Designing Immortal Software: A Vision for Lopecode

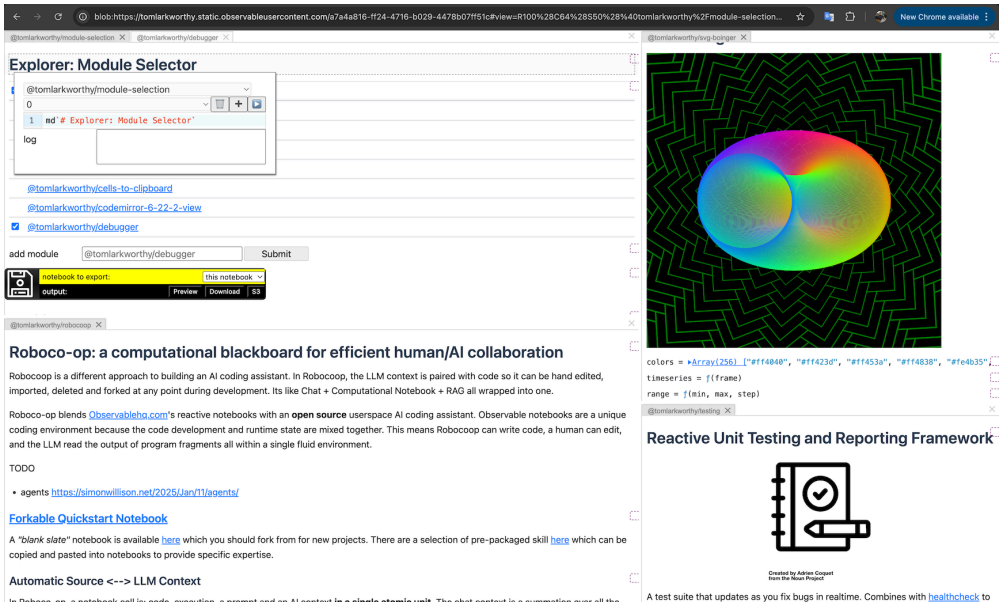
this is available as a lopecode document online

Overview

Lopecode is a platform for building and sharing reactive, interactive computational media that can live forever. It's forkable, self-contained, and build-free. It borrows the liveness and expressiveness of systems like Observablehq Notebooks, Smalltalk, HyperCard, Visual Basic, and Spreadsheets—but updates them with web-native technologies and a focus on long-term sustainability.

The goal of this statement is to share what I'm exploring, why I think it's valuable, and what kinds of projects or collaborations I hope to connect with.

What Lopecode Is



Screenshot showing the multi-notebook view, served from a blob domain (offline). On the top left we see the inline cell code editor

Relationship to Observable

Observable is an amazing evolution of the spreadsheet and code. It provides instant preview without any setup and offers a searchable corpus of reusable examples. Lopecode builds directly on this foundation—it reuses the Observable reactive runtime engine and shares much of its programming model and visual aesthetics.

Where Lopecode diverges radically is in its focus on permanence, malleability, and whole-runtime composition:

- **Offline-first and Durable**: Lopecode notebooks are static HTML files that can be archived, emailed, or hosted anywhere—no platform dependency.
- **No iframe Separation**: In Observable, the user code runs inside a sandboxed iframe, separate from the platform code, isolated for security reasons. In Lopecode, there is no such separation. The entire system, including the editor, is built in userspace and is fully modifiable. You can live-edit the platform itself and do anything the web can (e.g. SharedArrayBuffer, which not possible on Observable enabling WASI tooling). Lopecode notebooks as-a-whole are isolated by running in isolated domains. There is no domain host that needs protection from userspace *because* they are offline, the browser itself providing isolation of the webpage from the host system.
- **Multi-Notebook and Reflective by Default**: Observable focuses on a single notebook at a time. Lopecode supports multiple notebooks open simultaneously, communicating on a shared reactive runtime that supports self-reflection. This encourages modular design and enables global metaprogramming capabilities. For example
  - a **testing notebook** can scan the entire runtime for tests and generate unified test results across the aggregate.
  - a **debugging notebook** can generate a time series of all cell transitions across notebooks, helping debug interdependencies.
  - an **editor notebook** enabled in-place mutation of other cells in any other notebook
  - an **LLM** notebook can help explain how things work and help fix bugs and apply software modifications.

With Lopecode you build up a network of cooperating notebooks that collaborate. Each is a orthogonal literate programming slice of the whole programming system.

