

This README is better viewed online (<https://model-view-self-modify.netlify.app/README.html>) with interactive iframes, than on [github](#).

You can also `git clone https://github.com/cben/model-view-self-modify` and serve locally by e.g. `python3 -m http.server` though current implementation won't load offline (I used CDNs).

what: Model |> View |> Self-Modify architecture

Representing user actions as source code modification is an under-explored approach to state management. I built a JS live coding environment to play with it. Here is a minimal example (<https://model-view-self-modify.netlify.app/?load=counter.js>):

The screenshot shows a live coding interface. On the left, a code editor displays a JavaScript file with numbered lines. Lines 1-18 show the creation of a counter model and its view. Lines 19-20 show the creation of multiple instances of the counter. The right side features a UI with a large number '2' and two buttons: 'increment' and 'decrement'. Below the UI, there are two sections: 'As a node:' and 'As object:', each with a 'Create counter' button and a 'type' dropdown menu set to 'button'.

```
1 var model = 0
2   + 1
3   + 1
4 var here = LINE_START(HERE())
5
6 const View = (model, where) => html`<div>
7   <h1>${model}</h1>
8   <button onclick={() => where.WRITE(' + 1\n')}>increment</button>
9   <button onclick={() => where.WRITE(' - 1\n')}>decrement</button>
10 </div>`
11 yield View(model, here)
12
13 var instancesHere = LINE_START(HERE())
14
15 const newInstance = `\
16 var model = 0
17 var here = LINE_START(HERE())
18 yield View(model, here)
```

1. Try clicking [increment] [decrement]. User actions `WRITE(...)` computation steps into the source, which is immediately re-run and UI is updated. (I'm using UPPERCASE names for source-handling helpers)
2. Click [Create counter], observe how now each counter can be inc/decremented separately.

✍️ To edit your own code(s) and persist after reload, open [without ?load= param](#); each `?id=...` you pick is independent.

I'm excited about it for 2 reasons:

1. Cognitive simplicity: It requires grokking only one concept of change for code & data evolution, and it doesn't force one above the other.
2. By implementing "[Always Already Programming](#)" literally, it is conducive to [blurring the boundaries](#) between language/env authors | developer | end-user.

There are obvious concerns including Δ security, scalability, and software updates. Yet if you want to build malleable, bi-directional, home-cooked, end-user-empowering experiences, I propose this is a fruitful starting point.

Focus: "Append-mostly" over in-place overwriting

A counter could also be implemented by over-writing `var model = 0` to become `= 1`, `= 2` and so on. Both are interesting and under-explored! I choose to focus on approaches that append "transcript(s)" of computation steps corresponding to user actions, because:

- Non-destructive, easier to clean up after shooting yourself in the foot.
- Capturing your actions in text is a gateway to programming [by demonstration].
- It smuggles advanced-but-somewhat-mainstream practices like MVU/redux and [event sourcing](#) into "muggle" hands 😺. It enables fun workflows notably live reload and [time traveling debugging](#).

Compare to "**Model-View-Update**" architecture popularized by [Elm](#) / [Redux](#):

app state is immutable ("Model" / "store"), View renders UI as a pure function of state,
user actions are [defunctionalized](#) into pure data e.g. { type: "rotateRight" },
and "Update" / "reducer" function dispatches on (action, old model) → to compute new model.

The difference here is we represent the same user intent as code e.g. `model = rotateRight(model)`, not data, and actually append them in designated place(s) in app code. (see "Where" section below)

Bi-directionality here is not some "magical" output->source inference, you write explicit UI code that generates source pieces. You can encapsulate it in components, not unlike React+Redux.

Prior Art: colorForth [magenta variables](#) are over-writable pointers into code. [James Vaughan](#) and [Jason McGhee](#) shared several projects that overwrite, which are also interesting for using Language Server Protocol to de-couple execution from a specific editor.

