Living Papers: A Language Toolkit for Augmented Scholarly Communication

Jeffrey Heer
jheer@uw.edu
University of Washington
Seattle, United States

Matthew Conlen
mpconlen@gmail.com
University of Washington
Seattle, United States

Vishal Devireddy
vishald@cs.washington.edu
University of Washington
Seattle, United States

Tu Nguyen
tu21897@uw.edu
University of Washington
Seattle, United States

Joshua Horowitz
joho@uw.edu
University of Washington
Seattle, United States

ABSTRACT

Computing technology has deeply shaped how academic articles are written and produced, yet article formats and affordances have changed little over centuries. The status quo consists of digital files optimized for printed paper—ill-suited to interactive reading aids, accessibility, dynamic figures, or easy information extraction and reuse. Guided by formative discussions with scholarly communication researchers and publishing tool developers, we present Living Papers, a language toolkit for producing augmented academic articles that span print, interactive, and computational media. Living Papers articles may include formatted text, references, executable code, and interactive components. Articles are parsed into a standardized document format from which a variety of outputs are generated, including static PDFs, dynamic web pages, and extraction APIs for paper content and metadata. We describe Living Papers' architecture, document model, and reactive runtime, and detail key aspects such as citation processing and conversion of interactive components to static content. We demonstrate the use and extension of Living Papers through examples spanning traditional research papers, explorable explanations, information extraction, and reading aids such as enhanced citations, cross-references, and equations. Living Papers is available as an extensible, open source platform intended to support both article authors and researchers of augmented reading and writing experiences.

CCS CONCEPTS

• **Human-centered computing** → **Interactive systems & tools**.

KEYWORDS

Academic Publishing, Augmented Reading, Interactive Articles

ACM Reference Format:
Jeffrey Heer, Matthew Conlen, Vishal Devireddy, Tu Nguyen, and Joshua Horowitz. 2023. Living Papers: A Language Toolkit for Augmented Scholarly Communication. In *The 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23), October 29–November 01, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3586183.3606791

UIST '23, October 29–November 01, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0132-0/23/10.
https://doi.org/10.1145/3586183.3606791

1 INTRODUCTION

For centuries, the format of academic articles has adhered to conventions compatible with the movable type printing press: a "user interface" consisting of textual, mathematical, and graphical content organized into sections, figures, and linkages such as footnotes, bibliographic citations, and cross-references. In recent decades, computational technologies such as word processing, digital typesetting, and the Internet have had a tremendous impact on how research articles are written, produced, archived, and accessed, yet had relatively little impact on the structure of articles themselves. Physical paper has multiple virtues as a format—it is tangible and archival, with no batteries required. But academic articles are now often read on a screen [51], using a proprietary format optimized for print (PDF) that suffers from accessibility concerns [6] and complicates computational extraction and analysis [31].

In contrast, visions of alternative publishing formats have been a staple of Human-Computer Interaction since the inception of the field, from initial hypertext designs [42], to the World Wide Web [5], to interactive documents now published regularly by data-driven journalists [9]. Despite exciting innovations in augmented reading aids [14, 22, 23, 50] and experiments with online-first research venues [59, 66], academic publishing remains resistant to change.

Meanwhile, both corporations [19] and non-profits [2] have indexed large swathes of the literature, often applying vision and NLP methods for not-always-accurate extraction of paper content and metadata [57]. These efforts enable large-scale search and scientometric analysis [17]; however, robust and flexible tools for content extraction and reuse remain out of reach for many researchers.

We seek to bridge present and future publishing through novel authoring tools. We contribute Living Papers, a framework for writing enhanced articles that encompass multiple output types: *interactive web pages* to enable augmented reading experiences, accessibility, and self-publishing; *static PDFs* to align incentives and participate in existing publishing workflows; and *application programming interfaces (APIs)* to enable easy extraction and reuse of both article content and executable code. In sum, Living Papers is a "language toolkit" consisting of a standardized document model and a set of extensible parsers, transforms, and output generators.

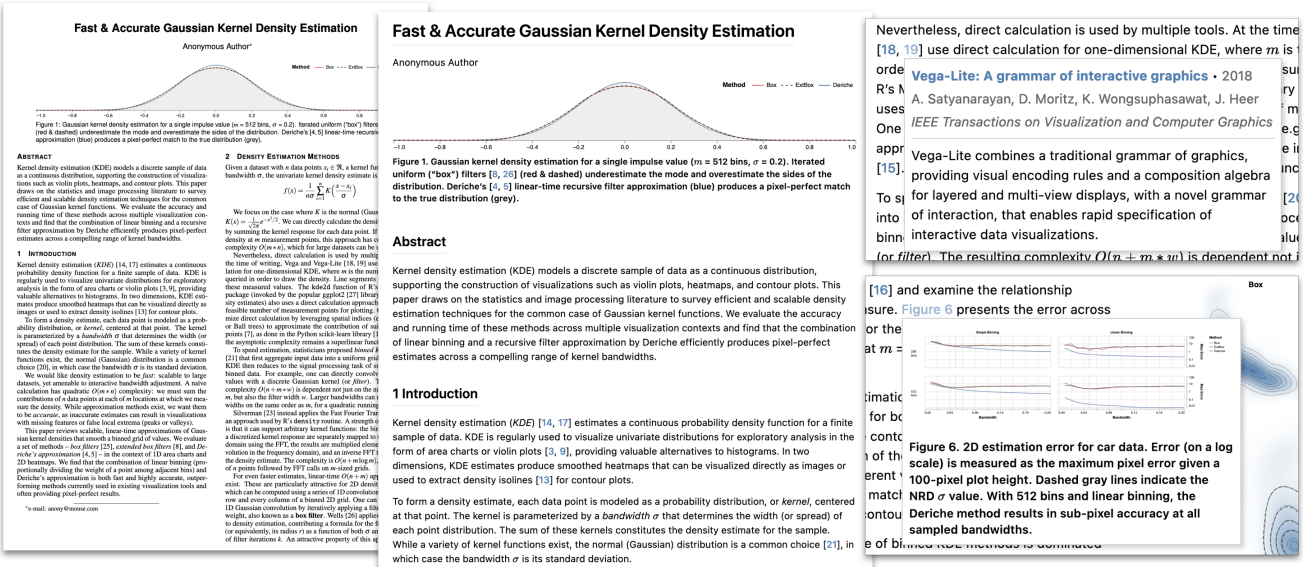


Figure 1: Living Papers version of an IEEE VIS 2021 paper, with PDF (*left*) and HTML (*center*) output generated from the same source document. Web output includes reading aids for citations (*top right*) and cross-references (*bottom right*).

To support dynamic reading aids and explorable explanations [62], Living Papers produces web-based articles with a reactive runtime and extensible component system. We use Markdown [20] as a default input format, with syntax extensions for custom components. Articles may include executable code in languages such as JavaScript, R, and Python to generate static or interactive content. To support "backwards compatibility" with current publishing practices, the Living Papers compiler automatically converts interactive and web-based material to static content, and generates LaTeX [36] projects or compiled PDFs using extensible journal and conference templates. To assist not only people but also computers to more easily interpret papers, Living Papers can compile article content into accessible data structures, APIs, and software modules.

We present our design objectives for Living Papers, honed in conversations with publishing tool developers and researchers of both augmented reading aids and information extraction from academic articles. We seek to balance tensions among dynamic content, accessible authoring, participation in existing publishing workflows, and research into novel techniques. We evaluate the system by demonstration, sharing articles by ourselves and others that span formal research papers (including this one!), explorable explanations, and enhanced content extraction and reuse. These examples highlight augmentations such as enhanced previews for citations and cross-referenced material, equations with interactive term definitions, and articles with dynamic content such as explorable multiverse analyses [14]. Living Papers is available as open source software, and intended to support both article authors and researchers exploring augmented forms of scholarly communication.

2 RELATED WORK

Living Papers connects prior research on augmented reading, article authoring tools, and information extraction from academic papers.

2.1 Augmented Reading

Augmented reading interfaces have long been a topic of HCI research. In addition to the development of hypertext [42] and HTML [5], earlier works include Hill et al. [25]'s Edit Wear and Read Wear—a document viewer showing traces of social reading and writing activity—and Xerox PARC projects including Fluid Documents [67] and eXperiments in the Future of Reading (XFR) [21].

More recent research focuses specifically on scholarly communication. A common aim is to provide contextual information about references, technical terms, and mathematical symbols where they are used, without having to break one's flow by jumping to another part of the document. ScholarPhi [22] annotates papers with definitions of terms and symbols. Other projects study math augmentations to improve the readability of formulas [23], perform rich linking of text and tables [4, 30], provide context by surfacing citation text from later papers [50], support skimming via automatic highlights [16], and produce plain language summaries for broader audiences [3]. Some of these techniques are now available in the online Semantic Reader application [1]. Dragicevic et al. [14] prototype "multiverse" analyses of varied data analysis choices by interacting with the paper itself. Living Papers provides a platform for the development and deployment of such techniques.

Elsewhere, interactive articles with dynamic figures, annotations, and embedded simulations have gained prominence, particularly in data-driven journalism [9]. Distill.pub [59], a journal for explaining machine learning, and the IEEE VISxAI workshop [66], provide academic venues for online-first, interactive web content. However, Distill is now on indefinite hiatus [59], in part due to the editing and mentoring costs of high-quality interactive articles. Living Papers seeks to make it easier to write and self-publish such articles, but also align incentives by simultaneously producing static outputs for submission to traditional research venues.

