Silence inside Systems:
Roots and Generativity Consequences

Amrit Tiwana and Hani Safadi University of Georgia, Athens, Georgia 30602, USA; tiwana@uga.edu/ hanisaf@uga.edu

ABSTRACT

Silence as a system behavior has evaded attention in IS discourse, where interaction predominates. It merits examination as systems grow brittle, brute-force countermeasures falter, and autonomous systems risk misbehaving. We conceptualize silence as dynamic systemwide behavior, theorizing *how* it drives IS generativity by coalescing developers' attention on temporally-shifting slivers of a system's codebase.

Using a supercomputer-scale model, we analyzed nearly 20 million tweaks over a quarter-century in over a thousand systems, constructing 67,000+ evolving human-to-artifact networks to infer such behavioral dynamics. We show silence breeds generativity by dynamically funneling developers' attention.

We make three novel contributions. First, we introduce silence as a dynamic, systemwide behavior, rooted in system architecture. Second, we show silence's generativity consequences—halving degenerativity, doubling evolution speed, quadrupling releases, and spawning 600 times as many forks. Third, we uncover attention funneling as the mechanism linking silence to generativity.

Keywords: Attention, silence, IT evolution, IS generativity, chattiness, loose coupling, architecture, quietness, evolutionary dynamics, supercomputing methods, Miles Davis, Moore's law, Shikantaza, trace data, system behavior.

1. INTRODUCTION

Small tweaks can fuel giant leaps in systems. A new technology wave such as generative AI, a novel design philosophy like containerization, or an infrastructural innovation such as hyperscaling lets us extend systems in powerful new ways, fostering what scholars call IS *generativity* (Fürstenau et al. 2023; Vial 2023). While prior research emphasizes the architectural foundations of IS generativity (Rolland, Mathiassen and Rai 2018; Tiwana and Safadi 2024; Yoo, Henfridsson and Lyytinen 2010), systems' behavior—potentially just as crucial—has yet to receive much attention. A growing recognition that, as systems evolve, their behavior diverges from their original design puts this in the spotlight (e.g., Fürstenau et al. 2023; Lindberg et al. 2016).

A key aspect of a system's behavior is how much its parts interact with each other—what we envision as a system's "chattiness" or "silence." Silence, we contend, is an underappreciated trait of systems' behavior. We retrace silence to an overlooked nuance in Simon's (1962: 481) seminal work, where he coveted infrequent interactions among a system's parts ("silence"). Understanding this theoretically could unlock IS generativity. We initiate the conversation, asking: *How* and *why* does silence inside a system shape its generativity?

Brittleness, waning countermeasures, and autonomous systems drive the urgency to understand evolving systems' understudied behavior. First, systems grow brittle, as illustrated large-scale IT meltdowns increasingly precipitated by one tweak.[1] Evolution breeds unpredictable behaviors (Fürstenau et al. 2023; Lindberg et al. 2016; Rolland et al. 2018; Tiwana and Safadi 2024), often defying architectural expectations (Rahwan 2019). So, we might see chattiness where we expect silence, and silence where we expect chattiness. Second, the waning of historic countermeasures—Moore's Law hitting the limits of physics and bandwidth costs soaring (Lundstrom 2003)—aggravates challenges with chatty systems.[2] Netflix's 900 microservices illustrate this challenge. Brute force can demand deep pockets; Microsoft built a $400 million, 4,100-mile trans-Atlantic undersea cable for Office-365.[3] Increasing silence behavior inside systems might be an elegant alternative. Third, our nascent understanding of evolving systems' behavior (Rahwan 2019: 478) induces anxiety about AI-fueled autonomous systems; as anecdotes of ChatGPT hallucinating, Teslas running over pedestrians, Boeings crashing, smart munitions going haywire, and DaVinci surgeries going awry illustrate.[4] As metallurgist Smith (1981: 54) presaged, the essence of complex structures lies in how their parts interact. Understanding behaviors of evolving IT systems requires a uniquely-IS mindset where humans and IT artifacts are simultaneously front and center.

Software engineering overlooks this phenomenon, for three likely reasons. First, it prioritizes immediate operational performance (latency, congestion, defects, reliability) over long-term generativity (e.g., Abgaz et al. 2023; Jin et al. 2022). Second, it analyzes static code snapshots rather than systems evolving behavior (Rahwan 2019)—using a camera where we need a video recorder. Third, causal mechanisms—the "why" (Ashworth, Berry and de Mesquita 2021: 48)—lie outside its foci and repertoire (Rahwan 2019: 482).

<< INSERT FIGURE 1 HERE >>

Figure 1 illustrates our focal gaps: ① *how* silence shapes generativity and ② *why* (the blackbox mechanism). Resurrecting March and Simon (1958), we conceptualize *silence* inside a system as a systemwide behavior of minimal chattiness among its modules. Such silence, we theorize, promotes IS generativity by dynamically funneling teams' attention on narrow slivers of a system's codebase. While rooted in causally-upstream studies of IS architecture (the left side of Figure 1), our theory moves nomologically further along the causal chain.

We tested these ideas using a supercomputer to construct novel, computationally-intensive datastreams. We analyzed 274 million lines of code, 9.7 million dependencies, 19.8 million code changes across 194,105 modules in 1,356 open-source systems. We examined every human-artifact interaction as these millions of IT artifacts evolved over a quarter-century, constructing 67,000+ systemwide social-network measures to infer our constructs.

We advance IT evolution research in three ways. First, we introduce silence as a dynamic systemwide behavior, enlarging the architecture-dominated conversation. Second, we show that silence enhances IS generativity—spawning more forks, curbing degenerativity, and speeding evolution—while revealing the consequences of unexpected chattiness and silence. Third, we unblackbox attention funneling as a powerful mechanism that coalesces scarce developer attention into IS generativity. We found shifting between extremes doubles system quality and evolution pace, quadruples release cadence, and increases generative forking 600-fold. To software engineering, we contribute silence as a theoretical concept rooted in architecture, its long-term effects, novel evolutionary measures, and insights into how system behavior shapes developer-code interactions. Subsequent sections develop these ideas, methods, analyses, and our contributions.

2. THEORY

The gist of our forthcoming theory is: Silence inside a system—by dynamically funneling team attention—fosters IS generativity. Our unit of analysis is a system. Table 1 defines our key constructs and Figure 2 summarizes our model. Here, codebase refers to uncompiled source-code, while architecture—causally-upstream to our model's focus—encompasses the systemwide arrangement of modules, including their coupling, nesting, and scope in prior studies.

<< INSERT FIGURE 2 HERE and TABLE 1 >>

2.1. Foundation: Simon's Systems Theory

Our theoretical foundation is Simon's (1962) general systems theory. His theory applies across all varieties of systems—from biological and mechanical to social and organizational. His central insight is that organizing systems as collections of subsystems ("modules") that interact minimally with each other but are internally tight-knit makes systems more evolvable. Simon idolized interactions among modules so weak that they become nearly-independent.

Simon's ideas profoundly influenced modern IT systems (Rolland et al. 2018), making modularity a defining trait of modern systems (Chidamber and Kemerer 1994). Lost in the contemporary translation of Simon's theory is the nuance that Simon emphasized systems' internal *behavior* rather than just systems' architecture. Recent studies in both software engineering (e.g., Aiomar et al. 2024) and IS (e.g., Fürstenau et al. 2023; Lindberg et al. 2016) have rediscovered that systems' behavior often diverges from their architecture as they evolve.

Silence inside a system—which we conceptualize ground-up from its Simonian roots—is about the dynamic interaction *behavior* among modules constituting a system; it's distinct and separable from a system's architecture. To illustrate, consider a building's architecture. It can influence—but isn't deterministic of—how its occupants interact. Similarly, a system's architecture influences but doesn't dictate its silence behavior. We therefore conceptualize silence as a systemwide behavior that's rooted in, but not entirely determined by, its architecture. This focuses us on how systems actually behave, rather than how their architecture suggests they should.

