Live Programming in Hostile Territory

Orion Reed
me@orionreed.com

Chris Shank
chris.shank.23@gmail.com

"[People] make their own history, but they do not make it as they please; they do not make it under self-selected circumstances, but under circumstances existing already, given and transmitted from the past."

— Karl Marx

Abstract

Live programming research gravitates towards the creation of isolated environments whose success is measured by domination: achieving adoption by displacing rather than integrating with existing tools and practices. To counter this tendency, we advocate that live programming research broaden its purview from the creation of new environments to the augmenting of existing ones and, through a selection of prototypes, explore three *adversarial strategies* for introducing programmatic capabilities into existing environments which are unfriendly or antagonistic to modification. We discuss how these strategies might promote more pluralistic futures and avoid aggregation into siloed platforms.

1. Introduction

Live programming research is broadly concerned with the creation of programming systems which provide immediate feedback on the dynamic behavior of a program even while running [1]. This promise of immediate feedback requires the ability to modify, inspect, and manipulate programs while they execute —capabilities that established programming environments, designed around edit-compile-run cycles, cannot reliably provide. We believe this fundamental mismatch drives live programming research to face inwards, towards the *creation of fully circumscribed universes* — often viewed as the most pragmatic means to ensure the runtime malleability that liveness requires. This inward focus produces systems which can be operated on from within themselves, but neglect their participation in wider contexts of use [2], encouraging what Kell describes as a success-by-domination strategy [3] where systems achieve adoption by displacing rather than integrating with existing tools and practices.

Whereas traditional programming leverages ubiquitous plaintext infrastructures that resist single-system dominance through their simplicity and interoperability [4], live programming's visual requirements largely preclude utilizing this pluralistic foundation. This threatens to shift the experience of programming into one mediated through siloed platforms, losing the freedom and plurality that plaintext infrastructures provide. Rather than accept this trajectory, we advocate for this community to extend its research from the creation of new environments to the augmenting of existing ones, situating new systems in their present contexts of use.

We explore three strategies for live programming in 'hostile territory'—environments that are unfriendly or antagonistic to modification. Central to these strategies is *free addressability*—a property we argue is essential for augmenting systems without requiring cooperation from their original creators. We demonstrate, through a selection of prototypes from the *folkjs* research project [5], how we can exploit the addressable surfaces of user interfaces to situate new affordances in environments that were never designed to accommodate them. These interventions are not ends in themselves, but create fragile bridges that demonstrate the potential of more robust infrastructure and, by setting expectations of interoperability, make it harder to retreat into isolation.

2. Free Addressability

The practice of information hiding—originally advocated by Parnas to support "centralized management process for large, disconnected teams" [6]—creates challenges for software evolution, particularly in contexts where multiple authors work across organizational boundaries rather than within coordinated teams. As Ostermann et al. observe, it is unclear how to decide up-front which design decisions should be hidden versus exposed, and software evolution often brings new stakeholders who need access to previously hidden information [7]. This results in what Basman et al. call "hermetic" systems—isolated environments that "give insufficient consideration to what lies outside the system" [8]. While information hiding serves its intended purpose within coordinated development teams, contemporary software ecosys-

tems increasingly demand cross-system integration and external extensibility. Our approaches require reaching into systems whose internal components these design choices obscure.

We believe *free addressability*—a term we adopt from Basman et al. [8]—is key to enabling outward-facing integration and moving beyond success-by-domination strategies. Free addressability embraces transparent, publicly addressable state through queries, selectors, names, or other mechanisms that make parts of a running system targetable from the outside without requiring permission or coordination from the original creators, seeking to "maximally advertise the structure of the application via a transparent addressing scheme" [8].

Our adversarial strategies exploit the fact that user interfaces often expose more addressable surfaces than the underlying program—through DOM elements, accessibility trees, and visual components. This disparity creates crucial leverage points for live programming interventions, allowing us to exploit addressability where it exists and demonstrating where additional addressability would be beneficial. These addressable surfaces provide the basis for working in hostile territory by offering ways to situate live programming capabilities within environments that were never designed to accommodate them.

3. Strategies

Our strategies draw inspiration from what Doctorow calls *"adversarial interoperability"* - interfacing with systems without the permission of their original creators [9]. By exploiting the addressable surfaces of user interfaces, we can introduce live programming capabilities through three distinct approaches that work around, rather than require, system cooperation.

We explore three approaches that differ in their relationship between *system* and *environment*:

1. **Annotating** existing environments with new affordances
2. **Embedding** systems into heterogenous host environments
3. **Extending** closed systems through re-appropriation of available addressing schemes.