2.2 Authoring Tools

Academic articles are typically written using word processors or digital typesetting tools, notably TeX [34] and LaTeX [36]. TeX's mathematical notation has become a *de facto* standard for writing formulae, also applicable on the Web via packages such as KaTeX [15]. Collaborative authoring is supported via web applications, including Overleaf [47]. Typst [61] is a more recent alternative to TeX with its own markup language and integrated scripting language, runnable online via WebAssembly. For formatted text, Markdown [20] is a popular alternative to both HTML and LaTeX.

Pandoc [39] is a document converter with many-to-many (though sometimes lossy) transformations among formats. Pandoc parses input documents into an internal abstract syntax tree (AST) representation, from which it then produces converted outputs. Living Papers follows a similar approach, and even uses Pandoc to parse input Markdown files, but differs by providing built-in reading augmentations, an integrated reactive runtime, an extensible component system, and API outputs for information extraction.

Authoring tools may include executable code to enable interactivity or to support computational generation of content such as figures, tables, and statistical models. Knuth's Literate Programming [35] popularized the interleaving of code and narrative within a single document and has had a strong influence on computational notebooks and related formats [53]. Computational notebooks including Jupyter [33] and Observable [44] structure a document into "cells" that may contain text (with Markdown syntax) or runnable code. CurveNote [13] builds on Jupyter notebooks to produce online articles, while JupyterBook [28] uses a Markdown-based format (MyST [41]) with references to external notebook content.

RMarkdown [52] and its successor Quarto [49] interleave executable code blocks into Markdown syntax. Code is extracted, evaluated, and results are stitched back into the document. Code is typically evaluated at compile time (e.g., running an R script), though Quarto also supports "live" JavaScript in the browser. Living Papers similarly supports interleaved text and code, and can execute code either at compile time or within an integrated reactive runtime and component system. Both Living Papers and Quarto independently chose to incorporate the JavaScript dialect of Observable notebooks [44] for interactive content. Quarto does not provide a component library and relies primarily on Pandoc for its implementation (Living Papers uses Pandoc only for parsing).

Manubot [26] provides a toolchain for scholarly publishing that takes Markdown files as input, supports automatic resolution of Digital Object Identifiers (DOIs), and generates static Web and PDF output using Pandoc and GitHub actions. Living Papers similarly supports retrieval of bibliographic metadata given DOIs or other paper identifiers. Nota [12] is a tool for writing web documents with augmented reading aids and interactive figures implemented within the React [40] framework. Nota uses a variant of Markdown syntax extended with constructs for defining terms and inline scripting.

Idyll [9] is a language for interactive articles, including explorable explanations [62]. Idyll uses Markdown syntax, extended to include arbitrary components, and produces web applications implemented using React. Living Papers similarly supports an extensible component model, though using W3C-standard custom HTML elements rather than React. Living Papers uses a modified version of Idyll's abstract syntax tree (AST) format, and provides an extended tool chain to support academic articles, including citation processing, multiple output formats, and conversion of interactive content to static output. In addition, Living Papers uses Observable's reactive runtime for linked interactive content. In contrast to Idyll, Living Papers authors can write code directly in their articles and import content from existing Observable notebooks, enabling custom interactives with much less software engineering. Fidyll [10] provides a higher-level syntax for Idyll focused squarely on narrative visualization, and also targets slideshow, video, and PDF output.

Living Papers and the projects above share significant overlaps. Markdown syntax is prevalent across projects and multiple tools use identifiers such as DOIs to automatically resolve bibliographic data. Living Papers differs from the other projects by combining both support for academic papers (including "built-in" reading augmentations) and an integrated reactive runtime and component system. To the best of our knowledge, Living Papers is also unique in generating APIs for context extraction and reuse.

Meanwhile, other tools for interactive documents use graphical interfaces rather than textual markup and code. Idyll Studio [11] provides a WYSIWYG editor for Idyll articles (though not custom components). Webstrates [32] support collaborative editing to a shared, network-accessible document model. VisFlow [58] uses text-chart links to support dynamic layouts for narrative visualization. Scalar [60] is a web-first tool that provides authoring interfaces and content reuse, primarily serving the digital humanities community. Here we focus on the language toolkit provided by Living Papers, upon which future graphical and collaborative editors might build.

2.3 Information Extraction & Reuse

Other projects focus on analyzing and extracting content from papers, sometimes available only in PDF form. Augmented reading techniques (§2.1) and literature review tools [27] depend upon accurate identification or synthesis of term definitions [29], citation sentences [50], summary text [3], and more. Extraction tools include GROBID [37] and the infrastructure behind the open Semantic Scholar graph [2]. Extraction tools can "unlock" content to convert PDF documents to more screen reader accessible HTML [63] or interpret bitmap images of charts [56]. However, automatic extraction from PDFs is a difficult, error-prone task [57].

Living Papers instead supports extraction and reuse directly from published results. In addition to articles intended for people to read, Living Papers produces outputs for computational use. Living Papers generates a structured AST format in JavaScript Object Notation (JSON) and an application programming interface (API) that provides convenient access to paper metadata (title, authors, etc.) and content (section text, figures, captions, references, *in situ* citations). Moreover, the interactive content of a Living Papers article compiles to a separate, importable JavaScript module, enabling reuse of computational content in other articles or web pages.

3 DESIGN GOALS & PROCESS

Living Papers seeks to balance sometimes competing goals, such as supporting both interactive web articles and standard print workflows. Through our iterative development we have discussed our

goals and progress with multiple stakeholder groups. Over the period of a year we spoke with augmented reading and accessibility researchers from the CHI, UIST, VIS, and ASSETS communities; information extraction and knowledge base researchers (many associated with the Semantic Scholar team [2]); and publishing tool developers, including contributors to Quarto [49], Distill.pub [59], Nota [12], Jupyter [33], Observable [44], and the New York Times. We use Living Papers to write our own research articles and observed its use by graduate students in a Fall 2022 course on the Future of Scholarly Communication. Through this process we arrived at the following design considerations.

***Augmented** reading experiences.* We seek to aid contextual understanding of references, formulas, and other content without "bouncing" between paper sections (§2.1). By default, output web articles include contextual previews for both citations and cross-references. We also demonstrate extensions for augmented equations, term definitions, and alternative reading interfaces.

***Computational** media.* We seek to recast scholarly publications from static articles to computational artifacts more amenable to both people and machines. Authors should be able to incorporate reproducible results such as models and data visualizations, which can then be reused as-is in other media. Living Papers supports interaction via a reactive runtime that integrates executable code blocks and an extensible component library that includes augmented citations, cross-references, equations, and interactive text. While tools like Semantic Scholar rely on accurate information extraction to provide reading aids, Living Papers side-steps this issue via language design and enables downstream extraction by producing APIs to query paper content and reuse reactive web content.

***Approachable** writing and content generation.* We sought a familiar yet sufficiently expressive markup language, leading us to use Markdown as our default input format. We follow Pandoc's [39] Markdown syntax, which is familiar to users of RMarkdown [52] and includes constructs for tables, math blocks, and citations. In addition, syntax highlighting (e.g., in VSCode) is already supported. We want to simplify inclusion of computer-generated models and figures, for example using executable code blocks. In addition, Living Papers' citation processor performs automatic lookup of bibliographic metadata from DOIs and other identifiers (e.g., PubMed and Semantic Scholar ids) to help ease reference management, while still supporting standard citation formats such as BibTeX.

***Compatible** with existing publishing norms.* We hypothesize that the print-focused needs of current publication workflows is a major impediment to the adoption of augmented formats. Distill.pub editors, for instance, emphasized the issue of aligning to existing incentives. Living Papers supports both interactive web-based content and traditional print-based media. Authors should be able to write their content once and generate both augmented web pages and submission-worthy PDFs. To accommodate these needs, Living Papers automatically converts interactive material to static text or images for print output, while also supporting output-specific blocks when authors wish to specialize content for different media.

***Accessible** and **archivable** content and interactions.* Accessibility researchers expressed a strong preference for HTML over PDF, as HTML output with semantic tags better supports screen readers than standard PDF output. Living Papers' default web page template uses a responsive layout that adjusts for desktop or mobile viewing. By publishing to the Web, Living Papers is also applicable to more informal genres such as blog posts. Meanwhile, static output enables printing to paper for both reading and archival purposes.

***Collaborative** authoring and review.* Much academic work is collaborative in nature. Living Papers uses plain text formats that operate well with revision control and diff'ing tools such as Git, supporting awareness and integration of collaborators' work. We also support anchored annotations [48] to a Living Papers AST, providing infrastructure for collaborative commentary or annotations of terms or named entities of interest. Though beyond the scope of this paper, we plan to build on these features to support collaborative authoring and reviewing interfaces in future work.

***Extensible** platform for research.* While Living Papers is intended to be useful as-is, our collective understanding of the design space of augmented reading aids and effective use of interactivity is still developing. For example, augmented reading researchers desired an extensible research platform for better dissemination and testing of techniques. To support continued research and evaluation, Living Papers provides an open architecture with flexible parsing, transforms, and output formats. Living Papers can support exploration of new input markup languages, AST transforms, reading aids, custom interactive components, output types, and more.