2.2. Silence Inside a System

Our conceptualization builds on Simon's (1962: 481) notion of weak interaction behavior *among* a system's modules—fundamental to his idea of near-independence. His emphasis on *near* recognized that modules must interact somewhat to constitute a system, advocating for weak interactions among subsystems. In his posthumous work spanning broad classes of systems, Simon (2002) explicitly used the term "silence," a concept we adapt here specifically for IT systems. He reiterated minimizing interactions among modules, warning that "talk drowns out silence" (page 611).

This idea of limited interaction *behavior* among a system's modules is central to organizing systems yet has never explicitly received theoretical attention. However, we must first conceptually differentiate silence from adjacent concepts, demonstrate its unique value, and build it from the ground up in ways cumulative on related work.

2.2.1. Adjacent concepts

Three architectural concepts—loose-coupling, accidental coupling, and architectural drift—are nomologically-proximate but conceptually-distinct from our conceptualization of silence. They're architectural properties, while silence is a system's understudied behavior. Architecture can influence

behavior, but it does not, by itself, constitute behavior. Our model preserves this crucial distinction while assimilating these causally-upstream insights. Table 2 summarizes these.

<< INSERT TABLE 2 HERE >>

a. Loose-coupling. Silence is nomologically proximate yet differs conceptually from loose-coupling (e.g., Subramanyam, Ramasubbu and Krishnan 2012). While loose-coupling describes structural independence among modules, silence characterizes their actual interaction behavior.[5] Loose-coupling circumscribes—nondeterministically—interaction behavior among modules. Loosely-coupled modules can be chatty (e.g., in REST APIs or event-driven systems), while tightly-coupled modules can be silent (e.g., error-handling modules that rarely activate). Consider social platforms like Slack, streaming services like Spotify, and cloud systems like Gmail—all feature loosely-coupled modules that exchange high volumes of data through APIs.

Conversely, automotive systems contain tightly-coupled modules that run briefly at startup then remain silent.[6] As a system evolves, loosely-coupled modules might become chatty and tightly-coupled modules might go silent. Loose-coupling makes silence more likely but doesn't guarantee it. This positions loose-coupling causally-upstream to silence in our theory in Figure 1. Only Goren (2020) previously used the term "silence," but differently—to formalize a theorem for information transfer through non-communication.[7]

b. Accidental coupling. Silence differs from "accidental coupling" (Abgaz et al. 2023) or "possible dependencies" (Jin et al. 2022), which emerge from unrecorded *structural* dependencies from dynamic-typed code and design violations. In dynamic-typed code, the variable type is determined at run-time rather than compile-time; we eliminate this confound by focusing exclusively on a static-typed language.[8] Design violations occur when implementation actions deviate from design intentions—for example, when expedient shortcuts and patchwork across modules create technical debt (Rolland et al. 2018).[9] Such structural coupling is an architectural attribute that can lead to reducing silence but itself is not silence—a downstream *behavior*. Refactoring curbs such architectural degradation (Aiomar et al. 2024), controlled in our analysis.

c. Architectural drift over time. Recent IS work shows systems' behavior diverging from architectural expectations, revealing unexpected module interactions (Lindberg et al. 2016), and new or vanishing interactions as they evolve (Fürstenau et al. 2023; Tiwana and Safadi 2024). As systems drift, silent codebases may grow chatty, and chatty ones silent.

While silence shares conceptual space with these adjacent concepts, as a behavior rather than an architectural property they represent, it provides a new lens for how systems actually behave rather than how their architecture suggests they should.

2.3. Conceptualization of Silence

Literally, silence means refraining from talking. Building on Simon, we define *silence* as the absence of observable interaction behavior among a system's modules. In software, silence manifests as reduced traffic—e.g., fewer calls and lesser data flow—among its modules. Practitioners implicitly advocate for silence when advocating that systems minimize data passed between modules (e.g., Richards and Ford 2020: 127). The opposite is a chatty system, exhibiting extensive interaction behavior among modules to achieve system functionality (Richards and Ford 2020: 110). Silence and chattiness form a continuum, with silence being a location on it.

Silence is a *systemwide* property that describes the interaction behavior among its modules. Unlike interaction among human developers, interaction among modules entails one module invoking functionality in other modules or passing data to them. Since one module might invoke another but not the other way around, silence halves when an interaction between two modules goes from unidirectional (○→○ call) to bidirectional (○↔○ call).[10] As silence increases, interactions among modules progressively decrease.

Figure 3 contrasts two systems with four modules (○), one more silent (a) and one chattier (b). In realizing the system's functionality, modules in the silent system invoke each other far less than in the chattier one. A system's architecture can affect silence—but nondeterministically—as it evolves over time. Consequently, as a system evolves, we can have a chatty system despite loosely-coupled modules or a silent system despite tightly-coupled modules.

<< INSERT FIGURE 3 HERE >>

2.3.1. Java code illustration

Figure 4 illustrates this idea using two modules (*p1* and *p2*), containing classes *A* and *B,* with their code underneath (files *A.java* and *B.java*). Cell A shows two uncoupled modules exhibiting silence as expected. Cell C is two tightly-coupled modules that are chatty as expected. Here *A* invokes *B* and module *p1* invokes *p2* using a Java *import* statement. Behavior deviates from architecture in cells B and D. Cell B shows loosely-coupled modules being unexpectedly chatty because class *A* uses *B* without an explicit module-level

invocation (import). Cell D shows tightly-coupled modules being unexpectedly silent, as *p1* invokes *p2* through an import statement, but this import is unmaterialized in actual class usage. This architecture-behavior divergence is more apparent in systems that have evolved over some time, which is why we subsequently examine such systems.

<< INSERT FIGURE 4 HERE >>

2.3.2. Silence's conceptual value

Silence's unique theoretical value-added is two-fold. First, it directly taps into evolving, observable system behavior, which software engineering scholars describe as understudied and atheoretical (Jin et al. 2022; Rahwan 2019). The urgency to understand the evolving behavior of systems is because they become brittle over time, distributed designs make brute-force countermeasures untenable, and autonomous systems heighten concerns about unpredictable behaviors. Second, it rekindles Simon's (1962) underappreciated emphasis on silence as a system behavior. Like Simon, Smith (1981: 54) also emphasized that grasping a complex artifact demands understanding the interaction behaviors within it; these change over time as IT systems evolve. Our conceptualization acknowledges architecture's essential yet nondeterministic role in shaping system behavior, broadening the theoretical discourse beyond architecture that predominates, while cumulatively building on it. This later allows counterfactually exploring the "what-if" generativity consequences of chattiness where silence is expected and silence where chattiness is anticipated.

2.4. Attention Funneling

Attention literally means focusing the mind on something. Simon defines attention as the set of elements that enter into consciousness at any given time; elements to which effort is directed (Van Knippenberg et al. 2015; Tonellato et al. 2024). Funneling literally means to channel to a focal point. Attention funneling, then, is channeling of the team's effort to some subset of the systemwide codebase, such as a module.

We define *attention funneling* as the degree to which a project team temporarily concentrates its code changes on fewer modules of a system. While modularity enables *individual* developers to focus on a specific segment of a system's codebase, attention funneling dynamically coalesces *teamwide* effort on a narrow sliver such as a module or a file. This transient concentration shifts over time to different slivers of the codebase. While silence characterizes how code artifacts behave, attention funneling characterizes how team members collectively behave.

Our conceptualization differs from attention concepts in management and software engineering by being both dynamic—as attention shifts between modules from one period to the next—and inward-focused within teams.[11] Like Harvey's (2014: 332) emphasis on dynamic team processes, attention funneling evolves over time to create what she describes as deep collective engagement, reflecting different knowledge types' distinct attention needs (Zhang et al. 2025). It's a team-level analog to March and Simon's (1958: 154) idea of narrowing individuals' attention spans, recognizing that attention operates simultaneously at individual and team levels (Zhang et al. 2025). Unlike outward-focused narrowing of attention of organizations in management, inward narrowing of attention of teams is akin to switching from a wide lens to a macro lens.

Funneling attention translates into developers collectively concentrating effort on fewer modules, while its opposite—attention scattering—splinters effort across the codebase. Figure 5 contrasts this using two three-person (■) teams working on three modules (○) in a codebase. The team on the left (a) shows funneled attention with all members focused on one module while the one on the right (b) has scattered attention. Attention funneling mirrors the individual-level Zen practice of *Shikantaza* (只管打坐) at the team level; a heightened state of concentrated awareness. It characterizes human developers'—not a system's modules—collective behavior.