Typst aspires to be a better TeX, designed for fast incremental rendering. Typst creators used appending to make "interactive" games [icicle](#) & [badformer](#), where user types a sequence of WASD letters & document is re-rendered each time. Also picked up in community [soviet-matrix package](#)

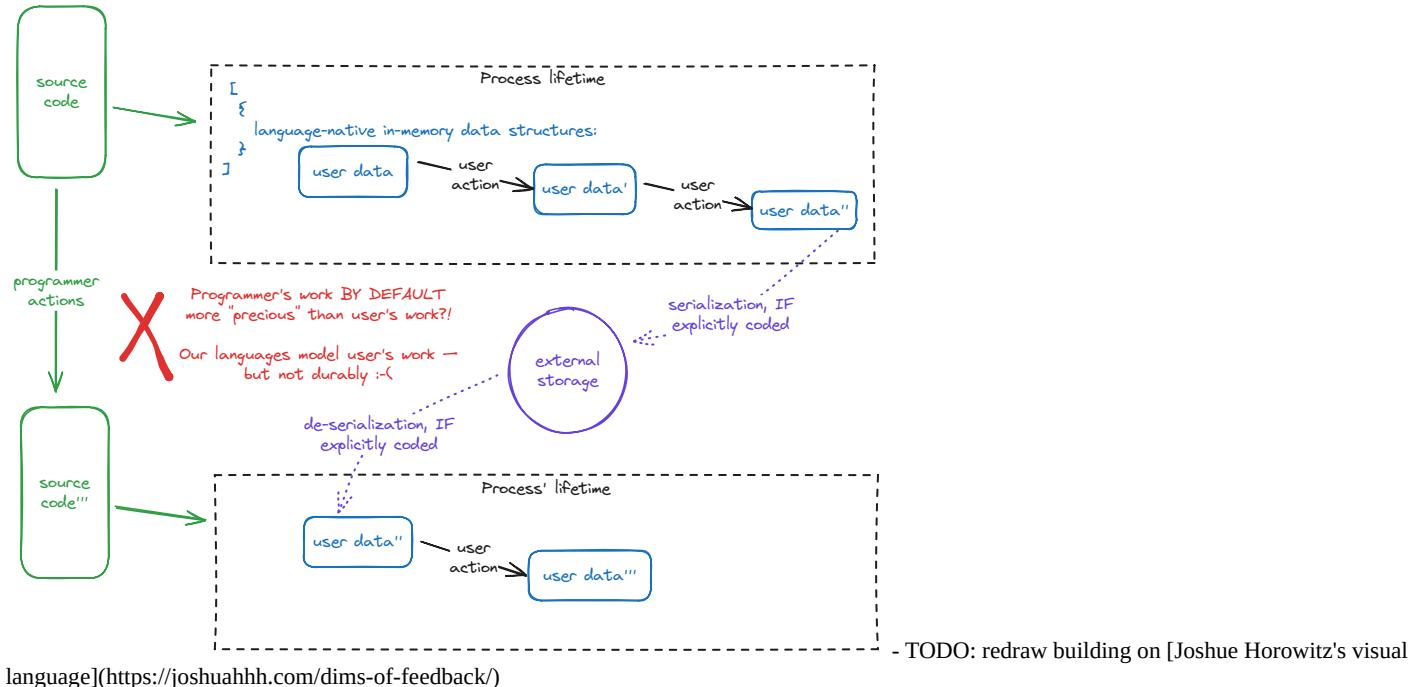
implementing Tetris.

I consider PhotoShop layers to be a grand success in showing the public they can be more productive manipulating a *recipe* than directly manipulating the final result. I've used several EDA software translating menu actions to statements in a Tcl console. Replayable Dockerfiles brought reproducibility to many more devs. I want more of these!

The core idea is so simple that I'm sure more people independently discovered it before, but I'm not aware of an agreed-upon term; I hope "Model View Self-Modify" name might stick by comparison to well-understood MVU?

Why (persistence): User's work deserves being 1st-class

Our languages encourage us to store user's work in runtime data structures (lists, dicts etc.) but when process dies or code changes, we discover developer's work was durable, but user's work is lost — and we need a whole other toolbag (file I/O, serialization, pointer swizzling, networked storage APIs, databases, ORMs...) to tackle that 😞.



That creates perverse incentive against fine-grained mixing of software use & modification/automation.

Even as experienced programmer running 100% open source OS, I'd rather keep my exact state as a user than use my superpowers if that means restarting the process! The contexts where I do mix them, routinely & fearlessly, are: (1) browser devtools to tweak layout/styling — even if those tweaks won't persist (2) shell prompt, where use is always already textual — and code is frequently disposable (yet retrievable from shell history).

