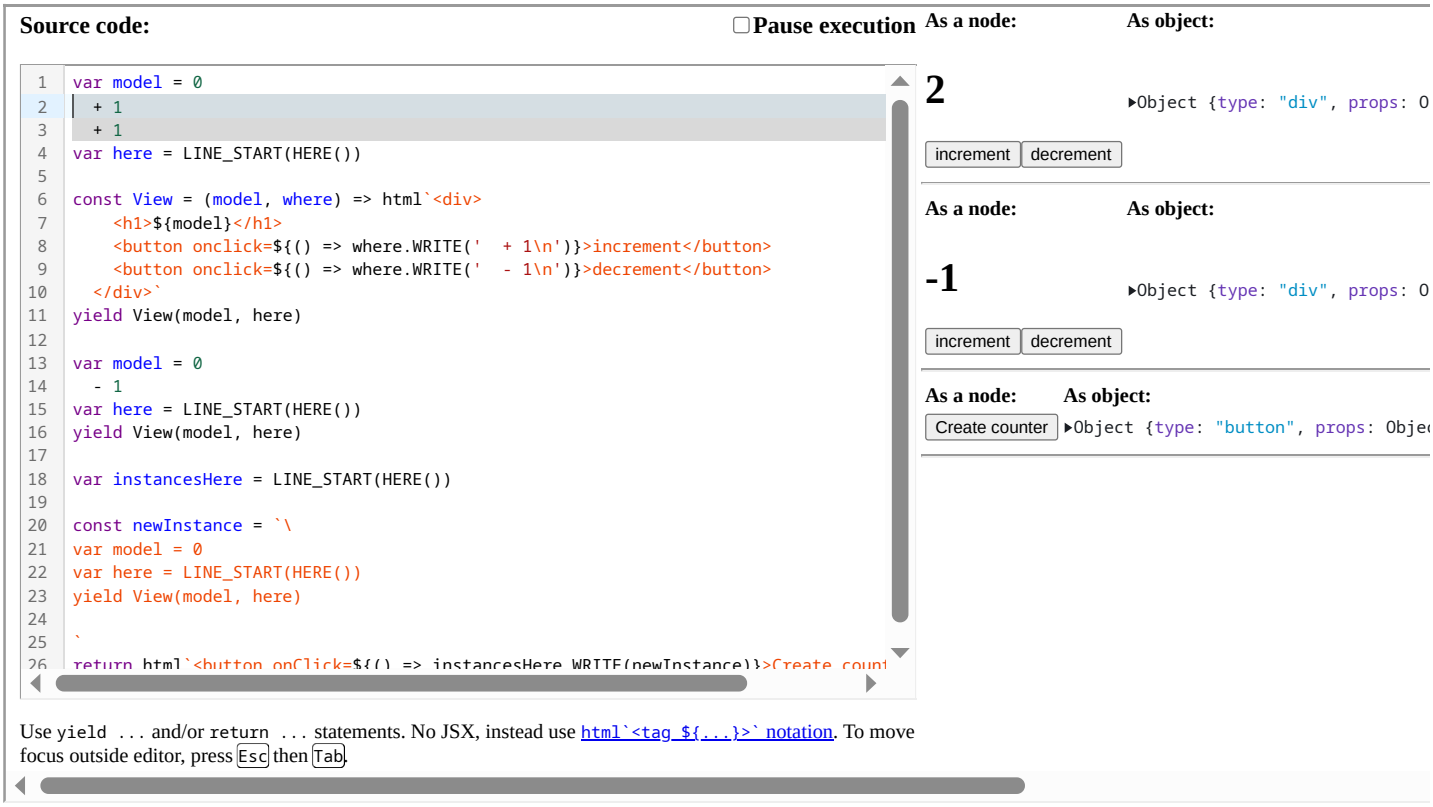
This is version I presented at Substrates 2026 workshop, plus some updates per reviewer feedback, especially sections 9–10.