In terms of *non-goals*, web application frameworks, static site generators, and narrative visualization tools (e.g., for rich "scrollytelling" [9]) overlap with Living Papers. We do not attempt to cover this space, but instead focus squarely on academic articles. We prioritize familiarity and practicality (including use of Markdown, the Observable runtime, BibTeX, and leaky abstractions over LaTeX) over formal elegance—"evolution, not revolution." This focus was honed by discussions with notebook and publishing tool creators, and by the accretive history of the Web versus other document systems, all the way back to Nelson and the "curse of Xanadu" [65]. Living Papers intentionally embraces a "polyglot" syntax.

4 EXAMPLE ARTICLES

Before detailing Living Papers' technical building blocks, we present a set of example articles that span—and blur the distinction between—traditional research papers and explorable explanations. All article sources and outputs are included as supplemental material. Beyond the examples presented here, both ourselves and others have used Living Papers to write research papers for course projects and for venues such as IEEE VIS, ACM CHI, and UIST—including this paper.

4.1 Fast & Accurate Kernel Density Estimation

Figure 1 shows an IEEE VIS 2021 paper on kernel density estimation [24]. The original LaTeX manuscript includes text, equations, and figures created with the Vega visualization grammar [55], which were manually converted from SVG to PDF format. The Living Papers version is written in a more *approachable* Markdown syntax, produces *compatible*, *archivable* PDF output, and converts source SVG images to PDF output for inclusion in LaTeX. Living Papers uses extended Markdown syntax to define a `figure` component:

```
::: figure {#id .class property=value}
![alt text](image.svg)
| Caption text
:::
```

In the one-dimensional registration problem, we wish to find the horizontal disparity h between two curves $F(x)$ and $G(x) = F(x + h)$. This is illustrated in Figure 2.

Our solution to this problem depends on $F'(x)$, a linear approximation to the behavior of $F(x)$ in the neighborhood of x, as do all subsequent solutions in this paper. In particular, for small h, (h : disparity vector)

$$F'(x) \approx \frac{F(x+h) - F(x)}{h} = \frac{G(x) - F(x)}{h} \quad (1)$$

so that

$$h = \frac{G(x) - F(x)}{F'(x)} \quad (2)$$

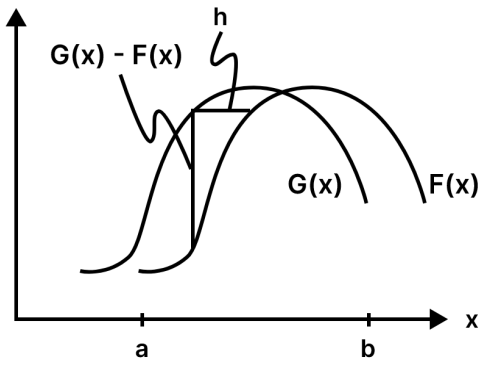


Figure 2. Two curves to be matched.

Figure 2: Colored terms with nested, interactive definitions (*left*) aid equation understanding in Lucas & Kanade's classic optical flow estimation paper [38]. A "sticky" margin figure (*right*) stays fixed while scrolling the current section, maintaining context.

References in BibTeX format can be included in-document, in a separate file, or retrieved using an inline identifier (e.g., using DOIs, `@doi:10.1109/VIS49827.2021.9623323`). Living Papers' more *accessible* `html` output includes *augmented* reading aids that provide previews for citations and cross-references (Figure 1, *right*).

Living Papers also helps manage figure layout across media types. The Web-based version includes margin figures (indicated by a `.margin` class on figure components), which, for the `latex` output, are instead placed within a two-column layout. LaTeX is notoriously finicky with figure layout: to ensure desired placement, authors may need to move figure source definitions far from the content they reference. Living Papers provides a `\place{id}` directive that indicates where to place a referenced figure or table within the output `latex` source. This allows `html` output to place the figure where it is defined in the original Living Papers source, while repositioning the figure as desired within generated LaTeX. To adjust layout and prevent undesirable gaps or overflows, the `latex` output module also accepts a `vspace` option that systematically inserts vertical offset instructions for figures, captions, or other named node types. We have found these extensions valuable for expediting article production, including for this current paper.

4.2 An Iterative Image Registration Technique

Figure 2 recreates Lucas & Kanade's paper on optical flow estimation [38]. The Living Papers version provides math augmentations [23] to help readers make sense of the paper's formulas, demonstrating Living Papers' *extensibility*. Equations include colored terms; in-situ definitions are revealed on mouse click. A custom AST transform first parses `definitions` components in the document source:

```
~~~ definitions
@F :blue: First stereo image
@x :red: Position vector in an image
~~~
```

Mathematical notation can reference defined terms within `$`-delimited inline math (e.g., `$@F(@x)$`) and equation components (`~~~ equation`). Augmented formulas are then rendered using Web components that inject color annotations into KaTeX [15] source

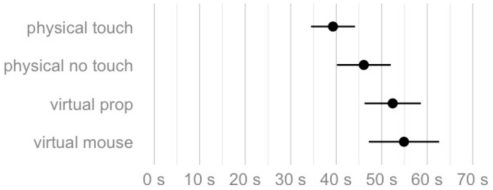


Figure 3. Average task completion time (geometric mean) per condition. Error bars are 95% t-based CIs.

We focus our analysis on task completion times, reported in Figure 3 and Figure 4. Dots indicate sample means, while error bars are 95% confidence intervals computed on log-transformed data [8] using the t-distribution method. Draggable text

Figure 3: Interactive text enables explorable multiverse analysis of data analysis choices [14]. Dragging or clicking the text cycles through statistical procedures and resulting plots.

and bind event listeners to the rendered terms. *Compatible* `latex` output shows normal, unaugmented math.

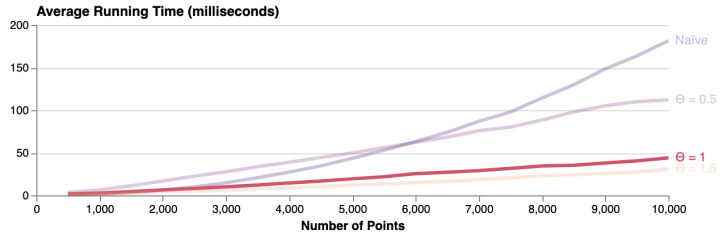
The article includes *augmented* "sticky" figures that persist to maintain context while reading. For `html` output, adding the attribute `sticky-until="#sec4_2"` to a figure component causes that figure to stay on-screen until the section with id `sec4_2` is reached. The `latex` output ignores these web-specific annotations.

4.3 Explorable Multiverse Analysis

Living Papers articles can include interactive *computational* content. Here we recreate Dragicevic et al. [14]'s explorable multiverse analyses for assessing the sensitivity of different statistical analysis decisions. Readers can explore a range of data analysis choices by interacting with built-in components for draggable and toggleable text (Figure 3). For example, the following inline component syntax adds draggable text for a choice of confidence interval levels:

```
[:option-text:]{
  options=[50,68,80,90,95,99,99.9]
  suffix="%"
  bind=confidenceLevel
}
```

The reactive runtime binds the component value to the runtime's `confidenceLevel` variable, causing all dependent components to



The running time results confirm that the Barnes-Hut approximation can significantly speed-up computation. As expected, the naïve approach exhibits a quadratic relationship, whereas increasing the Θ parameter leads to faster calculations. A low setting of Θ = 0.5 does not fare better than the naïve approach until processing about 6,000 points. Until that point, the overhead of quadtree construction and center of mass calculation outstrips any gains in force estimation. In contrast, for Θ = 1 and Θ = 1.5 we see significant improvements in running time.

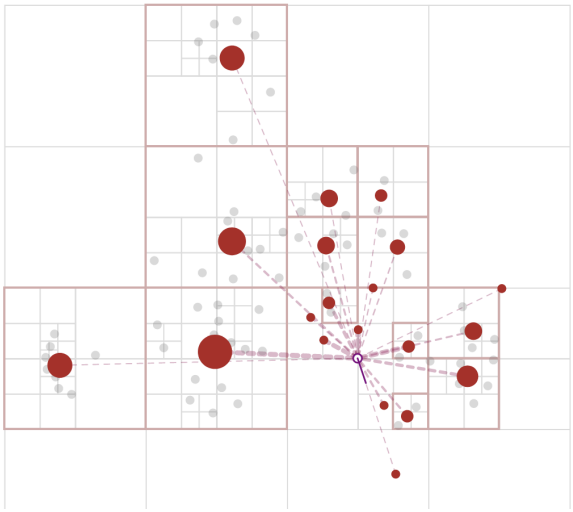


Figure 4: An explorable explanation of the Barnes-Hut approximation for N-body forces. Linked text and chart elements update the Θ variable in the Living Papers reactive runtime, driving highlighting and simulation parameters.

update. The inline JavaScript code `` `js confidenceLevel` ``, for example, dynamically displays the current value as output text. Other bound components similarly update via two-way bindings.

The article uses pre-computed images for all confidence level and analysis procedure combinations (e.g., bootstrapped vs. parametric intervals). The image syntax `![alt text](`figA`)` binds code output—here, the variable `figA`—to the image source URL. Alternatively, result images could be generated during article compilation by using the `knitr` transform to evaluate R code blocks (§7.2).

This article also compiles to *compatible* LaTeX. An output-specific AST transform (§7.3) evaluates the reactive runtime in a headless web browser, generates static content, and rewrites the article AST by replacing all dynamic elements. The resulting PDF is a coherent article describing just the default set of analysis choices.

