The Lopecode Tour

online link

Lopecode is a document you can double-click or throw on the web. It’s readable prose and a live programming environment. It resists bit-rot because it's a local-first, single HTML file that works without network.

It's a tool-for-thought and a distribution format: a reactive runtime that self-serializes with all dependencies bundled. The core is a tiny UI-less microkernel; features are userspace modules you can inspect, edit, and change live.

It’s made for experimentation: extremely extensible, engineered to be fast (no network round trips), durable (plain-text, git-friendly), and immediate — change code and see the system update in place.

▶ Video tour presented at Feeling of Computing

Self-Serializable

Lopecode can serialize itself, so you fork into a new browser tab or download a hardcopy to a file recursively.

Programmable Prose

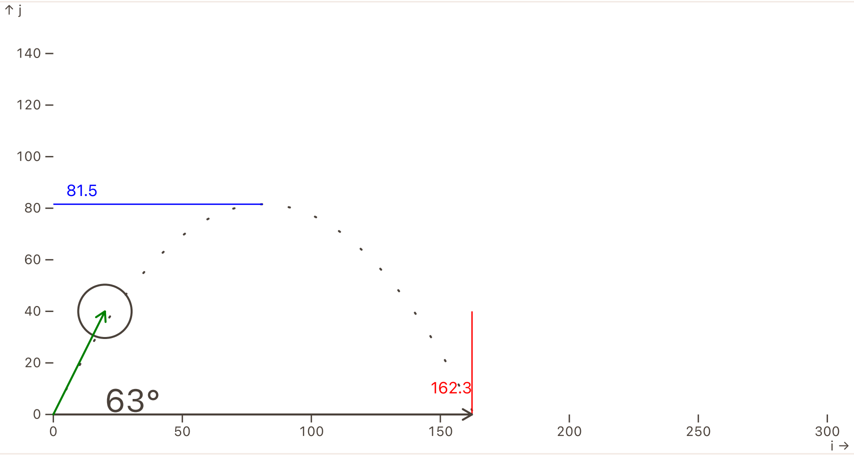
You can include the result of computation in prose. For example, the time *now* is: Thu, 19 Feb 2026 11:37:38 GMT

You can write in markdown, and reference program data inline with `${<expr>}`. When `<expr>` is a DOM element, it is inserted, so you can embed interactive widgets within markdown click me!

```
timeNow = 2026-02-19T11:37:38.075Z
```

Interactive Dataviz

Lopecode includes Observable Plot, a flexible primitive for building interactive data visualizations.



Interactions on plot can be read and referenced and processed by code. Try moving the points around in the above diagram and note how program data updates.

```
▸ Array(2) [162.28338430173338, 81.54943934760449]
```

See manipulate for more examples.

```
import {viewof plot_trajectory, plot_trajectory, max_arc, trajectory_manipulate} from "@tomlarkwo
```

Overview

Lopecode is like a document that can be opened **without the internet**, but it's a programming environment where the authoring tools are bundled so anyone can change it after that fact.

Lopecode is **reactive**, like a spreadsheet. The code that renders and edits the document is written in userspace and are **live** editable. It is designed for simplicity, legibility, durability and composability. The Lopecode file format is plain text and git friendly and can be opened directly by any of the major desktop or mobile operating systems without extra software. The file *can also* be uploaded to static file hosting for distribution on the internet, where it can be redownloaded and remixed by anyone.

The Tour started by showing you some of the high level features that have been developed, answering why you might find the format useful *today*. The real power though, is that you can modify or add your own features, by yourself or with help from the AI. Lopecode *core* is a UI-less microkernel architecture that delegates all high level features to userspace. A Lopecode document is really multiple userspace documents cooperating on a reflective, hot-patchable reactive runtime. So the second part of the tour explains the meta-programming architecture and the third part calls our important userspace modules that you can *mold* live.

Multi-document

A Lopecode document is really many documents, navigated by weblinks. You can work on several documents at the same time, or hide supporting computation behind a curtain. Reactivity is cross module.

open the module-explorer

Simplifying the Programming Process

Modern development is full of friction: big builds, long test cycles, fragile global state, and separate tools for editing, running, and debugging. Lopecode tackles that by making the runtime the primary surface you edit. Code is organized as small reactive cells; changing a cell recomputes only its dependants, so edits are local, fast, and predictable. Reactive tests and an embedded AI harness give immediate, targeted feedback — only the tests causally affected by a change are rerun, and the AI gets the same fast signals you do. Everything lives in userspace: editors, debuggers, importers — you can swap any part. For power users: no black-boxed tooling you can't change, extreme extensibility, and single-file offline export for archiving experiments. In short: hot-swap code, keep state, and iterate a lot faster.

