Reconstructing Program Genealogies from Abstract Program Descriptions: The ELIZA Case Study

Jeff Shrager
Bennu Climate, Inc.
Palo Alto, CA 94301
jshrager@gmail.com

December 3, 2025

Abstract

Software phylogeny – the project of trying to understand how earlier programs influence later, similar ones – is not a new concept, but it turns out to be elusive. Whereas organism have real (if often obscure) evolutionary pathways via inheritance or horizontal gene transfer, software artifacts evolve through many different pathways, including revision, partial direct copying, translation into different languages, and even de novo conceptual reconstruction. It may even be difficult to define what it means for a program to be the same as or similar enough to another program to be considered a clone or knock-off. Analysis of version control histories is (mostly) clear, but much too narrow to encompass many interesting cases of program inter-influence. I hypothesize that it is useful to construct software phylogenies based upon shared abstract features, rather than at the code level. I use ELIZA, Joseph Weizenbaum's 1966 conversational program, as a model system for exploring this hypothesis. ELIZA has accumulated nearly sixty years of ports, rewrites, and conceptual imitators, producing a large and diverse family of specimens whose similarities and differences can be described in terms of abstract characteristics that differ from instance to instance, even though they all are ELIZAs in that they do essentially the same task, ignoring details such as the programming language in which the program was written, specifics of the conversational domain, and so on. To explore this hypothesis I gathered eighteen different ELIZAs from the collection in Elizagen.org, and used an LLM to develop a set of useful abstract characteristics of these programs. I input the resulting characteristic matrix PAUP, a common tool for the construction of phylogenetic trees. I also copied two of these exactly as a manipulation check – these pairs should cluster together, otherwise the results are suspect. The results accord with many of the intuitive relationships between these various ELIZAs, although there were some head-scratching surprises. Overall, I conclude that there is enough

support for the hypothesis to support putting effort into gathering many more ELIZAs and working up a well-refined set of characteristics for a more complete experiment.

1 Introduction

This project set out to ask a simple, methodological question: how can we reconstruct a genealogy of programs? Not the history as described in manuals or recollections, but a genealogy grounded in the artifacts themselves—the code, the structure, the linguistic patterns, and the operational behaviors that survive across generations of rewrites.

Software phylogeny – the project of trying to understand how earlier programs influence later, similar ones – is not a new concept, but it turns out to be elusive. Whereas organism have real (if often obscure) evolutionary pathways via inheritance or horizontal gene transfer, software artifacts evolve through many different pathways, including revision, partial direct copying, translation into different languages, and even de novo conceptual reconstruction. It may even be difficult to define what it means for a program to be the same as or similar enough to another program to be considered a clone or knock-off. Analysis of version control histories is (mostly) clear, but much too narrow to encompass many interesting cases of program inter-influence.

ELIZA, the early conversational program has generated an unusually large family of rewrites, ports, parodies, and reinterpretations. There are dozens and probably hundreds of ELIZA near-descendants scattered across dozens of programming languages, employing a variety of language-processing algorithms. The family of ELIZAs comprises a sprawling, poorly documented software "clade" that might be understood through systematic taxonomic analysis ELIZA provides an ideal proving ground for this. The family of ELIZA-like programs is large, diverse, inconsistent, and full of partial or fragmentary examples. Some versions survive in LISP or SLIP listings from the 1960s; others from 1980s hobbyist BASIC magazines; others as parodies, pedagogical rewrites, or one-off re-creations on long-dead university servers.

2 Method

2.1 Corpus Assembly and Normalization

Over almost two decades I have been assembling a corpus of historical and contemporary ELIZA implementations (http://elizagen.org) drawn from public archives, academic collections, personal code repositories, and recovered sources. Each program was treated as a *specimen* in a software–genealogical sense. To enable comparison across languages and coding styles, all specimens underwent a light normalization pass in which indentation and comment formats were standardized, superfluous whitespace was removed, and file metadata was stripped.

These steps preserved the structural and logical features of each program while eliminating irrelevant surface variation.

2.2 Initial Feature Extraction and YAML Encoding

Each specimen was abstracted into a structured, analyzable form through a feature–extraction process driven by a large language model (LLM). For each program, the model was prompted to identify salient structural and behavioral properties, including control-flow patterns, pattern-matching mechanisms, rule formats, data structures, and characteristic idioms associated with ELIZA-style systems.