4.4 The Barnes-Hut Approximation

Figure 4 shows an explorable explanation of the Barnes-Hut approximation for N-body forces, originally written using Idyll [9], which demonstrates *augmented* reading, *computational* media, and *extensibility*. A custom simulation component—showcasing force-directed layout and interactive visualizations of quadtree structures and force calculations—is included as a margin figure that persists throughout the article. The simulation and linked plots update in response to interactions with the article, including input from *action links* (e.g., `[Θ = 0.5](`theta=0.5`)`) and bound sliders (created with the Observable standard library's `Inputs.range` method):

```
~~~ js {bind=theta}
Inputs.range([0, 2], { step: 0.1, label: 'Theta' })
~~~
```

Similar to the Idyll version, the force simulation and Vega-based plots are implemented as custom components. Converting the original React components to W3C custom HTML elements was straightforward, involving changes only to "wrapper" code. However, the Idyll version also involves custom components for sliders and action links (implemented in separate JavaScript files) and integration logic

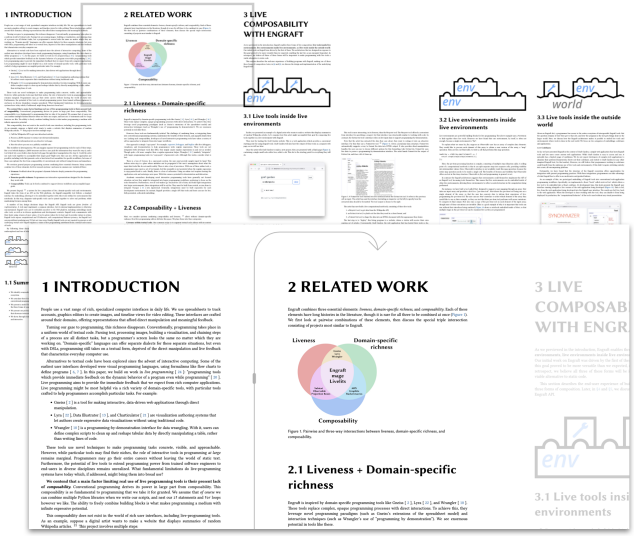


Figure 5: A zoomable paper reading interface created by post-processing Living Papers `html` output.

that requires knowledge of Idyll internals. Living Papers supports these features, including reuse of common Observable components, directly within the primary article source.

The article compiles to LaTeX, but, if done naïvely, can produce an illegible article due to the lack of interaction. Using *output-specific* blocks, Living Papers authors can designate content that should be included only for target output types. Here, we can annotate the persistent interactive simulation as `html:only`. We can then use `latex:only` blocks to include *compatible* simulation snapshots with desired keyframe parameters in the static output.

4.5 Zoomable Paper Reader

Figure 5 shows an *augmented* reading interface created by a student, demonstrating *extensible* output. JavaScript code post-processes

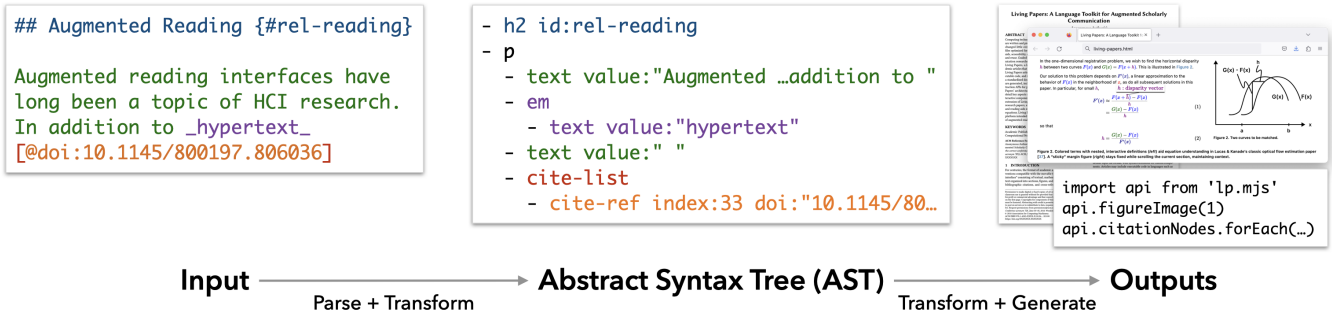


Figure 6: Overview of Living Papers compilation. Input files (e.g., using extended Markdown syntax) are parsed to produce an Abstract Syntax Tree (AST) that is further updated by AST transforms for citations, executable code, and more. Modules for `web`, `latex`, and `api` output apply output-specific AST transforms and then generate output files.

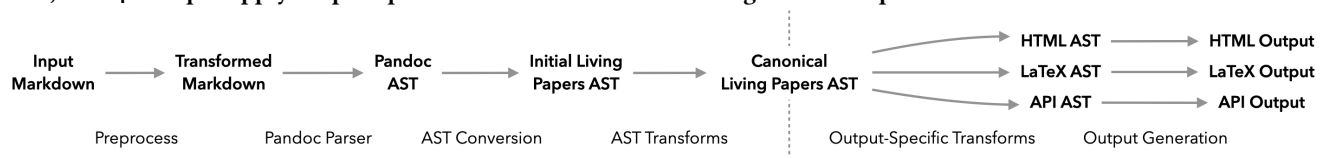


Figure 7: Pipeline to parse a Markdown file (§6), perform AST transforms (§7), and generate HTML, LaTeX, and API outputs (§8).

Living Papers `html` output into a zoomable, column-oriented layout. The initial view provides an overview of the full paper. A reader can freely pan and zoom the canvas. For a linear reading experience, the reader can click a region of the article (Fig. 5 *top-left*) and zoom in to a tracked, scrollable navigation mode (Fig. 5 *bottom-center*). When the reader reaches the end of a column, they can continue scrolling to trigger automatic panning along a "track" to the top of the next column. The content in each column is standard Living Papers output, with the same reading aids and reactive content options previously discussed. We are now refactoring this layout and navigation code into a reusable `html` output template.

5 LIVING PAPERS ARCHITECTURE

Living Papers uses the compilation pipeline illustrated in Figures 6 & 7. Input files are *parsed* into an abstract syntax tree (AST). AST *transforms* then analyze and update the AST, including citation processing and analysis of executable code, to form a *canonical AST* that represents the article in standalone fashion. One or more *output modules* take the canonical AST as input, apply *output-specific transforms*, and then generate output files. Living Papers is implemented in JavaScript and provides a command line utility (`lpub`) for article compilation. Here we describe the core abstractions of the document model, reactive runtime, and extensibility mechanisms. Later sections further detail the compilation steps of parsing (§6), AST transformation (§7), and output generation (§8).

5.1 Document Model

Living Papers uses an AST format adapted from Idyll [9], analagous to the Document Object Model (DOM) used by web browsers. An AST is representable in JSON format, facilitating both processing and serialization. At the top-level, an AST consists of three properties: *metadata*, *data*, and the *article* tree. Article *metadata* includes information such as title, authors, and keywords, as well as processing options for outputs, transforms, and components. Article *data*, such as resolved citations or term definitions, can be accessed by downstream components and generated APIs. Extensions can add their own data properties as needed. Living Papers includes a stand-alone library for AST creation, modification, and traversal.

The document tree is rooted at the AST *article* property. A document node may contain a *type*, *name*, *properties*, and either a *value* or *children*. Currently, we only use the types *textnode* and *component*. A text node contains verbatim text as a string *value*. Component nodes include a *name* (e.g., `p` for paragraph or `em` for emphasis) and may include *children* as an array of child AST nodes. Figure 6 (*middle*) illustrates this AST structure. All Living Papers start with a root node with the component name `article`.

Node properties consist of key-value pairs where the *values* take one of three types. *Value*-typed properties simply contain a static value. *Expression*-typed properties contain a reactive JavaScript expression, which can be used to dynamically set component properties. *Event*-typed properties contain JavaScript event handler code. Unlike expression properties, event handlers can update variable assignments in the reactive runtime (§5.2). The special `class` property may contain one or more named classes used to style content. Living Papers includes a standard set of classes for layout and sizing shared by `html` and `latex` output.

Our design prioritizes Web output, while maintaining flexibility for other outputs. Basic AST component nodes adhere to matching HTML elements. Living Papers' `p`, `link`, and `image` nodes map to `p`, `a`, and `img` HTML tags, with properties that align to corresponding HTML attributes. Similarly, formatting (`em`, `strong`, `blockquote`, …) and list (`ul`, `ol`, `li`) nodes mirror their HTML equivalents.

Other nodes are specific to Living Papers. The `cite-list` and `cite-ref` nodes represent citations. A `cross-ref` node represents a reference to a section, `figure`, `table`, or `equation` elsewhere in the article. The `code` and `codeblock` components represent source code, often with syntax highlighting. Meanwhile, `math` and `equation` nodes represent expressions in TeX math notation.

Component nodes for executable code, math formulas, or other specialized syntax may include content in a `code` property or child text node. For example, a `math` node with the formula for the golden ratio ϕ could contain a single child text node with the string `\phi=\frac{1+\sqrt{5}}{2}`. Downstream transforms or output modules then process this content as needed. With `html` output, each of these is ultimately displayed using a custom HTML element.