Reactive Dataflow

Lopecode is based on reactive dataflow programming (the Observable Runtime). Its the same model that spreadsheets use. You change something, everything updates. Code is organised into cells; cells update when their upstream dependencies change. The order in which you display the cells is not relevant to the computation -- you are free to order the cells how you like.

```
greet_name = "hello Tom"
```

```
greeting = "hello"
```

if you change name (edit link), the change cascades

```
name = "Tom"
```

```
your_name_is = "Tom is your name"
```

AI ❤️ Dataflow

Lopecode was designed to speed up human programming, but the affordances also transfer to AI. The fact that a dataflow runtime is robust to individual cell failures means we can code up an AI inside the dataflow graph and let it modify other cells without breaking itself.

Prompt Agent config Provider

Roast my code (the Tour of Lopecode).

Clear Cancel

Conversation

started: 2/19/2026, 12:32:57 PM • ended: 2/19/2026, 12:33:39 PM • duration: 42s

tokens: total 0 (in 0, out 0)

Step 2 • unknown

tokens: total 0 (in 0, out 0) responseId: resp_07dd804a1cb1265e006996f4eeaa7c81939c12385e30b12582

```
Nice — you shipped an entire reactive OS inside a single HTML file and then asked to be roasted. Brave. I in
just fling insults blindfolded): I looked through @tomlarkworthy/lopecode-tour and skimmed variables like ti
createFileAttachment/setFileAttachment (112/113), the decompilation test import (test_all_cells_decompilable
(currentModules id=58 / liveCellMap id=68). So this roast is targeted, not random.

Roast (with receipts and fixes)
- Ambitious, not apologetic — which is great. Also terrifying.
  - Receipt: You’ve got ~50 modules in currentModules (id=58) and liveCellMap with dozens of cells (id=68).
subtle breakages.
  - Fix: Keep the core tiny. Extract experimental toys into optional plug-ins that opt-in at runtime.

- You vendored the whole web like a hoarder who’s also a performance engineer.
  - Receipt: exporter, fileattachments, codemirror, escodegen, acorn, etc., are vendored (multiple module im
(mentioned in the text).
  - Roast: Congratulations, you can run offline in a bunker, but you packed the bunker with five different e
the bunker. That’s one devops nightmare away from glorious entropy.
  - Fix: Split heavy deps into lazy-loaded modules or optional attachments. For the export use a “slim” mode

- Function.toString as the canonical source? Cute. Also brittle.
  - Receipt: You rely on Function.prototype.toString() to treat the runtime as the source (see runtime-is-th
  - Roast: That works until you (or an AI agent) write anonymous closures, native-bound functions, or functi
“source” becomes a ransom note.
  - Fix: Add a small serialization contract: require exported cells to expose a source string or an AST. Add
test_all_cells_decompilable id=83, but it’s undefined/pulled from a toolchain — make it a reachable safety n

- The visualizer steals DOM nodes like a polite cat burglar.
  - Receipt: visualizer(runtime, { detachNodes: true }) (visualize id=126) — you explicitly detach nodes to
  - Roast: If two UI components want the same node you either run a visualizer war or clone nodes incorrectl
equivalent of “who gets custody of the button.”
  - Fix: Instead of detaching DOM nodes wholesale, provide a render proxy or a declarative virtual DOM repre
visualizer produce serializable representations and let each host rehydrate.

- timeNow is sampling at Nyquist, which is cute math, not necessary human compassion.
  - Receipt: timeNow generator (id=8) yields every 500ms.
  - Roast: Your CPU loves you for giving it work every half-second. Your battery does not.
  - Fix: Sample once per second unless you actually need 2Hz updates. Or make frequency configurable.

- File attachments: convenient, leaky, and slightly magical.
  - Receipt: createFileAttachment + URL.createObjectURL usage (attachment id=114 and then setFileAttachment
  - Roast: You create object URLs and hand them to the runtime, but do you revoke them? If not, memory leak
  - Fix: Revoke object URLs when the attachment is no longer needed. Add a finalizer or use Blob.text() and

- Meta-programming + agentic AI = power user features, and an existential test case.
  - Receipt: robocoop3 is instantiated (id=31) and it can read/modify runtime (text plus a running agent DOM
  - Roast: You gave the AI scissors and keys to the repository and then said “be helpful.” That’s promising
existential bugs.
  - Fix: Add constrained sandboxes for experimental agent edits. Use the reactive tests (you have them, ids
mode for automated edits (simulate patch, run decomp/behavior tests, then commit).

- Your decompilability test is a weak link (or an optional import that can be absent).
  - Receipt: test_all_cells_decompilable is imported from observablejs-toolchain but is currently undefined
  - Roast: You depend on a noble invariant but wire it to an optional module. It’s like having a smoke alarm
arrive.
  - Fix: Make decompilation checks reachable and fail-safe. If the checker is optional, add a local fallback
non-decompilable constructs.

- UX: visualizer detaches nodes and editors are everywhere — delightful for power users, confusing for morta
  - Receipt: cellEditor used in-page and every cell has an editor (cellEditor id=105; my_value cell editor d
  - Roast: Multiple live editors for the same value is a great stress test for consistency and race conditio
and the other editor is mid-undo.
  - Fix: Make single-writer mode the default for novice flows; give advanced users “multi-editor” toggles. S
active.
```

```
- You love Maps and reactive globals and that’s honestly inspiring and terrifying.
  - Receipt: options Map from currentModules (id=122) and many module-level reactive variables.
  - Roast: The entire runtime-as-source-of-truth model is brilliant but the mental model is heavy: users wil
notebook now belongs to a different timeline.
  - Fix: Add a “safe-edit” mode where edits are performed in a forked runtime that can be previewed and merg
change to the main runtime.”

Quick prioritized action items (conservative, high impact)
1. Make decompilation tests always reachable and failing-safe. Prevent agents from committing changes that b
infrastructure.)
2. Add objectURL revocation for attachments (ids 112/114/115).
3. Turn timeNow sampling down to 1Hz by default (id=8).
4. Make visualizer detachNodes optional and provide a safe clone/serialize path (id=126).
5. Add an “agent dry-run” workflow: let robocoop3 propose patches, run reactive tests, show diffs, then appl
tests id=81).

Final roast one-liner
You built a magnificent, slightly narcissistic reactive runtime that says “I can run anywhere” and then pack
scissors to make the internet nervous. It’s the software equivalent of a Swiss Army
```