<< INSERT FIGURE 5 HERE >>

2.5. Hypotheses

2.5.1. How attention funneling shapes IS generativity

IS generativity broadly refers to a system's capacity for doing new things (Vial 2023). Recent IS studies conceptualize generativity as emerging both from and within a system, although they diverge on whether agency resides with external third parties or the original developers. *From-system* generativity refers to a system spawning innovations via the uncoordinated third-party actors independent of its originators (Fürstenau et al. 2023; Yoo et al. 2010), such as when forking creates derivative systems. Systems can also experience degenerative change that degrades quality (Fürstenau and Baiyere 2019). *Within-system* generativity encompasses internal changes, measured through its rate of change and new version releases (Tiwana 2015; Vial 2023). We'll therefore cast a wide empirical net spanning all four conceptualizations of generativity from (forking and degenerativity) and within (evolution rate and releases) systems.[12]

Recent IS generativity studies suspect that misallocating attention has repercussions, emphasizing cues for developers to focus future effort (Fürstenau and Baiyere 2019: 1322). Our focus on attention

funneling mechanism stems from the cognitive load of tracking module interactions on individual developers' finite attention as they coalesce development effort. Realizing IS generativity thus depends heavily on how teams allocate their attention across the system. When individual developers' attention is scattered across multiple modules, consensus-building on future development priorities becomes difficult (Fürstenau and Baiyere 2019; Tonellato et al. 2024). This dispersion of effort impedes the team's ability to realize focused, high-quality code changes within discrete modules during each development period. Consequently, the team struggles to produce sufficiently valuable code for generative extension by other developers (Vial 2023), while simultaneously risking codebase degradation

Conversely, when all developers in a team temporarily concentrate their effort on fewer modules, they make more tangible progress. Allocating attention drives groups' productivity (Van Knippenberg et al. 2015). Team fixation on specific codebase slivers indicates consensus on development priorities, increasing the likelihood of realizing envisioned code changes (Tonellato et al. 2024). This focused development effort leads to higher-quality, release-worthy code that other developers can build on, while also curbing codebase degradation—both key aspects of from-system generativity. Coalesced teamwide effort enables expedient implementation of code changes and fulfillment of future development commitments (Van Knippenberg et al. 2015), improving both the evolution rate and release frequency aspects of within-system IS generativity (Tiwana 2015; Richards and Ford 2020: 116). Overall, funneling of teamwide attention helps extend systems' functionality beyond their original design. We therefore expect it to enhance overall IS generativity.

Hypothesis 1: *Attention funneling enhances IS generativity.*

2.5.2. How silence shapes teamwide attention

With greater silence among a system's modules, changing one module requires fewer simultaneous adjustments elsewhere (Simon 1962). It frees up team members' scarce attention that would otherwise be allocated to tracking modules' interactions (Fürstenau and Baiyere 2019: 1322; Harvey 2014)). Zhang et al. (2025) showed that different types of knowledge require different attention patterns; and silence enables teams to match their cognitive focus to development priorities, which is difficult in chatty systems where multiple module interactions compete for attention.

With higher silence, a team can focus collectively on one salient module at a time in relative isolation, achieving Harvey's (2014: 332) deep collective engagement, without needing time-consuming

iteration to preserve interoperability with other modules. This allows the whole project team to temporarily focus its development effort on a sliver of the codebase (Van Knippenberg et al. 2015; Tonellato et al. 2024). Individual developers in the project team can thus pay more attention to fewer modules on average, consistent with the attention allocation ideas developed in §2.4.

In contrast, a chattier system requires tracking how even a single change in one module affects other modules in the system's codebase. Multiple modules then simultaneously compete for team members' scarce attention (Simon 2002), scattering team attention. We therefore expect greater silence among the system's modules to enhance the funneling of its project team's attention.

Hypothesis 2: *Silence inside a system enhances funneling of its project team's attention.*

2.5.3. Mechanism

Silence breeds IS generativity by shaping how teams collectively engage with the system's codebase. It creates conditions to coalesce teamwide attention (Fürstenau and Baiyere 2019; Van Knippenberg et al. 2015), which in turn improves generative outcomes. Since attention operates simultaneously at individual and team levels (Zhang et al. 2025), attention funneling serves as the connective mechanism to coalesce individual developers' attention into team-level effort concentrated on one sliver of the codebase at a time. Rather than directly causing generativity, silence influences team attention allocation patterns (§2.5.1), which subsequently foster generative outcomes (§2.5.2). Thus, attention funneling mediates the effect of silence on IS generativity.

Hypothesis 3: *Attention funneling mediates the influence of silence on IS generativity.*

3. METHODS

3.1. Data Collection

We used bleeding-edge trace analysis with data streams collected as part of the OKRA initiative. Open Knowledge Relics Archaeology (OKRA) is a larger, decade-long research program studying IT evolution using open-source digital relics using supercomputing-driven induction as a form of digital archeology (Tiwana and Safadi 2024). Trace-data are longitudinal, field data collected in source-code management systems over the course of development activities. We assembled time-series records spanning each system's lifespan (Salganik 2017: 26). Like frames in a video, these quarterly snapshots captured both module-to-module and developer-to-module interaction behaviors. This approach enabled the type of causal inference Rahwan et al. (2019: 478) advocated to understand interaction behavior inside systems.

To mitigate confounding due to programming languages, dynamic-typing, and heterogeneity in how systems' modules interface, we focused exclusively on Java-language systems. Such narrowing of context permits more precisely disentangling the mechanism at work (Ashworth et al. 2021: 6). Modules (called *packages*) in a Java system interact in highly standardized ways.[13] Java uses static typing, thus mitigating confounding by dynamic-typing (wherein the type of a variable is determined at runtime rather than at compile-time such as in Ruby and Python) that impedes maintainability (Aiomar et al. 2024).

From Github's 46.7 million projects database, we randomly sampled 1,356 Java projects that met four winnowing criteria: (a) coded in Java, (b) actively developed (defined as at least a monthly code commit over the preceding five-year window), (c) original (non-forked) systems, and (d) non-private, open-source repositories. Using Github standardized the development environment; while open-source permitted constructing quarter-century data streams implausible with proprietary systems. We analyzed in each system's source-code, every line of code that ever existed, every change, the location of every file and every module in each system it affected over a quarter-century (1996-2016 plus another five years for our generativity outcomes).[14]

<< INSERT FIGURE 6 HERE >>

Using a supercomputing cluster, we analyzed 274 million lines of source-code and comments spanning 1.4 million files in 194,105 modules, with 9.7 million dependencies, 19.8 million commit records, and 7,200 developers—approximating 163,000 issues (40,000 years) of *ISR*. We assembled quarterly snapshots into 1,356 time-series trace data streams. This balances observing silence with preempting confounds, while remaining computationally feasible since a quarterly window consumed several weeks of supercomputer processing time. We wrote a custom source-code analyzer to extract different networks from each project and used the NetworkX Python package (networkx.org) to construct a triad of 22,534 networks—systemwide module-to-module for silence, developer-to-module linkages for attention, and file-to-module for instruments—for every system spanning 83 quarters. Following Cunningham (2021: 94), this aggregates granular microdata to the higher, system-level; as trace datastreams comprised of a long time-series of short observation intervals.

3.2. Measures

We measured all constructs using data accreted from reconstructing for every observation interval the digital fingerprints developers left behind in their code changes over each system's lifetime. We leveraged Java

systems' organization as modules, each with a unique folder containing *.java* source-code files. Dependencies among two files in different folders are module-to-module dependencies. We focused only on system-native modules, excluding imports from the Java library and external modules. We analyzed every line of this source-code for each system in every quarter, which was the input for our novel social-network measures unlike any used in software engineering. Table 1 summarizes the measures for our theoretical constructs, instruments, and controls.