LISPs, Smalltalk, Self famously put code & user's work on equal footing in a single persistent "image" of data structures.
Here I'm unifying in the other direction, storing both as textual code - this direction is *under-explored*!
Cf. also Jamie Brandon's [no strings on me](#) on runtime state vs. legibility tradeoffs.

Challenge: Schema evolution NOT solved, though cognitively flattened

[Cambria](#) and [Subtext](#) attack a hard problem:

For example, many live programming techniques treat state as ephemeral and recreate it after every edit, but when the shape of longer-lived state changes then the illusion of liveness is shattered – hot reloading works until it doesn't. — <https://arxiv.org/pdf/2412.06269.pdf>

I punt on that hard problem and expect user=dev resolve conflicts, just in a conceptually simple way.

Consider event-sourcing DB migration changing the format of past events, or a refactor changing Redux actions structure, invalidating the recorded history. Fixing those requires thinking of both "code change" and "action on data" concepts at once :-/ In Redux devtools, you could download the actions log as JSON, process, and load new actions; it's tedious and in my dev experience I used to just discard the log.

In this self-modify paradigm you get same issues — but *history is regular code*, so regular "debug / refactor after an API change" skills apply! (Including the option of making API backward-compatible)

Why: Reduce barriers between app "end user" / developer

First, note the live environment responsible for re-evaluating code upon every change and rendering the result is no longer a "dev tool" — it's now essential part of the app **runtime**.

(Distributing dev env to ALL users may feel weird in compiler circles, but is 100% normal in Excel circles.)

The source could be hidden by default, but it does give user some powers! First, undo/redo for free.

- Is time-travel debugging important for end-users? I think it varies by domain.
For example, if your "program" is algebraic chess notation, these were being published in journals for the sole purpose of "users" replaying

them step-by-step (on wooden boards — that language got standardized before computers were invented!) to look for "bugs" & "fixes" during "execution" 😊

Prior Art: Graphite.rs image editor has [language-centric architecture](#), IIUC any direct manipulation creates re-playable scene graph nodes that are exposed to user.

Why: Reduce barriers between app developer / IDE developer

If user-facing UI actually edits/inserts code, same skills translate to developer making mini-UIs for themself!

- TDD helpers: visualize pass/fail/rich results, buttons to JUMP() to test's code: <https://model-view-self-modify.netlify.app/?load=test-helpers.js>
- Help yourself to [Babylonian-style Programming](#) without hard-wired IDE support? Call a function, render the results. Write examples as part of the language, not special metadata.
- Literate/notebook helpers? Below in Tetris example, the code & outputs became long and I added H1(), H2() functions that render a large heading and sync cursor to source location.
- Level/asset editors. Below in Tetris example, I express the tetraminoes as arrays e.g.

```
[1,0], [1,1], [1,2], [1,3],
```

When rendering boards, I've wired all cells to (1) show coordinates (2) insert coordinates at cursor when clicked. This allowed me to "draw" the shapes by clicking.

Prior art: [livelits](#) render custom UIs inline in code.

Can we say here we have "poor man's livelits", only rendering side-by-side with code? Still useful.

TODO: This is an area I hope to explore more, e.g. [moldable inspectors](#), a GUI builder, number/color scrubbers...

Prior Art: [Mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks](#) prototyped a Jupyter extension that allows UI user actions to edit back the cell's code.

Why: internal/external DSL perspective

By admitting input, a program acquires a control language by which a user can guide the program through a maze of possibilities.
— [Chuck Moore](#) (for particular definition of "input")

The redux append-only log of user actions *is code*, in an *app-specific language*.

The pattern-matching we do in Update / reducer functions *is* an explicit interpreter for an "external DSL":

```
... onclick="dispatch({ type: 'rotateRight' })" ...  
  
switch (action.type) {  
  case 'rotateRight': ...
```

which adds ceremony & cognitive load.

The architecture I propose here is an "internal DSL" alternative. Supported actions are written as regular Model → Model functions; you chain them using regular function call syntax.

Prior Art: VisiCalc's original save format was [a series of keystrokes](#) replaying which would re-create the spreadsheet. That's more of an internal hack, [less suitable](#) as a stable language, and they did later introduce an [interchange format](#).

