTeX}$ compilation results are strongly dependent on instruction order, $\text{\LaTeX}$ introduced engineering improvements to allow users or package authors to insert and execute custom code at specific execution timing points. These mechanisms are not based on an explicit semantic event model, but rather attach to $\text{\TeX}$'s sequential execution flow,

coordinating behavior between packages and document structure by "intercepting execution timing." This enables functions such as lazy initialization, global parameter patching, auxiliary file processing, and layout adjustment. In the $\text{\LaTeX}$ core system, the commonly used hook commands primarily include the four types listed in Table 4:

Table 4: Common $\text{\LaTeX}$ Hook Directives and Descriptions

Directive	Description
<code>\AtBeginDocument</code>	Executes code at the beginning of the document body (after <code>\begin{document}</code>).
<code>\AtEndDocument</code>	Executes code before the document ends (before <code>\end{document}</code>).
<code>\AtEndOfPackage</code>	Executes code immediately after the current package finishes loading.
<code>\AtEndOfClass</code>	Executes code immediately after the current document class finishes loading.

These directives provide a relatively non-invasive method of collaboration between packages and user documents, avoiding the need to forcibly control loading order or directly modify user source code to achieve functional extension.

C.1.3 Lagged Manifestation and Ambiguous Localization of Errors

$\text{\TeX}$'s error detection mechanism is fundamentally an "**execution-driven, post hoc**" paradigm. This is a direct consequence of its compilation model, where linear macro expansion is inextricably interwoven with immediate typesetting. Since the system does not maintain a structured intermediate state capable of real-time verification during execution, errors manifest passively only when execution becomes impossible, exhibiting a set of systemic characteristics.

First, error types are highly conflated at the diagnostic level. Structural defects at the semantic level (e.g., mismatched environments, abnormal parameter expansion) are often reported indistinguishably from failures at the typesetting level (e.g., box overflows, mode-switching errors). Consequently, error messages fail to reflect the relevant level of abstraction, significantly increasing debugging difficulty.

Second, the reported location of an error is rarely its actual point of origin. The line numbers and context provided by $\text{\TeX}$ typically correspond to the point where the system finally "crashed" or stalled,

rather than the source where the issue was introduced. Furthermore, the point of interruption and the origin of the error may be separated by a significant distance: an early, local error might only be exposed in a completely different form after multiple cycles of macro expansion and typesetting decisions. The result is a globalization of local issues, where a subtle defect in a single environment or command is sufficient to cause an irreversible interruption of the entire compilation process.

It must be emphasized that this is not an oversight in specific implementation details, but a characteristic rooted in the fundamental design of $\text{\TeX}$: the system lacks a structured state model independent of the execution process that enables continuous consistency checking. In other words, $\text{\TeX}$ possesses no "prior knowledge" regarding the document's validity; it reports failure only when execution becomes untenable. Listing 10 shows a confusing error case:

Listing 10: Nested section commands

```
1 \documentclass{ctexart}
2 \begin{document}
3 \section{aaa \subsection{bbb}}
4 \end{document}
```

From the perspective of document structure, this usage of `\section` is clearly illegal: a subsection is nested within the parameter of a section title. However, $\text{\LaTeX}$ does not report this as a "structural nesting error." Instead, the system enters an inconsistent state during the macro expansion and typesetting execution, eventually throwing a confusing error message such as "LaTeX Error: Not allowed in LR mode." This message neither identifies the violating structural relationship nor points to the location where the issue actually occurred.

The following examples highlight a typical characteristic of $\text{\TeX}$'s error detection mechanism: a significant offset often exists between the true origin of the error and the reported location. The unclosed-brace example promoted to the main text (Listing 1) illustrates this issue: the second argument of `\frac` lacks a closing brace. Semantically, this is a distinct local structural error. Yet, $\text{\TeX}$ does not immediately report an error upon reading this line; instead, it continues to treat subsequent input as part of the argument, maintaining math mode and attempting to complete macro expansion. It is not until `\end{equation}` is reached that the system detects the failure of the grouping and mode states to converge properly, triggering a fatal "Run-

away argument?" error. By this point, the reported location is far removed from the actual occurrence, forcing users to manually backtrack through the context to verify closed groups and locate the root cause.

A similar latency phenomenon occurs with mismatched delimiters, such as using `\left(` while omitting the corresponding `\right)`. In this scenario, $\text{\TeX}$ persists in waiting for a command to close the extensible delimiter. Superficially, the formula continues to be accepted and parsed, but its internal state is already inconsistent. Ultimately, the error is triggered at `\end{equation}`, potentially interrupting compilation with a message like "You can't use `\eqno` in math mode." It is crucial to emphasize that such errors do not directly point to the logical cause ("missing `\right)`"); rather, they reflect the inability to finalize the typesetting process at the closing stage. This transforms an issue of local structural incompleteness into a failure at the mode or typesetting level.

Both scenarios illustrate a fundamental constraint: $\text{\TeX}$ does not maintain a structural state model capable of immediate verification. Instead, it relies on the passive exposure of issues when the execution process fails. This is one of the root causes of its "lagged manifestation and ambiguous localization" of errors.

Remark C.1. The `.tmu` format effectively mitigates this issue at the structural level. By representing content with implicit around tags and explicit structural wrapping, the format maintains a verifiable structural state during the input phase, preventing error propagation. See Section 3.1 for details.

C.2 Compatibility Dilemma Beneath a Unified Syntax

To address varying requirements (such as modern fonts and Unicode support), the $\text{\LaTeX}$ ecosystem has evolved multiple parallel compilation engines, such as `pdfLaTeX`, `XeLaTeX` and `LuaLaTeX`. This differentiation, rather than being a strength, is a manifestation of system design fragmentation.