3.1. Adversarial Annotation

Adversarial annotation challenges the assumption that live programming requires purpose-built universes, making it possible to embed new affordances where people already work. Rather than creating destinations for users to visit, annotation distributes programming capabilities as lightweight augmentations that attach to existing structure—demonstrating that environments are not the only path to liveness.

While web-based systems often break when their DOM tree structure is modified, they often tolerate the addition of *new* attributes that encode interactive functionality. This tolerance creates one path for escaping isolated environments — annotations can introduce liveness without requiring users to abandon their tools or migrate their work.

Our first prototype demonstrates how we can annotate regions of text with a custom HTML attribute that binds a Language Server Protocol (LSP) server—a standardized interface for providing language-specific programming assistance—directly to existing web content. This annotation adds syntax highlighting, diagnostics, and auto-completion to web pages or text editors that lack these capabilities, without requiring any structural modifications to the host document. The CSS Custom Highlight API enables syntax highlighting and diagnostic underlines to be rendered as visual overlays, while tooltips display auto-completion suggestions and error messages without altering the underlying text. Some LSP functionality, such as code folding, cannot be implemented through pure annotation since it requires structural changes, but this approach demonstrates how substantial programming capabilities can be introduced through minimally invasive interventions.

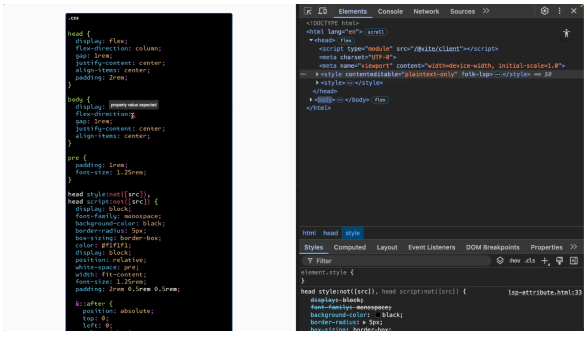

Figure 1: A custom HTML attribute that binds an LSP server to an editable style tag

The flexible pattern matching of CSS selectors enables these annotations to discover and interact with their surroundings, working opportunistically with available document structure rather than requiring pre-negotiated structural agreements. Unlike environments that must control their entire context,

annotations can situate themselves within foreign systems and coexist with existing features.

Our second prototype ports event propagators [10] to a custom HTML element, creating computational relationships between interface elements that enable spreadsheet-like reactivity between arbitrary UI components of existing websites. Through CSS selectors, these elements can define connections between DOM nodes, transforming static web pages into reactive documents where changes propagate automatically across components.

Annotations can also encourage the decomposition of functionality trapped within monolithic systems, making it available as reusable components. Our folk-sync attribute exemplifies this approach by extracting collaborative editing capabilities from systems like Webstrates and exposing them through a simple HTML annotation. This attribute makes document subtrees collaborative across devices, enabling real-time shared editing of any web content without requiring migration to dedicated platforms.

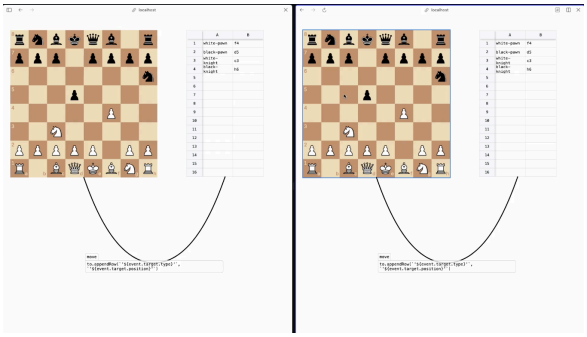

Figure 2: A chess board, event propagator, and spreadsheet

The composability of these annotations becomes apparent when multiple augmentations work together to create capabilities that exceed the sum of their parts. The figure above shows a chess board, event propagator, and spreadsheet—each authored as standard HTML with appropriate annotations —that not only synchronize state across multiple windows through folk-sync, but also react to each other's changes through event propagators. Moving a chess piece triggers the event propagator to log the move in the spreadsheet, creating a real-time game log that updates across all connected devices. This demonstrates how lightweight interventions can compose into rich, interactive systems that exhibit the computational relationships and collaborative capabilities typically associated with purpose-built environments, yet remain situated within ordinary web pages that can be inspected, modified, and extended through standard web technologies.

These interventions succeed by creating the experience of an environment without requiring one — users encounter live programming capabilities that feel indigenous to their current tools rather than isolated systems forcing them to move elsewhere.