Tests (1)

▶ Run tests (3/3 passed)

Step 1 • completed

tokens: total 3227 (in 3114, out 113, reasoning 64, cache r2688/w0) responseId: resp_07dd804a1cb1265e006996f4eb52a481938f454808

No assistant text yet for this step (streaming or tool-only step).

Tools (1)

▶ list_variables (ok) • list_variables (154 variables)

Step 0 • completed

tokens: total 2086 (in 2034, out 52, cache r1536/w0) responseId: resp_07dd804a1cb1265e006996f4e97fdc8193aff4ee7c121b7a13

No assistant text yet for this step (streaming or tool-only step).

Tools (1)

▶ list_modules (ok) • List modules (50 found)

☐ ⬆ ⬇ 🗑 ✚ ▶ 📄 📋

inputs: robocoop3

```
robocoop3({ prompt: 'Roast my code (the Tour of Lopecode).' })
```

```
import {robocoop3} from "@tomlarkworthy/robocoop-3"
```

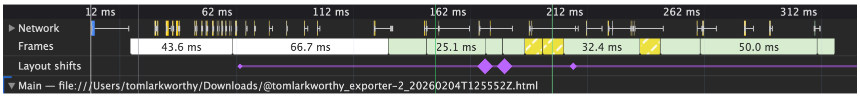
robocoop3 is an Agentic AI that can read and modify the executing program. It is aware of the code that every cell executes and values they produce. You can ask it why something is happening and it very capable at figuring it out! It can explain code, fix bugs, write plans, and implement features.

robocoop2 is a normal dataflow program written in Lopecode that you can change *right now*. If you make changes they will automatically propagate and update the one here. Dataflow is capable of expressing complex live programs.

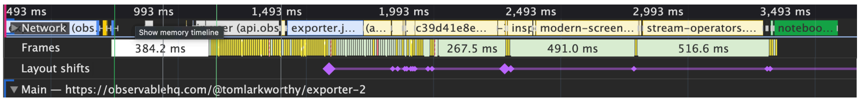
Save to file, open offline

Lopecode saves to a *single* HTML file that requires nothing special to open (download). Double clicking the file will open the page on a `file://` domain where it *just works*, no local-server required.

The experience of opening the file is insanely fast, because no networking is involved.



320ms to load exporter-2 notebook from file



3500ms to load the same notebook from Observablehq.com. 10x slower!

Lopecode's ability to self-serialize encourages low risk experimentation, instead of serializing to file you can serialize to an in-memory fork that manifests as a new browser tab to try things out on. Let the AI run wild in an isolated tab!

Self-serialization has other uses, it is possible to copy the environment to your clipboard, and paste into another website, bringing your Lopecode environment into a 3rd party domain, see the "*Copy To JS*" button on the full *exporter* UI.

fork

this notebook

@tomlarkworthy/bootloader

parchment

Copy as JS Fork Download

```
import {downloadAnchor, forkAnchor, exporter} from "@tomlarkworthy/exporter-2"
```

Web Standards

Lopecode is *not* a new programming language, or a software-as-a-service. It's a HTML document that can be saved to disk for archiving. Old file exports work after years. Here is the first commit to the Lopecode repository on Jan 3, 2025, to `webpage.html`, which looks embarrassing but still runs today! Just save the file to disk and click it!