5.2 Reactive Runtime

To support interaction, Living Papers includes a *reactive runtime* in which changes to variables or code outputs automatically propagate to dependent elements. Though the runtime is browser-based, it is tied to features of Living Papers' core document model, including *expression*- and *event*-typed properties.

Unlike Idyll, which uses its own basic reactive variable store within a React [40] context, Living Papers uses the same reactive runtime as Observable notebooks [46]. In addition to providing a robust and performant reactive engine, we chose this approach to leverage Observable's standard library (with built-in access to libraries including D3 [7] and Vega-Lite [54]) and import content from existing Observable notebooks. Authors can create content such as dynamic figures within an external notebook and then reuse that work, importing it directly in a Living Papers article.

After initial parsing, an AST transform identifies executable code blocks using Observable's JavaScript dialect and converts them to `cell-view` component nodes. Multiple named cells (reactive units) can be defined within one code block by using a single-line `---` delimiter. Only the last cell in a block is mapped to visible output.

Other components can also participate. *Expression*-valued properties map to reactive variables in the runtime, updating their corresponding components upon change. *Event*-valued properties are evaluated upon component input events (such as click), and can assign new values to named reactive variables, triggering article updates. Both custom components and JavaScript-defined elements (e.g., Observable Inputs [45]) can serve as input widgets, so long as the resulting Web element exposes a gettable and settable *value* property. Living Papers AST nodes also support a `bind` property that instructs the runtime to instantiate a two-way binding between a named reactive variable and the input component *value*.

Depending on the input format, Living Papers provides syntactic sugar for runtime integration. In Living Papers Markdown, the span `$$x^2 = ${v}^2 = ${v*v}$$` specifies a dynamic equation: JavaScript string interpolation is performed for the templated code `${v}` and `${v*v}`, the resulting TeX formula is then typeset. We use a double `$$` delimiter here to enable these internal template variables. Instead of a normal hyperlink, the link syntax `[click me](` v+=1`)` specifies an *action link* with an *event*-typed `onclick` property invoked upon click. Similarly, the image syntax `![alt text](` image_src`)` creates an *expression*-valued `src` property that dynamically sets the source URL to the `image_src` variable.

5.3 Extensibility

The Living Papers compiler marshals a number of extensible modules. Articles may specify AST *transforms* to apply. Web output may include *components* implemented as custom HTML elements. Both `html` and `latex` output are generated using *templates*.

```
::: figure {#overview .page position="t"}
![alt text](image.png)
| Figure caption text.
:::

~~~ equation {#kde}
f(x) = \frac{1}{n\sigma} \sum_{i=1}^{n}
  K{\Big (}\frac{x - x_i}{\sigma}{\Big )}
~~~

[:range-text:]{min=1 max=10 bind=var}
```

Figure 8: Living Papers Markdown syntax for block and inline component elements. Fenced blocks (`:::`) contain Markdown content to be parsed. Verbatim blocks (`~~~`) pass child content as-is. Inline elements may include parsed child content in the span after the `:component-name:`.

The compiler maintains a *context* object across parsing, transforms, and output generation to provide access to needed resources and services. The context provides access to the source file paths, directories, and external options (to complement or override options provided as article metadata) as well as caching, logging, and a resolution method for extension lookup.

Living Papers supports third-party extensions using a resolution scheme to lookup external transforms, components, templates, parsers, or output modules. If an extension is specified as a file path, it is looked up directly, typically within the same article project. Otherwise, the extension specification is treated as an npm (Node Package Manager) package name and looked up using Node.js' built-in resolution algorithm. Third-party packages can include a special `living-papers` entry in their `package.json` file, providing a manifest for any extensions (transforms, components, etc.) provided; these are then added to the compiler's internal registry.

6 INPUT PARSING

Parsing is the first phase of article compilation. The Living Papers architecture supports arbitrary parsers dispatched by file extension or a specified input type, including the "non-parser" of reading in an existing canonical AST JSON file. That said, our current implementation focuses on an extended Markdown format.

Given its familiarity and support for citations, references, tables, and more, we use Pandoc's Markdown variant. The parser module calls the Pandoc binary to parse inputs and produce a Pandoc AST in JSON format, then transforms the Pandoc AST to a Living Papers AST. Prior to invoking Pandoc, a pre-processor is used to handle our customized component syntax, including block and inline components (Figure 8), and to properly escape backtick-quoted code in component attributes, action links, and images. The pre-processor emits Pandoc-compatible Markdown.[1]

The parser performs additional interpretation tasks when mapping the Pandoc AST to a Living Papers AST. Notably, it classifies different references by type. While `@Knuth:84` cites a reference by id (e.g., in BibTeX), `@doi:10.1093/comjnl/27.2.97` instead cites

[1]We adopted Pandoc to expedite development. However, the parser does not track input token positions, which are valuable for linking input and output. Pandoc and the pre-processor may later be replaced by a JavaScript implementation. That said, parsing Pandoc ASTs provides an avenue for conversion from other formats (see §9).

the work by its DOI. Meanwhile, references such as `@fig:overview` or `@eqn:kde` are cross-references to article content. The parser distinguishes among these based on the prefix.

Though the Markdown parser is not internally extensible at present, Living Papers can also accept entirely new parsers. In any case, novel components are supported via AST transforms. For example, a transform can extract and process code blocks with custom component names (e.g., the math term definitions in §4.2). As such transforms of verbatim content are applied downstream of the initial parse, they can be reused with any future parsers as long as those parsers produce compatible ASTs.

7 AST TRANSFORMS

After an initial parse, the compiler applies AST transforms to map the parser output to a canonical AST representation of the article. Different output modules may also apply subsequent AST transforms. An AST transform is created by calling a constructor method that passes in options, resulting in a function that takes an AST and context (§5.3) as input and returns a modified AST as output.

7.1 Post-Parse Transforms

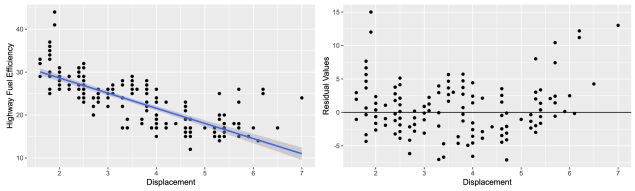
The first transform applied after parsing handles *inclusion* of additional content. Depending on the specified options, additional source files are loaded, either parsed or left verbatim, and then added to the AST. As in many other document processors, this allows article content to be spread over multiple files.

The next transform performs *runtime code extraction*, identifying executable code (e.g., `` `js value.toFixed(2)` ``) and mapping it to `cell-view` components for inclusion in the reactive runtime. Subsequent parsing and generation of runtime code occurs later during `html` output generation (§8.1).

The *citations* transform provides citation processing. Any external bibliography (BibTeX) files referenced in the article metadata are first loaded and parsed. Next, the transform finds `bibliography` component nodes in the AST and parses their verbatim content. Bibliographic data is parsed using the citation.js library [64], with CSL-JSON [8] as our canonical format. The transform then traverses the AST to visit all `cite-ref` nodes. If a citation refers to a work by an internal id, the transform attempts to look up that id among the parsed entries. If the citation instead uses an external id such as a DOI, the transform attempts to retrieve a CSL-JSON entry from the Web. For DOI lookup we use the REST API of doi.org. Given a resolved external id, the transform also queries the Semantic Scholar API [2] for additional information, including abstracts and summary ("tldr") snippets. Network request results are cached across iterative compilations for improved performance.

The *citations* transform produces multiple results. Bibliographic entries for all cited works are included in a formatted bibliography at the end of the article. All `cite-ref` nodes are updated to include an integer index into the sorted bibliography and a resolved id.[2] To support in-context reading aids and information extraction, bibliographic data (CSL-JSON, BibTeX, and Semantic Scholar data) are added to the article AST's top-level `data` property under the `citations` key. Citation lookup failures are also recognized, resulting in informative error messages.

[2]If a DOI and internal id refer to the same article, they are resolved to a single id.



(Left) Displacement vs. fuel efficiency, with fit $y = -3.5306x + 35.6977$. (Right) Residual plot.

Figure 9: The *knitr* transform evaluates R code blocks at compile time and rewrites the AST. Fitted parameters bind to the runtime for inclusion in formula text.

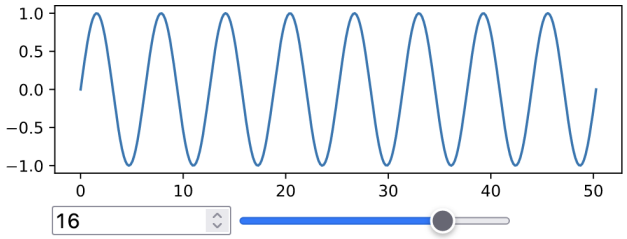


Figure 10: Using the *pyodide* transform: a dynamic Python Matplotlib chart is parameterized by a JavaScript slider.

7.2 Opt-In Transforms

Opt-in AST transforms specified in an article's metadata run after the standard transforms. Either custom third-party transforms or the following built-in transforms may be invoked.

The *knitr* transform extracts executable code written in the R programming language, synthesizes and evaluates an R script, and weaves the results back into the Living Papers AST. The transform writes extracted R code to blocks in an external Markdown file, invokes the knitr program (also used by RMarkdown and Quarto) to evaluate the code, and parses the resulting output Markdown to extract generated content. Adding a `bind` property to an R code block causes JSON-serialized output to be bound to a named variable in the reactive runtime. As in Figure 9, one can fit a statistical model in R and pass the fitted parameters and other data for processing by JavaScript. In the future, we hope to provide analagous functionality for compile-time evaluation of Python code blocks.

