Designing postmodern substrate architectures

Position paper for ‹Programming› 2025's *Software Substrates* workshop

Camille Gobert[a]

a Université Paris-Saclay, CNRS, Inria, Laboratoire Interdisciplinaire des Sciences du Numérique

Abstract Manipulating information with a computer requires reading, transforming and writing that information using a series of constrained encodings in the computer's memory—*information substrates*. However, to perceive and act upon that information, humans need to interact with representations and instruments that differ from the encoded information itself—*interaction substrates*. Reconciling these two types of substrates is difficult: accessing, observing and transforming the underlying digital data is conceptually and technically challenging, especially in the major software architectures in use. To address these difficulties, researchers often suggest replacing various parts of established software with arguably better alternatives, following a *modern* view of computing. In this position paper, I critique this approach and argue in favour of a more *postmodern* view of computing, which acknowledges the diversity of computing in the 21st century and encourages research that embraces the constraints that come with it rather than rejecting them altogether. Following this claim, I give examples of modern failures and postmodern successes and present a few research directions that I explored recently and would like to explore in the future to foster discussions and future collaborations on this topic.

ACM CCS 2012

- **Software and its engineering** → *Software creation and management*; *Software organization and properties*;
- **Human-centered computing** → *HCI theory, concepts and models*;

Keywords information, interaction, substrates, operating systems, postmodernism

The Art, Science, and Engineering of Programming

Computers are, by definition, apparatuses for manipulating information that we encode using sequences of symbols stored in the computer's memory—bits, usually. In modern computers, this manipulation is both mechanised and programmable. It is mechanised as, unlike other tools that we can use to encode information using symbols such as abacuses and paper, moving from one symbolic state to another is the result of the machine executing hardware-enforced instructions that do not (necessarily) require any human intervention. It is also programmable, in the sense that the way symbols are transformed can be changed over time, such as by running a program rather than another.

A consequence of that definition is that any information that we manipulate with a computer—such as a piece of text or an image—must be encoded somewhere, somehow, in some of its memory. Often times, though, a single piece of information is actually encoded in several locations and formats in parallel—in other words, in different *information substrates*. To illustrate this, consider a situation—that I currently experience myself—in which one writes a text document using the LaTeX typesetting system and renders it as PDF using a web-based editor such as Overleaf[1] or Visual Studio Code with a plug-in for previewing the generated PDF in a webview.

In this situation, each encoding serves a different purpose. The content of the document typed by the author is first encoded in the array of String-like objects managed by the text editing component that executes in the webpage. Every time the text changes, the editor updates the DOM, to which the web browser reacts by rendering the page as shapes (following CSS rules) and painting them as pixels to be displayed on the author's screen. Similarly, whenever the document is saved, its content is written as text in a LaTeX file (either locally or on a remote server). The LaTeX compiler that watches for changes is automatically triggered and produces a new PDF file on the same filesystem, whose change is notified to the PDF viewer embedded in the same webpage. The program parses the new PDF into a list of objects, renders them as shapes in a canvas, possibly updating the DOM to, e.g., add invisible text on top to make it selectable, and eventually paints the shapes as pixels.

Although each of these encodings contains different data—either because they encodes the same information in different ways or because some contain extra information—, they are nonetheless kept in sync by the computer. In other words, these transformations are used to enforce *constraints* between substrates. In the above situation, every time the author modifies the LaTeX code, such as by pressing a key on their keyboard, a series of transformations map the hardware interrupt processed by the operating system to an event in the DOM, which is itself processed by a function registered by the text editor's script in order to update the internal array of String-like objects. This change triggers a cascade of transformations to update all the other information substrates that depend on this one, eventually resulting in an updated image on the author's screen. Similarly, whenever the document is saved, a similar series of transformations are automatically performed by the computer to keep the information substrates used by the PDF viewer.

[1] https://www.overleaf.com.