The resulting feature lists were stored in a structured `YAML` representation that served as an intermediate "phenotypic" description of each specimen. At this stage, we also performed *manual feature augmentation*, adding expert-recognized traits that the model sometimes omitted or generalized too broadly. These additions included historically known algorithmic markers, distinctive syntactic quirks, and subtle divergences from canonical ELIZA logic. The YAML files resulting from this phase constituted the initial character matrices for downstream inference.

2.3 Round-Robin Vocabulary Expansion

Because initial model outputs vary in the features they surface, we employed an iterative, round-robin expansion procedure to harmonize and enrich the shared feature vocabulary across the corpus. In each iteration, a specimen's current feature list was presented to the LLM together with the aggregate vocabulary drawn from all other specimens. The model was asked to identify potentially relevant but missing features, propose distinctions, and ensure that each specimen was evaluated in light of an increasingly comprehensive, cross-specimen vocabulary.

Following each automated expansion cycle, we again conducted *manual curation*. This included removing spurious or over-abstracted features and adding additional domain-specific distinctions that the model sometimes conflated. Iteration continued until the vocabulary stabilized and no significant new traits were produced.

2.4 Final Feature Matrix Construction

After the round-robin process converged, the final feature list for each specimen was assembled by merging all automatically inferred features with all manually augmented ones. Features were encoded either as binary or categorical characters, depending on their form, yielding a specimen-by-character matrix analogous to a morphological character matrix in biological systematics.

Pairwise dissimilarities between specimens were computed by treating each character as an independent trait and scoring mismatches in character state.

The resulting distance or character-state matrix served as the input for phylogenetic analysis.

2.5 Phylogenetic Inference Using PAUP*

Phylogenetic trees were inferred using PAUP* under a parsimony criterion. The character matrix was imported into PAUP*, and heuristic tree searches were performed using random addition sequences, tree-bisection–reconnection (TBR) branch swapping, and multiple replicates to mitigate the risk of entrapment in local optima. All most-parsimonious trees were retained.

Strict and majority-rule consensus trees were then computed from the set of optimal trees. These consensus trees represent hypothesized genealogical relationships among ELIZA implementations, based solely on shared and divergent software features encoded in the final character matrix.

2.6 Interpretation

The resulting phylogenies should be interpreted as hypotheses of software relatedness rather than literal historical lineages. The pipeline—from normalized source programs, through LLM-derived and manually curated feature sets, and finally to PAUP*-based phylogenetic reconstruction—was designed to parallel biological systematics while accommodating the distinctive evolutionary dynamics of software artifacts.

What this project demonstrates is that software artifacts can be analyzed genealogically—not narratively or historically, but computationally. If one can extract a consistent set of characters across variants, one can apply the same tools used in evolutionary biology to visualize relationships among programs. ELIZA, because of its long history of transformation, provides an unusually rich test case, but the workflow generalizes. Any software ecosystem with many descendants or reinterpretations—adventure games, early operating systems, AI toolkits, interpreters—could be analyzed in this way.

The broader idea is that software, like biological organisms, carries traces of its ancestry. Those traces persist even when the syntax is rewritten, the structure reorganized, or the implementation style completely changed. And by approaching software genealogically—through careful artifact recovery, character extraction, and computational inference—we can uncover those traces in a principled way. The ELIZA Genealogy Project is still in progress, but the methodology is already clear: collect, normalize, extract, encode, and visualize. The goal is not to produce the definitive history of ELIZA, but to develop a method for computational taxonomy—an approach that brings rigor to the study of how software ideas evolve across time, languages, and cultures. ELIZA simply turns out to be the ideal proving ground for that experiment.

