

Codesign

Between Code and Design



Concept by Dmitry Vostokov

Codesign

Between Code and Design

Codesign with LLMs and the Software Codesigner Profession

Dmitry Vostokov · DumpAnalysis.org

Developed with AI assistance (GPT-6 Astra Max and Fable 5.1 Extra) and checked against the sources listed in this article.

Version 9 · October 8, 2026

Abstract

I now rarely code explicitly and spend much more time on design. Here, I use Codesign to describe this activity between code and design, closer to the design side. This adds an interpretation to a term already established in hardware/software co-design. The word can be read in three ways: CODE overlaps with DESIGN in CO[DE]SIGN; CODE + SIGN treats generated code as an executable sign; and CO-DESIGN means designing together with large language models (LLMs). Generated code gives me a proposal that I can examine, execute, and compare with my requirements. I also propose the Software Codesigner profession. CODE + SIGNER adds responsibility for signing off on an implementation for a specified use. I connect these ideas to software diagnostics and prior work in programming as design, semiotics, co-design, program synthesis, human–LLM co-creation, agency, and code review. The proposed contribution combines these readings with a design-side way of working.

From Explicit Coding to Codesign

I now spend less time writing individual statements and more time deciding what a system should do, how its parts should relate, and what evidence shows the result corresponds to the intended behavior. Coding assistants produce much of the explicit code. My activity remains connected to code, but its center has moved toward design.

Here, LLM refers to the underlying model and coding assistant to the system through which I use it. An agentic assistant can also carry out actions autonomously in response to a prompt [16].

I use Codesign here for this activity. It lies between code and design, nearer to design. I spend more time on concepts, relationships, constraints, and assessing implementations. Much of the source text is produced by coding assistants.

In this article, the capitalized Codesign names my combined reading. I use co-design for the collaborative and established engineering senses.

Code and design already overlap. Choosing a data structure, separating components, or placing error handling can be a design decision made while writing code. Naur's programming as theory building and Reeves's source code as design provide earlier accounts of this connection [1, 2]. My position between code and design describes how I participate in development.

At the code end, I express most decisions by constructing and modifying source text. Toward the design end, I work through requirements, models, constraints, examples, and the assessment of candidate implementations. Codesign sits closer to this end while staying connected to implementation.

During a task, I may design a component, inspect generated code, diagnose an unexpected result, and manually correct it. An implementation detail can pull my attention back to explicit coding. Codesign describes where I spend most of my attention.

Three Readings of the Same Word

The first reading is visible in the spelling. CODE ends with DE, and DESIGN begins with DE. Letting these letters overlap gives CO[DE]SIGN. The shared letters provide a small visual model of the shared activity. Code and design remain recognizable within one word.

The second reading separates the word after CODE: CODE + SIGN. Here the emphasis moves from the activity of producing code to the interpretation of its result. Generated code can function as a sign of a possible design, a proposed solution, an assumption introduced by the assistant, or a misunderstanding that has acquired an executable form.

The third reading is CO-DESIGN: designing together. I am also co-designing with LLMs. Their contribution can include a proposed decomposition, a question about an omitted condition, or an alternative that changes how I understand the problem. Design develops through the interaction as well as through my examination of the generated code.

THREE READINGS

CODE overlaps with DESIGN



CODE + SIGN

An executable proposal to interpret

CO-DESIGN

Developing the design with LLMs

A POSITION NEARER DESIGN



Figure 1. Three readings of Codesign and its proposed position nearer the design side. The lower diagram shows a qualitative emphasis, not a measured proportion. Code construction and design continue to overlap throughout.

The spelling helps me remember these connections; it is not a proposed etymology. Co-design in the participatory-design tradition provides an established collaborative meaning [3]. Here I also consider collaboration with LLMs. I need to distinguish their contributions from my own and explain how proposals are accepted. CODE + SIGN concerns meaning and interpretation. CODE + SIGNER, introduced below, adds the act of signing off on code.