From a user's perspective, all three engines accept `.tex` input and adhere to the $\text{\LaTeX}$ macro interface. However, at the system level, they exhibit fundamental differences in the key aspects summarized in Table 5:

These discrepancies imply that the same `.tex` source file does not correspond to the same seman-

Table 5: Comparison of L^AT_EX Engines

Dimension	pdfLaTeX	XeLaTeX	LuaLaTeX
Native Encoding Support	8-bit (requires inputenc)	Native Unicode	Native Unicode
Font System	Type1 / limited OpenType	System Fonts (OpenType)	OpenType (via Lua)
Extension Mechanism	T _E X primitives	T _E X + XeT _E X primitives	T _E X + Lua VM
External Programmability	Very Weak	Very Weak	Strong (Lua)

tic execution environment across different engines.

Compatibility Burden: A single document may require switching between engines to resolve specific issues, forcing users to understand the subtle differences and limitations of each engine.

Take the use of Emojis as an example. If a document requires the direct inclusion of Unicode Emojis (e.g., 🍌), pdfLaTeX is mechanically incapable of supporting such characters.² While XeLaTeX possesses native Unicode support, in practice, it often degrades Emojis to monochrome glyphs or suffers from missing characters due to insufficient font coverage. LuaLaTeX offers the theoretically most complete support path, handling complex characters via OpenType fonts and the Lua layer. However, this 'enhanced capability' comes at a price: LuaLaTeX's compilation speed is typically significantly slower than XeLaTeX, and it introduces an additional dependency on the Lua runtime, expanding both the source of errors and the scope of debugging. Under these constraints, a compromise workflow that actually exists in reality involves: compiling the main body with XeLaTeX for speed and layout stability; separately compiling pages or chapters containing Emojis using LuaLaTeX; and finally stitching the output results from different engines together at the PDF level.

A similar compatibility burden appears in the bibliography processing chain. While L^AT_EX superficially provides a unified .bib data source format, during the actual compilation process, users must choose between different backends like bibtex and biber. These two differ fundamentally in character encoding, sorting rules, and data models as shown in Listing 11:

Listing 11: BibTeX vs BibLaTeX

```
1 \bibliography{ref/refs} %
   References compiled using BibTeX
```

²Some packages such as hwemoji attempt to work around this limitation by pre-compiling emojis into PDF resources and replacing them during compilation. However, this is not native support—i.e., direct rendering from text and font files—but rather a workaround. We still consider pdfLaTeX to be fundamentally incapable of supporting such characters.

```
2 % \printbibliography %
   References compiled using BibLaTeX
   /Biber
```

Visually, these commands differ by only a single line, yet they correspond to two nearly incompatible toolchains. The former triggers the traditional bibtex workflow: the .bib file is parsed by bibtex to generate a .bbl file, employing a data model constrained by 8-bit encoding and a rigid .bst styling mechanism. The latter implicitly requires the biblatex package and the biber backend. It adopts an internal Unicode data model while offloading significantly more logic, specifically sorting, filtering, and formatting, to the macro layer.

Consequently, although both methods "appear to use the same .bib file," they diverge significantly in character encoding support, linguistic processing capabilities, style customization, and compilation steps. A smooth migration cannot be achieved by simply swapping commands. From the user's perspective, this implies that bibliography management is not a stable, interchangeable module, but rather deeply coupled with the engine and packages. Users must not only understand .bib syntax but also explicitly identify which call path their document is entering for bibliography processing. This further exacerbates the cognitive burden regarding tool selection and workflow configuration.

Ecosystem Fragmentation: Certain packages support only specific engines, degrading document portability. A common example is the fontspec package, used for loading system fonts (OpenType/TrueType) and Unicode settings. Under pdfLaTeX, this triggers a fatal error (cannot-use-pdf_TEX) because pdf_TEX lacks native Unicode support.

Increased Cognitive Load: These factors compel users to learn not just L^AT_EX syntax, but also how to select and configure the appropriate compilation engine to achieve their desired document output.

C.3 Alienation of the Tool Ecosystem

To compensate for the aforementioned core design deficiencies, the $\text{\LaTeX}$ ecosystem has spawned a series of complex compensatory mechanisms, which have in turn introduced new problems.

C.3.1 Multi-pass Compilation: A Fragile Stopgap

In $\text{\LaTeX}$, the accurate generation of cross-references, tables of contents, and bibliographies relies on multiple compilation passes to make up for the absence of internal state. The typical workflow involves a first pass that generates auxiliary files (like `.aux` and `.toc`) to record state, followed by subsequent passes that read these files to populate cross-references, page numbers, or chapter titles. If the document includes citations, external tools such as BibTeX or Biber must also be integrated into the process. For example, the standard procedure for a document with references often follows the sequence: `pdflatex` $\rightarrow$ `bibtex` $\rightarrow$ `pdflatex` $\rightarrow$ `pdflatex`.

This mechanism imposes several burdens:

- **State Fragmentation:** Document state is scattered across multiple external files.
- **Cognitive Load:** Users must master complex “compilation rituals.” For instance, failure to run BibTeX results in all citation numbers appearing as `??`.
- **Debugging Difficulties:** Error tracing is notoriously difficult. An illegal character in a `.bib` file might trigger an error in the `\bibliography{refs}` command, causing the error message to be completely detached from the actual root cause.

In extreme cases, cross-references remain generated during the first pass, leaving the user with a mass of undefined numbers or empty tables until the multi-pass cycle is complete. This reliance on external side effects to patch internal architectural flaws is, in essence, a form of technical debt.