Silence. For *silence*, we constructed 22,534 codebase silence networks of every module-to-module invocation across a system's codebase, one for each system for every quarter. We focused on the two primary object-oriented module-to-module dependencies, class composition and class inheritance. A file in a Java codebase represents at least one class. If one class invokes or inherits from another class, an interaction dependency exists between the two files of the two classes. We tallied dependencies between files in different modules to compute module-to-module chattiness. We also robustly replicated all models at a more granular file-level using method calls between files rather than modules.[15]

Following Figure 3, silence equals one minus the density of the module-to-module interaction network for an entire system's codebase. This converts density to sparsity, transforming chattiness to silence. In Java, one directed link from module A→B exists when module A invokes (in Java lingo, "imports") module B. If it also invokes in reverse (A←B), it adds another link. We aggregated this count of module-to-module interactions for each system's native modules in every quarter to estimate density. The lower this density, the higher the silence among a system's modules. It ranges from zero (where every possible module-to-module interaction exists) to 100% (where no modules talk). Aggregating this creates a systemwide measure for the intensity of module-to-module interaction behavior. Unlike software engineering's static, granular file/module measures, our novel social-network analyses of code artifacts capture dynamic systemwide behavior of evolving modules' interactions; complementing recent IS work on architectural changes over time (e.g., Tiwana and Safadi 2024)

$$Silence_t = \{1 - \{(\#\text{module-to-module invocations}_t) \div ((\#\ \text{modules}_t) \times (\#\text{modules}_t - 1))\}\}$$

Figure 7(a) visualizes silence of one sampled system—*autopsy* (github.com/sleuthkit/autopsy)—in one quarter (last quarter of 2011); the hollow circles (○) are modules; the arrows (→) are calls among modules (○). It had a high silence (0.7) in this observation window.

<< INSERT FIGURE 7 HERE >>

The appendix illustrates our strategy for inferring silence from the Java source-code of an enterprise application with three modules. It also shows how silence and loose-coupling can vary independently. The code snippets and the figure highlight the relationship between method invocation and import statements. To compose the functionality of the application, some modules need to invoke code from other modules using "import" statements we used to measure the intensity of module-to-module interactions.

Attention funneling. We measured *attention funneling* as developers' focus on fewer modules in each quarter, assessed using systemwide developer-to-file networks whose links represent code commits. The fewer the modules on which their commits are concentrated in an interval, the more funneled is the team's attention.[16] We then estimated this network's Freeman centralization, which is the difference between the most central node and other nodes in this network of commits each quarter, measuring teamwide attention. These developer-to-module interactions evolve across observation windows, tapping into evolving human developer-with-code artifact interactions.

$$\textit{Attention funneling} = \{\sum_{d \in developers}(\max_{v \in developers}(centrality(v)) - centrality(d))\} \div \#\text{developers}$$

Higher centralization indicates developers collectively focusing commits to fewer modules, reflecting teamwide attention funneled to fewer modules in a system's codebase. Like our silence networks for module-to-module interaction behaviors, we constructed 22,534 such attention funneling networks, each capturing every human developer-to-file relationship from each system's codebase commit history list for every quarter.

Figure 7(b) visualizes attention funneling in the same sample project as Figure 7(a). It had low attention funneling (0.2) as indicated by the arrows (→) representing code commits by different developers (■) to files in different modules (○). Thus, developers' code changes were concentrated on fewer files (which are nested in fewer modules in a Java codebase). To recap, silence means less aggressive invocation behavior across modules in a codebase; attention funneling means commits went to fewer modules.

Generativity. The narrowest conception of generativity is a system changing through unfiltered contributions from broad, varied audiences, while the broadest conception encompasses external, third-party modifications, technical improvements, and resistance to degenerative forces. The literature is also theoretically ambiguous about whether generativity stems from third-party modifications or from original developers actions. Reflecting this ambiguity, we used four different measures to cover the entirety of plausible generativity

manifestations. The first, most direct one was the lifetime-cumulative count of codebase forks measured five years beyond t_3 (the end of the first quarter of 2021) (*codebase generativity*) (Fürstenau et al. 2023; Vial 2023). Developers who fork a codebase as a foundation for derivative systems are the users of the codebase itself, analogous to end-users of a compiled system. Following Fürstenau et al.'s (2023) logic for growth in a system's users as generativity, we counted lifetime forks to measure codebase generativity. Of our four metrics, our stance is that the true hallmark of IS generativity is such developer-independent extension by external third-parties, which cleanly separates it from other evolutionary outcomes. We complemented this with a lagged measure of codebase quality degradation to measure *degenerativity* (Fürstenau and Baiyere 2019), using a lagged count variable of the number of cyclomatically-complex lines of code at t_3. These covered from-system generativity. We complemented these with within-system generativity quantified as change internal to a system's codebase as *evolution rate*, measured at t_3 as the count of quarterly code commits. This follows Vial's (2023) emphasis on system adaptation. Since more code changes alone might not suffice to produce release-quality outcomes, we complemented it with *release cadence*, measured five years after t_3 as the cumulative lifetime count of new releases of the whole system (Richards and Ford 2020: 116). Generativity and release cadence were lifetime-cumulative measures lagged for an additional five years beyond t_3; evolution rate and cyclomatic complexity were measured quarterly (t_3).

Instruments. We constructed our three instruments—loose-coupling, hierarchical depth (Chidamber and Kemerer 1994), and module coarseness (Subramanyam et al. 2012)—also by analyzing each system's codebase every quarter. These are each system's architectural properties—its modules' coupling, nesting, and scope—discussed later in §4.1, which Figure 10 visually illustrates.[17] They're time-variant, systemwide properties recomputed for every quarter, creating 22,534 networks similar to the silence and attention ones but using file-to-module linkages. This mitigated bleeding into our silence measure. We measured *loose-coupling* as the sparsity of the file-to-module network reflecting the boxes-within-boxes metaphor with near-independence between the boxes. In Java, files representing classes and interfaces are organized within packages representing modules. The sparsity of this containment network indicates that a system is composed of relatively independent modules (range: 0-1). *Hierarchical depth* was the average shortest path length in the file-to-module network in each system's entire codebase in every quarter; larger values indicate a deeper codebase hierarchy. *Module coarseness* was the centralization of the file-to-module network. High

centralization implies that few modules implement most functions on which various classes depend; it ranged from 0.5 to 12.15 in our non-normalized data.

3.3. Descriptive Statistics and Patterns of Attention and Silence over Time

< INSERT TABLE 3 HERE >

Table 3 shows low correlations among loose-coupling, silence, and attention funneling in our panel data, confirming their distinctiveness (instruments are shaded). Systems in the fourth quarter of 2016 averaged 4.75 (σ2.96) years in age, with 6 (σ8.9) developers making 537 (σ 1,106) quarterly changes to 241 (σ 465) files. Systems had 1,181 (σ1,611) files organized into 143 (σ 193) native modules with 7.44 (σ13.2) million lines of code and documentation; their modules invoked each other 7,190 (σ10,377) times. As of 2021, lifetime fork counts averaged 323 (σ1,456), with 7.9 (σ11.8) major releases. Stationarity tests using the *xtunitroot* procedure in Stata confirmed that at least one panel is stationary. Codebase files to which attention was funneled changed 90.7% (σ11.6%) from one quarter to the next, confirming highly dynamic attention funneling. The shifts over time in median silence and attention funneling in Figure 8 show that they go hand in hand.[18] Silence degraded as a system crossed the 15-year mark (60 quarters), scattering attention along with it. We speculate that many systems reached their end-of-life in about 15 years.

< INSERT FIGURE 8 AND 9 HERE >

The cross-correlogram in Figure 9 further shows that silence precedes attention funneling by 1-3 quarters, with no reverse relationship. This underscores our causality argument that prior silence affects the current attention funneling; and attention today is not associated with future silence. Granger-causality tests confirm that silence causes attention funneling (χ^2 5.21, p<.001) but attention funneling does not cause silence (χ^2 .063, n.s.). Further, silence consistently precedes attention changes, this pattern is robust under alternative specifications. Silence and attention funneling are therefore causally-related.