What It Means to Lean toward Design

When I lean toward design, I concentrate on decisions and their consequences. Which states are valid? Who owns a resource? What happens when a dependency fails? What observations distinguish successful completion from an incomplete operation? A decision about any of these questions can affect many lines of code.

A prompt is one way to express a design. I can also use an interface contract, a state diagram, a counterexample, an acceptance test, or a diagnostic scenario. An existing component may specify behavior that must be preserved. In each case, I decide what should remain invariant while implementations change.

The design side also includes learning from implementation. Generated code can expose a distinction that was missing from the initial request. A proposed cache forces me to decide what counts as stale. A background task forces me to decide who can cancel it. A new dependency forces me to consider the consequences of its failure. I can revise the design after inspecting such proposals.

The design can therefore develop while I examine generated implementations. I do not need to specify every detail before generation. However, I need to recognize the assistant's choices, decide which to retain, and understand the resulting system.

My technical involvement includes explaining these choices and assessing their consequences. Typing fewer statements does not measure that involvement. Merely accepting generated output does not establish that I have performed design. I can delegate code production while remaining responsible for the design decisions I accept.

Code as an Executable Sign

Here I use sign for something through which I understand something else in a particular context. A function, a type, or a collection of modules can represent a retry policy, a domain distinction, or an architectural boundary. Computer semiotics provides prior work for this interpretation [5, 6].

Consider a generated component called `ReliableClient`. Its name invites one interpretation. A loop within it suggests repeated attempts. A timeout parameter suggests bounded waiting. These are signs I use to begin reconstructing a design. They are not sufficient evidence that the component provides the reliability I require.

The name may promise more than the body implements. The loop may retry a permanent failure. The timeout may apply separately to each attempt, allowing the total operation to take much longer than expected. Generated code can therefore be informative even when it is wrong: it makes some assumptions available for inspection.

I can distinguish three perspectives on a generated implementation and its evaluation (Table 1).

Table 1. Perspectives on a generated implementation and its evaluation

Perspective	What is examined	What remains to be established
Design representation	Names, types, interfaces, component boundaries, and apparent patterns	Whether the represented decisions correspond to the intended system
Executable mechanism	Control flow, state changes, resource use, and interactions with dependencies	Whether the mechanism has the required properties under the relevant conditions
Diagnostic evidence	Test observations, traces, logs, snapshots, and externally visible effects	Which claims the observations support, and what their coverage leaves unresolved

These perspectives depend on what I want to understand. The same code fragment may be read differently by its author, a maintainer, an operator, and a coding assistant.

An executable sign also has behavior. When compiled or interpreted and run in its environment, code can consume resources, change state, and affect other systems. I therefore examine both what the code represents and what its mechanisms can do.

This interpretation does not require me to attribute human understanding to the assistant. I examine the generated code, what it appears to represent, and how I read it in the development context. The assistant's explanation is another artifact to examine. Agreement between that explanation and my prompt does not, by itself, establish correctness.

I can read human-written code in the same way. What has changed in my work is how often I encounter code as a proposal to interpret, without having composed its individual statements.

Co-Designing with LLMs

I am also co-designing with LLMs in the collaborative sense. I bring purposes, context, constraints, and judgments. The LLM suggests structures and alternatives, and may raise questions about decisions I have not yet considered. The design develops through these exchanges and through my examination of the results.

Consider a possible conversation about the diagnostic client used later in this article. I ask for reliable retrieval. The LLM proposes retries and asks which failures should be retried. I specify transient failures and a deadline. Its next proposal distinguishes a deadline for the whole operation from a timeout for each attempt. I examine the distinction and revise the requirement. The resulting design contains decisions that became explicit during the exchange.

The LLM's proposal can change my next question, and my answer can change its next proposal. I may revise the problem description, the implementation, and the criteria for assessing it. Conversational software-development research provides a precedent for this form of co-creation [7].

Calling the LLM a co-designer describes its functional role; it does not assign it the developer's experience, understanding, or responsibility. People and organizations still decide what to accept and what evidence is sufficient for the intended use. A suggestion can be useful even when I reject it after investigation.