3.2. Adversarial Embedding

Adversarial embedding is the approach of decoupling live programming systems from a specific host environment, like a top-level domain or desktop application. Unlike annotation, which augments existing systems, embedding introduces complete systems that can be composed alongside other tools across diverse host environments. This portability requires lightweight protocols that enable systems to communicate with their hosts without making strong assumptions about the surrounding environment.

Web applets [11] are one such protocol that web-based live programming systems can use to achieve embeddability. Through a small event-based protocol that wraps around an `iframe`, web applets enable any web page to externalize state and actions to a host environment. This requires no changes to how systems are designed, packaged, or distributed — they remain as standalone systems while gaining embeddability. The approach of web applets maintains the conventional boundaries between systems: authors control what can be accessed externally, constraining the kind of deep integration that free addressability enables.

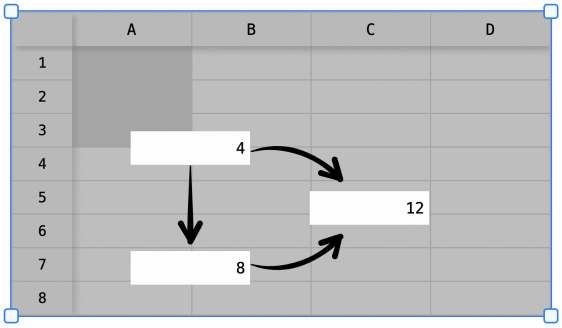

Figure 3: A freely-addressable HTML spreadsheet element.

One alternative is to embed freely addressable systems, which lessens the coordination requirements and provides a more compelling form of extensibility. For example, if the addressable space is the DOM, then a fully addressable system can be implemented as custom HTML elements. We demonstrate

above an *HTML-first* spreadsheet where each cell is addressable through CSS selectors. The state of the running system is exposed through the DOM, each cell having properties for its evaluated value and dependencies. Furthermore, each cell emits a DOM event when it is re-evaluated, supporting granular observability from outside of the spreadsheet element. This addressability extends to styling—any part of the system can be modified via CSS—making it possible to permissionlessly amend this system with an in-place visualization of the spreadsheet's dependency graph all while the system continues running.

3.3. Adversarial Extension

When systems provide no addressable surfaces, adversarial extension creates addressability by exploiting whatever infrastructure remains available. Unlike annotation, which works with systems designed to tolerate additions, extension operates on closed systems by re-purposing infrastructure that was not intended to support external modification.

Accessibility APIs represent one such exploitable infrastructure. Operating systems expose accessibility trees to support assistive technologies, creating a parallel addressable representation of every running application's interface. While this interface provides only a limited view of application state focused on user-facing elements rather than internal program logic, it offers near-universal coverage across all running applications.

Our prototype demonstrates how this infrastructure can be repurposed for external augmentation—a WebSocket server connects web interfaces to accessibility and windowing APIs, making it possible to query, subscribe to, and modify the interface state of any running application. This creates an addressable surface where none existed before.

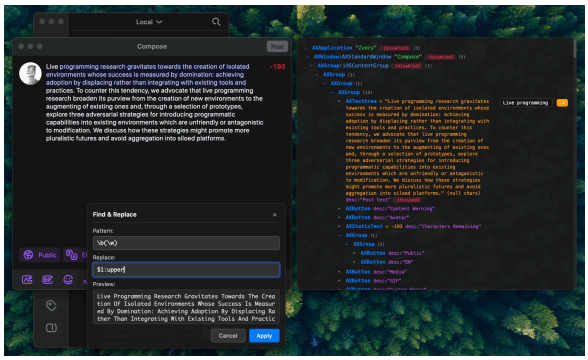

Figure 4: Ivory app extended with editable accessibility tree and regex-based text editing UI.

The accessibility tree prototype shows the Ivory messaging application augmented with two external interfaces: an editable outline view of its accessibility tree and a regex-based find-and-replace interface for text editing. This regex functionality, absent from Ivory itself, demonstrates the kind of read-write querying possible across boundaries we usually consider closed—proprietary applications with no APIs, closed source code, or deliberate restrictions on extensibility. The positioning system leverages accessibility coordinate information to spatially attach these augmentations to their target elements, making them feel more like native features than external overlays. Since applications cannot opt out of accessibility infrastructure without breaking assistive technology compliance, this approach works even with systems designed to resist external intervention, and the same augmentations can work universally across any application.

These three strategies are complementary rather than competing—each addresses different constraints in the landscape of existing systems. *Annotation* works with systems that tolerate additions, *embedding* enables portability across environments, and *extension* exploits mandatory addressable surfaces when no other options remain. The choice of strategy depends on the specific affordances and restrictions of the target environment.