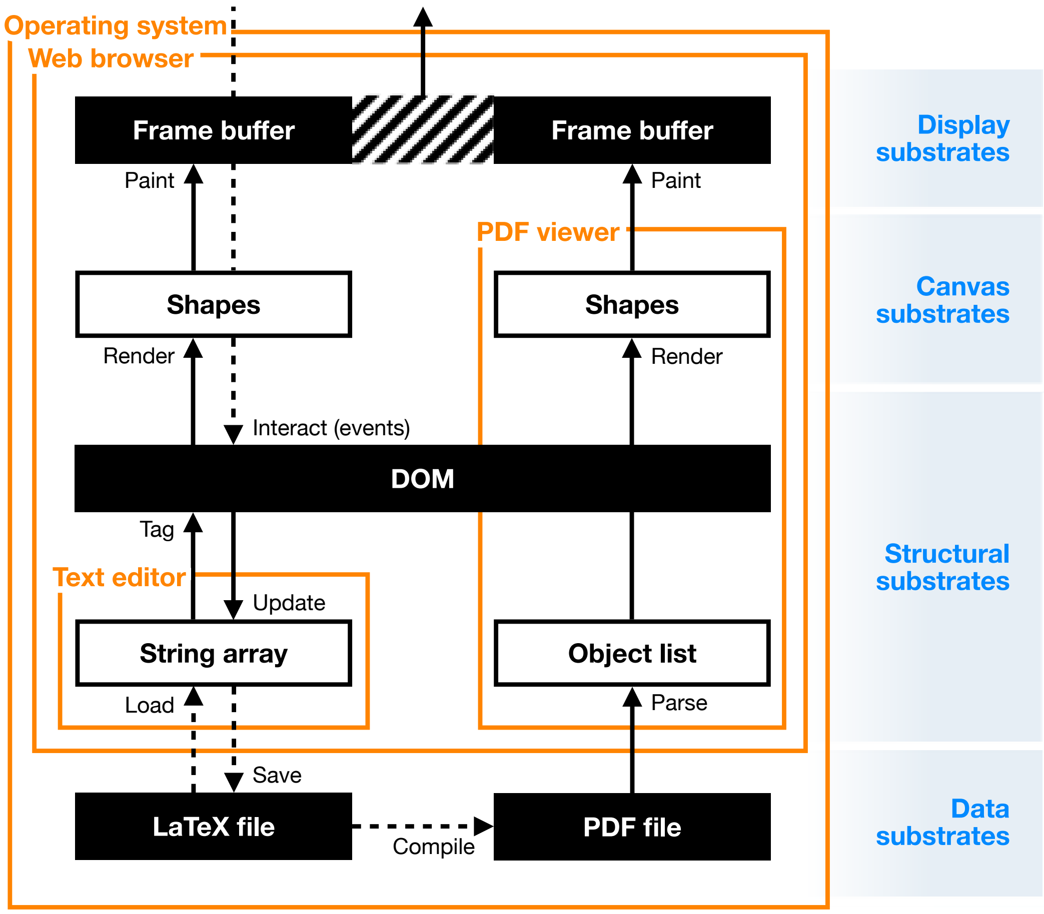

Figure 1 Graph of substrates of the interactive system described in the text. Rectangles represent information substrates that users can either interact with (black rectangles) or not (white rectangles). Arrows represent transformations performed by the machine that maintain constraints between substrates, either automatically (plain arrows) or when the user performs an additional action (dashed arrows). Orange frames represent independent systems that manage the substrates they enclose. Information substrates are vertically arranged according to the type of interaction substrates that can be used to interact with them, which are listed in blue on the right.

Taken together, the information substrates and their inter-dependencies form a graph, depicted in Figure 1, in which each encoding is a node and each arrow is a transformation between two encodings. Different encodings (and their related constraints) are managed by different subsystems running on the computer, shown as orange rectangles on the diagram. A key characteristic of these subsystems is that they are usually not aware of the encodings managed by other subsystems, even when they depend on them. For example, although the web browser is responsible for managing the memory used by the text editor and the PDF viewer programs, it has no understanding of the idiosyncratic meaning of the information that the programmer encodes and manipulates in this memory using JavaScript.

However, when we humans think about and interact with a computer, we usually only consider a limited subset of this graph of substrates, shown as black rectangles in Figure 1. There is no inherent reason for only being able to interact with only some of these substrates; it is only a matter of providing the human user with appropriate representations and instruments for doing so—or, as Mackay and Beaudouin-Lafon call them, *interaction substrates* [15]. Interaction substrates are equally meant to belong to this graph, but unlike information substrates, they are meant to be used by humans rather than by computers. As a consequence, we must be able to perceive the information they contain using output peripherals (such as screens and loudspeakers) and/or act upon it using input peripherals (such as mice and keyboards). This means that any information substrate can be exposed to a user as long as the data it contains can be mapped to the appropriate interaction substrate. In the figure, this is depicted by classifying each information substrate into one of the four categories of interaction substrates (shown in blue) proposed by Mackay and Beaudouin-Lafon [15, fig. 8] according to the type of information it contains.