Lopecode Features

To clarify the scope and implementation of Lopecode, here is a breakdown of its concrete features:

- **Modular Reactive Notebooks**: built-in reactivity and componentized structure
- **Live Interdependencies**: cells can reference and respond to changes in one another
- **Namespaces**: scope cells into modules for clean organization
- **Self-reflection**: cells have access to the runtime and can iterate and inspect other cells programatically.
- **Meta programming**: cells can rewrite and create new cells programatically.
- **Hypertext interface**: native support for hyperlinking between cell dependancies within the serialized graph
- **Literate programming**: first class support for markdown, HTML, d3 cells for documentation, with programatic interpolation from live runtime values.
- **Self-hosted**: no need for cloud services or remote runtimes, works from a `file://` domain
- **Internal IDE**: fully featured notebook editor embedded in the platform
- **Offline-first**: can be used and edited without network connectivity
- **Self-exportable**: notebooks can be modified and exported as a new single file
- **Data/Dataviz Enabled**:
  - Built-in support for **D3**, **Plot**, and other visualization libraries
  - **FileAttachment** support for embedding datasets
- **Plain Text Serialization Format**:
  - Fully **human-readable**
  - **Git diff friendly** for collaboration

- **Single HTML File Deployment**:
  - Easy to host—just upload the file to a static web host
  - Easy to run—works directly from the `file://` domain without a server
- **Modular Internals**:
  - Uses standard **JavaScript modules**
  - Supports inclusion of **static file assets**, **images**, and **binary data**

The result is a system that feels rich and modern, but is simple to preserve, inspect, and distribute.

Lopecode stands on the shoulders of Observable, but pushes the model toward a more open-ended, composable, portable, and system-level substrate for interactive media.

Why Durability Matters

Durability is not usually considered important in web based software stacks. In Lopecode, it's a core design principle we hope can unlock broader accessibility, ecosystem health, and long-term impact. Offline-operation prevents a large class of security concerns that mandates networked software has to be regularly patch to be secure.

Durability reduces costs

Platform churn (new OS versions, changing APIs, shifting runtime environments) makes software expensive to maintain over time. This increases the cost of development—especially for solo developers, educators, researchers, and other long-tail creators. By making software trivially hostable, portable and executable, Lopecode lowers the **total cost of ownership**.

Durability increases reuse

Software that works years later is software that gets read, learned from, and remixed. Each fork is a chance for reuse or reinterpretation. A reactive notebook that runs, and can be debugged, is a more valuable artefact than a blog post with outdated dead code blocks. A lopecode document you have a copy of, cannot be broken by software changes made by others, or taken away from you.

Durability as Ecosystem Infrastructure

Fast-moving platforms tend to forget their past. Durable software, by contrast, provides a stable base for long-term knowledge creation. In science, we build on papers written decades ago. Why can't we do the same with interactive software?

Durability as a Design Constraint

By removing modern tooling (frameworks, cloud services, devops), Lopecode pushes toward systems that are understandable, inspectable, and self-reliant. This is a plus for software intended as a tool-for-thought.

As an example: React is a widely used frontend framework, but it's a durability liability. React apps typically depend on large toolchains, break between versions, and require active infrastructure. That's makes sense for commercial software but a poor match for personal, commons owned software.

Lopecode aims for durability by radically reducing dependencies. First, it is a single file. Computer files have been around since the 60s, and are the simplest sharable replicable unit that have stood the test of time. Second, it has no network dependencies and therefore no server or DNS dependencies that require active network connectivity to work. Thirdly it's web, as opposed to an OS-specific native solution. Multiple vendors supply standards-compliant web clients, thus its only real external dependency has multiple free implementations for all major operating systems. Browser ship their own dev tools, enabling state of the art-of-the-art debugging experiences ecosystem compatability. Finally, it's self-hosting—there, is nothing else you need to work with it, as it bundles its own means of production.

What I'm Exploring

Lopecode is both a platform and a hypothesis: that radically lowering the friction to share and remix interactive software will open up new possibilities for personal software, computational blogging, scientific publishing and educational media.

Imagine:

- A layman curating data-driven applications that solve their individual needs, customised from ecosystem forked notebooks.
- A research paper that includes not just charts, but live models you can fork and adjust
- A blog post that's also a playground that a motivated reader can download, run locally and modify.
- A demo that doesn't decay that you can rely to operate the same way everytime you run it.

Most software today is brittle and ephemeral. Even well-crafted systems disappear when their hosting, dependencies, or toolchains break. By contrast, **great writing lasts**. Does programming need to be so fragile?

I'm interested in how much we can push toward this ideal by treating software more like durable documents.