Best viewed with interactive iframes, online (https://cben.github.io/model-view-self-modify/substrates-2026.html) or locally:

```
git clone https://github.com/cben/model-view-self-modify
cd model-view-self-modify
git checkout substrates-2026
python3 -m http.server  # or any other static server
```

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

Representing user actions as source code modification is an under-explored approach to state management.
In the current naive form the idea is roughly **language-agnostic** I built a JavaScript live coding environment to play with it. (But extending it would depend more, see sections 9—11.)
Here is a minimal example (https://cben.github.io/model-view-self-modify/substrates-2026/editor.html?load=counter.js):



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" [Hoff2020] literally, it is conductive to blurring the boundaries [Horowitz2026] between language/env authors | developer | end-user.

There are obvious concerns including ⚠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.

Prior art links are spread through the sections, trying to focus on relevant aspects. 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 Model-View-Update architecture?

1. 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 appoarches 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 [Fowler2005] into "muggle" hands 😈. It enables fun workflows notably live reload and time traveling debugging [James2014] by replaying action log.

Q: Will this architecture lead to a sane structure of code?
Hard to say as I only used it in toy examples, but I draw confidence from significant similarity to **"Model-View-Update" architecture** popularized by Elm [Poudel2021] and Redux [Redux2024]. The core of MVU is:
app state is immutable ("Model" / "store"), View renders UI as a pure function of state,
user actions are defunctionalized [Koppel2019] 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 sections 6–7 for deeper comparison)
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.

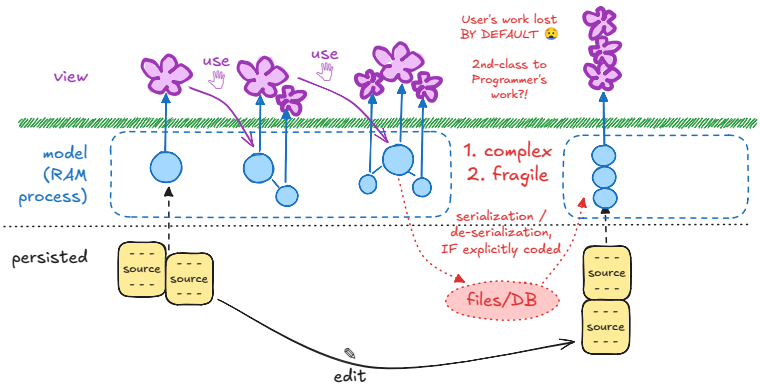
Prior Art: colorForth magenta variables [Oakford2003] are over-writable pointers into code. This was lower level (can store 1 machine word unless you start doing pointer arithmetic?) but allowed persisting (some) state in the source.

[Vaughan2025] and [McGhee2025] 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 [icicle2023] & [badformer2023], where user types a sequence of WASD letters & document is re-rendered each time. Also picked up in community [soviet-matrix2024] implementating Tetris. These are an almost exact implementation of Model-View-Self-modify! (except Typst has no concept of UI action binding, so they had to use single letters)

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

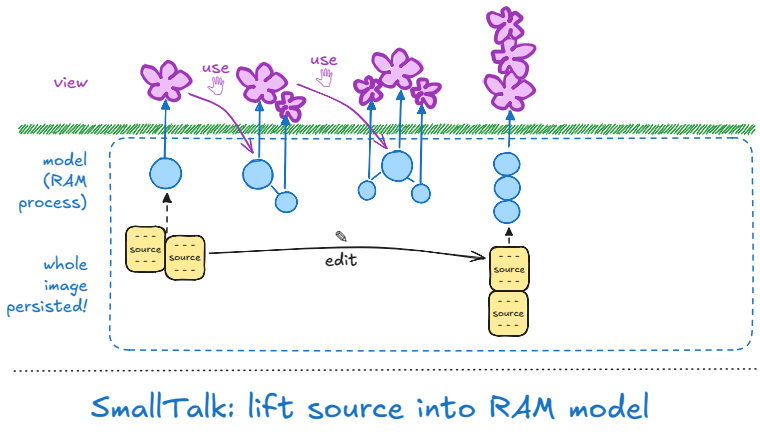
Our languages encourage us to store user's work in runtime data structures (lists, dicts etc.) but when process dies or code changes, we discover developer's work was durable, but user's work is lost — and we need a whole other toolbag (file I/O, serialization, pointer swizzling, networked storage APIs,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