Table 1: Program List

ID	**Program Name**
E01	Cosell66Lisp
E02	Trembly17Cobol
E03	SamsonXXLisp
E04	KMPXXLisp
E05	JonLXXLisp_Test1
E06	Babcock15BASIC
E07	Hay21CPP
E08	Jull23BASIC
E09	DavenportXXBASIC
E10	Srijak08Python
E11	Strout05Python
E12	Shrager73BASIC_Test1
E13	Norvig91Lisp
E14	Cherry82Lisp
E15	Bell79BASIC
E16	Dunlop97JS
E17	Platt79BASIC
E18	JonLXXLisp_Test2
E19	Weizenbaum65MADSLIP
E20	Shrager73BASIC_Test2

Table 2: Feature Definitions

rule_based_pattern_matching Indicates whether the program uses an explicit set of rules, each with patterns and associated responses, to determine its behavior, as opposed to purely procedural or statistical logic.
keyword_priority_ranking Indicates whether the program assigns numeric priorities to keywords or rules and uses these priorities to select which rule or phrase to apply.
synonym_translation_table Indicates whether the program maintains a mapping that normalizes different surface forms (e.g., synonyms, inflections) into canonical tokens before rule application.
backtracking_pattern_matcher Indicates whether the pattern matching engine can consume variable-length segments of the input and backtrack over them to satisfy complex patterns.
wildcard_and_class_tokens Indicates whether the pattern language supports wildcards (e.g., numeric segment placeholders) and token classes or property-based matches within patterns.
parse_segment_storage Indicates whether the matcher stores matched input segments in an auxiliary structure for later reference during response generation.
template_based_reconstruction Indicates whether responses are generated by filling templates that reference previously matched segments by position or index.
conversational_memory_rules Indicates whether the program stores selected past user inputs or derived phrases and later retrieves them via dedicated memory rules to generate responses.
fallback_last_resort_rule Indicates whether there is an explicit default or "last resort" rule that is applied when no higher-priority or more specific rule matches.
punctuation_sensitive_tokenization Indicates whether the input is tokenized using an explicit set of separator and breaker characters so that punctuation affects how tokens and phrases are formed.
ordered_memory Indicates whether the recall is ordered (True) or random (False) when memory is called upon to produce a response.
list&symbol_processing_v_string_processing A value of True for this feature indicates that the program primarily uses an list and symbol processing paradigm. A value of False indicates that the program uses a primarily string based representation.
separable_script A value of True for this feature indicates that the script is essentially separate from the language processing code, possibly (although not necessarily) in a separate file. A value of False indicates that the script is a built-in part of the program. (Just because the script appears in the given code doesn't necessarily mean that it is inseparable. If there is code that reads the script in at the beginning, even from its own file, the value here could be True.)

Table 3: Program Features Comparison

Program	Rule-Based Pattern	Keyword Priority	Synonym Translation	Backtracking Pattern	Wildcard & Class	Parse Segment Storage	Template-Based Recon.	Conversational Memory	Fallback Last Resort	Punctuation Sensitive	Ordered Memory	List & Symbol vs String	Separable Script
E01_Cosell66Lisp	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
E02_Trembly17Cobol	✓		✓			✓	✓		✓	✓	✓		
E03_SamsonXXLisp	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
E04_KMPXXLisp	✓		✓			✓	✓	✓	✓	✓	✓	✓	
E05_JonLXXLisp_Test1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
E06_Babcock15BASIC	✓	✓				✓	✓		✓	✓	✓		
E07_Hay21CPP	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
E08_Jull23BASIC	✓	✓	✓			✓	✓		✓		✓		
E09_DavenportXXBASIC	✓		✓		✓		✓		✓		✓		
E10_Srijak08Python	✓			✓	✓	✓	✓	✓	✓		✓	✓	
E11_Strout05Python	✓		✓		✓	✓	✓		✓		✓		
E12_Shrager73BASIC_Test1	✓	✓	✓			✓	✓		✓		✓		
E13_Norvig91Lisp	✓			✓	✓	✓	✓			✓	✓	✓	
E14_Cherry82Lisp	✓	✓					✓		✓	?	✓	✓	
E15_Bell79BASIC	✓		✓		✓	✓	✓		✓		✓		
E16_Dunlop97JS	✓		✓				✓		✓	✓	✓		
E17_Platt79BASIC	✓					✓	✓		✓				
E18_JonLXXLisp_Test2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
E19_Weizenbaum65MADSLIP	✓	✓	?	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
E20_Shrager73BASIC_Test2	✓	✓	✓			✓	✓		✓		✓		