During this exchange, I can refer to the same code fragment, diagram, test, or trace. A counterexample may lead to a new implementation, and an implementation may expose an incomplete requirement. These artifacts help me develop and discuss the design.

Generated Code as a Design Proposal

I can consider an assistant's output as a proposed implementation. It may satisfy my explicit requirements while introducing choices I have not discussed. Some choices are necessary to complete the implementation. Others may come from familiar examples, repository context, or defaults.

Suppose I ask for a service that processes requests in the background. The result may introduce an unbounded queue. The request did not explicitly ask for unbounded storage, but the code now includes that decision. Reading the code as a sign brings the decision back into view. I can then specify limits, overload behavior, and the observations needed to recognize a growing backlog.

I can repeat the process: express an intention, obtain an implementation, identify its decisions, examine its behavior, and revise the code or the intention. A failed experiment may reveal an implementation defect or a design omission.

Each artifact has a different role. My request describes what I want. Generated code implements a particular proposal. A test expresses an expectation for selected conditions, and a trace records selected observations from an execution. I need to preserve these distinctions when comparing the artifacts.

When the assistant generates both an implementation and its tests, the two may share the same mistaken assumption. I should therefore connect important checks to requirements, counterexamples, or external observations that can challenge that assumption. Independence concerns the basis of the check, not simply whether another model or another prompt produced it.

Generated code makes the proposal concrete enough to inspect and exercise. I can use the findings to correct the implementation or continue the design discussion.

Design Decisions in a Retry Example

Consider an illustrative request for a client that retrieves diagnostic data from a remote service: retry three times, apply a deadline, support cancellation, and log the result. The request appears specific, but it leaves several design questions open.

Does three mean three total attempts or an initial attempt followed by three retries? Does the deadline cover the whole operation or each attempt? Which failures are eligible for retry? What happens if cancellation arrives during a delay? Does a message saying that the request is complete distinguish success from an exhausted retry policy?

Suppose the assistant produces a reasonable-looking implementation with a retry loop, a timeout, and a final completion message. I can recognize familiar structures while still having the wrong design. Codesign makes the unresolved decisions explicit before treating the generated program as an accepted implementation. Table 2 relates possible decisions to evidence I can seek.

Table 2. Retry decisions and evidence for assessing the implementation

Design decision	Possible misleading sign in code	Evidence to seek
At most three total attempts	A parameter called <code>retries</code> with an unclear counting convention	A controlled failure scenario with an observed attempt count
One deadline for the whole operation	A timeout restarted for every attempt	Timing observations across failed attempts and waiting periods
Retry only designated transient failures	A broad exception handler surrounding the retry loop	Different responses to transient and permanent failure scenarios
Cancellation prevents new local attempts	A cancellation parameter that is checked only at entry	Cancellation during both an active attempt and a waiting period
Terminal outcomes remain distinguishable	A generic completion message on every exit path	Separate observations for success, failure, deadline expiry, and cancellation

The deadline and cancellation requirements must also respect the behavior of the dependency. Stopping local waiting does not establish that remote work has stopped. The design should say what is controlled locally and what remains uncertain remotely. This distinction may be invisible in a convenient function name.

This is a hypothetical example. The assistant may write the loop and the calls, while I determine their intended behavior and how to assess it. The work remains connected to code, with my attention nearer design.

Diagnostics as Feedback into Design

In software diagnostics, I infer system state and behavior from execution artifacts. Codesign adds a related task before execution: reading generated code as a proposal for the system I may construct.

In my earlier work on Bridging the Great Divide, I proposed using pattern catalogs to connect software construction with memory dump analysis and, through trace motifs, with trace and log analysis [8]. I can extend this connection to generated code. From a proposed design, I identify the evidence I expect to find and compare it with execution artifacts.

For the retry example, I can examine the relation between attempts, delays, and terminal outcomes. An extra attempt can contradict the chosen counting rule. Repeated activity after cancellation can reveal a missing control path. A completion record without a corresponding outcome can reveal either incomplete instrumentation or an incorrect reconstruction of the operation.