The *pyodide* transform extracts Python code to run directly within the reactive runtime. We use Pyodide, a WebAssembly port of Python and libraries including Pandas, Matplotlib, and Scikit-Learn. Pyodide evaluates Python code in the browser, including an object model with direct JavaScript bindings. The transform leverages Pyodide to create Python cells that run just like standard Observable JavaScript cells. Figure 10 shows a Python Matplotlib figure interactively driven by an Observable Inputs slider. However, Pyodide and associated libraries take a few seconds to initialize, delaying page loading time. Languages such as R might be similarly integrated if they are compiled to WebAssembly.

7.3 Output-Specific Transforms

Ouput modules apply transforms to prepare an AST for a specific output type. For example, `html` output applies multiple transforms to aid layout, section numbering, and other aspects (§8.1). Here

we focus on a particularly important transform shared by multiple output types: conversion of interactive and web-specific content to static assets such as text and images.

The *convert* transform first analyzes an input AST to form a *conversion plan*, consisting of nodes and properties that need to be converted to produce static output. For example, SVG images are supported in the browser but not by LaTeX; we want to convert those to PDF format prior to `latex` output generation. Other content is generated or parameterized by the reactive runtime (e.g., *expression*-typed properties). We must instantiate a runtime, evaluate the content, then extract and convert it.

To perform conversion we use Puppeteer [18], an instrumented, headless browser commonly used for automated web page testing. In order to associate AST nodes with resulting HTML DOM elements, the transform first annotates the AST nodes we wish to convert with a unique id attribute. This attribute passes through the `html` output module to the resulting web page, enabling dynamic lookup via CSS selectors. The transform then loads the compiled article in Puppeteer, extracts the rendered content, and rewrites the corresponding AST nodes. For expression-typed AST node properties and text content, generated values are simply transferred as-is. For all other content, the transform takes an image screenshot in either PNG, JPG, or PDF format.

Puppeteer provides a convenient API for extracting bitmap images of an identified DOM element. However, for PDF output, only full page printing is supported. To work around this limitation, the transform dynamically injects a new stylesheet into the Puppeteer page that hides all content but the desired target element. It also applies absolute positioning to override any local layout directives. The transform then retrieves the element's bounding box and "prints" a PDF page whose dimensions exactly match that of the target element. The result is a vector graphics PDF that can be directly included in `latex` output.

8 OUTPUT GENERATION

Given a canonical AST, Living Papers invokes output generation modules to produce both human- and machine-readable articles.

8.1 Web Output

To generate interactive web pages, the `html` output module first applies a sequence of output-specific AST transforms. These transforms prepare syntax-highlighted code listings; handle sections designated by the `abstract`, `acknowledgements`, and `appendix` component nodes; generate a header section with article title and authors; insert section and figure numbers; resolve cross-references; and process nodes with a `sticky-until` attribute, which causes content to persist on screen until an indicated section is reached.

Given a web-specific AST, the module first compiles code for the reactive runtime. Code from all `cell-view` nodes and *expression*- and *event*-typed properties is compiled to standard JavaScript functions for inclusion in the Living Papers runtime. While `cell-view` components and expression properties map to standard reactive variables, event handlers must be dealt with separately. As the Observable runtime does not allow a cell to be redefined internally, event handlers must modify the runtime by re-defining variables externally. The code generator inserts a proxy object to collect all

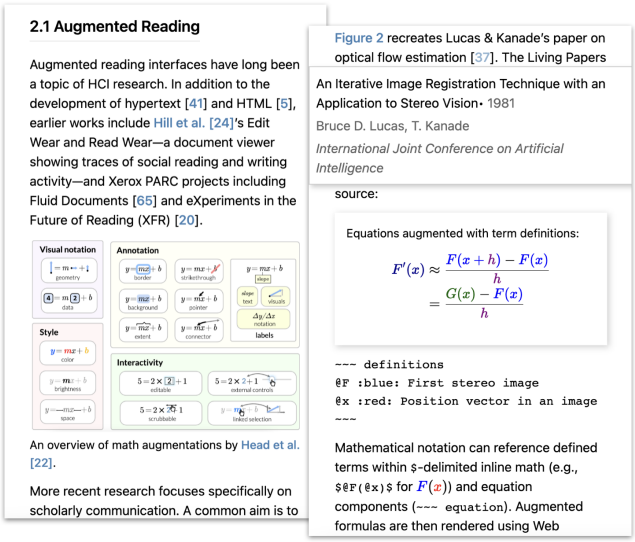


Figure 11: A web-based article viewed at a mobile form factor. The default theme provides a responsive layout.

variable assignments made by a handler; these assignments are then applied to the runtime in batch when the handler completes.

Next, the module marshals all components that map to custom HTML elements. Living Papers includes components for citations (`cite-ref`), cross-references (`cross-ref`), reactive runtime output (`cell-view`), syntax-highlighted code (`code-block`), and math blocks (`math`, `equation`). KaTeX [15] is used to process and typeset TeX notation in the browser. To aid reading, citation and cross-reference components provide tooltips with contextual information: title, authors, venue, and summary for citations, and a live content snapshot and caption for cross-references. The component library includes other interactive elements; for example `range-text` to select from a range of values by dragging and `toggle-text` to cycle through values upon click or touch. These input components can be bound to reactive variables (§5.2) to drive dynamic content. External custom components are also supported (§5.3).

The `html` output module then generates entry code to register any custom components used, instantiate the runtime, load generated runtime code, and assign the top-level AST *data* property to the root `article` element for subsequent lookup. All generated code and component definitions are run through a bundler that packages and optionally minifies the code for use. If an article does not contain interactive content or custom components, this process is skipped and no output JavaScript is generated.

Finally, the module generates output HTML and CSS by walking the AST and mapping nodes to corresponding HTML elements or text nodes. Output-specific nodes or properties (e.g., those flagged for `latex` output) are ignored. To generate CSS, both base CSS definitions shared by all articles and the CSS for a named theme are loaded. Alternative themes can be used via Living Papers' extension mechanisms. The default layout uses multiple columns with a main column for primary content and a right margin column for footnotes and marginalia (tagged with the `.margin` class). Media queries collapse all content to a single-column layout for accessible

mobile reading. An article's *metadata* may include a `styles` property, indicating a custom CSS file to also include. The collected CSS files are then bundled and optionally minified. Depending on the article *metadata* settings, the resulting HTML, CSS, and JavaScript are either written as separate files to an output directory, or as a `selfContained` HTML file that includes CSS and JavaScript inline.

Living Papers also provides a `static` output module, which generates HTML without any interactive elements. Interactive content is first converted to static assets by an output-specific transform (§7.3), after which the same generation process above is performed. This option can be used to generate static, no-motion content for both accessibility and archival purposes.

8.2 Print Output

Print output is generated by the `latex` output module. The module can produce a full LaTeX project, consisting of a directory with generated source files and assets. By default, the module generates a LaTeX project in a temporary directory and invokes the external `pdflatex` command to produce a PDF.

The *convert* transform first maps interactive or Web content to LaTeX-compatible assets, as described in §7.3. The output module then walks the AST to generate output LaTeX content. Standard text is processed to map to LaTeX special characters and escape sequences as needed, while nodes containing "raw" TeX content are written verbatim. The module segments generated content into named variables (e.g., preamble, abstract, body, acknowledgements, appendix) and passes these to a selected template. Living Papers provides an extensible set of LaTeX templates for various publishing venues, including built-in templates for ACM and IEEE journals and conferences. Living Papers includes additional directives to aid TeX-specific layout concerns, described earlier in §4.1.

8.3 Computational Output

Living Papers also supports outputs intended for computational consumption. The `ast` output module simply writes the canonical AST in JSON format. The AST can then be loaded and analyzed to extract article content, data, and metadata.

The `api` output module generates an API for more convenient access. We envision these modules being used for information extraction (e.g., for construction of academic knowledge bases) and to enable new applications (e.g., content for research lab websites, course syllabi, or curated libraries). The generated API includes methods for accessing metadata (title, authors, etc.), querying article content (abstract, section text, captions, citations), and exporting figure content (including generated images). The generated API can be easily imported as a standard ECMAScript module, including directly from a URL. The `api` output module first runs an output-specific transform to generate static PNG images for all figures, tables, and equations. Next it annotates the AST with these images in the form of base64-encoded data URLs. It then generates a JavaScript module that loads the AST and provides methods to query and access the content.

Like computational notebooks, Living Papers articles can contain executable code (e.g., models and dynamic figures) that people may wish to reuse in other articles or web pages. The `runtime` output module generates a standalone JavaScript module that contains the compiled runtime code of an article and is compatible with Observable notebooks. With this functionality, a Living Papers article can directly import (or *transclude* [43]) the computational content of other papers. Examples of runtime transclusion and the extraction API are included as supplemental material.

9 DISCUSSION & FUTURE WORK

We presented Living Papers, a framework to bridge academic publishing of printed papers, interactive web pages, and machine-readable APIs and assets. Living Papers provides an extensible infrastructure for parsing, transformation, and generation of scholarly articles, coupled with a reactive runtime and component system supporting augmented reading aids and interactive texts. To the best of our knowledge, Living Papers is unique in its combination of reading augmentations, language-level interaction support, asset conversion, and output API generation for academic articles.