3.3.1. Rival explanations

Our fixed-effects model holds constant time-invariant system and team properties, which are the bulk of rival explanations in prior studies. We controlled for time-variant (a) code volume (Subramanyam et al. 2012), (b) module count, (c) system age (Fürstenau and Baiyere 2019), and (d) team size from prior studies. We reestimated them for every quarter. *Module count* is Simon's (1962) span of a system, which is the number of subsystems a system's codebase is partitioned into. Greater *code volume* (measured as the logged

codebase line count (Subramanyam et al. 2012)) has more plausible instances for code change, thus higher generativity. With increasing *system age*, accumulation of technical debt and changes in its surrounding infrastructure impede changes (Rolland et al. 2018), lowering generativity. More developers actively changing the codebase (*team size*) provides more manpower to refine code, for which we counted the number of unique developers committing code in each quarter. Finally, rationalizing the codebase through *architecture refactoring* (Aiomar et al. 2024) (proxied as the ratio of files that differ from the preceding quarter[19]) can reduce its complexity to bolster generativity. This accounts for recent software engineering advances in accidental coupling and possible dependencies (e.g., Abgaz et al. 2023; Jin et al. 2022). Stage 3 in Table 4 shows that four of the five controls were significant.

4. ANALYSIS

Our unit of analysis is a system. Our trace-data are longitudinal; criterion variables are count-based; and our predictor is likely endogenous. To isolate the effect of silence on attention funneling, we accounted for the endogeneity of silence and for time-variant rival explanations of generativity. Our model's focus on the effects of within-system silence on generativity makes within-system analysis appropriate.

Three tests led us to fixed-effects using heteroskedasticity-mitigating robust standard errors (Cunningham 2021: 77). A significant Hausman test (χ^2 2.77; p<.001) led us to fixed-effects over random-effects, simultaneously eliminating time-invariant rival explanations. Our data was heteroscedastic (χ^2 = 1.5e+06; p<.001; rejected null of homoskedasticity) and autocorrelated ($F_{Wooldridge}$ 19.76; p<.001; rejected the null of zero autocorrelation).

We used a three-stage model using *xtivreg2* in Stata 16: (1) the 2SLS *silence model* with instruments predicting silence, (2) the 2SLS *attention model* using silence from Stage 1 predicting attention funneling, and (3) the negative binomial *generativity model* using predicted attention from Stage 2 plus controls. Table 4 shows the results.

<< INSERT TABLE 4 HERE >>

4.1. Stage 1 (silence model): Endogeneity of Silence inside a System

Silence is likely endogenously shaped by a system's architecture, reflecting its architectural roots. Rahwan (2019: 480) singled out systems' architecture—the internal arrangement of its modules by its designers—affecting its behavior. Lacking empirical precedent, we used three architectural system attributes in Figure 10

as theoretically-guided instruments: ① loose-coupling, ② hierarchical depth, and ③ module coarseness. They address omitted variables bias crucial for causal inference (Cunningham 2021: 11), while recognizing the causally-upstream roots of silence in systems' heteromorphic architectures. This approach assimilates prior architectural studies (e.g., coupling, nesting (Chidamber and Kemerer 1994), and granularity (Subramanyam et al. 2012)) into our downstream systems behavior. *Loose-coupling* among a system's modules increases silence by lowering modules' need to interact. *Hierarchical depth*—how many layers deep the module-to-subordinate module relationships run—ranges from flat to deep hierarchies (Simon 1962: 468) (depth of trees in code hierarchy in Chidamber and Kemerer 1994). Flatter hierarchies make most modules peers in a system's hierarchy, requiring greater interaction (Simon 1962: 469); and deeper hierarchies thus increase silence. Finally, as Figure 10 illustrates, a system can have a mix of smaller, fine-grained modules and larger, coarse-grained modules. *Module coarseness*—how large the average module in a system is (Subramanyam et al. 2012)—mirrors Simon's (2002) notion of a system being comprised of larger subsystems. Larger modules contain more functionality internally, reducing cross-module traffic compared to smaller modules that must interact more with other modules to accomplish system functions (Richards and Ford 2020: 110). Therefore, silence increases with coarser modules. Instruments should influence silence but not directly affect attention funneling. Our instruments satisfy this conceptual requirement because they affect how modules with a system interact but are unlikely to *directly* affect how members within the project team direct their attention. Equation (1) shows the Stage 1 model. Stage 1 in Table 4 shows that all three instruments were significant.

$$\text{Silence}_{t1} = \alpha_0 + \alpha_1 \text{loose-coupling}_{t0} + \alpha_2 \text{hierarchical depth}_{t0} + \alpha_3 \text{module coarseness}_{t0} + \varepsilon \ldots\ldots\ldots (1)$$

<< INSERT FIGURE 10 HERE >>

4.1.1. Model identification

Our instruments satisfied the *relevance criterion* of high correlations with silence ($r_{\text{loose-coupling}} = .36$; $r_{\text{hierarchical depth}} = .51$; and $r_{\text{module coarseness}} = .45$; all $p<.001$) and the *exclusion restriction* of low correlations with attention ($r_{\text{loose-coupling}} = .08$; $r_{\text{hierarchical depth}} = .12$; and $r_{\text{module coarseness}} = .07$; all $p<.001$). The large F-statistic (286 >10; $p<.001$; Stage 1) showed no instrument weakness, while Kleibergen-Paap ($\chi^2 = 697.71$; $p<.001$), Cragg-Donald ($F_{\text{Wald}} = 850.16$; $p<.001$), and Sargan-Hansen ($\chi^2 = 0.27$; ns) tests confirmed proper identification, strong instruments, and valid overidentification.

4.2. Stage 2: Attention Funneling Model

Stage 2 (equation 2) used predicted silence from Stage 1 to estimate attention funneling, thus accounting for systems' heteromorphic structures known from prior studies.

$$\text{Attention\_funneling}_{t2} = \beta_0 + \beta_1 \text{silence\_}\hat{y}_{t1} + \varepsilon \quad \ldots\ldots(2)$$

The significant, positive effect of silence on attention funneling (β = .48, T-value = 8.00, p<.001) highlighted in Stage 2 in Table 4 supports Hypothesis 2. Effect sizes are also crucial in drawing causal inferences in such voluminous data (Cunningham 2021: 51). In our Stage 2 linear regression model where silence and attention have a 0 to 1 range, the effect size of .48 ~ .5 indicates that each unit increase in silence increases attention funneling by ½ unit.

4.3. Stage 3: Generativity Model

Our criterion variables are count variables, for which both negative binomial regression (NBR) models and simpler Poisson models are plausible. (Codebase generativity and release cadence are non-panel, lifetime-cumulative count variables; and degenerativity and evolution rate are panel, count variables.) Their dispersion parameter $\alpha>0$ indicates over-dispersion, for which a NBR model is more appropriate. It was 3.37 (T= 29.3; p<.001) for codebase generativity, 0.5 (T= -4.85; p<.001) for degenerativity, 1.31 (T =170; p<.001) for evolution, and 5.6 (T =32.5; p<.001) for release cadence, leading us to NBR models.

Stage 3 (equation 3; used for all four criterion variables) estimated generativity using NBR fixed-effects models, also including controls discussed in §3.3.1. Stage 3 of Table 4 shows incidence rate ratios (IRRs) appropriate for NBR models in lieu of conventional regression coefficients. IRRs are exponentiated coefficients, so negative coefficients become IRRs ≤1 and positive ones become IRRs >1.

$$\text{Generativity}_{t3} = \gamma_0 + \gamma_1 \text{attention\_funneling\_}\hat{y}_{t2} + \{\gamma_2 \text{code volume}_{t2} + \gamma_3 \text{module count}_{t2} + \gamma_4 \text{system ag}_{t2} + \gamma_5 \text{team size}_{t2}\} + \gamma_6 \text{silence\_}\hat{y}_{t1} + \varepsilon \quad \ldots\ldots(3)$$

Attention funneling significantly increased codebase generativity in Table 4's Stage 3 (IRR = 616.87; T = 6.36; p<.001); reduced degenerativity (IRR = .50; T = -4.85; p<.001); enhanced the system's evolution rate (IRR = 1.98; T = 5.08; p<.001); and increased release cadence (IRR = 3.68; T = 2.08; p<.05). This strongly supported Hypothesis 1 across all four metrics of IS generativity. Significant mediation tests using the results from Stages 2 and 3 showed that attention funneling completely mediated the effect of silence on codebase generativity (T_{Sobel} = 4.98; T_{Aroian} = 4.94; $T_{Goodman}$ = 5.02; all p<.001). Similarly, mediation tests were significant across degenerativity (T_{Sobel} = 4.15; p<.001), evolution rate (T_{Sobel} = 4.29; p<.001), and release cadence (T_{Sobel} = 2.01; p<.05). Hypothesis 3 was therefore supported across all four generativity metrics.