Lopecode was designed not to bit-rot. By eliminating the network and bundling everything into a standards compliant file, I hope that programs that worked into the past will continue to work long into the future. The web has a good track record of backwards compatibility, and all the evergreen browser implementations work with Lopecode.

Several other advantages come for free with the local Web execution.

- The `debugger` statement works. Furthermore, it synergizes with reactive programming. When you add a `debugger` statement the cell is rerun and often immediately triggers a breakpoint.
- The other rich development features of the web work (performance profiling, page inspection). Modern browsers are essentially IDEs which can be fully leveraged when working with Lopecode.
- You can easily share you work by uploading it to a webserver. Any static hosting works.
- You can leverage the vast existing Javascript ecosystem, e.g. dynamic `import` works. ⚠️ you have to copy them in to make them offline-first.

The Architectural Principles of Lopecode Core

So far we have discussed a selection of cool things Lopecode can do. But we have not explained what it is. Curiously none of those things are integral to the core of Lopecode. In this section we describe the guiding principles shaping the technical architecture.

> *"If a system is to serve the creative spirit, it must be entirely comprehensible to a single individual… Any barrier that exists between the user and some part of the system will eventually be a barrier to creative expression. Any part of the system that cannot be changed or that is not sufficiently general is a likely source of impediment."*

Design Principles Behind Smalltalk, `Byte` 1981 (Ingalls)

The runtime is the source of truth

Traditional programming has the programmer edit a source code, which is compiled to a program, which is then run. This is true even of the hosted Notebook environment on Observablehq.com.

Lopecode does not have a distinct concept of source code. Instead, it leverages Javascript's Function.prototype.toString() to decompile variables on the fly when source code is needed. Whatever is currently executing in the runtime, will be what is exported.

This is how Lopecode exports still work on the hosted Observable platform, despite no access to the hosted source code. The export scans the runtime programatically to create a serialisable representation.

Everything runs in userspace

For a system to be *truly* moldable. Everything must be changeable. Lopecode adopts a µ-kernel design where all features are added in userspace. For example, a minimal Lopecode file can be small as 48Kb.

es-module-shims@2.6.2 (20kb) and networking_script (7.7Kb)
ES-Module-Shims installs module loading hooks, allowing all Javascript module imports to be intercepted and the source rewritten. Lopecode leverages this to load modules from HTML content embedded in the source instead of over the network. It is the machinery that enables a single file to provide the illusion of networked module loading.

The custom `networking_script` adds a few more network orientated patches so that CommonJS modules and vanilla fetch also can be served with the impression of network connectivity.

@observablehq/inspector@5.0.1 (7Kb)
The observable inspector is an off-the-shelf module that Observable uses to render Javascript values as a DOM representation.

@observablehq/runtime@6.0.0 (6.5Kb)
The runtime is the reactivity dataflow engine developed by Observable. It is surprisingly small. It provides the programming module of variables and modules, and handles scheduling recomputations.

@tomlarkworthy/bootloader-min (3.5K)
The bootloader is a nominated userspace module that should run first. It is responsible for setting up the standard library and then loading the first real modules specified by the `bootconf.json`. The normal lopecode bootloader is much larger (>300kb) and includes the markdown renderer, katex, Plot etc.

The core starts an Observable Runtime engine and sets up the internal networking and loads nominated userspace modules. At the end of the boot sequence nothing will have been rendered to the DOM. All the initial machinery is without UI.

The notebook experience you are reading now happens after the Lopecode boot sequence. It is not what Lopecode is, more a demonstration of what is possible in userspace. We have chosen to recreate the Observable Notebook 1.0 experience, but it is not coupled to that. You can totally replace the UI for a radically different experience if desired.

What Lopecode provides is packaging Observable's reactive runtime with mechanisms to run it from a single file and a collection of important userspace modules. An important implication being anything is reactively hot-swappable at runtime.

The Lopecode file format and boot sequence is documented in exporter.

Realtime all the things

Lopecode is built under the philosophy that reactive state is just a better way of programming. The state should always be up to date, and instantly responsive to changes made anywhere else. The 40+ year appeal of spreadsheet is clear evidence. Lopecode is all-in on reactivity, and has extended the model to close expressivity gaps and adds missing engineering guardrails.

Lopecode lives reactivity, for example, meta-programming facilities like the currently loaded modules, is not a function you poll, but a variable `currentModules` that all interested tools subscribe to. Reactive testing allows tests to be run on causally linked implementation changes and surface breakages instantly. Technical documentation can be reactively linked to the real implementation.