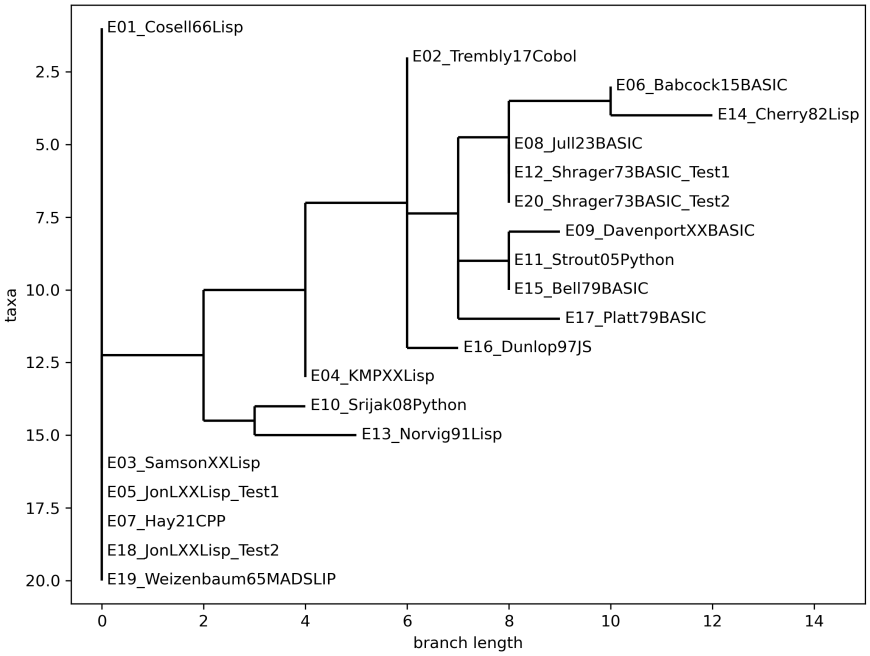

Figure 1

3 Results

The clustering results reveal several features of the ELIZA lineage that align well with historical expectations while also surfacing a few surprising connections. First, the two versions labeled "Test1" and "Test2" emerge exactly as they should: because they are byte-for-byte identical aside from their filenames, they appear together in the same equivalence class. Their relative ordering within that set is arbitrary and holds no interpretive value, so the fact that the JONL pair is interspersed with other programs in the display does not signal any discrepancy. The system correctly treats these copies as a single lineage branch.

The original ELIZA source, Cosell's 1966 reconstruction, and Jon L. White's ("JONL") version cluster in the same group as well. This is consistent with the known chronology: White's Lisp rewrite was produced only a few years after Cosell's, in the same MIT environment, and the shared ancestry is reflected in the structural features that drive the phylogenetic analysis. Also notable is that Ant's modern reconstruction—which he has long described as a faithful reproduction of the earliest code—indeed clusters tightly with the original and with the earliest knockoffs. This supports his claim that his version preserves both the spirit and structure of the 1960s implementations.

Beyond these expected results, the tree exposes several intriguing relationships that warrant further investigation. Versions written in different program-

ming languages occasionally cluster together, suggesting either direct transliteration or some common intermediary ancestor circulating informally across communities. In particular, Samson's Lisp version sits unexpectedly close to the originals, despite its unclear provenance. Whether this reflects an unrecognized historical connection or simply convergent structural similarity remains to be determined, but these early signals point to a richer and more complex software genealogy than the canonical narratives suggest.

4 Prior Art

The study of software genealogy draws on several overlapping research traditions, each of which offers methods or conceptual frameworks that help illuminate the problem of reconstructing ancestry among dispersed, informally transmitted code artifacts—precisely the situation in which many historical ELIZA variants reside. Early work on clone detection recognized that duplicated or near-duplicated fragments could reveal latent relationships among programs. Baxter et al.'s AST-based clone detection system Baxter et al. (1998) showed that structural similarities in code, even when lexical tokens differ, can expose deep shared lineage. Kamiya et al.'s CCFinder Kamiya et al. (2002) extended this insight by demonstrating that multi-linguistic, token-based similarity analysis can robustly detect large-scale duplication in the wild. Roy and Cordy's survey Roy and Cordy (2007) codified the idea that code similarity is not merely a question of plagiarism or redundancy but can support inference about evolutionary relationships. For the present project, these works collectively underscore that structural and token-level patterns in ELIZA variants, even if superficially noisy or idiosyncratic, carry usable genealogical signals. This supports both the initial clustering of variants and the rationale behind using LLM-based vocabulary normalization to regularize surface diversity into comparable forms.