C.3.2 Long-Term Suppression of the Tool Ecosystem

The batch processing design of $\text{\TeX}$ has stifled the development of peripheral tools. Real-time preview in $\text{\LaTeX}$ editors is merely a simulation achieved by frequently triggering background compilations, which suffers from significant latency and layout

inconsistencies. For example, in documents containing numerous floating objects and formulas, even modifying a small paragraph can trigger a global re-calculation of pagination, causing the preview to lag. Furthermore, features like intelligent code refactoring or syntax completion are virtually impossible to implement effectively without a structured document tree. Most automation tools rely on regular expression matching rather than syntax tree analysis, leading to limited error-checking capabilities and a high propensity for false positives or missed errors.

This structural dilemma is not accidental; rather, it is rooted in the batch-processing paradigm of $\text{\LaTeX}$ itself. As quoted in the main text, Frank Mittelbach, the technical lead for $\text{\LaTeX}$, candidly admitted in an interview (de Souza, 2021) that the separation between local and global processing “is right now hard in $\text{\TeX}$.” For completeness, the fuller exchange runs:

PN: “I try to look at this issue from the point of view of global and local, and interactivity is just like a... This is probably a change that happens very fast, and that you worried only about the local stuff, but the separation between local and global in $\text{\LaTeX}$ seems to be hard.”

FMi: “First of all, it is right now hard in $\text{\TeX}$...”

C.4 Comparison: Design Paradigms of Structured and Incremental Systems

Systems like GNU $\text{\TeX}$ macs adopt a distinct architecture: the document maintains a structured tree representation in memory, where local modifications trigger immediate local repainting, and structural validity checks occur during the editing phase. For instance, modifying a formula or a floating object redraws only the affected region. Cross-references and numbering update instantly, ensuring the user always interacts with a predictable and previewable document state. Errors are isolated within local modules, preventing global system crashes.

This contrast shows that $\text{\LaTeX}$ ’s need for multiple compilation rounds and external tools is not unavoidable and instead results from a compensatory design strategy. These limitations could be completely bypassed by refactoring the architecture. Ultimately, the fundamental problem of $\text{\LaTeX}$ lies not the quality of its typesetting, but the de-

fects of global dependency and state management inherent in its execution model.

D Limitations of TeX in User Experience Design

TeX and its derivative, the L^ATeX ecosystem, have long occupied a central position in the field of academic typesetting. However, the overall user experience has failed to evolve in sync with the advancement of computing environments and user expectations. From the continuous bloating of distribution sizes to the performance and interaction barriers imposed by the compilation model, and further to the long-standing engineering defects in language design, the TeX ecosystem falls short in modern writing contexts—particularly regarding "usability," "maintainability," and "adaptability to modern workflows." This section analyzes the limitations of TeX's user experience design from multiple dimensions, including deployment costs, language structure, and practical user experience.

D.1 Issues with Distribution Scale and Deployment Models

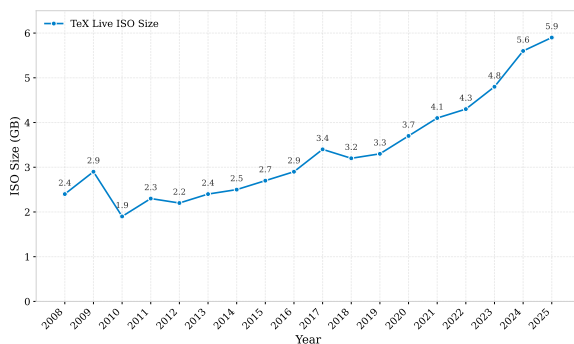


Figure 6: TeX Live ISO Size Trends

Figure 6 illustrates the trend in file size for the most popular TeX Live ISO from 2008 to 2025. The horizontal axis represents the year, and the vertical axis represents file size in GB. Starting at 2.4 GB in 2008, the line shows an overall upward trajectory, reaching 5.9 GB by 2025. Despite minor fluctuations (such as a drop to 1.9 GB in 2010 and a slight dip to 3.2 GB in 2018), the long-term trend demonstrates a nearly twofold increase in size, reflecting the continuous expansion of software packages.

This volumetric growth is primarily driven by TeX Live's role as a comprehensive L^ATeX distribution, which constantly integrates new fonts, documentation, multi-language support, and packages

to adapt to user needs and technological advancements. For instance, while early versions focused on core functionality, later iterations incorporated extensive PDF support, graphics libraries, and extensions, resulting in volume bloat.

However, this continuous expansion warrants critical reflection. Taking TeX Live as an example, a significant portion of its distribution volume consists of documentation sets and font resources. For beginners or light users requiring only basic typesetting functions, such overhead constitutes a clear redundant burden, consuming installation time, disk space, and maintenance effort. Although the TeX ecosystem emphasizes modularity at the package level, the distribution strategy (as illustrated in Figure 7) still prioritizes full installation as the default. Minimalist installation options, while available, are neither highlighted nor widely adopted. Consequently, most users default to the full version, effectively "normalizing" the issues of bloat and resource waste.

While distributions like MikTeX attempt to address this, a more aggressive promotion of lean versions in default configurations by making documentation, examples, and large font sets optional is far more rational in terms of efficiency, resource utilization, and environmental impact, particularly in network- or storage-constrained scenarios.