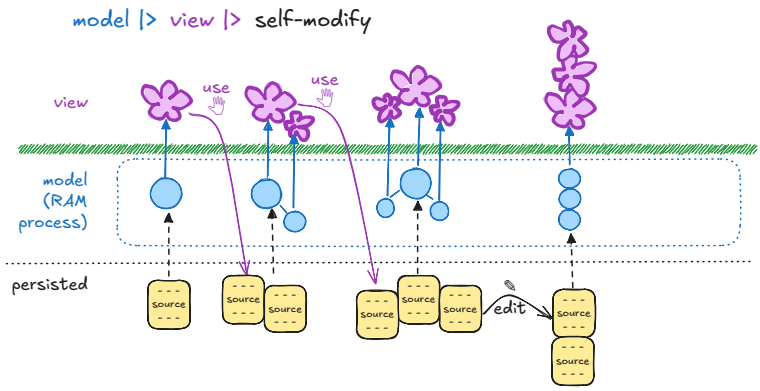
(I'm building on "Dimensions of Feedback" visual language [Horowitz2024] but splitting runtime state arising from source from user-visible feedback.)

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 lift code into runtime data structures, on equal footing with user's work (allowing orthogonal persistence as a single "image"):



Here I'm unifying in the other direction, lowering both to textual code (**the code is the substrate**) - this direction is *under-explored*!



Cf. also [Brandon2023] on runtime code modification vs. legibility tradeoffs (his takeaway is keeping source/data separation & mitigating issues by specific language design; but the frank discussion of cognitive difficulties understanding live systems is rare and interesting).

3. Challenge: Schema evolution NOT solved, though cognitively flattened

Cambria [LHH2023] and Subtext [Edwards2025] 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. [EPSL2024] (emphasis mine)

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

Saving a log of intent-ful actions does prepare us better for schema changes than just storing current state would.

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)

4. 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: 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.
Graphite.rs image editor doubles down on a language-centric architecture [Graphite2025], IIUC any direct manipulation creates re-playable scene graph nodes that are exposed to user (as a structured editor; a textual form exists but is less user-facing).

5. 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://cben.github.io/model-view-self-modify/substrates-2026/editor.html?load=test-helpers.js
- *Help yourself* to Babylonian-style Programming [RRRLH2019] 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 [OMBVCC2021] 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 [KRHMWP2020] prototyped a Jupyter extension that allows UI user actions to edit back the cell's code. Unfortunately their code doesn't seem published(?), but the goal was specifically helping "tool builders" to enrich Jupyter with UI->code tools. (Simple usage appends user actions, though they include a more advanced API for overwriting.)
However, they approach code->UI sync by the tool *parsing* cell code; this is less general, and implies some separation of "tool" code — whereas I celebrate ability of arbitrary "user-land" code to generate UI.

6. 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. [Moore2011] (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 MVU update/reducer functions *is* an explicit interpreter for an "external DSL":