4. Related Work

Systems like Sifter [12], Vegimite [13], Rousillon [14], Wildcard [15], and Joker [16] exemplify adversarial strategies by enabling end-users to customize web pages, scraping data into spreadsheets and tables, and reflecting modifications back to the original page. By packaging themselves as web extensions rather than standalone applications, these systems situate themselves inside the environments they augment rather than requiring users to bring their data elsewhere.

Whereas the systems above try to abstract away web technologies behind familiar interfaces, Webstrates [17] takes the opposite approach, creating a collaborative authoring environment where "the state of the DOM itself corresponds to the authorial shared state" [8]. Webstrates demonstrates the potential of exploiting the DOM's inherent addressability as a foundation for live programming in shared authorial environments. Our DOM sync attribute explores similar territory, enabling real-time collaborative editing

of DOM structures without requiring migration to a dedicated platform.

Engraft [18] explores composition between live programming tools by creating interfaces that allow different systems to be embedded within each other. While Engraft acknowledges that live programming systems should integrate with the outside world, its focus on inward composition—maintaining properties within controlled environments—contrasts with our emphasis on outward integration into hostile territory.

5. Limitations & Future Work

Our current exploration focuses on additive modifications and does not address removing or replacing parts of running programs. The approaches we present also concentrate heavily on UI-level intervention points. Significant work remains in applying adversarial techniques at other levels of the software stack, from runtime systems to operating system primitives. Kell's work on liballocs suggests one promising direction for free addressability at the level of Unix processes [19].

A key limitation emerges around the relationship between addressability and modifiability. While *free addressability* focuses on making system parts targetable, it doesn't address how those parts can actually be modified. The DOM is exploitable because CSS selectors provide addressing while referenced elements expose clear manipulation interfaces—a relationship that remains undertheorized in our work.

Most of our examples target web and browser contexts, limiting their applicability to the broader software ecosystem. Future work should explore how these strategies translate to desktop applications, mobile environments, and system-level software. The fragility of some approaches—such as relying on unstable CSS selectors or working around obfuscated DOM structures—highlights the need for more robust addressing schemes.

An important direction for future research involves enabling interoperability and co-existence between different live programming models that may have conflicting guarantees or execution models. What primitives enable different computational paradigms to work together? These questions become urgent as we move toward ecosystems where multiple live programming systems must coexist and collaborate.

Perhaps most ambitiously, we envision extending these principles to operating system design. What would it look like if accessibility trees provided stable, rich addressing schemes for all running applications? How might we design OS-level APIs that assume external composition rather than treating it as an afterthought?

6. Conclusion

When users experience live programming capabilities situated in place rather than sequestered in dedicated environments, we hope they begin to see such integration as normal rather than exceptional. We believe pluralistic practices that subvert intended boundaries create pressure like water finding cracks — persistent forces that gradually reshape systems toward openness.

Much of live programming research focuses on creating better environments without considering how change actually happens in computing ecosystems. We believe the community needs to confront the question of *change*: how do isolated programming tools evolve into integrated, composable ecologies without falling into success-by-domination strategies? Our approach rests on the belief that fragile bridges and adversarial interventions create social pressures that drive systemic change. By demonstrating what becomes possible when addressable surfaces are exploited, we establish expectations of interoperability and integration. These prototypes point toward a future where external composition is a design assumption rather than an afterthought.

The scale of this challenge becomes clear when we consider the difficulty of departing from tradition. Plaintext infrastructures resist single-system dominance, but this resistance was not inevitable. As Hall observes, what we think of as "human-readable plaintext" is actually the massive set of text encoding, display, manipulation, and processing artifacts currently ubiquitous in computing: *"ASCII, UTF8, text editors, text-field or text-area UI widgets, terminals, keyboards, String types, object-to-String rendering functions, human-readable format libraries, tokenizers, parsers, escape sequences and input sanitization, Base64 encoding, line-ending and whitespace conventions, and the fallback data-flavor of the copy/paste clipboard"* [4]. This ubiquity required decades of standardization, adoption, and gradual convergence—it did not emerge from any inherent philosophical commitment to openness. The challenge is achieving similar ubiquity for live

programming systems. This is not to advocate that specialized environments like Blender or Ableton should be decomposed—such tools serve important purposes within their domains. Rather, we argue that *all systems* should consider their participation in broader ecosystems rather than operating in complete isolation.