A missing log message does not prove that the corresponding activity did not occur. Collection may be incomplete, the event may not have been instrumented, or the message may refer to a different stage. Similarly, a success message does not prove that the intended external effect occurred. I also need to examine the instrumentation.

I can ask diagnostic questions while designing the component. What evidence will be available if it behaves unexpectedly? Can I correlate an operation across its parts? Can I distinguish an absent activity from an absent record? When would I need a memory snapshot, an observation of a dependency, or a controlled experiment?

Observability is therefore part of Codesign. Diagnostic findings may lead me to change component boundaries, state models, or failure policies, as well as individual statements.

The Software Codesigner Profession

I now extend Codesign from an activity to a proposed profession: Software Codesigner. A Software Codesigner develops designs with coding assistants, examines the resulting implementations, and decides whether to accept them within an agreed scope. I may write comparatively little source text while remaining closely involved in what the software means and does.

The name has a similar overlap: CODE and DESIGNER give me CO[DE]SIGNER. I can also read it as CO-DESIGNER, someone who participates in the design process with others, including LLMs. CODE + SIGNER describes the person who signs off on the resulting code.

I first interpret the generated code as an executable sign. I then decide whether the implementation is acceptable for its intended use. Sign-off records this judgment for a particular version and purpose. The act of approval does not make the code correct.

Sign-off starts influencing my work during design. For example, to approve cancellation behavior, I need to decide what cancellation means and what evidence would support that decision. This affects the questions I ask the LLM, the alternatives I consider, and the diagnostics I request.

I can describe the role through four recurring activities (Table 3).

Table 3. Activities of the proposed Software Codesigner

Activity	Work of the Software Codesigner	Record that supports later review
Form the design	Establish purposes, constraints, and criteria for acceptance	Requirements and the reasons for consequential decisions
Develop it with LLMs	Explore proposals and identify choices introduced during generation	Accepted alternatives, rejected alternatives, and unresolved questions
Assess the implementation	Inspect relevant code and relate analysis, tests, and diagnostics to claims	Evidence tied to the implementation and conditions examined
Decide on acceptance	Approve a defined scope, request changes, or withhold approval	The version, intended use, decision, limitations, and responsible reviewer

Returning to my retry example, I can record acceptance of a particular client revision for a stated service configuration. The record refers to the agreed attempt count, deadline, cancellation behavior, and observations of terminal outcomes. I also record what remains unresolved, such as whether remote work continues after local cancellation. If the retry policy or dependency changes, I need to reconsider which parts of my judgment still apply.

This work requires enough software knowledge to understand important implementation decisions, question the evidence, and recognize when another specialist is needed. Formulating a prompt is only part of the work. I must also be able to assess the result and explain my sign-off. I can return to explicit coding whenever it helps me investigate or express a decision.

Sign-off requires authority to request changes or withhold acceptance. Several people may share the work, and independent review is useful when I also helped develop the design. Approval of a component for integration need not authorize deployment of the entire system. The Codesigner works within the review and release responsibilities assigned by the organization.

This sign-off needs to be distinguished from other forms of signing. Apple's code-signing guide describes checks of origin and integrity and explicitly states that signing does not guarantee freedom from security vulnerabilities [9]. Apple also uses codesign as the name of a command-line tool [9]. For my purposes, a valid signature leaves a separate question: does the software meet its requirements? The Developer Certificate of Origin concerns the basis for submitting a contribution and its public record [10]. Here I am concerned with an engineering judgment about fitness for a specified use. These acts may overlap, but each serves a different purpose.

These activities already overlap with the work of software designers, developers, reviewers, and technical leads. The name Software Codesigner brings design participation and code sign-off together. Whether this combination develops into a separate profession remains an open question.

Prior Work and Conceptual Neighbors

Established Meanings of Co-Design

Hardware/software co-design is an established engineering discipline. De Micheli and Gupta describe concurrent hardware and software design directed toward system-level objectives [4]. The hardware and software decisions are considered together because each affects the resulting system.