4.3.1. Effect sizes

The exponentiated-form IRRs in our Stage 3 NBR models are the number of events associated with a unit of increase in the response variable. They show the effect of moving from completely-scattered attention (zero) to perfectly-funneled attention (one). (Attention funneling IRR is the rate ratio for a one unit increase in attention funneling at t_2, as predicted by silence in t_1.) Codebase generativity's IRR of 616.8 indicates that one unit increase in attention (i.e., moving from completely-scattered to perfectly-funneled attention) leads to 617 times more ports. Degenerativity's 0.5 IRR indicates that one unit increase in attention funneling halves the number of cyclomatically-complex lines of code i.e., doubles the system's code quality. Evolution rate's 1.98 IRR implies that perfectly funneling scattered team attention roughly doubles quarterly commits. Release cadence's 3.6 IRR indicates that this nearly quadruples future releases of the system. In summary, fully focusing a scattered-attention team doubles system quality and evolution rate, quadruples release cadence, and generatively propagates its codebase for derivative systems by over 600 times more.

4.3.2. File-level reanalysis for silence and attention funneling

We replicated all models using file-to-file interactions focused on method calls instead of module-to-module interactions. High correlations of file-to-file with module-to-module measures ($\rho = .81$, $p<.001$) signaled what our reanalysis subsequently confirmed—they're tapping into the same underlying construct. The silence→attention effect was robust across all models using the instruments from Table 4 as well as different ones subjected to more stringent tests appropriate for the different measurements. We replicated the Stage 3 NBR same-instruments and different-instruments models respectively with three and two of the four criterion variables in Table 4, given the nature of the data. Their relationships were robust across all analyses, with only release cadence showing weaker significance (one-tailed) in the same-instruments model. This assures robustness, irrespective of computing silence and attention across a system's modules or files.

The effect of silence on attention funneling held up ($\beta = .28$, $T = 2.89$, $p<.01$) when we instead control for the three architectural attributes using a two-step model. Thus, silence impacts attention funneling above and beyond architecture. Attention funneling effects on the generativity variables also held up (one-tailed $p<.05$ to .001); except for evolution rate being untestable due to model non-convergence.

4.4. Five Limitations

First, while attention funneling provides one explanation, alternative mechanisms merit consideration. For example, simpler systems might naturally exhibit both silence and generativity; and team expertise could shape developer-code interactions independently of silence. Second, our findings from open-source systems may not generalize to closed-source systems. Third, trace-data, lags matching our model's causal order, and Granger causality does not truly establish causality. Fourth, our instruments reflect temporal snapshots of actual architecture but have no baseline for what the intended architecture might have been at each system's inception. Fifth, using a single language mitigates cross-language confounds but bounds generalizability.

5. DISCUSSION

IS theory—rapt with communication—has been dead silent about silence, which is a central pillar of Simon's theory. This behavior merits attention as systems grow brittle, brute-force plateaus, and autonomous systems misbehave. While software engineering prioritizes operational performance and static code properties, IS must examine long-term dynamics and causal mechanisms. This initial foray examined how and why silence in a system shapes IS generativity.

We theorized that silent systems achieve higher generativity *because* they coalesce IS teams' attention on littler parts of a system. Our novel insights are into *how* evolving codebase behavior drives IS generativity by shaping the behavior of humans maintaining it. Our theory's novel attention funneling mechanism focuses on human-with-artifact dynamics, simultaneously bringing human developers and IT artifacts to the forefront.

To test these ideas, we constructed novel social-network-theoretic measures from our datastreams. We analyzed using supercomputing methods 274 million lines of source-code in ~190,000 modules in 1,356 systems as they underwent nearly 20 million changes over a quarter-century. From this, we constructed over 67,000 human-to-artifact and artifact-to-artifact social networks to uncover dynamics of behavior as these systems evolved over time. Our empirical findings support our theory, with the temporal sequence matching our causal chain, effect sizes suggesting practical significance, and robust patterns across diverse projects over time.

We introduced silence as a system behavior distinct from architecture, show its IS generativity consequences; and unmask the underlying mechanism of attention funneling. Silence reflects the dynamic

interaction behaviors among modules, complementing the predominant architectural lens for understanding systems' evolution. Our attention-centric explanation of how silence inside systems shapes IS generativity makes three novel theoretical contributions.

5.1. Contributions and Implications

5.1.1. Theoretical notion of silence inside systems

Our novel theoretical notion of silence inside systems enlarges the IT evolution conversation from architecture to behavior. We conceptualized it as an evolving *systemwide behavior* of its modules constraining their mutual interactions, building on Simon's (1962) early, underappreciated emphasis on the interaction *behavior* among a system's modules. Smith (1981: 54) once emphasized that the meaning of everything in a complex structure is in interactions. Interactions inside a system change as it evolves, leading to behaviors that its original architecture can't account for (Fürstenau et al. 2023; Lindberg et al. 2016; Rolland et al. 2018). This can cause the system's behavior to deviate over time from what's expected, a phenomenon recognized in both IS and software engineering.

While in IT project teams communication is key, within the IT artifact itself silence is golden. Silence, rather than being merely auxiliary, is causally-downstream from a system's architecture encompassing architectural attributes such as coupling, hierarchy, and granularity (e.g., Chidamber and Kemerer 1994; Subramanyam et al. 2012). Our theory of silence thus enlarges the IS generativity discourse beyond architecture and governance to encompass systems' behaviors (e.g., Fürstenau and Baiyere 2019; Rolland et al. 2018; Tiwana and Safadi 2024). Our concept of silence also bridges the artifact focus of software engineering with the human developer focus of IS, helping explain why similar systems' generative outcomes vary.

While our theory's basic concern is interaction behaviors among a system's modules, it *theoretically* assimilates their causally-upstream, architectural arrangement—the leftmost side of Figure 1—that's predominated prior studies. While not part of our theory of silence, using them as a theoretical scaffold enabled cumulative theory building in ways empirical controls can't. This theoretically, causally, and empirically assimilates apriori-known heteromorphism (differences in structure) of systems in explaining our focal silence→generativity relationship. This expands a historical emphasis in IS on dialog among human developers with dialog among a system's IT artifacts and between artifacts and humans. More broadly,

silence theoretically expands IS scholars' historical focus on interaction and communication to also encompass isolation and quietness.

5.1.2. Generativity consequences of silence

Our second novel contribution is causally linking silence in systems to IS generativity. While prior research examines management (e.g., Rolland et al. 2018) and architecture influences (e.g., Fürstenau and Baiyere 2019; Yoo et al. 2010), how internal system behaviors shape long-term generativity is unbroached. Our results confirm that silence breeds generativity, manifested as greater improvements in forking, degenerativity, evolution, and releases in silent systems relative to chattier ones. Silence fostered the most recognized form of generativity—forked derivatives (e.g., Fürstenau et al. 2023)—that grew by a dramatic 600 times; while also halving degenerativity (e.g., Fürstenau and Baiyere 2019). Further, it doubled system evolution speed and quadrupled lifetime releases of new versions.

These findings extend insights about how changing system interactions can either be generative or "dialog-halting" among developers (Fürstenau et al. 2023: 16). Our focus on interactions among IT artifacts constituting a system rather than among developers themselves revealed generativity benefits from minimizing internal system dialog. This introduces a novel mechanism into the IS generativity conversation for understanding the *coevolution* of human and systems' behaviors.

Counterfactual generativity consequences of unexpected silence. What happens when we see chattiness in systems where we expect silence and silence where we expect chattiness? Our theory underscores that architectural choices alone do not determine silence. Both modular and monolithic architectures can exhibit varying degrees of silence, thus architecturally-similar systems can display divergent silence behavior as they evolve. Our model allows for nuanced, counterfactual "what-if" analyses, juxtaposing silence anticipated on a system's architecture with its evolving, observed behavior. This back-and-forth retroduction between empirics and theory leads to novel insights (Ashworth et al. 2021: 4), helping us dissect how plausible mismatches affect generativity.