To use a certain interaction substrate S_1 implemented in system X to interact with the data D contained in and constrained by a certain information substrate S_2 implemented in system Y, the underlying systems must exhibit a number of properties: readability (R), interpretability (I), observability (O) and writeability (W). In order to present D to the user via S_1, X must be able to read the memory that contains D (R), interpret the format it is encoded in (I) and observe changes that may occur in it (O). In addition, to let users modify D, S_1 must be able to write D in such a way that S_2 is notified of every modification performed by S_1 (W + O).

Achieving these properties is easier in certain types of programming systems than in others [10]. For example, using Smalltalk [9] as an operating system guarantees that every object is globally accessible by any other object and encoded in the same format. Since substrates must be implemented as objects, any substrate can read, interpret and write any other substrate they get a hold onto. In addition, while Smalltalk's object properties are not reactive by default, they can be made observable by exploiting the high level of reflexivity supported by the system. The high level of malleability permitted by this architecture is well demonstrated in the Glamorous Toolkit,[2] a

[2] https://gtoolkit.com.

system based on Pharo[3] (a modern Smalltalk implementation) that encourages users to systematically create ad-hoc representations for every piece of digital information they reason about in their minds, a paradigm called *moldable development*.

Although they were visionary back in the 1970s and keep being developed and researched nowadays, this type of system is very different from those that constitutes most of the computing of the early 21st century. Today, all the major operating systems in use strongly isolate the different subsystems they execute from each other (think Unix processes). In these systems, unless two subsystems were specifically designed to exchange information (using, e.g., an IPC channel provided by the OS), even achieving readability or writeability has become challenging. Moreover, apart from a few common file formats, the information processed in these systems is encoded in very diverse ways, making interpretation not only costly but also extremely case-specific—and therefore hard to reuse across situations—, as illustrated by the difficulty to implement mechanisms such as foreign function interfaces, which let a programmer call a function written in a different language than the one they program in.[4] Despite some attempts to break walls between subsystems in a systematic way, as did Apple with OpenDoc and Microsoft with OLE, no such attempt succeeded. As a result, the few main ways to let subsystems communicate together that are in use today rely on mechanisms that are old and standard enough to be available on every major operating system: file systems and network protocols.

In light of these observations, stances from researchers can be arranged on an ideological axis ranging between two opposite views of computing: modernism and postmodernism. Modern computing supports unified visions of what computing should be and argues in favour of replacing what exists with something designed to be inherently better: a new programming language, a new data format, a new operating system, etc. For example, a modern viewpoint may argue that since operating systems in which everything is an object are more adapted for interacting with any piece of data with the interaction substrate of our choosing, we shall replace the operating systems we already use everywhere with something closer to Smalltalk to address the limitations we currently face. In contrast, postmodern computing, which may be characterised by *the absence of a "grand narrative"* [16, p. 8], encourages to adapt to the computing world as it currently is by designing *for*—rather than *against*—the constraints we inherit from the stack of established practices and technologies we must deal with, such as programming using plain text and common data formats.

The story of JavaScript is a good example of postmodern computing. Although it was initially created for simple scripting and designed during a 10 days rush in 1995 [19, §2.2], it has become one of the most popular programming technologies in use today. Despite its imperfections, JavaScript was never replaced by arguably "better" alternatives (among which ActiveX, Flash, Silverlight, etc.) that all ceased to exist. Instead, by accepting that JavaScript was here to stay and by acknowledging

[3] https://pharo.org.

[4] Polyglot runtime environments such as GraalVM (https://www.graalvm.org) are making this progressively easier, but they are still far from being used at large.

the constraints that came with it, work that we keep benefiting from today was put into *improving*—rather than *replacing*—the user and developer experience, such as by developing interactive inspectors, faster interpreters and gradual type systems.