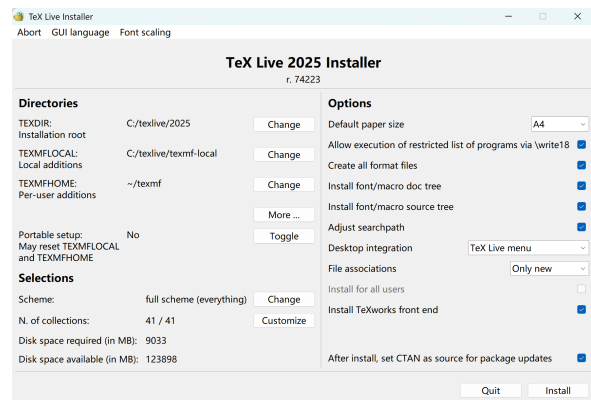


Figure 7: TeX Live Installer

Figure 7 illustrates the complex installation interface. In contrast, because the .tmu format encodes document structure explicitly, an editor for .tmu can adopt an on-demand loading mechanism, where features are only loaded when a document structure actually requires them. This significantly reduces the installation size and startup time.

Compounded by its underlying compilation model, L^ATeX presents a series of severe usability

ity obstacles. Stemming from inherent contradictions in language design, these issues, manifested as massive installation footprints, slow compilation speeds, and steep learning curves, collectively form a “deterrent barrier” for modern users.

D.2 Real-World Performance Dilemma

L^AT_EX’s performance disadvantages warrant discussion, as they render the system incongruous with fast-paced modern workflows. Here, ‘performance’ is not defined by interaction latency or incremental feedback, but rather serves a typesetting model centered on **batch processing**. The underlying assumption is that the user provides a relatively complete and stable source file, which the system processes through one or more full compilation cycles to generate quality-controlled output. Under this batch-oriented model, optimization prioritizes the correctness and consistency of the final result, rather than immediate responsiveness during the writing process.

However, this throughput-oriented performance goal aligns poorly with modern documentation patterns that prioritize low latency, localized feedback, and continuous interaction. Frequent, minor modifications during writing often trigger a re-parsing and re-typesetting of the entire document, introducing significant wait times that disrupt the continuity between editing and thinking.

Beyond performance unpredictability within a single platform, L^AT_EX exhibits significant discrepancies in compilation time across different operating systems. For instance, under identical document and package configurations, compilation on Windows is often noticeably slower than on Linux or Unix-like systems. This disparity stems not from hardware differences, but from underlying factors such as file system efficiency, process instantiation overhead, font and I/O management, and the optimization level of the T_EX toolchain for specific platforms.

For users, this cross-platform inconsistency undermines system comprehensibility and predictability. The same project may yield vastly different compilation experiences on a personal computer, a lab server, or a cloud environment, making ‘performance issues’ difficult to reproduce, isolate, or optimize. In collaborative scenarios, this variance amplifies into actual collaboration friction, effectively influencing workflow rhythm and tool selection.

Furthermore, L^AT_EX’s reliance on a multi-stage

compilation workflow exacerbates runtime performance costs. Functions such as bibliography management, indexing, and cross-referencing are typically offloaded to multiple external tools, requiring users to run multiple compilation rounds to achieve stable output. This design choice is not driven by performance optimization, but by the language’s inherent inability to express complex dependencies within a single compilation pass. Consequently, the overhead of repeated disk I/O, redundant scanning, and process startup is systematically offloaded onto the user.

D.3 Intrinsic Weakness: Absence of Engineering Standards

L^AT_EX 2_ε is fundamentally a collection of macros built upon the T_EX macro expansion mechanism. Its abstraction capabilities rely primarily on untyped text substitution and grouping scopes, rather than explicit language-level structures. Although the community has actively advanced the L^AT_EX3 project in recent years to ameliorate this situation, the project has not been released as a standalone version. Instead, it follows a strategy of progressive evolution, gradually integrating into the existing L^AT_EX 2_ε kernel. Currently, the expl3 programming layer provided by L^AT_EX3 serves as the foundation for numerous large-scale packages. Consequently, users often utilize its mechanisms indirectly and unconsciously in their daily workflows. This transition primarily results in improved interface consistency, maintainability, and engineering capabilities, accompanied by a degree of performance enhancement.

However, this progressive integration fails to eliminate the long-standing structural contradictions within L^AT_EX 2_ε’s language design. As a system built on a macro expansion language, its approach to modularity, interface expression, and state management stands in fundamental tension with modern engineering practices. Table 6 systematically outlines these inherent contradictions and their impact on usability and system stability.

L^AT_EX lacks formal engineering standards for package development. While the lpp1 license provides some guidance, there are no enforced standards for:

- Package documentation quality
- API stability and versioning
- Error handling and reporting

- Testing and validation

This has led to a fragmented ecosystem where package quality varies widely, and compatibility issues are common (de Souza, 2021).

In summary, $\text{\LaTeX} 2_{\epsilon}$ represents a macro language with a severe deficit in engineering rigor. Its system stability, maintainability, and user experience depend heavily on informal conventions and user expertise, rather than structural guarantees provided by the language design itself. This deficiency not only increases the complexity of extension and debugging but also establishes an unavoidable historical burden that constrains future systemic evolution.

D.4 User Experience Barriers from a Practical Perspective

The structural contradictions in language design discussed above do not remain abstract concepts. Instead, they directly translate into tangible user experience barriers repeatedly encountered during document composition, package combination, and troubleshooting. These issues do not stem from isolated implementation defects or inadequate documentation, but are intrinsic to the macro expansion mechanism and engineering constraints upon which $\text{\LaTeX} 2_{\epsilon}$ relies.