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

switch (action.type) {
   case 'rotateRight': ...
```

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.

The reduction in ceremony is pleasant to me but minor; I'm more interested in this shift because it makes UI actions & programming *look* similar, and encourages liberal inter-mixing (as internal DSLs tend to do).

It also reveals a guiding principle for building with this architecture: Beside black-box functionality, assume user will read&edit the produced code! => **Optimize APIs for making sense to the user**. E.g. is `rotateRight(model)` a good function? Yes *iff* it matches how the user would call it. Hmm perhaps `rotateRight(game)` might be better 🤔.

7. 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!

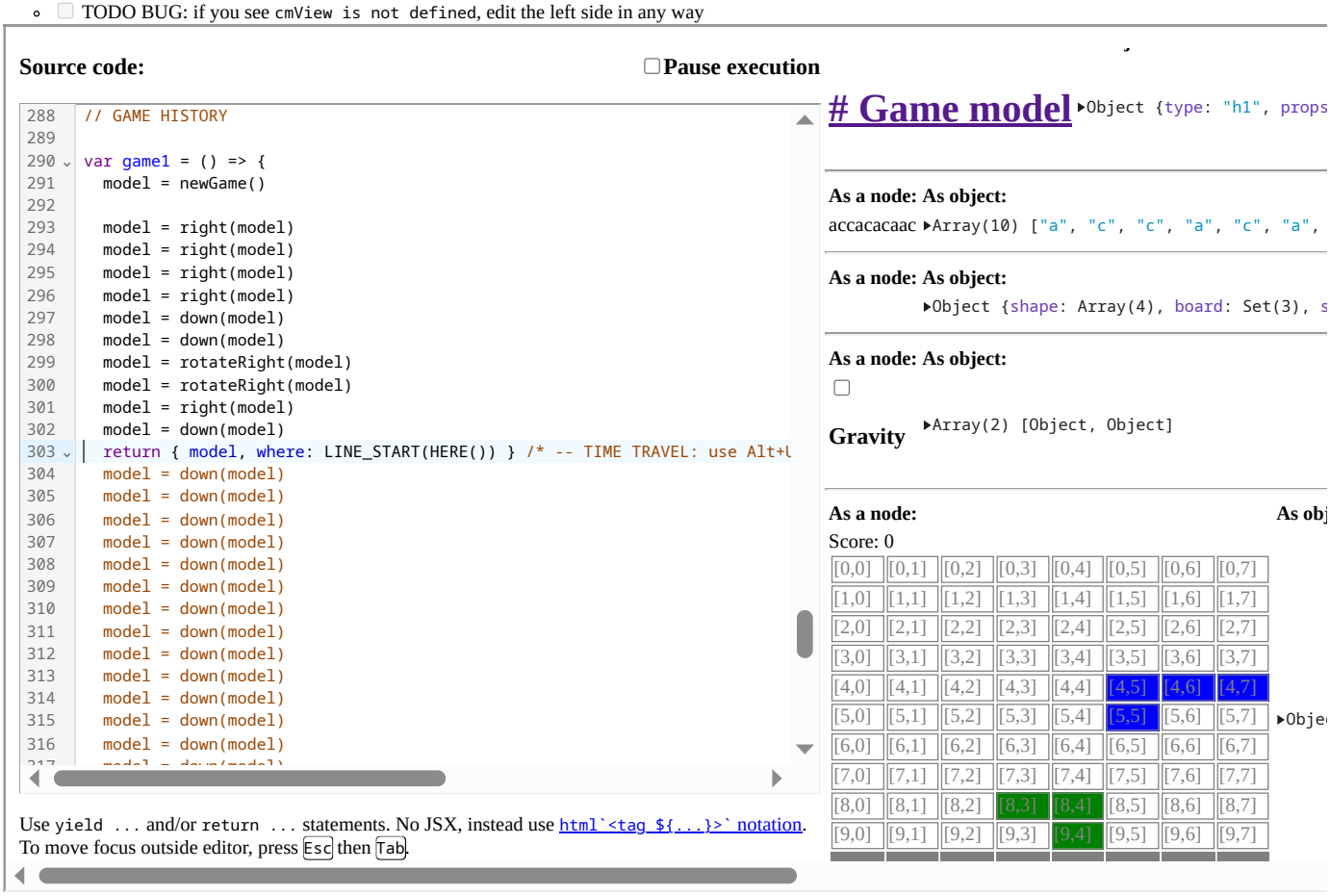
- ☐ The helpers are still geared for inserting, not yet fine-grained enough to target specific parts of a line for in-place overwriting.

(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 indentity 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 [Inkbox2024]). 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.

8. Putting it all together: Tetris

1. https://cben.github.io/model-view-self-modify/substrates-2026/editor.html?load=tetris.js
   - ☐ TODO BUG: if you see `cmView is not defined`, edit the left side in any way



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

   (Initially I thought time travel will be a feature of the IDE, stopping execution early. But the execution I want cut short is just deriving the model; the View logic should still run *after* that, and the IDE doesn't know which code does what. Hence, the cheesy user-space implementation with a comment and/or early `return` statement.)

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

4. Enable "Gravity" checkbox above the board to enable pieces descending after 1 second if you make no move.

5. Put cursor inside `RCSet([...])` in `newGame` function, or above in `shapes.I`, `shapes.J` etc. definition. 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.

[The following sections discuss weak points with **speculation on future work**.]

9. Challenge: O(n²) slowdown! Incremental computation?

Well, O(n) per interaction. 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.

As reviewers note, this is the cost of my preferences (see section 1) to append actions; overwriting state in-place would scale better.

I'm hopeful incremental computation could help. Don't re-run code from start, especially when appending near the end. (Snapshotting derived state is standard practice in long-lived event-sourcing systems, achieving OK performance while still allowing retroactive replay when needed.)

Feasibility & accuracy of dependency analisys **depends on the language**!

- Generally, immutable data structures are safer for snapshotting & analysis than imperative side effects. My Tetris example is mostly functional, non-mutating — but there is a big difference for analysis between what a functional language could *guarantee* vs. self-imposed discipline.
- 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...

