The Journal of Open Source Software

Litrepl: Literate Paper Processor Promoting Transparency More Than Reproducibility

Sergei Mironov [1]

1 Independent Researcher, Armenia

DOI: 10.21105/joss.08384

Software

- Review
- Repository
- Archive

Editor: Matthew Feickert

Reviewers:

- @itepifanio
- @agoose77

Submitted: 05 January 2025
Published: 24 July 2026

Summary

Litrepl is a lightweight text processing tool designed to recognize and evaluate code sections within Markdown or Latex documents. This functionality is useful for both batch document section evaluation and interactive coding within a text editor, provided a straightforward integration is established. Inspired by Project Jupyter, Litrepl aims to facilitate the creation of research documents. Following developments in software deployment theory, however, we shift our focus from informal reproducibility to enhancing transparency in communication with interpreters by relying on POSIX interfaces: named pipes are accessible via the file system, asynchronous process identifiers are visible in status reports. The tool provides a comprehensive command-line interface, making it easier to integrate with code editors. The project repository includes a reference Vim plugin.

Statement of need

Literate Programming, formulated by Donald Knuth, shifts the focus from merely coding to explaining computational tasks to humans. This approach is seen in the WEB system (Knuth, 1984) and its successor tools, which use a document format that interleaves code sections with explanatory text. These systems can produce both readable documentation and executable code, and over time, this concept has evolved towards simplification (Ramsey, 1994).

The Read-Evaluate-Print Loop (REPL), a key concept in human-computer interaction, gained importance in the LISP and APL communities (Iverson, 1962; McCarthy, 1960). By combining a command-line interface with a language interpreter, REPL enables incremental and interactive programming, allowing users to modify the interpreter state directly. This approach is believed to enhance human thought processes by keeping users actively involved (Granger & Pérez, 2021).

A pivotal development in this field was the IPython interpreter (Perez & Granger, 2007), which led to the Jupyter Project and its Jupyter Notebook format (Kluyver et al., 2016). This format consists of logical sections like text and code that can interact with language interpreters, enabling REPL-like programming for well-structured, shareable documents. This concept became a part of *Literate Computing* (Pérez & Granger, 2015), which aims to reach broad audiences, enhance reproducibility, and promote collaboration. Key technical aspects include bidirectional communication between the Jupyter Kernel and Notebook renderer, alongside client-server interactions between the web server and user browser.

We believe that reproducibility is indeed crucial in the Literate Computing framework, enhancing communication among dispersed researchers. However, as illustrated by (Dolstra et al., 2010), we argue that this challenge exceeds the capacity of a single tool or library, needing a system-scale solutions. Such a solution would define a reproducible environment in a formal language and would include a closure of dependencies of every installed software component. The

responsibility of tool writers remains minimizing the added dependencies and keeping the communication transparent to user (Vallet et al., 2022).

Software developers, researchers, and technical writers might need a way to combine such environment descriptions with an equally transparent record of their REPL-driven work: something that behaves like an ordinary process managed by the operating system, produces plain text artefacts suitable for version control, and can be reliably re-deployed and replayed.

In Table 1 we list popular literate programming tools together with a new **Litrepl** tool, developed by the author. We note that most existing tools depend on Jupyter kernels technology adding Web client-server and the Xeus/ZeroMQ message passing library to the environment. Moreover, both Web and ZeroMQ protocols require maintaining mutable states in computer memory aiming to support multi-user modes and various network conditions.

Litrepl, in contrast, focuses on a simple single-user communication held via bi-directional text stream objects available in a modern operating system. We reduce the internal communication state down to a few entities all visible via the file system by leveraging POSIX (IEEE and The Open Group, 2024) features. By excluding network protocols from the pipeline, we reduce the size of the project's dependency closure and thereby improve alignment with formally reproducible frameworks, batch processing environments, and continuous integration systems.

Table 1: Litrepl compared to other literate programming tools .

Name	Document format	Data format	Interpreter	Backend	Frontend
Jupyter	Jupyter[j1]	Rich	Many[j2]	Client-server via ZeroMQ[j3]	Web, Editor
Quarto	Markdown[q1]	Rich	Few	Modified Jupyter[q2]	Web, Editor
Codebraid	Markdown[c1]	Rich	Many[c2]	Similar to Jupyter[c3]	Editor
R markdown	Markdown[r1]	Rich	Few[r2]	Linked libraries[r3]	Editor
Litrepl	Markdown, LaTeX	Text-only	Few	POSIX Pipes	Command line, Editor

[j1] Jupyter Notebook Format
[j2] Available kernels
[j3] Details on Jupyter backend communication link
[q1] Quarto Markdown extensions
[q2] Quarto uses the slightly modified Jupyter kernel protocol. See Jupyter kernel support and Quarto echo kernel
[c1] Codebraid Markdown extensions are described as code chunks formatting
[c2] Codebraid supports Jupyter kernels link
[c3] Codebraid supports interactive editing via Jupyter kernels.
[r1] The R Markdown language
[r2] In RMarkdown most languages do not share a session between code sections, the exceptions are R, Python, and Julia link
[r3] For Python, RMarkdown relies on Reticulate to run, which uses low-level Python features to organize the communication link.

Interfacing Interpreters

Litrepl communicates with interpreters using POSIX uni-directional pipes making the following general assumptions:

- The streams implement synchronous single-user mode.
- The presence of an echo command or equivalent. The interpreter must be able to echo an argument string provided by the user in response to such command.
- Command line prompts are disabled. Litrepl relies on the echo response rather than on prompt detection.