Lack of Usability and Interface Consistency:

Since command interfaces are constrained by convention rather than language-level mechanisms, $\text{\LaTeX} 2_{\epsilon}$ lacks a unified standard for parameter syntax, optional argument placement, and starred variants across different packages. A typical example is `\newcommand`, which supports only a single, fixed-position optional argument, whereas more complex interface requirements necessitate the use of packages like `xparse` (via `\NewDocumentCommand`). This deficiency in interface expressiveness forces users to frequently switch mental models between different packages, significantly increasing the cost of learning and usage.

Fragility and Implicit Errors Induced by the Global Namespace: $\text{\LaTeX} 2_{\epsilon}$ lacks language-level namespaces and encapsulation mechanisms; all commands and variables share a global symbol table. This design makes it difficult to guarantee independence between packages, with loading order often directly affecting document behavior. In the worst-case scenario, definitions of the same command by different packages may result in “silent overwriting,” causing subtle but impercepti-

ble changes in document output, thereby weakening system predictability and reliability.

High Opacity in Error Diagnosis and Debugging: Under an execution model where macro expansion and typesetting decisions are deeply intertwined, error messages are often detached from their root causes. Common alerts like “Undefined control sequence” or “Overfull \hbox” typically indicate the surface location where the issue was triggered, rather than the origin of the error. Troubleshooting often forces users to rely on empirical methods like step-by-step commenting and regression testing. This process is both inefficient and difficult to systematize, further exacerbating the maintenance burden for complex documents or large projects.

Limited Extensibility and Non-Linear Growth in Implementation Complexity:

When users attempt to implement features with even slight complexity (e.g., multiple optional arguments, starred variants, conditional interfaces, or nested logic), they are often compelled to utilize extensive low-level macro hacks to bypass the language’s expressive limitations. Such implementations typically rely on implicit state, special naming conventions, and opaque expansion orders, leading to a significant degradation in code readability and maintainability. Consequently, a linear increase in feature complexity is often accompanied by an exponential rise in implementation complexity.

High Learning Costs and the Accumulation of “Tacit Knowledge”:

Collectively, these issues cause the learning path for $\text{\LaTeX} 2_{\epsilon}$ to depend heavily on tacit knowledge, including grouping scope rules, internal naming conventions (such as `@-class` commands), macro expansion timing, and execution order. This knowledge is not explicitly expressed through language mechanisms but is scattered across package implementations and community lore, making it difficult for new users to establish a stable, transferable cognitive framework.

D.5 Contributions and Constraints of $\text{\LaTeX} 3$

The $\text{\LaTeX} 3$ project emerged as a response to the structural dilemmas described above. It attempts to introduce a programming layer with distinct software engineering characteristics while preserving the underlying $\text{\TeX}$ macro expansion model. By systematically standardizing and constraining the macro programming practices of $\text{\LaTeX} 2_{\epsilon}$ through modular naming conventions, an explicit data type

Table 6: Core Structural Contradictions in $\text{\LaTeX} 2_{\epsilon}$ Language Design and their Engineering Consequences

Contradiction	Description	Impact
Global Naming vs. Modularity	$\text{\LaTeX} 2_{\epsilon}$ lacks namespaces; all commands and variables share a global symbol table	High risk of package conflicts; loading order directly affects behavior; system fragility
Text Substitution vs. Structured Interfaces	Macros are essentially untyped token substitutions, lacking parameter signatures and type systems	Static checking is difficult; function interfaces are uncomposable; parameter passing is error-prone
Local State via Grouping vs. Compiler Awareness	State management relies on $\text{\TeX}$'s grouping and rollback mechanism rather than explicit variable scoping	Compilers cannot resolve scope; state leakage and logical errors are difficult to detect
Conventional Interfaces vs. Automatic Verification	Interface constraints rely on documentation and voluntary compliance, not language-level enforcement	Parameter types/counts cannot be statically verified; errors are only exposed at runtime
Compatibility vs. Modern Features	To maintain compatibility with $\text{\LaTeX} 2_{\epsilon}$, implementation of new features is cumbersome	Requires reliance on advanced packages like <code>xparse</code> or complex hacks; high learning and maintenance costs

system, and controlled scope management, the project aims to bring order to the system. As its core component, the `expl3` programming layer improves interface consistency and code maintainability, thereby reducing the risks of naming conflicts and implicit state interference during package integration and extension.

However, the improvements offered by $\text{\LaTeX} 3$ remain incremental and limited. Its commitment to long-term backward compatibility with $\text{\LaTeX} 2_{\epsilon}$ dictates that the new and old programming paradigms coexist for a considerable period. Users are required not only to master the new interface model but also to interact with existing packages and historical conventions. This coexistence complicates the ecosystem and renders the migration process slow and uncertain. It serves as a testament to how the historical baggage accumulated over $\text{\LaTeX}$'s long evolution continues to constrain its further engineering development.

D.6 Limitations of Modern Collaboration Platforms: A Case Study of Overleaf

In response to $\text{\LaTeX}$'s structural deficiencies regarding collaboration support and usability, the academic community has advanced a series of improvement measures. Among these, online collaboration platforms, represented by **Overleaf**, have emerged as the most influential solution in recent years. By providing an integrated cloud-based environment, Overleaf significantly improves the onboarding experience and multi-user collaboration workflows for $\text{\LaTeX}$. However, viewed from the perspective of system architecture, Overleaf's design does not address the fundamental issues of $\text{\LaTeX}$'s core model. Its role is closer to the encapsulation and optimization of existing workflows

rather than a revolution of the underlying paradigm.