Purity Q: Suppose a language where you *can* fool the optimizations, are expected not to, and have a manual "rerun from start" button — could that be a good-enough experience?
Based on my Jupyter experience, I feel it'd miss a lot of the cognitive simplicity this architecture aspires to 😴. OTOH both Observable & Marimo are foolable, yet come much closer to "magic" in an imperative language than Jupyter does, so perhaps?

10. Challenge: Determinism of external/spontaneous events

The whole idea of re-running code frequently from scratch is a big **bet on deterministic, reproducible execution**. Feasibility may vary by problem domain, and by language/system APIs...

Purity Q: Snapshotting & incremental computation might mitigate this a bit(?) Arguably, Dockerfiles became popular precisely because they took a pragmatic "up to you" stance to determinism + covered it up with image snapshotting. However, reproducibility is much more crucial when your code may re-run every second than rebuilding an image monthly...

- Already with Tetris, I met question how to randomly select incoming shapes 🎲. As an example of poor language & stdlib fit, `Math.random()` is unacceptable — JS gives no way to seed it, and if you replay same moves with different shapes you can get entirely different positions!
- The next improvement was to find a seedable PRNG library, and carry its state in the model. [soviet-matrix2024] does that too.
  Pretty OK for Tetris. But imagine a game where user actions may affect how many times RNG is advanced — then if you *time-travel* and change the past, you risk mixing up the future.
- The more robust approach would be to write back the specific choices made into the source!
  Writing has to be guarded so it's self-inhibitting to prevent infinite loops, e.g. change `down(model)` into something like `down(model, { nextShape: T })` which would use the recorded value.

One of the reviewers asked about Tetris shape descending down on a **timer**. I made it work, by spontaneously appending `model = down(model)` lines, same as if user would press it. Some lessons learnt:

1. Env must NOT move user's cursor to where new source got inserted! If it steals it, user may get no chance to edit/delete such auto-acting code. Without stealing, mixing user actions with spontaneous edits work quite well (though I'm far from stressing them)! Note that each `WRITE()` is atomic, so we don't suffer from character-level merge conflicts.
2. Many ways to trigger infinite loops => added both internal "[ ] Gravity" checkbox and IDE "[ ] Pause execution" to help recover.
3. Closures on a timer may refer to outdated text locations that shifted since... I improved helpers like `LINE_START()` return declarative objects that only resolve exact location when executing `.WRITE(...)` / `.JUMP()`.
4. Confusingly, you want `setTimeout()` not `setInterval()` to avoid an avalanche. The timeout handler executes once => modifies source => source runs again => installs a new handler.
5. I wanted every code run (e.g. due to manual action by user) to cancel any outstanding timer, but doing that required stashing its handle in global state, breaking out of the simple model that every run is isolated:

```
if (window.activeTimeout) { // KLUDGE: from a *previous* run!
  clearTimeout(window.activeTimeout)
  window.activeTimeout = undefined
}
if (gravity) {
  // KLUDGE: Persist for next run
  window.activeTimeout = setTimeout(() => { here.WRITE(`  model = down(model)\n`) }, 1000)
}
```

⚠ As reviewer 2 notes, these are difficulties wrt. environmental state (DOM handles, OS handles, closures...) and indeed SmallTalk-style object graphs handle these easier than flat text.
I think the difficulty does depend on particular API styles, generally functional passive-data one may be an easier fit than OOP-style wrappers whose identity matters? But that's speculation.

Q: What about external inputs, e.g. responses to **network requests**? A popular approach with Redux is stuffing the results into redux state, so a corresponding naive approach here is to write the results into the source code!

- Again, the write'd have to be self-inhibiting: if unknown make network request, once written just use that.
- Responses may be impractically large.

11. 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.

References

1. [Hoff2020] Melanie Hoff, *Always Already Programming* (https://gist.github.com/melaniehoff/95ca90df7ca47761dc3d3d58fead22d4) ↩
2. [Horowitz2026] Joshua Horowitz, *The Blurry Boundaries Between Programming and Direct Use* (https://joshuahhh.com/paper-plateau-2026-blurry/), PLATEAU 2026 workshop ↩
3. [Fowler2005] Martin Fowler, *Event Sourcing* (https://martinfowler.com/eaaDev/EventSourcing.html) ↩
4. [James2014] Michael James, *Time Travel made Easy — Introducing Elm Reactor* (https://elm-lang.org/news/time-travel-made-easy) ↩
5. [Poudel2021] Pawan Poudel, *Beginning Elm*, section 5.2 *Model View Update - Part 1* (https://elmprogramming.com/model-view-update-part-1.html) ↩
6. [Redux2024] Mark Erikson & contributors, *Redux Fundamentals, Part 2: Concepts and Data Flow* (https://redux.js.org/tutorials/fundamentals/part-2-concepts-data-flow) ↩
7. [Koppel2019] Jimmy Koppel, *The Best Refactoring You've Never Heard Of* (https://www.pathsensitive.com/2019/07/the-best-refactoring-youve-never-heard.html), Compose 2019 talk ↩
8. [Oakford2003] Howerd Oakford, The colorForth Magenta Variable (http://www.euroforth.org/ef03/oakford03.pdf), EuroForth 2003 ↩
9. [Vaughan2025] James Vaughan, *Code⇄GUI bidirectional editing via LSP* (https://jamesbvaughan.com/bidirectional-editing/) ↩
10. [McGhee2025] Jason McGhee, Hacker News thread (https://news.ycombinator.com/item?id=44437770) ↩
11. [icicle2023] *Help the Typst Guys reach the helicopter pad and save Christmas!* game (https://typst.app/universe/package/icicle/) ↩
12. [badformer2023] *Retro-gaming in Typst. Reach the goal and complete the mission.* game (https://typst.app/universe/package/badformer/) ↩
13. [soviet-matrix2024] YouXam & contributors, *Tetris game in Typst* (https://github.com/YouXam/soviet-matrix) ↩
14. [Horowitz2024] Joshua Horowitz, Technical Dimensions of Feedback in Live Programming Systems (https://joshuahhh.com/dims-of-feedback/), LIVE 2024 ↩
15. Jamie Brandon, *no strings on me* (https://www.scattered-thoughts.net/writing/there-are-no-strings-on-me/) ↩
16. [LHH2023] Litt, Hardenberg, Henry, *Project Cambria — Translate your data with lenses* (https://www.inkandswitch.com/cambria/) ↩
17. [Edwards2025] *Subtext Retrospective*, summarizing 2004–2020 research (https://www.subtext-lang.org/retrospective.html) ↩
18. [EPSL2024] Edwards, Petricek, Storm, Litt, *Schema Evolution in Interactive Programming Systems* (https://arxiv.org/pdf/2412.06269) ↩
19. [Graphite2025] Chambers, Kobert, *Rust-Powered Graphics Editor: How Graphite's Syntax Trees Revolutionize Image Editing* (https://www.youtube.com/watch?v=ZUbcwUC5lxA), interview on Developer Voices podcast ↩
20. [RRRLH2019] Rauch, Rein, Ramson, Lincke, Hirschfeld, *Babylonian-style Programming: Design and Implementation of an Integration of Live Examples into General-purpose Source Code* (https://arxiv.org/abs/1902.00549) ↩
21. [OMBVCC2021] Omar, Moon, Blinn, Voysey, Collins, Chugh. *Filling Typed Holes with Live GUIs* (https://hazel.org/papers/livelits-pldi2021.pdf), PLDI 2021 ↩
22. [KRHMWP2020] Kery, Ren, Hohman, Moritz, Wongsuphasawat, Patel, *mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks* (https://dl.acm.org/doi/abs/10.1145/3379337.3415842), UIST 2020 ↩
23. [Moore2011] Chuck Moore, *Programming a Problem-Oriented Language* section 1.2 (http://forth.org/POL.pdf) ↩
24. [Inkbox2024] Inkbox software, *I built my own 16-Bit CPU in Excel* video (https://youtu.be/5rg7xvTJ8SU?t=91) ↩