My use concerns a different relationship: my position between code and design. I develop designs with LLMs, interpret generated code as an executable sign, and decide whether to sign off on an implementation. These meanings can coexist. I can work in this way while also co-designing a hardware/software system.

In the participatory-design tradition, Sanders and Stappers describe co-design as collective creativity across the design process, including participation by people who are not trained designers [3]. Co-designing with an LLM connects my use to the collaborative meaning. I still need people affected by the system to participate to understand its purposes and consequences.

The spellings codesign and co-design are dictionary variants for designing jointly; Merriam-Webster records the verb's first known use in 1943 [19]. My spelling convention is local to this article. Chlipala's Codesign for Legibility (to AI and Everyone Else) argues that improving models requires changing the worlds they model to make them easier to understand [20]. My focus is on design participation, generated code, and the decision to accept it.

Programming as Design and Understanding

Naur's Programming as Theory Building places the programmer's understanding of how a program addresses its subject matter at the center of programming [1]. This distinction between producing text and developing understanding is useful for Codesign. Can I explain why the generated system has its particular structure and how it should change?

Reeves's What Is Software Design? treats source code as an engineering design [2]. Codesign retains this connection. The change is in how I develop and examine that design when an assistant generates much of the source text.

Knuth's literate programming connects programs with explanations for human readers [11]. In Codesign, I may need to understand a substantial implementation before I have worked through its detailed construction myself.

Computer Semiotics and Semiotic Engineering

Andersen's Computer Semiotics presents computer systems as sign systems open to interpretation and describes programming as sign creation [5]. Tanaka-Ishii's Semiotics of Programming examines programs through models, kinds, and systems of signs [6]. The reading of code as a sign therefore has substantial prior work.

De Souza's semiotic engineering examines communication between designers and users through interactive systems [12]. Here I apply a related perspective during development. I interpret generated code in relation to my intentions, the implementation choices, and the evidence available to assess them.

Picca's Not Minds, but Signs proposes a semiotic account of LLMs, emphasizing sign manipulation and contextual interpretation [21]. This is a close conceptual neighbor. Here I focus on the generated program as an executable sign and on the evidence needed to decide whether to accept software.

Program Synthesis and Exploration with AI

Gulwani, Polozov, and Singh survey program synthesis as the automatic construction of programs from specifications of user intent [13]. This connects to my shift from writing implementations toward expressing and refining requirements. Any guarantees depend on the specification, synthesis method, and their assumptions. They do not transfer to an informal request to a generative assistant. A claim about that assistant's output needs its own justification.

Barke, James, and Polikarpova identify acceleration and exploration modes in their study of programmers using Copilot [14]. Exploration is close to Codesign: generated code can help me discover and assess possible approaches.

Vaithilingam, Zhang, and Glassman report difficulties understanding, modifying, and debugging generated code [15]. These findings support the need to examine how developers interpret such code. Both studies focus on specific tools, participants, and tasks; their results do not generalize to every current coding assistant.

Conversational Development and Co-Creation with LLMs

Ross and colleagues developed the Programmer's Assistant to study extended, contextual conversations with a code-capable LLM. Their evaluation with 42 participants explored capabilities beyond code generation and presented conversational interaction as a basis for co-creative software development [7]. This is direct prior work for my collaborative reading of Codesign.

In my account, the conversation develops the design, while code and execution artifacts give me concrete objects to examine. This links collaboration to interpreting code as a sign.

Agency and Delegation in Agentic Development

Feng, Yun, and Wang's 2026 study examines agency in software engineering with agentic AI. It involved five senior engineers in task development, ten junior engineers in an AI-assisted debugging task, and five other senior engineers reviewing junior prompt histories. Their findings include experienced developers maintaining control through detailed delegation and organizational policies constraining the allocation of agency [16]. The study concerns these participants and activities. It informs my discussion of directing and assessing generated implementations without establishing a general model of the proposed profession.