Connections I'm Seeking

Lopecode is still early, but I'd love to find collaborators or idea partners working on:

- **New programming languages or programming models** that want single file distribution and that can be compiled to JS (or WASM?)
- **Computational publishing** (interactive articles, literate programming, explorable explanations)
- **Novel ideas worth replicating** Is there innovative worth redoing on Lopecode.
- **Multimodal or LLM-based interaction** (future directions I moving in)

If you're building systems where:

- Forking matters
- Modifiability is a goal, not a risk
- Rich interactivity shouldn't mean platform lock-in

...then maybe Lopecode could help. Or maybe your ideas could push it further.

Research Areas

The grand vision is self-perpetuating, never breaking, forever useful software. There are many research threads that could contribute towards that goal.

How to make modification easy?

Lower the barrier of entry. Improve the development feedback loop. Simplify the programming model

- Spreadsheet like reactivity
  - New debugging paradigms as reactivity is complex at scale
- Data viz augmented programming e.g. direct manipulation
- Self-documenting architecture.
- Merging histories
- LLM modifications

- Polyglot language support

How to make reproduction easy

Engineering a format that is powerful but easily run. Immortal software is programming + developer tooling + data

- DOM representation
- Single file web bundling tricks
- Size minimization
- Cross domain local-first data transfer
- Multiplayer
- Social transmision

How to make software useful?

- Data visualisation
- Underserved niches
- Valuable abstractions
- Teaching aids

Inspiration

local-first

https://www.inkandswitch.com/local-first/

"You own your data, in spite of the cloud" - Martin Kleppmann et al.

Seven ideals for local-first software

1. No spinners: your work at your fingertips
2. Your work is not trapped on one device
3. The network is optional
4. Seamless collaboration with your colleagues
5. The Long Now
6. Security and privacy by default
7. You retain ultimate ownership and control

Digital Gardening

https://maggieappleton.com/garden-history

- Topography over Timelines
- Continuous Growth
- Imperfection & Learning in Public
- Playful, Personal, and Experimental
- Intercropping & Content Diversity
- Independent Ownership

Malleable Systems Collective

https://malleable.systems/

- Software must be as easy to change as it is to use it
- All layers, from the user interface through functionality to the data within, must support arbitrary recombination and reuse in new environments
- Tools should strive to be easy to begin working with but still have lots of open-ended potential
- People of all experience levels must be able to retain ownership and control
- Recombined workflows and experiences must be freely sharable with others
- Modifying a system should happen in the context of use, rather than through some separate development toolchain and skill set
- Computing should be a thoughtfully crafted, fun, and empowering experience


```
import {exporter} from "@tomlarkworthy/exporter"
```

********** PAPER 4 **********
AUTHORS: Tom Larkworthy
TITLE: Designing Immortal Software: A Vision for Lopecode

++++++++++ REVIEW 1 (Gilad Bracha) +++++++++
I'll probably come back to this, since there is a lot of substance here. So these comments are just a first draft.

I'm aligned on much of what is discussed here. Below are points on which I'm less aligned and may merit discussion.

I don't share the enthusiasm for computational notebooks. Their main positive role has been to familiarize a broad audience with a certain, limited degree of liveness. Observable is better than others, but I would regard your choice of building on it as an implementation convenience and nothing more.

A theme across several of the submissions is the tension between purity and pragmatism. It's a very real trade-off, and I've made compromises here myself. Using Javascript directly is one of the areas where I draw the line. I'm a PL person, and a key part of this exercise is better PLs. I've built my own language, Newspeak, in the Smalltalk tradition, but with changes in support of modularity, security and interoperability with the ugly world outside.

Security and modularity are themselves often in tension with ease of use. In my document system, Ampleforth, I've made things more convenient (and less secure) than in traditional Newspeak development. I see you've done something analogous wrt Observable's security structure. I feel there is need for more thought in this area.

Collaboration seems a bit neglected here. This is where local-first gets hard. It's also where some of the internal tensions in the local-first vision manifest. Its' great to say that you have your own copy that no one can mess with, but if your collaborators want new features from later versions of the software and you don't, it's not clear that collaboration and sync will continue to work. My view is that these two things contradict each other, and the system should allow you to choose which one to prioritize: either you stop syncing and keep things exactly as they were, or you keep collaborating and give up on the idea that your software never changes and will never break.

I tried playing with the live version of the paper. It wasn't as intuitive as I'd wish. Editing the text seems to be done only in the markup (and even then I hit some difficulties). I'd expect WYSIWYG editing by default (unless the doc is read-only, which is a legitimate choice in some cases; it wasn't clear if this was a read-only document). May well be my lack of familiarity.

I'd like to hear more about the AI integration.

Overall, I commend you for building a working system and using it.