Following Cunningham's (2021: 10) strategy of counterfactual thinking using ŷs, we contrast architecturally-expected silence vis-à-vis observed silence. Expected silence is the level that our theory predicts based on a system's architecture (Stage 1 of our model), counterfactually contrasted with observed silence following Cunningham (2021: 42). This approach creates a productive dialog between theory and

empirics as advocated by Ashworth et al. (2021: 3). The 2x2 in Figure 11 shows generativity for matched (cells ❶, ❸) and mismatched (cells ❷, ❹) expected versus observed silence. (The high-low is mean splits; the *italicized* number in each cell is generativity measured as fork counts.) Focusing on forks reflects our stance that, among its many conceptions, third-party extension—without original developer involvement—best epitomizes and demarcates IS generativity from other evolutionary outcomes.

< INSERT FIGURE 11 HERE >>

A *silent* system (cell ❶)—that is architecturally-expected to be silent and also behaves silently—exhibits most generativity, mirroring our core thesis. A *chatty* system (cell ❸)—that is chatty and is also architecturally-expected to be chatty—exhibits the second lowest level of generativity. The other two hatched cells are systems whose observed behavior mismatches architectural-expectations. A *garrulous* system (cell ❷)—that behaves less silently than expected—has generativity poorer than the matched cell ❶. A mute system (cell ❹)—that is unexpectedly silent—has the poorest generativity. The most generative systems have higher-than-expected silence (the upper half of Figure 11), favoring overly silent architectures. Designers should therefore pare interactions among modules to the bare minimum required to realize system functionality.

5.1.3. Attention funneling mechanism

Our third contribution is our mechanism—attention funneling—linking silence to IS generativity. Mechanisms answer "why" questions (Ashworth et al. 2021: 48), addressing the blackbox in Figure 1's causal chain. Silence in a system enhances IS generativity *because* it dynamically coalesces teams' attention on a small codebase sliver—such as a module or a file—at one time.

This mechanism bridges two historically-separate research streams: IS studies of human developer interactions and software engineering studies of code artifacts, underscoring their interplay. It connects silence with attention—fundamental yet overlooked notions in Simon's (1962: 470) theory—to show how previously-unobserved behaviors inside IT artifacts causally shape teamwide attention dynamics.

While prior work emphasized divergent thinking (Harvey 2014) and knowledge sharing (Zhang et al. 2025), our mechanism shows how system properties shape teams' collective cognition. It also enlarges the conversation on the dynamics of IT evolution, complementing Fürstenau et al.'s (2019: 1322) insights on misallocated attention. This mechanism for generatively harnessing developers' attention integrates

individual and collective attention, in the recent footsteps of Zhang et al. (2025). More broadly, it echoes Simon's thinking that attention scarcity requires concentrating on select activities while deliberately ignoring others.

5.2. Contributions to Software Engineering

First, our conceptualization of silence as a systemwide behavior links design to behavior, a connection sought after by software engineering scholars (Aiomar et al. 2024). This insight is particularly timely given the growing trend towards refactoring monolithic systems into distributed microservices (Abgaz et al. 2023). Second, our focus on long-term consequences—such as forks, complexity creep, codebase evolution, and release tempo—complements their short-term concerns like congestion, latency, performance, and reliability (e.g., Abgaz et al. 2023; Jin et al. 2022). Third, our novel social-network measures of systemwide dynamics expand their repertoire beyond file- or module-level static analysis; addressing Rahwan's (2019: 482) recognized need. Fourth, our mechanism—which operates outside the typical radar of software engineering scholarship—reveals *how* system behaviors shape human-code interactions in ways that can spawn derivatives and curb degradation.

6. CONCLUSION

For IS practice, cultivating silence inside systems catalyzes IS generativity, and this silence can be seeded through thoughtful architecture and disciplined maintenance as systems evolve. Such an approach not only coalesces scarce developer attention but also circumvents the unsustainable Band-Aid fix of compensating for chattiness by increasing processing power and bandwidth. A silent revolution?

Five questions merit future work. First, how does silence *across*—rather than within—systems influence IS generativity? Second, when does chattiness inside modules foster silence inside systems? Third, how can silence be regained when technical debt breaks it? Fourth, what are the evolutionary consequences of intertemporal increase or decrease of silence in a system? Finally, beyond silence in this initial foray, what other system behaviors merit consideration as funnels for developers' attention?

In conclusion, silence—rooted in thoughtful modularity—breeds generativity by focusing teams on one codebase sliver at a time. Jazz legend Miles Davis said, in music, silence is more important than sound. In systems, silence is as crucial as interaction.

REFERENCES

Abgaz, Y., McCarren, A., Elger, P., and Solan, D. 2023. Decomposition of Monolith Applications into Microservices Architectures. *IEEE Transactions on Software Engineering*. **49**(8) 4213-4242.

Aiomar, E., Mkaouer, M., and Ouni, A. 2024. Behind the Intent of Extract Method Refactoring: A Systematic Literature Review. *IEEE Transactions on Software Engineering*. **50**(4) 668-694.

Ashworth, S., Berry, C., and de Mesquita, E. 2021. *Theory and Credibility: Integrating Theoretical and Empirical Social Science*. Princeton University Press.

Chidamber, S., and Kemerer, C. 1994. A Metrics Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*. **20**(6) 476-493.

Cunningham, S. 2021. *Causal Inference: The Mixtape*. Yale University Press, CT.

Fürstenau, D., Baiyere, A., Schewina, K., Schulte-Althoff, M., and Rothe, H. 2023. Extended Generativity Theory on Digital Platforms. *Information Systems Research*. **34**(4) 1686–1710.

Fürstenau, D., and Baiyere, A.K., N. 2019. A Dynamic Model of Embeddedness in Digital Infrastructures. *Information Systems Research*. **30**(4) 1319-1342.

Goren, G., and Moses, Y. 2020. Silence. *Journal of the ACM*. **67**(1) 1-26.

Hansen, M., and Haas, M. 2001. Competing for Attention in Knowledge Markets: Electronic Document Dissemination in a Consulting Company. *Administrative Science Quarterly*. **46**(1) 1-28.

Harvey, S. 2014. Creative Synthesis: Exploring the Process of Extraordinary Group Creativity. *Academy Of Management Review*. **39**(3) 324-343.

Jin, W., Zhong, D., Cai, Y., Kazman, R., and Liu, T. 2022. Evaluating Impact of Possible Dependencies on Architecture-Level Maintainability. *IEEE Transactions on Software Engineering*. **49**(3) 1064-1085.

Lindberg, A., Berente, N., Gaskin, J., and Lyytinen, K. 2016. Coordinating Interdependencies in Online Communities. *Information Systems Research*. **27**(4) 751-772.

Lundstrom, M. 2003. Moore's Law Forever? *Science* **299**(5604) 210-211.

March, J., and Simon, H. 1958. *Organizations 2/e*. Wiley, New York.

Rahwan, I. 2019. Machine Behavior. *Nature*. **568**(7753) 477-486.

Richards, M., and Ford, N. 2020. *Fundamentals of Software Architecture*. O'Reilly, Sebatapol, CA.

Rolland, K., Mathiassen, L., and Rai, A. 2018. Managing Digital Platforms in User Organizations. *Information Systems Research*. **29**(2) 419-443.

Salganik, M. 2017. *Bit by Bit: Social Research in the Digital Age*. Princeton University Press, Princeton, NJ.

Simon, H. 1962. The Architecture of Complexity. *Proceedings of the American Philosophical Society*. **106**(6) 467-482.

Simon, H. 2002. Organizing Talk and Silence in Organizations. *Industrial and Corporate Change*. **11**(3) 611-618.

Smith, C. 1981. *A Search for Structure*. MIT Press, Cambridge, MA.

Subramanyam, R., Ramasubbu, N., and Krishnan, M. 2012. In Search of Efficient Flexibility: Effects of Software Component Granularity on Development Effort, Defects, and Customization Effort. *Information Systems Research*. **23**(3) 787-803.