By working in hostile territory, we hope to demonstrate that live programming need not retreat into isolated environments to achieve its goals. The strategies we explore—annotation, embedding, and extension—offer different paths for engaging with the existing software landscape as it exists today. While our prototypes remain fragile and limited, they point to a future where live programming capabilities become as ubiquitous as plaintext itself. The question is not whether any single system will achieve domination, but whether we can work within our inherited circumstances. In this view, live programming research becomes work of *transformation* rather than *escape*.

References

[1] P. Rein, S. Ramson, J. Lincke, R. Hirschfeld, and T. Pape, "Exploratory and Live, Programming and Coding: A Literature Study Comparing Perspectives on Liveness," *The Art, Science, and Engineering of Programming*, vol. 3, no. 1, Jul. 2018, doi: 10.22152/programming-journal.org/2019/3/1.

[2] C. Clark and A. Basman, "Tracing a Paradigm for Externalization: Avatars and the GPII Nexus," 2017.

[3] S. Kell, "Convivial Design Heuristics for Software Systems," in *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming*, Porto Portugal: ACM, Mar. 2020, pp. 144–148. doi: 10.1145/3397537.3397543.

[4] C. Hall, "Rethinking the Human-Readability Infrastructure," in *Proceedings of the Workshop on Future Programming*, in FPW 2015. New York, NY, USA: Association for Computing Machinery, Oct. 2015, pp. 1–6. doi: 10.1145/2846656.2846657.

[5] C. Shank and O. Reed, "Folkjs." Accessed: Jul. 22, 2025. [Online]. Available: https://folkjs.org/

[6] P. Tchernavskij, "Designing and Programming Malleable Software," 2019.

[7] "Revisiting Information Hiding: Reflections on Classical and Nonclassical Modularity," in *Lecture Notes in Computer Science*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 155–178. doi: 10.1007/978-3-642-22655-7_8.

[8] A. Basman, C. Lewis, and C. Clark, "The Open Authorial Principle: Supporting Networks of Authors in Creating Externalisable Designs," in *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, in Onward! 2018. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 29–43. doi: 10.1145/3276954.3276963.

[9] C. Doctorow, "Adversarial Interoperability." Accessed: Aug. 09, 2020. [Online]. Available: https://www.eff.org/deeplinks/2019/10/adversarial-interoperability

[10] O. Reed, "Scoped Propagators." Accessed: Jan. 16, 2025. [Online]. Available: https://www.orionreed.com/posts/scoped-propagators

[11] M. Rupert and V. Steven, "Unternet-Co/Web-Applets." Accessed: Jul. 21, 2025. [Online]. Available: https://github.com/unternet-co/web-applets

[12] D. F. Huynh, R. C. Miller, and D. R. Karger, "Enabling Web Browsers to Augment Web Sites' Filtering and Sorting Functionalities," in *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology*, Montreux Switzerland: ACM, Oct. 2006, pp. 125–134. doi: 10.1145/1166253.1166274.

[13] J. Lin, J. Wong, J. Nichols, A. Cypher, and T. A. Lau, "End-User Programming of Mashups with Vegemite," in *Proceedings of the 14th International Conference on Intelligent User Interfaces*, Sanibel Island Florida USA: ACM, Feb. 2009, pp. 97–106. doi: 10.1145/1502650.1502667.

[14] S. E. Chasins, M. Mueller, and R. Bodik, "Rousillon: Scraping Distributed Hierarchical Web Data," in *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, Berlin Germany: ACM, Oct. 2018, pp. 963–975. doi: 10.1145/3242587.3242661.

[15] G. Litt and D. Jackson, "Wildcard: Spreadsheet-Driven Customization of Web Applications," in *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming*, Porto Portugal: ACM, Mar. 2020, pp. 126–135. doi: 10.1145/3397537.3397541.

[16] K. Katongo, G. Litt, K. Jin, and D. Jackson, "Joker: A Unified Interaction Model For Web Customization," 2022.

[17] C. N. Klokmose, J. R. Eagan, S. Baader, W. Mackay, and M. Beaudouin-Lafon, "\mkbibemph{Webstrates}: Shareable Dynamic Media," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, Charlotte NC USA: ACM, Nov. 2015, pp. 280–290. doi: 10.1145/2807442.2807446.

[18] J. Horowitz and J. Heer, "Engraft: An API for Live, Rich, and Composable Programming," in *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, San Francisco CA USA: ACM, Oct. 2023, pp. 1–18. doi: 10.1145/3586183.3606733.

[19] S. Kell, "The Inevitable Death of VMs: A Progress Report," in *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*, Nice France: ACM, Apr. 2018, pp. 61–62. doi: 10.1145/3191697.3191728.