Studies of software evolution have long recognized that programs change over time through copying, modification, and gradual accumulation of differences. Godfrey and Tu's analysis of open source software evolution Godfrey and Tu (2000) demonstrates that software systems exhibit branching patterns and divergence similar to biological lineages, though the mechanisms differ fundamentally. While this work focused on version control histories within single projects, it established that evolutionary patterns in software can be traced and analyzed systematically. The present work extends these observations by applying classical phylogenetic reconstruction methods to variants that lack formal version control metadata, using structural and behavioral features as evolutionary characters. These efforts demonstrate the feasibility of producing evolutionary trees of code artifacts when direct historical documentation is incomplete, providing a methodological foundation for treating the ELIZA corpus as an evolving population of software "organisms."

These efforts demonstrate the feasibility of producing evolutionary trees of code artifacts when direct historical documentation is incomplete. They provide a methodological and conceptual foundation for treating the ELIZA corpus

as an evolving population of software "organisms," subject to informal copying, mutation, recombination, and drift across decades and platforms. While these studies mostly concern contemporary, large-scale ecosystems, the analogy holds strongly for ELIZA, whose variants proliferated through hobbyist networks, academic archives, bulletin boards, and early internet repositories in a manner closely resembling biological or cultural diffusion.

The third major tradition—malware lineage reconstruction—might seem distant at first glance, but it is methodologically the closest analog to the ELIZA problem. Malware families evolve through rapid, decentralized, and poorly documented copying, which makes reconstructing their ancestry using code similarity and structural deltas an essential forensic task. Walenstein et al.'s design-space analysis Walenstein et al. (2007) surveys the similarity metrics used to detect relatedness among polymorphic variants, while Bailey et al. Bailey et al. (2013) present one of the first practical systems for reconstructing malware family trees directly from code bodies. These papers demonstrate that coherent genealogies can be inferred even when artifacts are heavily mutated, distributed without provenance, and subject to opportunistic reuse—precisely the conditions under which ELIZA variants propagated. For the present project, malware lineage work provides both an empirical precedent and a methodological toolkit: the logic of identifying minimal mutations, shared scaffolding, inherited routines, and branching divergence is directly applicable to the ELIZA corpus, and validates the notion that lineage can be inferred even when artifacts are partial or degraded.

A fourth line of work emerges from mining software repositories, where researchers use version-control systems to reconstruct histories of files, functions, and subsystems. German and Hassan German and Hassan (2009) and Bird et al. Bird et al. (2006) demonstrate that even messy, inconsistent repository data contains enough signal to reveal complex evolutionary patterns. Although the ELIZA ecosystem lacks such structured metadata—its variants often circulated before version control was ubiquitous—the methodological stance is instructive: software artifacts carry historical traces even when explicit documentation is incomplete or unreliable. Techniques such as semantic differencing, temporal clustering, and ancestry inference, originally developed for large VCS-backed projects, help frame the ELIZA effort as a special case of a broader problem in recovering the hidden structure of software evolution.

Finally, historical and cultural analyses of software, represented by work such as Marino's Critical Code Studies Marino (2020), emphasize that source code is also a cultural and historical object whose genealogy reflects social networks, educational lineages, and the circulation of ideas. This perspective is crucial for ELIZA, which was not merely copied but taught, rewritten, and reinterpreted across multiple communities—early AI laboratories, computer clubs, commercial microcomputers, hobbyist magazines, and eventually the open internet. Understanding ELIZA's genealogy therefore requires not only technical similarity metrics but also contextual reasoning about author intent, pedagogical transmission, and cultural reuse patterns. This is where the present project's hybrid approach—combining structural analysis, contextual metadata, code ar-

chaeology, and LLM-assisted normalization—extends beyond prior work.

Taken together, these bodies of literature demonstrate both the feasibility and the novelty of the current project. Clone detection research shows that even noisy lexical and structural patterns carry genealogical information. Software phylogenetics establishes that code artifacts can be treated as evolving populations. Malware lineage reconstruction proves that coherent trees can be inferred from informal, uncurated ecosystems. Mining-software-repository work demonstrates that lineage persists in artifacts even when metadata is sparse. And cultural code studies remind us that genealogies reflect not just code mechanics but the social pathways through which programs spread. The ELIZA genealogy project synthesizes these strands by building a reconstruction of a historically significant software family whose variants lack formal version control, whose mutation history spans half a century, and whose textual diversity requires normalization beyond conventional tools. In doing so, it occupies a unique position in the landscape of software evolution research, extending known methods to a domain where code, culture, and history are deeply entangled.