Code Review and Responsibility for Acceptance

Google's code-review guidance places overall design at the center and asks reviewers to examine functionality, tests, and the context of a change. It also distinguishes review scopes and calls for qualified reviewers where specialist knowledge is needed [17]. Its review standard assigns responsibility for the code under review and treats approval as a judgment about improving the codebase, without requiring perfection [18].

These practices already include responsibility for code written by someone else. Software Codesigner brings this responsibility together with designing through LLM interaction and interpreting generated implementations. People and organizations retain the authority to accept the result.

Novelty and Limits

I propose an additional interpretation of an existing term for software work with LLMs. Programming as design, code as a sign, program synthesis, human–LLM co-creation, and code review also have substantial prior work. My proposal connects these ideas and gives the practitioner a corresponding name.

The combination starts with the overlap in CO[DE]SIGN. CODE + SIGN lets me examine generated code as an executable proposal, and CO-DESIGN describes developing the design with LLMs. The position nearer design describes where my attention is concentrated. Software Codesigner extends these readings to the practitioner: the overlap in CO[DE]SIGNER, collaboration in CO-DESIGNER, and responsibility for acceptance in CODE + SIGNER.

I also extend my earlier work connecting software construction and diagnostics [8]. I read a generated implementation, identify the decisions it appears to express, and compare them with what its execution artifacts support. This connects interpreting proposed code with interpreting observed behavior.

My novelty claim concerns this combination. It does not concern the invention of code review, diagnostic feedback, or responsibility for acceptance [17, 18]. A targeted review through October 7, 2026 did not identify the exact formulation, including Software Codesigner as CODE + SIGNER. The search was limited and cannot establish priority or exclude equivalent formulations. I propose a way to understand the activity and a possible profession; I have not validated a new development method or established that a distinct occupation has emerged.

I can investigate the proposal through several questions. Does reconstructing the decisions in generated code help developers find requirement mismatches? Does designing the expected diagnostic evidence improve fault localization? How do requirements change during LLM dialogue? Does recording the scope and evidence of sign-off help later reviewers understand an acceptance decision, and what knowledge is needed to make it? These are questions for further work.

The position between code and design remains qualitative. A small source edit may contain a major design decision, while a long prompt may leave important decisions unspecified. Any measurement would need to examine the decisions and their supporting evidence, rather than count manually written lines.

Working from the Design Side

Codesign suggests a set of questions for examining generated work:

- Which design decisions does this code express?
- Which additional assumptions have entered the implementation?
- How has the interaction with the LLM changed the design?
- What observations would distinguish the intended behavior from a plausible substitute?
- What evidence could contradict my present interpretation?
- What must remain understandable when the implementation changes?
- Which version and intended use am I accepting, and what justifies my sign-off?

I can keep the answers, counterexamples, and diagnostic expectations with the code. This helps a later reader reconstruct the design and understand why I accepted a particular implementation.

Codesign describes my present work more accurately than a count of coding hours. I approach implementation more through design decisions, interaction with LLMs, and artifact examination.

Software Codesigner adds responsibility for deciding whether to accept the result. I may write fewer statements myself, but I still need to understand the important decisions and explain why I would sign off on the implementation.