D.6.1 Key Improvements by Overleaf

Overleaf has effectively mitigated several obstacles inherent in traditional $\text{\LaTeX}$ usage through the following mechanisms:

- **Simplified Environment Configuration:** Users are no longer required to install a full $\text{\TeX}$ distribution (such as `TeX Live` or `MiKTeX`) locally. Document composition commences via a web browser, thereby eliminating the technical barrier of local deployment.
- **Automated Compilation Workflow:** The system automatically manages multi-pass compilation and external tool invocations (e.g., `BibTeX`, `MakeIndex`, `Biber`). Users need only trigger a single "Recompile" action; the platform's backend manages complex dependencies, reducing the technical knowledge required of the user.
- **Enhanced Collaboration Features:** The platform supports multi-user real-time editing, commenting/annotation, and version history tracking. It provides a foundational experience comparable to modern documentation tools (such as `Google Docs`), significantly outperforming traditional collaboration methods based on email or `Git`.

These platform-level improvements have significantly expanded the applicability of $\text{\LaTeX}$ in non-professional typesetting scenarios, such as education and scientific research.

D.6.2 Constraints of Underlying Architecture

Despite Overleaf’s significant progress in user interface and workflow, its foundation still relies entirely on the traditional $\text{\LaTeX}$ compilation mechanism (Overleaf Team, 2023), which introduces several insurmountable limitations:

- **Preview Latency Issues:** The so-called “real-time preview” is essentially periodic remote compilation and PDF retrieval, not true instant rendering. When document size increases or complex packages (e.g., TikZ, algorithm2e) are included, compilation time rises significantly, leading to a degraded interactive experience.
- **Limited Conflict Resolution in Collaboration:** Since $\text{\LaTeX}$ source files are unstructured plain text, the platform cannot parse the document structure at a semantic level. When multiple users simultaneously modify the same paragraph, table, or macro definition, the system can only perform line-based text merging. It is unable to identify logical conflicts, ultimately necessitating manual intervention for resolution.
- **Restricted Functional Extensibility:** Overleaf cannot transcend the boundaries of $\text{\LaTeX}$ ’s own expressive capabilities. For instance, it fails to support true WYSIWYG editing modes, nor does it easily integrate dynamic content (such as interactive charts or real-time data binding). Implementing these features requires a structural overhaul of the typesetting engine, rather than mere adjustments to the frontend interface.

D.6.3 Impact on Trajectory of Technological Evolution

The success of Overleaf serves as a double-edged sword: while it enhances user experience, it simultaneously masks $\text{\LaTeX}$ ’s inherent technical debt and structural defects, thereby diminishing the motivation to explore next-generation typesetting systems. In contrast, systems such as $\text{\TeX}$ macs or Typst attempt to refactor the document model and syntax design from the foundation up, yet they struggle to disrupt $\text{\LaTeX}$ ’s dominant position due to entrenched user habits.

Consequently, although Overleaf optimizes the usability of the existing ecosystem and resolves

certain surface-level pain points, it objectively establishes a form of “path dependence,” delaying the demand for more fundamental technological transformation. This platform-driven optimization renders the legacy system more usable but fails to propel it toward a revolutionary direction characterized by greater structure, extensibility, and interactivity.

E Related Work

The challenges of using $\text{\LaTeX}$ in LLM-driven workflows have been documented from multiple angles. On the generation side, benchmarks reveal a consistent pattern: LLMs struggle with $\text{\LaTeX}$ ’s long-range syntactic dependencies and implicit semantic conventions. The TeXpert benchmark (Kale and Nadadur, 2025) reports only 15% accuracy on complex $\text{\LaTeX}$ tasks, with logical errors—not superficial typos—accounting for 54% of failures, demonstrating that even state-of-the-art LLMs struggle with $\text{\LaTeX}$ code generation. Similar difficulties arise in modality-crossing tasks: Ling et al. (2025) found that reinforcement-learning-based table-to- $\text{\LaTeX}$ conversion suffers from persistent token inefficiency rooted in $\text{\LaTeX}$ ’s verbose syntax, while Xia et al. (2024) performed large-scale benchmarks revealing that Equation-to- $\text{\LaTeX}$ and Table-to- $\text{\LaTeX}$ conversion tasks remain challenging even with extensive training data (Edit Distance > 0.21), attributing these high edit distances to the information entropy of $\text{\LaTeX}$ source code. These generation-level failures become systemic when $\text{\LaTeX}$ is embedded in end-to-end scientific pipelines: Lu et al. (2024) demonstrated that in a fully automated discovery framework (The AI Scientist), the $\text{\LaTeX}$ writing stage is a critical bottleneck—the model cannot perceive the rendered PDF, producing overflowing tables and placeholder text that require manual correction. Jain et al. (2026) arrived at a complementary conclusion from the editor side: their Bibby AI system found that raw compiler logs are insufficient for error localization and proposed combining logs with a live AST, effectively acknowledging that $\text{\LaTeX}$ ’s batch feedback model must be augmented with structural representations to support reliable AI assistance. Lyn and Graham (2025) further identify an “execution illusion” where LLMs produce linguistically fluent but unexecutable $\text{\LaTeX}$ code for scientific formatting, introducing TransLaTeX, a reasoning-and-control framework with compiler-level verifi-

cation. Our work extends this line of reasoning by arguing that such augmentation is inherently limited—a document format designed around explicit tree structure, rather than one retrofitted with AST recovery, can more fundamentally resolve the feedback and semantic transparency problems these systems encounter.