Technical Contributions of Lopecode to Live programming

The Reactive Dataflow ensures code changes are applied immediately but selectively. We will not explain it in too much detail here, it is *prior art*. The Observable documentation, JavaScript in Observable↗, explains the programming model in detail. Observable notes contains additional technical experiments on the programming model.

Lopecode is a substantial reactive program and its development has revealed some limitations. Lopecode has added to the state-of-the-art with the addition of meta-programming and dataflow-templating, each which solves a different expressivity problem with reactive programming.

In addition, we have added additional engineering facilities that help with common development problems with reactive programming, Reactive Testing and Reactive Debugging.

Finally we have turbo-charged Lopecode development with a userspace AI that leverages the reactive microkernel architecture. Unlike external LLM harnesses, it executes inside the runtime alongside all the other userspace code and is able to read runtime state, the page DOM, as well read and modify code.

Meta-programming in Lopecode

For Lopecode to implement userspace code editors, it is necessary to obtain references to reactive variables and to patch them dynamically. The Runtime SDK is a library that other modules use to get meta-programming access to the running runtime.

Meta-programming is also used to create reactive visualizations, serialize the runtime, and create debuggers.

```
import {thisModule, runtime, lookupVariable} from "@tomlarkworthy/runtime-sdk"
```

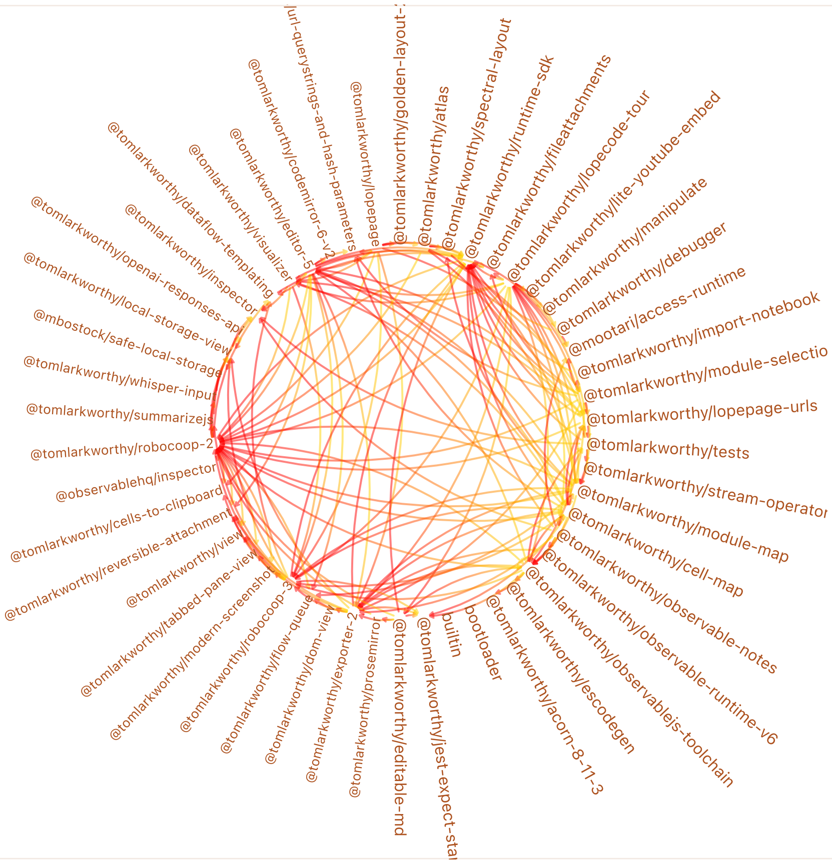
moduleMap

The current set of modules in the runtime is available as a reactive variable, so that any code can reactively update to changes in the current module listing. For instance, the plot below visualizes the *current* set of modules, and reacts to new modules being loaded into the runtime.

```
import {visualizeModules, viewof currentModules, currentModules} from "@tomlarkworthy/module-map'
```

```
▸Map(50) {Module => Object, Module => Object, Module => Object, Module => Object, Module => Obje
```

title	name	dependsOn
Tabbed Pane View	@tomlarkworthy/tabbed-pane-view	@tomlarkworthy/view
Reversible attachment	@tomlarkworthy/reversible-attachment	@tomlarkworthy/view
Save source to clipboard	@tomlarkworthy/cells-to-clipboard	
OpenAI Whisper Input	@tomlarkworthy/whisper-input	@tomlarkworthy/local-storage-v
Safe Local Storage	@mbostock/safe-local-storage	
Programmatic importNotebook	@tomlarkworthy/import-notebook	@tomlarkworthy/runtime-sdk
Observable Runtime (v6)	@tomlarkworthy/observable-runtime-v6	
Vendored CodeMirror 6 + Javascript	@tomlarkworthy/codemirror-6-v2	
Observable Inspector	@observablehq/inspector	
Atlas: Runtime Overview	@tomlarkworthy/atlas	@tomlarkworthy/runtime-sdk,@t
How the Observable Runtime works	@tomlarkworthy/observable-notes	@mootari/access-runtime,@tom