Can we learn from this story and put the same kind of work in other *things* that are here to stay in order to foster malleability of existing systems, rather than trying to replace them? More specifically, regarding interaction substrates (and operating systems), can we, as Kell [11, §7] puts it, *"accept the complex reality of existing ('found') software, developed in ignorance of our system, and [shift] our system's role to constructing views, including Smalltalk-like ones, of this diverse reality"*? This is the research direction on interaction substrates and their implementations that I want to explore.

In my own doctoral work [6], I worked on augmenting text editors with additional interaction substrates—that I call *projections*—to interact with specific fragments of code that are hard to manipulate as text, such as colours, tables and images. This work led me to implement two systems, *i*-LaTeX [7] and Lorgnette [8], which respectively apply this idea to four kinds of LaTeX fragments and let users freely map any textual or syntactic pattern to any projection provided by the text editor using a specification language. Although I initially believed that complementing text with other notations was novel, I quickly realised that a myriad of related concepts, techniques and software had been developed since the mid-1990s, including intentional programming [18], Barista [13], presentation extension [5], visual syntax [1], mage [12], Livelits [17], notational programming [2] and visual replacements [3]. Yet, putting this long line of work in perspective with the rarity of non-textual projections in the main programming systems in use today [10], as well as my own experience in adding new projections to existing text editors, made me realise that the biggest challenge is perhaps not to establish that completing text with projections can be helpful but to find ways to do so without requiring users and communities to switch to entirely different artifact ecologies in order to benefit from these projections.

Since then, this challenge has made me interested in evolving OS-level concepts that currently hinder malleability and richer interaction using a postmodern approach. For example, the in-memory file system that features real-time collaboration and cross-file reactivity I am currently working on makes me wonder how it could be mounted on a traditional OS file system, therefore enabling users to use any program installed on their system to interact with the files that it contains without having to re-create such software for yet another data format.[5] In addition, I am also increasingly interested in investigating how bringing uni- and bidirectional data format conversion to the operating system could help programmers and users connect software and data that were *not* designed to work together in the first place, a goal that echoes recent work on lenses and schema evolution [4, 14]. I am looking forward to discuss these ideas and issues during the workshop which, I hope, will lead to new ways of thinking about information and interaction substrates and pave the way for fruitful collaborations.

[5] This, in turns, questions what are the minimal non-breaking changes to, e.g., the Linux filesystem API, that are required to make this possible, at least to some extent.

References

[1] Leif Andersen, Michael Ballantyne, and Matthias Felleisen. “Adding Interactive Visual Syntax to Textual Code”. In: *Proceedings of the ACM on Programming Languages*. Volume 4. ACM, 2020, pages 1–28. DOI: 10.1145/3428290.

[2] Ian Arawjo, Anthony DeArmas, Michael Roberts, Shrutarshi Basu, and Tapan Parikh. “Notational Programming for Notebook Environments: A Case Study with Quantum Circuits”. In: *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*. UIST ’22. ACM, 2022, pages 1–20. DOI: 10.1145/3526113.3545619.

[3] Tom Beckmann, Daniel Stachnik, Jens Lincke, and Robert Hirschfeld. “Visual Replacements: Cross-Language Domain-Specific Representations in Structured Editors”. In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments*. PAINT 2023. ACM, 2023, pages 25–35. DOI: 10.1145/3623504.3623569.

[4] Jonathan Edwards, Tomas Petricek, Tijs van der Storm, and Geoffrey Litt. “Schema Evolution in Interactive Programming Systems”. In: *The Art, Science, and Engineering of Programming* 9.1 (2024), 2:1–2:33. DOI: 10.22152/programming-journal.org/2025/9/2.

[5] Andrew D. Eisenberg and Gregor Kiczales. “Expressive Programs through Presentation Extension”. In: *Proceedings of the 6th International Conference on Aspect-Oriented Software Development - AOSD ’07*. ACM, 2007, pages 73–84. DOI: 10.1145/1218563.1218573.

[6] Camille Gobert. “Projecting Computer Languages for a Protean Interaction”. PhD thesis. Université Paris-Saclay, 2024.

[7] Camille Gobert and Michel Beaudouin-Lafon. “I-LaTeX: Manipulating Transitional Representations between LaTeX Code and Generated Documents”. In: *CHI Conference on Human Factors in Computing Systems*. CHI ’22. ACM, 2022, pages 1–16. DOI: 10.1145/3491102.3517494.