The simplicity is a key advantage, but it also has drawbacks. There's no parallel evaluation at the communication level. We expect users to employ interpreter-specific parallelism, such as Python's subprocess module utilities. The text-only data type limitation is more fundamental; LaTeX and Markdown formats process non-text data via references. Litrepl is following these conventions directly, expecting users to organize the transfer e.g. by storing images on a file system.

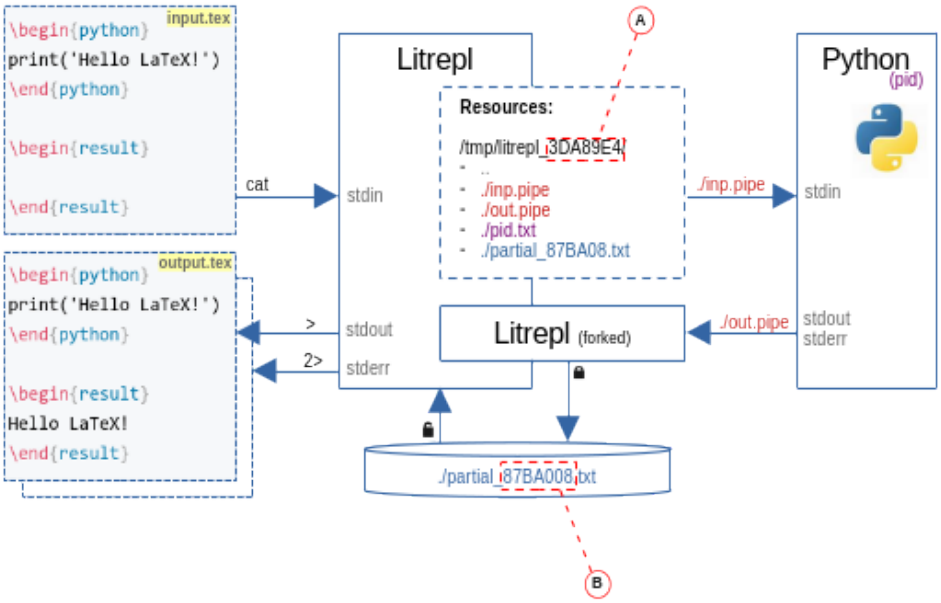

Figure 1: Litrepl resource allocation diagram. Hash **A** represents the interpreter session and is computed from the Litrepl working directory and the interpreter class. Hash **B** represents the asynchronous execution of a code snippet and is computed from the contents of the corresponding input.

Resources are stored in an auxiliary directory specified via command-line or environment variables. The editor-interpreter session is represented by pipe files, one for writing input and another for reading outputs, and a file storing the interpreter process identifier.

If the language interpreter takes longer than expected, Litrepl forks a readout process that redirects the output into a sink file. To keep the user interface responsive, Litrepl releases control after inserting a tag into the output, allowing the output to be updated during subsequent REPL execution rounds.

We avoid using document formatting or configuration files for configuration, preferring command-line options instead. Formatting is used exclusively for code and result sections, which we hope improves forward compatibility with text formats.

Conclusion

The tool is implemented in Python in under 2K lines of code according to the LOC metric, and has only two Python dependencies so far, at the cost of the dependency on the POSIX operating system interfaces. Needless to say, we used Litrepl to evaluate and verify the examples presented in this document.

References

Dolstra, E., Löh, A., & Pierron, N. (2010). NixOS: A purely functional linux distribution. *Journal of Functional Programming*, *20*(5–6), 577–615. https://doi.org/10.1017/S0956796810000195

Granger, B. E., & Pérez, F. (2021). Jupyter: Thinking and storytelling with code and data. *Computing in Science & Engineering*, *23*(2), 7–14. https://doi.org/10.1109/MCSE.2021.3059263

IEEE and The Open Group. (2024). *POSIX standard: Base specifications, issue 8* (Standard Issue 8; p. 4107). The Open Group. https://doi.org/10.1109/IEEESTD.2024.10555529

Iverson, K. E. (1962). *A programming language*. John Wiley & Sons, Inc. ISBN: 0471430145

Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B. E., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J. B., Grout, J., Corlay, S., Ivanov, P., Avila, D., Abdalla, S., Willing, C., & Team, J. D. (2016). Jupyter notebooks - a publishing format for reproducible computational workflows. *International Conference on Electronic Publishing*. https://doi.org/10.3233/978-1-61499-649-1-87

Knuth, D. E. (1984). Literate programming. *The Computer Journal*, *27*(2), 97–111. https://doi.org/10.1093/comjnl/27.2.97

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, *3*, 184–195. https://doi.org/10.1145/367177.367199

Perez, F., & Granger, B. E. (2007). IPython: A system for interactive scientific computing. *Computing in Science and Engg.*, *9*(3), 21–29. https://doi.org/10.1109/MCSE.2007.53

Pérez, F., & Granger, B. E. (2015). *Project jupyter: Computational narratives as the engine of collaborative data science*. Jupyter blog. https://blog.jupyter.org/project-jupyter-computational-narratives-as-the-engine-of-collaborative-data-science-2b5fb94c3c58

Ramsey, N. (1994). Literate programming simplified. *IEEE Softw.*, *11*(5), 97–105. https://doi.org/10.1109/52.311070

Vallet, N., Michonneau, D., & Tournier, S. (2022). Toward practical transparent verifiable and long-term reproducible research using guix. *Scientific Data*, *9*(1), 597. https://doi.org/10.1038/s41597-022-01720-9