Replace the following cell with

```
import { boinger } from "@tomlarkworthy/svg-boinger"
```

to see the live modules update with the addition of svg-boinger.

```
replace_me = "change to `import { boinger } from \"@tomlarkworthy/svg-boinger\"`"
```

cellMap

A module holds reactive variables in a namespace. When decompiling, groups of variables form cells that are defined by a snippet of Observable JS source code. The `cellMap` can calculate that grouping from scanning a runtime i.e. the runtime is the source of truth.

As often you want to run `cellMap` on the enclosing runtime, `liveCellMap` is a reactive variable that holds the latest cell groupings by module.

```
▸ Map(50) {Module => Array(45), Module => Array(26), Module => Array(11), Module => Array(14), Mo
```

With meta-programming we are able to build software components that work over the whole of the runtime, for instance, creating module listings UIs and navigators. the `cell-map` module itself has a live visualization of all the cell inter-connections.

Dataflow Templating

Dataflow Templating solves an expressivity gap with static datagraph programming models like Spreadsheets and Observable Notebooks. To scale up Dataflow to complex programs like Lopecode itself we needed to go beyond static dataflow graphs.

Dataflow Templating is the missing generative abstraction for Dataflow that enables dynamic replication of dataflow subgraphs at runtime. Dataflow Templating allows replicating a static prototype subgraph to become a reusable abstraction. It has been heavily leveraged by the code editor module, so that every cell can have an independent code editor.

Learn more about the technique in Dataflow Templating and see a complex deployment in the Cell Editor

Reactive Testing

Dataflow builds a dependency graph for recomputing answers efficiently. This can be leveraged to rerun just the unit tests that are causally linked to implementation.

throw? ☐

name	state	value
@tomlarkworthy/lopecode-tour#test_has_it_thrown	✅	"pass"
@tomlarkworthy/lopepage#test_url_roundtrip	✅	true
@tomlarkworthy/robocoop-3#test_cell_map_coverage	✅	"all variables can be decompile

```
test_has_it_thrown = "pass"
```

Reactive Testing can be combined with meta-programming to check invariants are maintained. We pipe the results of reactive tests into the embedded AI so that it gets fast feedback.

One invariant we expose to the AI is that all cells should be decompilable. The low-level API the AI uses can potentially author code that can break decompilability. The Reactive Testing provides fast feedback to the AI which will then repair damage that it may temporarily cause. Reactive Testing thus becomes a useful feedback signal for both humans and AI.

Learn more about the technique in Reactive Testing.

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

circuit breaking

Another use of reactive testing is as a circuit breaker for self-triggering reactive loops. When a cell throws an exception, dataflow halts propagation. Reactive testing also represents failures as exceptions. Hidden loops in Dataflow graphs are dangerous, but are often needed for bidirectional sync with external state. Bidirectional sync is safe when a change in one way, maps back to the original (an invertible bijection), but often bijection invariants are fragile to develop.

One practical solution to remove the risk of a coding mistake in a bijection, is to only run sync if your proposed change will invert back to itself. This can be represented as an invariant test, which if it throws, highlights the issue in the invariant test and doesn't perform the sync that could potentially begin a self-triggering loop.

This safety technique was used in lopepage where the page URL is bidirectionally synced to the page layout. A test prevents writing to current layout the page URL if the inverse transformation leads to a different layout. This logic was notoriously hard to keep bug free.

Reactive Debugging

Dataflow has great developer ergonomics when the computation has a clear ordering. However, looped computation can be tricky. Direct cycles are forbidden, but you can introduce cycles via sideffects. Occasionally you can create self-triggering loops and sometimes it can be hard to understand why cells are recomputing.

The Reactive Debugger records all variable transitions on a timeline which can help trace unexpected sources of recomputation.

clear pause ☑ window (secs) 30

delay (secs) 0

```
import {viewof pause, pause, _ndd} from "@tomlarkworthy/debugger"
```

Programmable Agentic AI Harness

Unlike an exterior LLM harness like `Claude Code`, the `robocoop3` agent executes inside the runtime as a normal dataflow program. Its unique features is that it can **read program state** as well as write reactively executed code.

It distributed as a `function` so you can instantiate multiple independent agents.