5 Why not a lineage tree?

Although this work is inspired by biological phylogenetics and textual stemmatics, the artifact we are reconstructing is neither a phylogeny nor a classical stemma codicum. Those frameworks assume a set of conditions that simply do not hold for software lineages. Biological phylogenies presuppose predominantly bifurcating descent, relatively clocklike mutation, and the absence of extensive horizontal transfer. Textual stemmatics assumes manuscripts are copied with occasional errors, that variants propagate largely through single-parent descent, and that contamination—borrowing from multiple sources—is exceptional enough to be treated as an anomaly. In contrast, the evolution of software such as ELIZA is fundamentally polyparental and reticulated. Programs are translated, ported, rewritten, partially copied, and recombined across languages and platforms; distinctive fragments are borrowed and reinserted; and entire structures are periodically obliterated and rebuilt. These properties violate the assumptions required for a tree-shaped genealogy in either biology or philology.

Moreover, whereas stemmatics analyzes variation directly from the textual surface, our reconstructions rely on abstract functional descriptions of each program—summaries of rule inventories, ordering, transformation pipelines, response templates, and architectural structure. These descriptors capture the conceptual design of each ELIZA variant, but they do not behave like independent characters in the phylogenetic sense. Their inheritance is purposeful rather than stochastic, often lossy, and frequently mediated by translation across languages or paradigms. The result is that shared features can reflect genuine descent, convergent reimplementation, or conscious borrowing from multiple ancestors. Under these conditions, a true stemma—a single rooted tree of descent—is not an appropriate model for the historical process.

For this reason, our goal is not to force the ELIZA corpus into a tree, but instead to construct a lineage graph: a directed acyclic network that allows multiple parents, partial inheritance, and reticulation. Where the historical record permits, we annotate edges with evidence such as shared idiosyncrasies, porting traces, or explicit documentation. Where uncertainty remains, we record alternative possibilities without collapsing them into a single branching structure. This approach better reflects how software actually evolves and allows us to capture the rich, hybrid, and multi-origin nature of ELIZA's diffusion, rather than imposing an overly simplified genealogical form that the evidence does not support.

Code Availability

All of the python programs, and ELIZA sources used in these experiments are available here: https://github.com/jeffshrager/elizagen.org/tree/master/genealogy

Acknowledgments

Thanks to the members of *Team ELIZA* (David Berry, Sarah Ciston, Anthony Hay, Rupert Lane, Mark Marino, Peter Millican, Art Schwarz, and Peggy Weil) for years of useful discussion about ELIZA, and especially Anthony Hay and Art Schwarz, for detailed technical advice on this particular project. Some of this work, including some of the programming and some of the writing of this paper, were aided by OpenAI and Anthropic LLMs.

References

Bailey, M., Dumitras, T., and et al. (2013). A system for malware lineage. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 73–89.

Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M., and Bier, L. (1998). Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, pages 368–377. IEEE.

Bird, C., Zimmermann, T., and Nagappan, N. (2006). Mining windows for software engineering tools. In *IEEE Software*.

German, D. M. and Hassan, A. E. (2009). Studying software evolution using cvs, subversion, and git. *Empirical Software Engineering*, 14(2):127–168.

Godfrey, M. W. and Tu, Q. (2000). Evolution in open source software: A case study. In *Proceedings of the International Conference on Software Maintenance*, pages 131–142. IEEE.

Kamiya, T., Kusumoto, S., and Inoue, K. (2002). Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. In *IEEE Transactions on Software Engineering*, volume 28, pages 654–670.

Marino, M. C. (2020). *Critical Code Studies*. MIT Press.

Roy, C. K. and Cordy, J. (2007). A survey on software clone detection research. *Queen's School of Computing Technical Report*.

Walenstein, A., Mathur, A., Chouchane, R., and Lakhotia, A. (2007). The design space of code similarity algorithms. In *Proceedings of the 2nd ACM Workshop on Recurring Malcode*, pages 1–10.