Converting user input to source code with descriptive function names is a step better. Long-term interchange is still tied to the host language and still depends on whether you can keep a stable "API".

Where: One vs. Multiple writable pointers into source

Purely functional MVU gravitates towards all actions forming a single history. Redux effectively does message-passing without a "receiver", with some benefits — and some modularity costs — compared to OOP.

But for me the overriding goal is *produced code should fit the user's mental model*! For example, if user is making moves in 2 games concurrently, do they want a single interleaved transcript ("Knight f3 on board 2"), or separate transcript of each game? Do they want global time-travel/undo, or separate for each game? (If separate, there is still editor's global Ctrl+Z.)

To support both, I extended the self-modification API with objects that represent locations in source code (CURSOR(), CALLER(), LINE_START(HERE()) etc.), each supporting editor actions (which currently consist of .WRITE(text) and .JUMP()). That's how the first example manages multiple counters!

(At this point I had to admit the architecture is not purely functional. Yes, *technically* you can lift all mutation out of *language* semantics into *IDE* — every change results in running a brand new program. But the system feel is of passing around handles to effectively mutable state, making their identity matter.)

Prior art: My attempts to google ideas like "purely functional self-modifying code" led nowhere, what with self-modifying code being shunned even in imperative circles for *being hard to reason about* :-)

However, Excel's surface layer is unidirectional dataflow (barring [cycles](#)). Turning a spreadsheet into "interactive app" may require macros, which can bind actions to editing cells & formulas. It's up to user whether they'd use a strict append-only log of actions, but either way Excel lets user fully edit the spreadsheet you got after invoking macros.

Putting it all together: Tetris

1. <https://model-view-self-modify.netlify.app/?load=tetris.js>

- TODO BUG: if you see `cmView` is not defined, edit the left side in any way

Source code:

```
272  var rotateRight = model => {
273    const { shape, board } = model
274    const newShape = [...model.shape.slice(1, 4), model.shape[0]]
275    // TODO: try other "wall kick" positions
276    const newPos = newShape[0]
277    if (intersectionRC(newPos, board).size == 0 &&
278        intersectionRC(newPos, fullBoard).size == newPos.size) {
279      return {
280        ...model,
281        shape: newShape,
282        lastMoveInvalid: false
283      }
284    }
285    return invalidMove(model)
286  }
287
288 // GAME HISTORY
289
290 model = newGame()
291
292 model = right(model)
293 model = right(model)
294 model = down(model)
295 model = down(model)
296 model = rotateRight(model)
297 here = LINE_START(HERE()) /* -- TIME TRAVEL: use Alt+Up / Alt+Down to move
298 model = rotateRight(model)
299 model = right(model)
300 model = down(model)
301 model = down(model)
```

Use `yield` ... and/or `return` ... statements. No JSX, instead use [html`<tag \\${...}>` notation](#). To move focus outside editor, press `F6` then `Esc`.

as a node.