`robocoop3` is agentic, and directly modifies the runtime using the low-level Javascript API. It is not exposed to the higher level Observable Javascript syntax. This contrasts to its predecessor `robocoop2` that reads and writes high-level Observable Javascript. Both are useful. `robocoop3` is much faster, with support for parallel tool calling and streaming responses, but `robocoop2` is able to write better high-level source code.

Learn more about the two coding assistants in their respective modules

Important Userspace Modules

Everything in Lopecode is implemented in userspace and thus is modifiable at runtime. Several modules add system features level features. You are free to add your own as needs demand.

Editor-5: Cell editor

The editor-5 module adds the `cellEditor`. It uses the popular Javascript library CodeMirror, which is vendored into the runtime as file attachment. The Editor uses two layers of Dataflow Templating so that every cell has a unique editor but the editor is lazily instantiated only when a user actually activates the edit feature.

Because the editor has a userspace implementation you can instantiate it in your own notebooks like a library. Consider the following editor that is synced with the reactive variable `my_value`:

☐ ⬆ ⬇ 🗑 ✚ ▶ 📋

```
my_value = 5
```

```
my_value = 5
```

`my_value` now has two editors. The one placed in the page, plus another cell editor all cells have. Because "Runtime-is-the-source-of-truth" both editors listen to the runtime to detect implementation changes, and thus are kept in sync.

```
my_value_variable = ▸Variable {ndd: true}
```

```
import {cellEditor} from "@tomlarkworthy/editor-5"
```

File Attachments: binary assets for offline-first operation

Modules support attaching binary assets through as files. This includes npm sourced Javascript modules. All of Lopecode's Javascript dependencies have been attached to modules, including the Observable runtime µ-kernel. This means an exported single file HTML file will open and fully operate without a network connection.

Learn more in the fileattachments.

The Lopecode Tour has vendored in Javascript dependencies for modern-screenshot, acorn, escodegen, Code mirror, Prose mirror, Jest expect, Observable runtime and Observable Inspector

FileAttachments are writable. This means that they can be used as a binary storage format. Changes are remembered after forking.

note

```
# Tom's Tasks

- write a demo for Lopecode
```

```
import {createFileAttachment, setFileAttachment} from "@tomlarkworthy/fileattachments"
```

```
attachment = ▸AbstractFile {name: "task.md", mimeType: "application/text"}
```

```
viewof tourModule = ▸EventTarget {tag: Symbol()}
```

Visualizer: projecting program state to the DOM

Reactive variables compute Javascript values. Javascript values have no inherent visual representation. The *visualizer* maps variable values to DOM nodes to present them in a notebook-like format.

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

```
options = ▸Map(50) {"bootloader" => Object, "builtin" => Object, "@tomlarkworthy/lopecode-tour"
```

You can pick a module to render, even one dynamically added. Note that DOM values can only have one parent, so using the visualizer here will often steal the DOM nodes from other pages. If you want the same UI in more than one place, this is where you elevate them to Dataflow Templates.

select a module to vizualize

@tomlarkworthy/summarizejs

```
219

@variable = Symbol(variable)

import {inspect, Inspector, src} from "@tomlarkworthy/inspector"

data = ▸Object {nested: Object, obj: Object, cls: class, instance: URL}

MouseEvent = ƒ()

summarizeJS = ƒ(…)

walked = "Object {nested: Object, obj: Object, cls: class, instance: URL}"

postProcess = ƒ(dom)

TEXT = 3

▸Array(18) ["Object {", "nested", ": ", "Object", ", ", "obj", ": ", "Object", ", ", "cls"

walk = ƒ(…)
```

Because the visualizer is a userspace module, we can change it on the fly, or supply a second mapping of program state. The visualizer leverages no special access to the runtime, it simply observes variable value changes with an API that available to every userspace service.

Lopecode core is not inherently a notebook environment, it is modules like `visualizer` that give it that look. It is possible for Lopecode to be both a notebook and something else, like a VR world, *at the same time*. Some of the other modules, like the debugger, are alternative projects on the lopecode core state.

Lopepage: a hypertext navigator over the runtime

Visualizer can render a single module as a notebook. *Lopepage* visualizes multiple modules on a docking UI, and allows internal navigation through hyperlinks. The layout of the page is synced to the URL, and serialized to notebook state, so you can easily pickup where you left.