The representation efficiency of $\text{\LaTeX}$ also poses challenges upstream, during data preparation and model training. [Paster et al. \(2023\)](#) documented the substantial engineering effort required to extract and normalize 14.7B tokens of mathematical content from web sources containing heterogeneous $\text{\LaTeX}$ fragments, attaching particular importance to mathematical content for LLM training. [Lin et al. \(2024\)](#) showed that direct LLM processing of documents incurs prohibitive costs, achieving up to 30% reduction only through semantic hierarchical indexing—a workaround necessitated by the verbosity of the underlying format, highlighting the need for more efficient document representations. That markup structure matters for model learning has been demonstrated more directly: [Li et al. \(2022\)](#) showed with MarkupLM that jointly modeling text and markup tags improves document understanding, while [Taylor et al. \(2022\)](#) found in training Galactica that $\text{\LaTeX}$ equations are a first-order component of scientific language whose representational overhead affects model behavior. Taken together, these findings suggest that $\text{\LaTeX}$ ’s syntactic redundancy—where equivalent renderings admit multiple source forms (e.g., $\text{\code{\frac{a}{b}}}$ vs. $\text{\code{\{a \over b\}}}$)—reduces next-token predictability and inflates training costs. Evidence from neural code modeling further supports the importance of structured representations: TreeDiff ([Zeng et al., 2025](#)) demonstrated that AST-guided span masking significantly outperforms token-level random masking for diffusion-based code generation, particularly for longer sequences (36.59% vs. 33.54% pass@1 on HumanEval with 1024-token prompts), suggesting that hierarchical representations enable models to better capture long-range dependencies. In Section 4.3, we provide direct experimental evidence for this hypothesis by comparing fine-tuning convergence on $\text{\LaTeX}$ versus the lower-entropy $\text{\code{.tmu}}$ format, which provides a more information-dense and structurally explicit representation than linear $\text{\code{.tex}}$, enabling more efficient learning.

On the systems side, dissatisfaction with $\text{\TeX}$ ’s batch compilation has produced two distinct architectural responses, neither of which fully ad-

dresses the problem our work targets. The first is incremental markup compilation: [Mädje \(2022\)](#) and [Haug \(2022\)](#) designed Typst with a functional type system and incremental compiler that achieves sub-second preview updates, resolving the latency problem but retaining a source-editing paradigm that still separates authoring from visual output. The second is interactive augmentation of existing $\text{\LaTeX}$ workflows: [Gobert and Beaudouin-Lafon \(2022\)](#) proposed i- $\text{\LaTeX}$, which overlays interactive “transitional” widgets onto a code editor to bridge source and rendered output—an approach that alleviates feedback delays without removing the underlying batch compilation dependency. Both strategies leave the core tension unresolved: the document’s authoritative representation remains either linear markup (Typst) or $\text{\TeX}$ source (i- $\text{\LaTeX}$), rather than an explicit semantic tree. The structured WYSIWYG approach pioneered by [van der Hoeven \(2001\)](#) with GNU $\text{\TeX}$ macs took a more radical position, demonstrating that high-quality mathematical typesetting can be achieved within a tree-based editor that maintains structure as its primary representation. Empirical evidence supports the case for such alternatives: [Knauff and Nejasmic \(2014\)](#) showed in controlled experiments that $\text{\LaTeX}$ users can be slower and produce more formatting errors than Word users, while [Tan and Rigger \(2024\)](#) systematically documented visual inconsistencies across $\text{\TeX}$ engines and TeX Live versions, substantiating the fragility of the ecosystem under real-world version drift. [Gardner et al. \(2025\)](#) implement a deep learning library entirely in pure $\text{\LaTeX}$; when the document is compiled, the $\text{\LaTeX}$ engine trains the network and generates figures, with their paper taking 48 hours to compile, illustrating the extreme runtime cost of $\text{\LaTeX}$ when used as a programmable substrate. Our work builds on the $\text{\TeX}$ macs lineage through the tree-structured $\text{\code{.tmu}}$ format (Section 3), and contributes new evidence that this representation not only improves editing responsiveness but also yields measurable advantages for LLM tasks—a dimension absent from prior structured-editor evaluations.

F The $\text{\code{.tmu}}$ format v.s. other editors

The $\text{\code{.tmu}}$ format builds on the document representation of the GNU $\text{\TeX}$ macs lineage ([van der Hoeven, 2001](#); [Liii Network, 2025b,a](#)), in which the document is stored as an explicit, tree-structured representation rather than as linear markup. This

family of formats underpins the WYSIWYG structured editors that maintain document structure as their primary representation. Table 7 compares an editor based on the .tmu format with several other editors currently available.

Table 7: Comparison between a .tmu editor and alternative editors. Sources: .tmu format (Liii Network, 2025b), T_EXmacs (van der Hoeven, 2001), T_EX/L^AT_EX (Lamport, 1994), LyX (Jackson and Voß, 2001), Typst (Mädje, 2022).

Editor	WYSI	Struct.	Unicode	T _E X Compat.
.tmu (this work)	Yes	Yes	Yes	Partial
T _E Xmacs	Yes	Yes	Partial	Partial
T _E X/L ^A T _E X	No	Yes	Partial	Full
Word	Yes	No	Partial	Incompatible
LyX	Partial	Yes	Partial	Full
Typst	No	Yes	Yes	Partial

The core technical challenge in supporting WYSIWYG editing from a structured representation lies in how mathematical formulas are represented and subsequently rendered. Unlike T_EX, which represents formulas through layout-oriented markup in the form of typesetting instructions, the .tmu format adopts a tree-based, functional representation for both mathematical formulas and the document structure itself. The following discussion evaluates this representation with specific examples.