[8] Camille Gobert and Michel Beaudouin-Lafon. “Lorgnette: Creating Malleable Code Projections”. In: *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. UIST ’23. ACM, 2023, pages 1–16. DOI: 10.1145/3586183.3606817.

[9] Daniel Ingalls. “The Evolution of Smalltalk: From Smalltalk-72 through Squeak”. In: *Proceedings of the ACM on Programming Languages* 4.HOPL (2020), 85:1–85:101. DOI: 10.1145/3386335.

[10] Joel Jakubovic, Jonathan Edwards, and Tomas Petricek. “Technical Dimensions of Programming Systems”. In: *The Art, Science, and Engineering of Programming* 7.3 (2023), pages 1–59. DOI: 10.22152/programming-journal.org/2023/7/13.

[11] Stephen Kell. “The Operating System: Should There Be One?” In: *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems*. PLOS ’13. ACM, 2013, pages 1–7. DOI: 10.1145/2525528.2525534.

[12] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. "Mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks". In: *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. UIST '20. ACM, 2020, pages 140–151. DOI: 10.1145/3379337.3415842.

[13] Amy J. Ko and Brad A. Myers. "Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '06. ACM, 2006, pages 387–396. DOI: 10.1145/1124772.1124831.

[14] Geoffrey Litt, Peter van Hardenberg, and Orion Henry. "Cambria: Schema Evolution in Distributed Systems with Edit Lenses". In: *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC '21. ACM, 2021, pages 1–9. DOI: 10.1145/3447865.3457963.

[15] Wendy Mackay and Michel Beaudouin-Lafon. "Interaction Substrates: Combining Power and Simplicity in Interactive Systems". In: *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. To Appear. ACM, 2025.

[16] James Noble and Robert Biddle. "Notes on Postmodern Programming". In: *17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*. ACM, 2002, pages 1–23.

[17] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. "Filling Typed Holes with Live GUIs". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, 2021, pages 511–525. DOI: 10.1145/3453483.3454059.

[18] Charles Simonyi. *The Death Of Computer Languages, The Birth of Intentional Programming*. Technical report MSR-TR-95-52. Microsoft Research, 1995.

[19] Allen Wirfs-Brock and Brendan Eich. "JavaScript: The First 20 Years". In: *Proc. ACM Program. Lang.* 4.HOPL (2020), 77:1–77:189. DOI: 10.1145/3386327.

********************** PAPER 11 **********************
AUTHORS: Camille Gobert
TITLE: Designing postmodern substrate architectures

++++++++++ REVIEW 1 (Gilad Bracha) +++++++++
This (and the other essay from INRIA) provide a very different perspective on substrates. Here the emphasis on document production and the difficulties around varying formats. Essentially nothing is said about computation; the idea of integrating documents and computation is core to the idea of substrates. So in a way this almost irrelevant; but I see real value in getting the experience of people dealing with real problems.

My own take on this is very modernist: build tools that do all this properly rather than battling all this legacy. However, humans are already tied to the old broken ways. AI will make resolving this quite straightforward in the foreseeable future.

One more comment. I think everyone understands the basic issues around data representation etc. So a lot of the essay isn't necessary for this audience.

++++++++++ REVIEW 3 (Tomas Petricek) +++++++++
The vision statement argues for a post-modern view of substrates - as something that has to be based on the existing ecosystems (artifact ecologies) and has to be built by gradually evolving the existing systems. It describes concrete work by the author to do this - building on technologies such as LaTeX and VS Code.

The paper is also based on a detailed analysis of the nature of programming substrates. It makes a distinction between "information substrate" and "interaction substrate". Substrates are also described in terms of notions such as "transformations" and "constraints". I think all those terms should serve as the basis for a more clear understanding of what a substrate is to be developed at the meeting.

I had various thoughts triggered by the vision statements that may be of interest:

* In urban planning, Jane Jacobs talks about systems that "work despite theory" (a city sidewalk life has unique characteristics that modernist urban planners cannot comprehend) - a topic I find interesting for SW too [1] - I wonder to what extent do successful post-modern (pre-)substrates share this characteristic?