All Notebooks in the Tour

The tour notebook contains a considerable volume of functionality, but serializes to 2MB. Here is the complete listing of modules contained within its single HTML file.

title	▴name
URL querystrings and hash parameters	@jashkenas/url-querystring

title	▴name
Safe Local Storage	@mbostock/safe-local-stor
Accessing a Notebook's Runtime	@mootari/access-runtime
Observable Inspector	@observablehq/inspector
Acorn@8.11.3	@tomlarkworthy/acorn-8-1
Atlas: Runtime Overview	@tomlarkworthy/atlas
cellMap	@tomlarkworthy/cell-map
Save source to clipboard	@tomlarkworthy/cells-to-cl
Vendored CodeMirror 6 + Javascript	@tomlarkworthy/codemirro
Dataflow Templating	@tomlarkworthy/dataflow-t
Debugger: Notebook Dataflow Debugger (ndd)	@tomlarkworthy/debugger
DOM view	@tomlarkworthy/dom-view
Inline editable md	@tomlarkworthy/editable-
Editor: Reactive Userspace Cell Mutator (v5)	@tomlarkworthy/editor-5
escodegen@2.1.0	@tomlarkworthy/escodege
Exporter 2: Single File Serializer	@tomlarkworthy/exporter-
Writable FileAttachments	@tomlarkworthy/fileattach
Convert cell computation to a Promise with cell flowQueue	@tomlarkworthy/flow-queu
golden-layout@2.6.0	@tomlarkworthy/golden-la
Programmatic importNotebook	@tomlarkworthy/import-no
@observablehq/inspector@5.0.1	@tomlarkworthy/inspector
jest-expect-standalone@24.0.2	@tomlarkworthy/jest-expe
lite-youtube-embed@0.3.3	@tomlarkworthy/lite-youtu
localStorageView: Non-invasive local persistance	@tomlarkworthy/local-stor
The Lopecode Tour	@tomlarkworthy/lopecode-
Lopepage	@tomlarkworthy/lopepage
lopepage urls	@tomlarkworthy/lopepage-
Direct Manipulation Plot	@tomlarkworthy/manipulat
modern-screenshot@4.6.6	@tomlarkworthy/modern-s
Module map	@tomlarkworthy/module-m
Explorer: Module Selector	@tomlarkworthy/module-s
How the Observable Runtime works	@tomlarkworthy/observabl
Observable Runtime (v6)	@tomlarkworthy/observabl
Bidirectional Observable JS <=> Runtime Toolchain	@tomlarkworthy/observabl
👋 Hello, OpenAI Responses API	@tomlarkworthy/openai-re
ProseMirror lib	@tomlarkworthy/prosemirr
Reversible attachment	@tomlarkworthy/reversible-
Roboco-op 2.1: Notebook collaborator	@tomlarkworthy/robocoop-
robocoop3: agents-as-functions	@tomlarkworthy/robocoop-
Runtime SDK	@tomlarkworthy/runtime-s
Minimize Crossings: Spectral Circular Graph for Layout	@tomlarkworthy/spectral-l
RxJS inspired stream operators for views	@tomlarkworthy/stream-op
SummarizeJS	@tomlarkworthy/summariz
Tabbed Pane View	@tomlarkworthy/tabbed-pa
Tests	@tomlarkworthy/tests
Composing viewofs with the view literal	@tomlarkworthy/view
Visualizer: Own Cell Renderer	@tomlarkworthy/visualizer

Work in progress

Lopecode is still quite fragile, it is being hardened and bugs trampled. Core development is on the Observable Notebook 1.0, in the Lopecode collection. That work (and many others) is synchronized to Github repository lopecode using Jumpgate.

Acknowledgements

Thank you *Mike Bostock* for building the Observable Runtime, d3, Plot and for seeing the future of a dataviz driven programming environment. Thanks *Philippe Rivière* for the outstanding work on Plot. Thanks *Fabian Iwand* for stretching what is possible on Observable Notebooks. Thanks *Thomas Ballinger*, *Tom MacWright*, *Jeremy Ashkenas*, *Toph Tucker*, *Visnu Pitiyanuvath*, *Allison Horst*, *Ian Johnson*, *Cobus Theunissen*, *Wayne Sutton* and many more for building Observable.

Thank you the Feeling of Computing community for feedback and support. *Jasmine Otto* for her positivity on the project. *Ivan Reese* for shepherding the Feeling of Computing community and providing space to Lopecode. Konrad Hinsen a kindred spirit making the lisp based Hyperdoc. Marek Rogalski building the mind blowing Automat. Chris Shank Orion Reed building folkjs and innovating new editable web technologies. All of these projects have been a source of ongoing inspiration.

Thanks to *Tomas Petricek* for introducing me to the computational *substrates* research discipline, which seems to fit what Lopecode is. Antranig Basman for connecting it to malleable systems and providing deep conversation about reactivity.

And Thank *YOU* for reading!

Content Helpers

```
aside = ƒ(title, module_names)
```

↗

```
externalLink = ƒ(…)
```

```
import {lite_youtube_css} from "@tomlarkworthy/lite-youtube-embed"
```