References

- [1] Naur, P. (1985). Programming as theory building. *Microprocessing and Microprogramming*, 15(5), 253–261. [https://doi.org/10.1016/0165-6074\(85\)90032-8](https://doi.org/10.1016/0165-6074(85)90032-8)
 - [2] Reeves, J. W. (1992). What Is Software Design? C++ Journal. Reprinted in *Code as Design: Three Essays* by Jack W. Reeves, Developer Dot Star, February 23, 2005. https://www.developerdotstar.com/mag/articles/reeves_design.html
 - [3] Sanders, E. B.-N., and Stappers, P. J. (2008). Co-creation and the new landscapes of design. *CoDesign*, 4(1), 5–18. <https://doi.org/10.1080/15710880701875068>
 - [4] De Micheli, G., and Gupta, R. K. (1997). Hardware/Software Co-Design. *Proceedings of the IEEE*, 85(3), 349–365. <https://doi.org/10.1109/5.558708>
 - [5] Andersen, P. B. (1992). Computer Semiotics. *Scandinavian Journal of Information Systems*, 4(1), Article 1, 3–30. <https://aisel.aisnet.org/sjis/vol4/iss1/1/>
 - [6] Tanaka-Ishii, K. (2010). *Semiotics of Programming*. Cambridge University Press. ISBN 978-0-521-73627-5. Author's overview and publication details: https://www.u-tokyo.ac.jp/biblioplaza/en/F_00238.html
 - [7] Ross, S. I., Martinez, F., Houde, S., Muller, M., and Weisz, J. D. (2023). The Programmer's Assistant: Conversational Interaction with a Large Language Model for Software Development. *Proceedings of the 28th International Conference on Intelligent User Interfaces (IUI '23)*, 491–514. <https://doi.org/10.1145/3581641.3584037> Author manuscript: <https://arxiv.org/abs/2302.07080>
 - [8] Vostokov, D. (n.d.). Bridging The Great Divide. *DumpAnalysis.org*. <https://www.dumpanalysis.org/the-great-divide>
 - [9] Apple Inc. (2016). About Code Signing. *Code Signing Guide*. Updated September 13, 2016. <https://developer.apple.com/library/archive/documentation/Security/Conceptual/CodeSigningGuide/Introduction/Introduction.html>
 - [10] The Linux Foundation and its contributors. (2004, 2006). *Developer Certificate of Origin*, Version 1.1. <https://developercertificate.org/>
 - [11] Knuth, D. E. (1984). Literate Programming. *The Computer Journal*, 27(2), 97–111. <https://doi.org/10.1093/comjnl/27.2.97>
 - [12] de Souza, C. S. (2005). *The Semiotic Engineering of Human-Computer Interaction*. MIT Press. ISBN 978-0-262-04220-8. <https://mitpress.mit.edu/9780262042208/the-semiotic-engineering-of-human-computer-interaction/>
 - [13] Gulwani, S., Polozov, O., and Singh, R. (2017). Program Synthesis. *Foundations and Trends in Programming Languages*, 4(1–2), 1–119. <https://doi.org/10.1561/25000000010>
 - [14] Barke, S., James, M. B., and Polikarpova, N. (2023). Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1), Article 78, 27 pages. <https://doi.org/10.1145/3586030>
 - [15] Vaithilingam, P., Zhang, T., and Glassman, E. L. (2022). Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. *CHI Conference on Human Factors in Computing Systems Extended Abstracts*, 7 pages. <https://doi.org/10.1145/3491101.3519665>
 - [16] Feng, D., Yun, B., and Wang, A. Y. (2026). From Junior to Senior: Allocating Agency and Navigating Professional Growth in Agentic AI-Mediated Software Engineering. *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems*, 1–24. <https://doi.org/10.1145/3772318.3791642> Author manuscript: <https://arxiv.org/abs/2602.00496>
 - [17] Google. (n.d.). What to look for in a code review. *Engineering Practices*. <https://google.github.io/eng-practices/review/reviewer/looking-for.html>
 - [18] Google. (n.d.). The Standard of Code Review. *Engineering Practices*. <https://google.github.io/eng-practices/review/reviewer/standard.html>
 - [19] Merriam-Webster. (n.d.). Codesign. *Merriam-Webster.com Dictionary*. <https://www.merriam-webster.com/dictionary/codesign>
 - [20] Chlipala, A. (2026, May 5). Codesign for Legibility (to AI and Everyone Else). *LessWrong*; crossposted from *Structure and Guarantees*. <https://www.lesswrong.com/posts/yAkidKLg5Ckiu5A5b/codesign-for-legibility-to-ai-and-everyone-else>
 - [21] Picca, D. (2025). Not Minds, but Signs: Reframing LLMs through Semiotics. *arXiv preprint 2505.17080*. Version 3, revised March 27, 2026. <https://arxiv.org/abs/2505.17080v3>
- Web sources consulted October 6–7, 2026. Publication years refer to the cited works; later online reprints and author summaries are identified where applicable.