G Fast Reference Rendering

Taking bibliographic citations as an instance: in the L^AT_EX ecosystem, \cite serves merely as a macro interface, with actual citation relationships established indirectly via intermediate files such as .aux and .bbl. Consequently, any modification to bibliography entries typically necessitates multiple global compilation cycles, exemplifying a classic batch processing model. In contrast, the .tmu format encodes citation relationships as structural links directly embedded in the document tree. To update a reference, a structured editor for .tmu performs only a local search and re-renders the specific leaf nodes involved, thereby avoiding a full re-parsing of the document.

H On-demand plugin loading

Because the .tmu format encodes document structure explicitly, a .tmu editor can adopt a monolithic installation paradigm, diverging from the package repository distribution model exemplified by T_EX

Live. A single installation package delivers a complete and self-contained editing and typesetting system, encompassing core executables, a built-in scripting runtime, a document model, and a rendering engine. Consequently, users avoid managing numerous independent packages or resolving granular macro dependencies. Although functional extensions exist as plugins, they do not load upon startup; instead, they load dynamically at runtime only when specific document structures in the .tmu tree trigger the corresponding requirements. This on-demand mechanism ensures that complexity is dictated by document content rather than a pre-configured feature set.

It is noteworthy that a .tmu editor requires an installation package of merely one hundred megabytes, occupying only several hundred megabytes of local storage upon extraction, yet it offers immediate, out-of-the-box support for the editing and export of complex mathematical documents. In stark contrast, even a minimalist L^AT_EX installation typically demands a local environment exceeding 1 GB, while a full distribution reaches approximately 6 GB in 2025.

This disparity stems from a fundamental divergence in system design. L^AT_EX front-loads "potential future capabilities" as an installation cost, maintaining compatibility through its macro language and package ecosystem. Conversely, the .tmu approach defers complexity to runtime, confining it within the actual execution path. Extensibility is achieved via a unified document tree model and a runtime plugin mechanism. Consequently, complexity is activated dynamically by specific documents at runtime, whereas L^AT_EX's complexity manifests primarily during the installation and configuration phases.

In summary, T_EX Live installs an ever-accumulating repository of historical packages, whereas the .tmu approach provides an evolvable document system. The complexity of the former is front-loaded to the installation phase, while the latter is triggered by the document at runtime.

I Benchmark on compiling/rendering time

Limited by the design of L^AT_EX, the compilation process requires significant time for documents that are rich in cross-references, tables of contents, and bibliographies. We chose 6 papers from arXiv that satisfy this richness criterion as benchmark

documents. Note that a .tmu editor is a WYSIWYG editor, so the comparison is unfair! The time consumption for the .tmu editor is

$$t_{\text{compiling}} + t_{\text{rendering}} + t_{\text{IO}},$$

where $t_{\text{compiling}}$ is the compiling time, $t_{\text{rendering}}$ is the rendering time, and t_{IO} is the *extra* I/O overhead for WYSIWYG editing. As shown in Figure 8, even with the extra I/O process, the .tmu editor outperforms L^AT_EX in compiling/rendering time for most documents. Note that for document arXiv:2502.17655, the .tmu editor compiles and renders slower than L^AT_EX; this exception is due to the size of the document, as it has 120 pages. Therefore, the I/O overhead accounts for a large proportion.

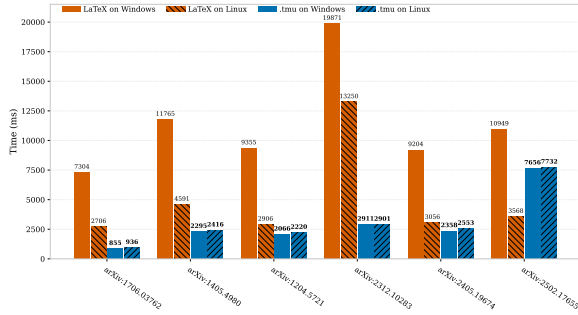


Figure 8: Benchmark on full compilation. Each bar represents the average of three trials.

Another limitation of L^AT_EX is slow incremental updates. The incremental-update experiment (adding new sections, adding tables in some paragraphs, modifying the relations between labels and references, and rearranging the positions of content slightly) is summarized in Section 4. Note that for document arXiv:2502.17655, the .tmu editor outperforms L^AT_EX on incremental updates, which seems to contradict the result of the full-compilation experiment above. The reason for such “contradiction” is due to the I/O overhead: with the .tmu format, the I/O process is only activated when the user opens the document. Therefore, for incremental updates, $t_{\text{IO}} = 0$.

J Prompts for evaluating structure locating

You are an expert in L^AT_EX. Your task is to read the main.tex and answer the following questions:

1. Count the number of sections.
2. Count the number of subsections.

3. Count the number of figures and tables.
4. Count the number of cross-references and bibliography references.
5. In which section does Equation 10.15 appear?
6. In which subsection does Equation 8.66 appear?
7. Is there a direct proof below Equation 12.1?
8. In what context does formula A.3 appear?
9. In which environment is Definition 4.4 first cited?
10. What is the number of the first equation after the first citation of Definition 4.4?
11. In which environment is Definition 7.1 first cited?
12. What is the number of the first equation after the first citation of Definition 7.1?
13. How many steps are there in the proof of Lemma 6.4?
14. Which definition or lemma numbers are directly used in the proof of Lemma 6.4?
15. How many steps are there in the proof of Lemma 8.3?
16. Which definition or lemma numbers are directly used in the proof of Lemma 8.3?
17. In which section does Citation 1 first appear?
18. In which subsection does Citation 6 first appear?
19. What was Citation 25 originally used to prove?
20. Has Citation 31 appeared in the article?