While we have not conducted a formal summative evaluation of Living Papers, our development process has been informed through consultation and collaboration with multiple stakeholders, spanning academic researchers and publishing tool developers. Over the past six months we have used Living Papers to write five submitted research articles in our own lab, and have observed student use in a graduate scholarly communication course. While we don't expect all users to react as positively, one external paper collaborator told us unprompted that "Living Papers is a bliss." We have particularly appreciated the directness of Markdown syntax, the ease of generating Web output with reading aids "baked in," and the ability to directly incorporate code to generate content such as models, figures, and tables. As showcased by our examples (§4), we have been able to readily incorporate varied levels of interactivity, ranging from standalone interactive graphics to richly linked explorable explanations. Going forward we hope to more deeply evaluate Living Papers and systematically study authoring experiences.

While rich interactivity can be attention-grabbing, finding an appropriate balance—in terms of both author effort and reader benefits—remains an open research question. Beyond citations, cross-references, and term definitions, what should be in the "standard library" of reading augmentations that authors can apply with little-to-no effort? And under what conditions do richer, explorable explanations significantly improve reader comprehension? We hope Living Papers will be used by ourselves and others to further develop and study the space of reading augmentations.

With respect to accessibility, we see Living Papers as a promising work in progress. The current offering includes HTML output with semantic tags, alt-text images, responsive design to common viewing form factors, and the ability to convert computational output to static content (including static HTML, not just PDF). The latter may be helpful for people with motion sensitivity. Generated paper APIs could enable additional accessibility aids. That said, our design goals (§3) can be in tension, particularly balancing accessibility with rich interactive output. There is no guarantee that authors' interactive content will be accessible. For example, generating more accessible and screen-reader navigable visualizations is an active area of research [68]. We see deeper engagement with accessibility stakeholders, the curation of accessible "standard library" components, and subsequent studies as vital future work.

Another area for future work is graphical and collaborative environments for article writing and reviewing. WYSIWYG editors (c.f., [11, 32]) provide an alternative to markup languages, and might operate directly on Living Papers AST structures. Interfaces for annotation and commenting would aid both collaborative writing and paper reviewing. Living Papers already includes infrastructure for selecting and excerpting AST segments that future interfaces might build upon. Another next step is to support multi-chapter books or multi-page websites in addition to single articles. We are interested in using Living Papers for end-to-end management of a workshop or conference, including study of peer (or post-publication) review and the generation of online proceedings.

Meanwhile, we seek to leverage the computational output of Living Papers. We envision laboratory web sites, curated reading lists, or novel literature search tools populated with content extracted from Living Papers APIs. However, this vision rests on either the network effects of wide-spread adoption or the ability to more effectively parse existing content to a shared machine-readable representation. We believe Living Papers can contribute to conversations about what such a shared document model can and should include, particularly with respect to interactive content.

One avenue may be to convert existing documents to the Living Papers format. Though parsing PDFs is difficult, existing tools that target HTML output (such as SciA11y [63]) might also target Living Papers. Meanwhile, other structured document formats can (sometimes lossily) be transformed. As Living Papers can parse Pandoc ASTs, with additional engineering we might leverage Pandoc to convert from LaTeX, MS Word, and other formats.

One potential concern is the large "syntactic surface" of Living Papers. To make full use of the system, paper authors must learn Living Papers Markdown, Observable JavaScript (for interactive content), BibTeX (for references), bits of TeX/LaTeX (for math equations, or when custom output-specific directives are needed), and so on. Developers of new transforms and components require further knowledge, such as standard JavaScript, HTML, and CSS. These complications are a direct consequence of Living Papers' evolutionary approach and its embrace of the Web and "literate programming" design patterns. While arguably complex, both "interleaved" syntax and these constituent languages are already in widespread use. We hope to build on this familiarity and infrastructure, while making many aspects "opt-in" rather than required.

Still, adoption is difficult and hard to predict. Even if Living Papers falls short of a widely-used framework, it can be deployed for real-world publications and websites, and also help influence the trajectory and feature set of other tools. For individual users, Living Papers provides natural escape hatches: one can produce a LaTeX project or interactive web page and, if desired, jettison Living Papers and move forward with the generated outputs.

More broadly, Living Papers can serve as a non-proprietary and extensible research system for experimentation—but one that also connects with existing publishing workflows, hopefully better aligning with author incentives. It offers a path for developers of novel reading or authoring techniques to integrate into an existing system for wider deployment in the wild. Living Papers is available as open source software at github.com/uwdata/living-papers.

REFERENCES

[1] Allen Institute for Artificial Intelligence, Semantic Scholar Team. 2023. Semantic Reader. https://www.semanticscholar.org/product/semantic-reader.

[2] Waleed Ammar, Dirk Groeneveld, Chandra Bhagavatula, Iz Beltagy, Miles Crawford, Doug Downey, Jason Dunkelberger, Ahmed Elgohary, Sergey Feldman, Vu Ha, Rodney Kinney, Sebastian Kohlmeier, Kyle Lo, Tyler Murray, Hsu-Han Ooi, Matthew Peters, Joanna Power, Sam Skjonsberg, Lucy Wang, Chris Willhelm, Zheng Yuan, Madeleine Zuylen, and oren. 2018. Construction of the Literature Graph in Semantic Scholar. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 3 (Industry Papers)*. Association for Computational Linguistics. https://doi.org/10.18653/v1/n18-3011

[3] Tal August, Lucy Lu Wang, Jonathan Bragg, Marti A. Hearst, Andrew Head, and Kyle Lo. 2022. Paper Plain: Making Medical Research Papers Approachable to Healthcare Consumers with Natural Language Processing. (2022). https://doi.org/10.48550/ARXIV.2203.00130

[4] Sriram Karthik Badam, Zhicheng Liu, and Niklas Elmqvist. 2019. Elastic Documents: Coupling Text and Tables through Contextual Visualizations for Enhanced Document Reading. *IEEE Transactions on Visualization and Computer Graphics* 25, 1 (2019), 661–671. https://doi.org/10.1109/tvcg.2018.2865119

[5] Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen, and Arthur Secret. 1994. The World-Wide Web. *Commun. ACM* 37, 8 (1994), 76–82. https://doi.org/10.1145/179606.179671

[6] Jeffrey P. Bigham, Erin L. Brady, Cole Gleason, Anhong Guo, and David A. Shamma. 2016. An Uninteresting Tour Through Why Our Research Papers Aren't Accessible. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. ACM. https://doi.org/10.1145/2851581.2892588

[7] M. Bostock, V. Ogievetsky, and J. Heer. 2011. D3 Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics* 17, 12 (2011), 2301–2309. https://doi.org/10.1109/tvcg.2011.185

[8] Citation Style Language. 2023. https://citationstyles.org/.

[9] Matthew Conlen and Jeffrey Heer. 2018. Idyll. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. ACM. https://doi.org/10.1145/3242587.3242600

[10] Matthew Conlen and Jeffrey Heer. 2022. Fidyll: A Compiler for Cross-Format Data Stories & Explorable Explanations. (2022). https://doi.org/10.48550/ARXIV.2205.09858

[11] Matthew Conlen, Megan Vo, Alan Tan, and Jeffrey Heer. 2021. Idyll Studio: A Structured Editor for Authoring Interactive & Data-Driven Articles. In *The 34th Annual ACM Symposium on User Interface Software and Technology*. ACM. https://doi.org/10.1145/3472749.3474731

[12] Will Crichton. 2023. A New Medium for Communicating Research on Programming Languages. https://willcrichton.net/nota/.

[13] Curvenote. 2023. https://curvenote.com/.

[14] Pierre Dragicevic, Yvonne Jansen, Abhraneel Sarma, Matthew Kay, and Fanny Chevalier. 2019. Increasing the Transparency of Research Papers with Explorable Multiverse Analyses. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM. https://doi.org/10.1145/3290605.3300295

[15] Emily Eisenberg and Sophie Alpert. 2023. KaTeX: The fastest math typesetting library for the web. https://katex.org.

[16] Raymond Fok, Hita Kambhamettu, Luca Soldaini, Jonathan Bragg, Kyle Lo, Andrew Head, Marti A. Hearst, and Daniel S. Weld. 2022. Scim: Intelligent Skimming Support for Scientific Papers. (2022). https://doi.org/10.48550/ARXIV.2205.04561

[17] Santo Fortunato, Carl T. Bergstrom, Katy Börner, James A. Evans, Dirk Helbing, Staša Milojević, Alexander M. Petersen, Filippo Radicchi, Roberta Sinatra, Brian Uzzi, Alessandro Vespignani, Ludo Waltman, Dashun Wang, and Albert-László Barabási. 2018. Science of science. *Science* 359, 6379 (2018). https://doi.org/10.1126/science.aao0185

[18] Google, Inc. 2023. Puppeteer. https://pptr.dev/.

[19] Google Scholar. 2023. https://scholar.google.com/.

[20] John Gruber. 2004. Markdown. https://daringfireball.net/projects/markdown/.