* I thought the hint of a formal analysis (substrate S, system X, and the R, I, O, W properties) suggest a possible valuable research project within the "normal science of programming substrates" - Can we use this scheme to describe a broader range of programming substrates?

* Another theme I find interesting is how substrates with good characteristics evolve and lose their good characteristics (such as Smalltalk as a source of inspiration for Java). Apple OpenDoc and Microsoft OLE would be interesting points to look at! (Often, I think the reason is "good engineering reasons" - but in case of those, it sounds like something else - perhaps just their technical complexity?)

[1] Petricek, Tomas. "Programming as architecture, design, and urban planning." Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. 2021.

++++++++++ REVIEW 4 (Antranig Basman) +++++++++
Cheers, Camille - I really appreciate the taxonomy of activities relating activities of interaction substrates to information substrates, and also that Michel and Wendy’s interaction substrates paper has finally arrived which I’ve

been looking forward to for a while. I resonate a good deal with the remarks on page 5 on how the modern practice of isolation has challenged our ability to put the subsystems of a system at the disposal of their user, and how we must rely on older mechanisms as a result. Also the framing of modern vs postmodern approaches, continuing from the line of Noble & Biddle, gives me a lot to think about especially re the definition of a substrate itself that we will shortly debate.

I see that there is something studiedly postmodern in the definitional style of the Parisian school, which for example determines that a substrate "(1) contains and structures objects of interest… etc.". This is indeed a kind of definition which accepts the complex reality of "found" software, in that it is prepared to accept essentially any structure as a substrate, and indeed the definition of information substrate is even more permissive than this, encompassing essentially any form of storage. I would call this a kind of "adjectival" definition of substrate which is prepared to respond to "substratiness" rather than a "predicative" one that determines what a substrate should and should not do. And despite his postmodern bent, I find this is at odds with Stephen's current submission "substratus unicus" which speculates, instead, on what could be "a substrate" and that, when found, it could enjoy a kind of canonicality. Despite this being a little startlingly "modern" I find myself aligned with this idea.

I think it is great to have both tendencies represented at this workshop since, likely, the important part of our work is to harmonise these extremes, and perhaps arrive at a view where the Parisian postmodern approach establishes a kind of "basin of attraction" for various practices, guiding and aligning them with the work of producing particular substrates. But I do find the intentionality here interesting - for example, when I in my own vision slide say "a substrate provides a complete ontology for elements in memory and on screen" I am intending to make a definitional statement describing the work of the substrate I am building, putting me in the "modern" rather than "postmodern" camp - but I see that quite easily the statement could also be seen with a postmodern lens.

This issue of "how to approach the world of found software" aka "compatibilism" which we see is rather vexed amongst the other vision statements and their responses was also at the heart of my previous vision slide on "stack goggles"
https://docs.google.com/presentation/d/1Qn42eK9vkpFikDM-yU7ytMVeZTrCaXPz0clQOdE0NUY/edit?usp=sharing
. This view I think connects with the goal of Lorgnette which is a very cool system.

In detail, I hope we will be able to talk about the need for improved reactive primitives which I expect is something you've already characterised in the need to make systems like Lorgnette composable. Current approaches to bidirectional dataflow, asynchrony, errors and related concerns are a poor fit for the demands of a substrate, modern or postmodern. Looking forward!

++++++++++ REVIEW 5 (Tom Larkworthy) +++++++++
I agree the software industry is too established to rip it out and boil the ocean with a new paradigm. Javascript continues to evolve with WASM/WASI, allowing native code to be transpiled to Javascript. For non-numerical computation, Javascript wins on distribution and tooling and ecosystem. I would add its a functional language, as it is idiomatic to pass functions around, and its non-blocking model is high performance and its event loop avoids various concurrency bugs that are common in other languages. I love Javascript, its surprisingly fast given the lack of an integer type ?!

I recently attended local-first conf 2025 and I think the author would be interested in braid-fs which is a layer on the existing filesystem for realtime collaboration. I dunno if I would call this post-modern though, I feel like using specialised programs for interacting with files is unixy and therefore modern, but still think its a good `thin waist` strategy! It is for this reason I make lopecode plain text and file base, for composition with git or anything else that appears post-git. I think its important we pick our battles wisely, the cost of software is increasing, and careful management of scope reduces risk of over commitment.