[0,0]	[0,1]	[0,2]	[0,3]	[0,0]	[0,1]	[0,2]	[1,0]	[1,1]	[1,2]	[1,3]	[1,0]	[1,1]	[1,2]	[2,0]	[2,1]	[2,2]	[2,3]	[2,0]	[2,1]	[2,2]	[3,0]	[3,1]	[3,2]	[3,3]	[3,0]	[3,1]	[3,2]	[4,0]
[1,0]	[1,1]	[1,2]	[1,3]	[1,4]	[1,5]	[1,6]	[1,7]	[1,8]	[1,9]	[1,10]	[1,11]	[1,12]	[1,13]	[1,14]	[1,15]	[1,16]	[1,17]	[1,18]	[1,19]	[1,20]	[1,21]	[1,22]	[1,23]	[1,24]	[2,0]	[2,1]	[2,2]	[2,3]
[2,0]	[2,1]	[2,2]	[2,3]	[2,4]	[2,5]	[2,6]	[2,7]	[2,8]	[2,9]	[2,10]	[2,11]	[2,12]	[2,13]	[2,14]	[2,15]	[2,16]	[2,17]	[2,18]	[2,19]	[2,20]	[2,21]	[2,22]	[2,23]	[2,24]	[3,0]	[3,1]	[3,2]	[3,3]
[3,0]	[3,1]	[3,2]	[3,3]	[3,4]	[3,5]	[3,6]	[3,7]	[3,8]	[3,9]	[3,10]	[3,11]	[3,12]	[3,13]	[3,14]	[3,15]	[3,16]	[3,17]	[3,18]	[3,19]	[3,20]	[3,21]	[3,22]	[3,23]	[3,24]	[4,0]	[4,1]	[4,2]	[4,3]
[4,0]	[4,1]	[4,2]	[4,3]	[4,4]	[4,5]	[4,6]	[4,7]	[4,8]	[4,9]	[4,10]	[4,11]	[4,12]	[4,13]	[4,14]	[4,15]	[4,16]	[4,17]	[4,18]	[4,19]	[4,20]	[4,21]	[4,22]	[4,23]	[4,24]	[5,0]	[5,1]	[5,2]	[5,3]
[5,0]	[5,1]	[5,2]	[5,3]	[5,4]	[5,5]	[5,6]	[5,7]	[5,8]	[5,9]	[5,10]	[5,11]	[5,12]	[5,13]	[5,14]	[5,15]	[5,16]	[5,17]	[5,18]	[5,19]	[5,20]	[5,21]	[5,22]	[5,23]	[5,24]	[6,0]	[6,1]	[6,2]	[6,3]
[6,0]	[6,1]	[6,2]	[6,3]	[6,4]	[6,5]	[6,6]	[6,7]	[6,8]	[6,9]	[6,10]	[6,11]	[6,12]	[6,13]	[6,14]	[6,15]	[6,16]	[6,17]	[6,18]	[6,19]	[6,20]	[6,21]	[6,22]	[6,23]	[6,24]	[7,0]	[7,1]	[7,2]	[7,3]
[7,0]	[7,1]	[7,2]	[7,3]	[7,4]	[7,5]	[7,6]	[7,7]	[7,8]	[7,9]	[7,10]	[7,11]	[7,12]	[7,13]	[7,14]	[7,15]	[7,16]	[7,17]	[7,18]	[7,19]	[7,20]	[7,21]	[7,22]	[7,23]	[7,24]	[8,0]	[8,1]	[8,2]	[8,3]
[8,0]	[8,1]	[8,2]	[8,3]	[8,4]	[8,5]	[8,6]	[8,7]	[8,8]	[8,9]	[8,10]	[8,11]	[8,12]	[8,13]	[8,14]	[8,15]	[8,16]	[8,17]	[8,18]	[8,19]	[8,20]	[8,21]	[8,22]	[8,23]	[8,24]	[9,0]	[9,1]	[9,2]	[9,3]

As a node:

As object:

Game model

As a node: As object:

accacacaac ►Array(10) ["a", "c", "c"]

As a node: As object:

►Object {shape: Array(4),

As a node:

Score: 0

[0,0]	[0,1]	[0,2]	[0,3]	[0,4]	[0,5]	[0,6]	[0,7]	[0,8]	[0,9]	[0,10]	[0,11]	[0,12]	[0,13]	[0,14]	[0,15]	[0,16]	[0,17]	[0,18]	[0,19]	[0,20]	[0,21]	[0,22]	[0,23]	[0,24]	[1,0]	[1,1]	[1,2]	[1,3]
[1,0]	[1,1]	[1,2]	[1,3]	[1,4]	[1,5]	[1,6]	[1,7]	[1,8]	[1,9]	[1,10]	[1,11]	[1,12]	[1,13]	[1,14]	[1,15]	[1,16]	[1,17]	[1,18]	[1,19]	[1,20]	[1,21]	[1,22]	[1,23]	[1,24]	[2,0]	[2,1]	[2,2]	[2,3]
[2,0]	[2,1]	[2,2]	[2,3]	[2,4]	[2,5]	[2,6]	[2,7]	[2,8]	[2,9]	[2,10]	[2,11]	[2,12]	[2,13]	[2,14]	[2,15]	[2,16]	[2,17]	[2,18]	[2,19]	[2,20]	[2,21]	[2,22]	[2,23]	[2,24]	[3,0]	[3,1]	[3,2]	[3,3]
[3,0]	[3,1]	[3,2]	[3,3]	[3,4]	[3,5]	[3,6]	[3,7]	[3,8]	[3,9]	[3,10]	[3,11]	[3,12]	[3,13]	[3,14]	[3,15]	[3,16]	[3,17]	[3,18]	[3,19]	[3,20]	[3,21]	[3,22]	[3,23]	[3,24]	[4,0]	[4,1]	[4,2]	[4,3]
[4,0]	[4,1]	[4,2]	[4,3]	[4,4]	[4,5]	[4,6]	[4,7]	[4,8]	[4,9]	[4,10]	[4,11]	[4,12]	[4,13]	[4,14]	[4,15]	[4,16]	[4,17]	[4,18]	[4,19]	[4,20]	[4,21]	[4,22]	[4,23]	[4,24]	[5,0]	[5,1]	[5,2]	[5,3]
[5,0]	[5,1]	[5,2]	[5,3]	[5,4]	[5,5]	[5,6]	[5,7]	[5,8]	[5,9]	[5,10]	[5,11]	[5,12]	[5,13]	[5,14]	[5,15]	[5,16]	[5,17]	[5,18]	[5,19]	[5,20]	[5,21]	[5,22]	[5,23]	[5,24]	[6,0]	[6,1]	[6,2]	[6,3]
[6,0]	[6,1]	[6,2]	[6,3]	[6,4]	[6,5]	[6,6]	[6,7]	[6,8]	[6,9]	[6,10]	[6,11]	[6,12]	[6,13]	[6,14]	[6,15]	[6,16]	[6,17]	[6,18]	[6,19]	[6,20]	[6,21]	[6,22]	[6,23]	[6,24]	[7,0]	[7,1]	[7,2]	[7,3]
[7,0]	[7,1]	[7,2]	[7,3]	[7,4]	[7,5]	[7,6]	[7,7]	[7,8]	[7,9]	[7,10]	[7,11]	[7,12]	[7,13]	[7,14]	[7,15]	[7,16]	[7,17]	[7,18]	[7,19]	[7,20]	[7,21]	[7,22]	[7,23]	[7,24]	[8,0]	[8,1]	[8,2]	[8,3]
[8,0]	[8,1]	[8,2]	[8,3]	[8,4]	[8,5]	[8,6]	[8,7]	[8,8]	[8,9]	[8,10]	[8,11]	[8,12]	[8,13]	[8,14]	[8,15]	[8,16]	[8,17]	[8,18]	[8,19]	[8,20]	[8,21]	[8,22]	[8,23]	[8,24]	[9,0]	[9,1]	[9,2]	[9,3]

2. Scroll both sides to bottom to see tetris game. Click source line opening "TIME TRAVEL" comment and try moving it up and down.

3. Click [rotateR] [left] [right] [down] buttons to play from that moment.

4. Put cursor inside `RCSet([...])` in `newGame` function. Start clicking board cells to mark them occupied.

⌚ If you want to edit freely, drop the `?load=...` from URL, otherwise your edits get overwritten on reload. You can append different `?id=...` to keep separate projects in browser localStorage.

Challenge: O(n²) slowdown

The longer you interact, the longer the code gets, and UI responsiveness will degrade! This may be bearable for "turn-based" apps and much worse for real-time games.

I'm hopeful incremental computation can help. Don't re-run code from start, especially when appending near the end. Feasibility of accurate dependency analysis depends on the language...

- TODO: Perhaps it could be helped by user-provided dataflow structure, e.g. treating functions or notebook cells as separate editors? Building on the Observablehq runtime, or Marimo could be nice.

This might also form a middle ground between single append point and arbitrary pointers — only append at end of a function/cell/file?

Some granularity is also interesting for receiving updated software and "opening" your existing data with it...

Challenge: Multi-player / security

Since both logic & user actions are stored in same text form, it's tempting to sync it by CRDT and gain distributed state *for free*.

I want to try it, but it may well be a dead end. In particular, the free-form source makes it **impractical to enforce any kinds of permissions**; to interact you need permission to edit, and if you can edit you can cheat.

Even in single-user setting injecting data as code is bug-prone. TODO: My current `WRITE('string')` helper is risky, should add safe parametrization like in good DB query-building APIs.