[21] Steve Harrison, Scott Minneman, Maribeth Back, Anne Balsamo, Mark Chow, Rich Gold, Matt Gorbet, and Dale Mac Donald. 2001. Design: the what of XFR. *Interactions* 8, 3 (2001), 21–30. https://doi.org/10.1145/369825.369829

[22] Andrew Head, Kyle Lo, Dongyeop Kang, Raymond Fok, Sam Skjonsberg, Daniel S. Weld, and Marti A. Hearst. 2021. Augmenting Scientific Papers with Just-in-Time, Position-Sensitive Definitions of Terms and Symbols. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM. https://doi.org/10.1145/3411764.3445648

[23] Andrew Head, Amber Xie, and Marti A. Hearst. 2022. Math Augmentation: How Authors Enhance the Readability of Formulas using Novel Visual Design Practices. In *CHI Conference on Human Factors in Computing Systems*. ACM. https://doi.org/10.1145/3491102.3501932

[24] Jeffrey Heer. 2021. Fast & Accurate Gaussian Kernel Density Estimation. In *2021 IEEE Visualization Conference (VIS)*. IEEE. https://doi.org/10.1109/vis49827.2021.9623323

[25] William C. Hill, James D. Hollan, Dave Wroblewski, and Tim McCandless. 1992. Edit wear and read wear. In *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '92*. ACM Press. https://doi.org/10.1145/142750.142751

[26] Daniel S. Himmelstein, Vincent Rubinetti, David R. Slochower, Dongbo Hu, Venkat S. Malladi, Casey S. Greene, and Anthony Gitter. 2019. Open collaborative writing with Manubot. *PLOS Computational Biology* 15, 6 (2019), e1007128. https://doi.org/10.1371/journal.pcbi.1007128

[27] Tom Hope, Doug Downey, Oren Etzioni, Daniel S. Weld, and Eric Horvitz. 2022. A Computational Inflection for Scientific Discovery. (2022). https://doi.org/10.48550/ARXIV.2205.02007

[28] Jupyter Book. 2023. https://jupyterbook.org/.

[29] Dongyeop Kang, Andrew Head, Risham Sidhu, Kyle Lo, Daniel S. Weld, and Marti A. Hearst. 2020. Document-Level Definition Detection in Scholarly Documents: Existing Models, Error Analyses, and Future Directions. (2020). https://doi.org/10.48550/ARXIV.2010.05129

[30] Dae Hyun Kim, Enamul Hoque, Juho Kim, and Maneesh Agrawala. 2018. Facilitating Document Reading by Linking Text and Tables. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. ACM. https://doi.org/10.1145/3242587.3242617

[31] Rodney Kinney, Chloe Anastasiades, Russell Authur, Iz Beltagy, Jonathan Bragg, Alexandra Buraczynski, Isabel Cachola, Stefan Candra, Yoganand Chandrasekhar, Arman Cohan, Miles Crawford, Doug Downey, Jason Dunkelberger, Oren Etzioni, Rob Evans, Sergey Feldman, Joseph Gorney, David Graham, Fangzhou Hu, Regan Huff, Daniel King, Sebastian Kohlmeier, Bailey Kuehl, Michael Langan, Daniel Lin, Haokun Liu, Kyle Lo, Jaron Lochner, Kelsey MacMillan, Tyler Murray, Chris Newell, Smita Rao, Shaurya Rohatgi, Paul Sayre, Zejiang Shen, Amanpreet Singh, Luca Soldaini, Shivashankar Subramanian, Amber Tanaka, Alex D. Wade, Linda Wagner, Lucy Lu Wang, Chris Wilhelm, Caroline Wu, Jiangjiang Yang, Angele Zamarron, Madeleine Van Zuylen, and Daniel S. Weld. 2023. The Semantic Scholar Open Data Platform. (2023). https://doi.org/10.48550/ARXIV.2301.10140

[32] Clemens N. Klokmose, James R. Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon. 2015. *Webstrates*. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. ACM. https://doi.org/10.1145/2807442.2807446

[33] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, and others. 2016. *Jupyter Notebooks-a publishing format for reproducible computational workflows*. Vol. 2016.

[34] D. E. Knuth. 1979. *TEX and METAFONT: New directions in typesetting*. American Mathematical Society.

[35] D. E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (1984), 97–111. https://doi.org/10.1093/comjnl/27.2.97

[36] Leslie Lamport. 1985. *LaTeX: A Document Preparation System*. Addison-Wesley Professional.

[37] Patrice Lopez. 2009. *GROBID: Combining Automatic Bibliographic Data Recognition and Term Extraction for Scholarship Publications*. Springer Berlin Heidelberg, 473–474. https://doi.org/10.1007/978-3-642-04346-8_62

[38] Bruce D. Lucas and Takeo Kanade. 1981. An Iterative Image Registration Technique with an Application to Stereo Vision. In *International Joint Conference on Artificial Intelligence*.

[39] John MacFarlane. 2023. Pandoc: A Universal Document Converter. https://pandoc.org/.

[40] Meta Open Source. 2023. React. https://react.dev/.

[41] MyST Markdown. 2023. https://myst-tools.org/.

[42] T. H. Nelson. 1965. Complex information processing. In *Proceedings of the 1965 20th national conference on -*. ACM Press. https://doi.org/10.1145/800197.806036

[43] T. H. Nelson. 1981. *Literary Machines*. Mindful Press.

[44] Observable. 2023. https://observablehq.com/.

[45] Observable Inputs. 2023. https://github.com/observablehq/inputs.

[46] Observable Runtime. 2023. https://github.com/observablehq/runtime.

[47] Overleaf. 2023. Online LaTeX Editor. https://www.overleaf.com/.

[48] Thomas A Phelps and Robert Wilensky. 2000. Robust intra-document locations. *Computer Networks* 33, 1-6 (2000), 105–118. https://doi.org/10.1016/s1389-1286(00)00043-8

[49] Quarto. 2023. https://quarto.org/.

[50] Napol Rachatasumrit, Jonathan Bragg, Amy X. Zhang, and Daniel S Weld. 2022. CiteRead: Integrating Localized Citation Contexts into Scientific Paper Reading. In *27th International Conference on Intelligent User Interfaces*. ACM. https://doi.org/10.1145/3490099.3511162

[51] Stuart Ritchie. 2022. The Big Idea: Should we get rid of the scientific paper? https://www.theguardian.com/books/2022/apr/11/the-big-idea-should-we-get-rid-of-the-scientific-paper. *The Guardian* 11 (2022).

[52] RMarkdown. 2023. https://rmarkdown.rstudio.com/.

[53] Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. Exploration and Explanation in Computational Notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM. https://doi.org/10.1145/3173574.3173606

[54] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (2017), 341–350. https://doi.org/10.1109/tvcg.2016.2599030

[55] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. 2016. Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. *IEEE Transactions on Visualization and Computer Graphics* 22, 1 (2016), 659–668. https://doi.org/10.1109/tvcg.2015.2467091

[56] Manolis Savva, Nicholas Kong, Arti Chhajta, Li Fei-Fei, Maneesh Agrawala, and Jeffrey Heer. 2011. ReVision. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM. https://doi.org/10.1145/2047196.2047247

[57] Zejiang Shen, Kyle Lo, Lucy Lu Wang, Bailey Kuehl, Daniel S. Weld, and Doug Downey. 2022. VILA: Improving Structured Content Extraction from Scientific PDFs Using Visual Layout Groups. *Transactions of the Association for Computational Linguistics* 10 (2022), 376–392. https://doi.org/10.1162/tacl_a_00466

[58] Nicole Sultanum, Fanny Chevalier, Zoya Bylinskii, and Zhicheng Liu. 2021. Leveraging Text-Chart Links to Support Authoring of Data-Driven Articles with VizFlow. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM. https://doi.org/10.1145/3411764.3445354

[59] Editorial Team. 2021. Distill Hiatus. *Distill* 6, 7 (2021). https://doi.org/10.23915/distill.00031

[60] The Alliance for Networking Visual Culture. 2023. Scalar. https://scalar.me/anvc/scalar/.

[61] Typst. 2023. Typst: Compose papers faster. https://typst.app/.

[62] Bret Victor. 2011. Explorable Explanations. http://worrydream.com/ExplorableExplanations/.

[63] Lucy Lu Wang, Isabel Cachola, Jonathan Bragg, Evie Yu-Yen Cheng, Chelsea Haupt, Matt Latzke, Bailey Kuehl, Madeleine N van Zuylen, Linda Wagner, and Daniel Weld. 2021. SciA11y: Converting Scientific Papers to Accessible HTML. In *The 23rd International ACM SIGACCESS Conference on Computers and Accessibility*. ACM. https://doi.org/10.1145/3441852.3476545

[64] Lars Willighagen. 2023. Citation.js. https://citation.js.org/.

[65] Gary Wolf. 1995. The Curse of Xanadu. https://www.wired.com/1995/06/xanadu/. In *Wired*.

[66] Workshop on Visualization for AI Explainability. 2022. http://visxai.io/.

[67] Polle T. Zellweger, Susan Harkness Regli, Jock D. Mackinlay, and Bay-Wei Chang. 2000. The impact of fluid documents on reading and browsing. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. ACM. https://doi.org/10.1145/332040.332440

[68] Jonathan Zong, Crystal Lee, Alan Lundgard, JiWoong Jang, Daniel Hajas, and Arvind Satyanarayan. 2022. Rich Screen Reader Experiences for Accessible Data Visualization. *Computer Graphics Forum* 41, 3 (2022), 15–27. https://doi.org/10.1111/cgf.14519