Tiwana, A. 2015. Evolutionary Competition in Platform Ecosystems. *Information Systems Research*. **26**(2) 266-281.

Tiwana, A., and Safadi, H. 2024. Atrophy in Aging Systems: Evidence, Dynamics, and Antidote. *Information Systems Research*. **35**(1) 66-86.

Tonellato, M., Tasselli, S., Conaldi, G., and Lerner, J. 2024. A Microstructural Approach to Self-organizing: The Emergence of Attention Networks. *Organization Science*. **35**(2) 496-524.

Van Knippenberg, D., Dahlander, L., Haas, M., and George, G. 2015. Information, Attention, and Decision Making. *Academy of Management Journal*. **58**(3) 649-657.

Vial, G. 2023. A Complex Adaptive Systems Perspective of Software Reuse in the Digital Age. *Information Systems Research*. **34**(4).

Yoo, Y., Henfridsson, O., and Lyytinen, K. 2010. The New Organizing Logic of Digital Innovation. *Information Systems Research*. **21**(4) 724-735.

Zhang, X., Fang, Y., Zhou, J., and Lim, K. 2025. How Collaboration Technology Use Affects IT Project Team Creativity: Integrating Team Knowledge and Creative Synthesis Perspectives. *MIS Quarterly*.

[1] wsj.com/business/airlines/flight-cancellations-delays-microsoft-outage-998f1c60
[2] Distributedness creates latency across the network, penalizing system responsiveness when the system's modules interact. Module interactions risk slowdowns, prompting admonishing of "chatty" APIs (e.g., Abgaz et al. 2023; Aiomar, Mkaouer and Ouni 2024).
[3] www.wsj.com/articles/google-amazon-meta-and-microsoft-weave-a-fiber-optic-web-of-power-11642222824
[4] https://www.wsj.com/opinion/boeing-737-max-crashes-mcas-pilot-error
[5] The modularity literature has no concept capturing system silence behavior, often assuming behavior mirrors architecture. Modularity-in-use describes end-users reconfiguring components, distinct from our focus on internal system behavior.
[6] Loosely-coupled modules exhibit strong interaction behavior through frequent communication or extensive data exchange. Systems using shared services, concurrent processes (like REST APIs), or event-driven architectures exemplify frequent interactions among loose-coupled modules. Here, modules often interact to synchronize states in response to events generated by other modules; but often through well-defined interfaces or APIs that preserve loose-coupling among modules. Conversely, tightly-coupled modules exhibiting silence includes batch-processing systems that interact only during batch jobs; and monolithic apps like Word contain tightly-coupled error-handling and reporting modules that activate only when errors occur, exchanging minimal data.
[7] His idea focused on reducing communication costs in fault-prone distributed systems through a message-optimal "stealth protocol" to improve fault-tolerance.
[8] Dynamic typing (in Python but not static-typed Java) creates dependencies invisible in source code (Jin et al. 2022). Our Java-only approach eliminated this confound.
[9] Modules' code becomes significantly coupled, rather than exhibiting interaction behavior we study.
[10] Software engineering calls incoming and outgoing dependencies *afferent* and *efferent* dependencies (Abgaz et al. 2023).
[11] It's unlike software engineering's *logical dependencies* that tracks co-modified source-code files; this captures human developers' collective dynamics. For example: When files A and B are frequently modified together, they have a logical dependency based on their change history rather than code structure.
[12] Evolution rate captures team productivity; and release cadence its tempo.
[13] Java's strict API, libraries, and predefined class interaction standards (formats, conventions) eliminate interface variance, reducing modularity to its loose-coupling dimension.
[14] Github hosts 200 million code repositories (172 million private) with 83 million developers. (Gousios-Spinellis' repository is at http://ghtorrent-downloads.ewi.tudelft.nl/mysql/mysql-2017-01-19.tar.gz) We used 2016 as cutoff as: (1) Java language changes in 2017 made earlier code comparisons infeasible, and (2) it allowed 5-year lagging of generativity. Of 1,679 qualifying projects, our final sample was 1,356 after excluding those deleted , or missing main branches or source-code files. The includes pre-Github projects migrated to it after 2008.
[15] We created file and module dictionaries that tracked nested classes and cross-referenced class references against imports, enabling construction of 22,500+ networks incorporating inheritance and composition relationships. The file-to-file dependencies used to constructing these networks parallel our module-level analyses.
[16] One developer-to-file linkage is created when a developer changes a file by committing code to it. Our approach is conceptually analogous to Hansen and Haas' (2001) attention measure counting document retrievals in intranets.
[17] Unlike software engineering snapshot studies of a single system like Linux with an explicitly-documented initial conceptual architecture, we examine concrete architecture across thousands of evolving systems inferred from their code. A system's architecture progresses in stages from *conceptual* (purpose/ scope/ constraints)→*logical* (entities to translate into artifacts)→*structural* (data/ protocols)→*concrete* (implementation language/ technologies/ services), our study centers on the final concrete stage as manifested in evolving codebases.
[18] We applied a 3-quarter moving average to both time series and excluded systems older than 75 quarters (<1% of data) for x-axis clarity. The cross-correlogram (Figure 9) is a correlation of two time-series. It shows attention funneling correlates most strongly with silence from one to three quarters prior.
[19] We tracked both content changes (adding/removing lines in existing files) and structural changes (adding/removing files). High change ratios often indicate active refactoring.

FIGURES AND TABLES

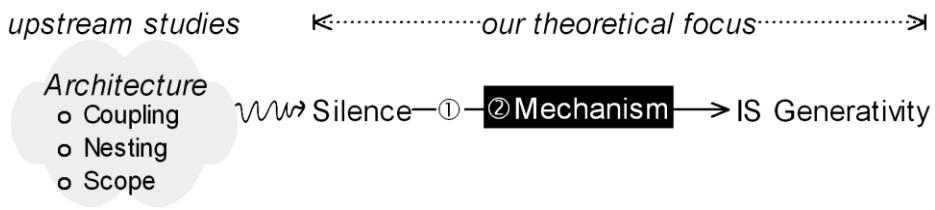

Figure 1: Our research question encompasses the ① nature ("how") and the ② blackboxed mechanism ("why") of the relationship.

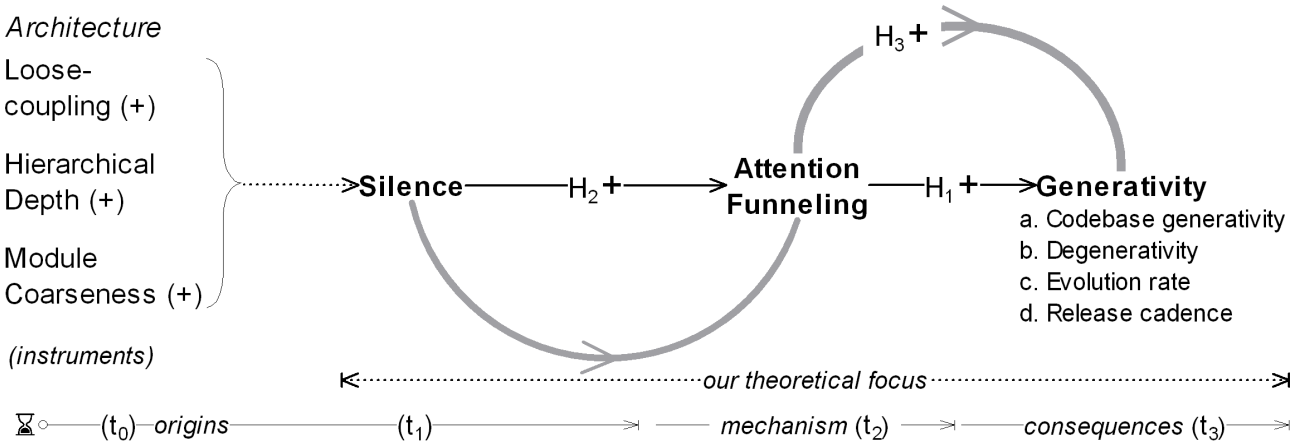

Figure 2: Research model illustrating the central mechanism of attention funneling.

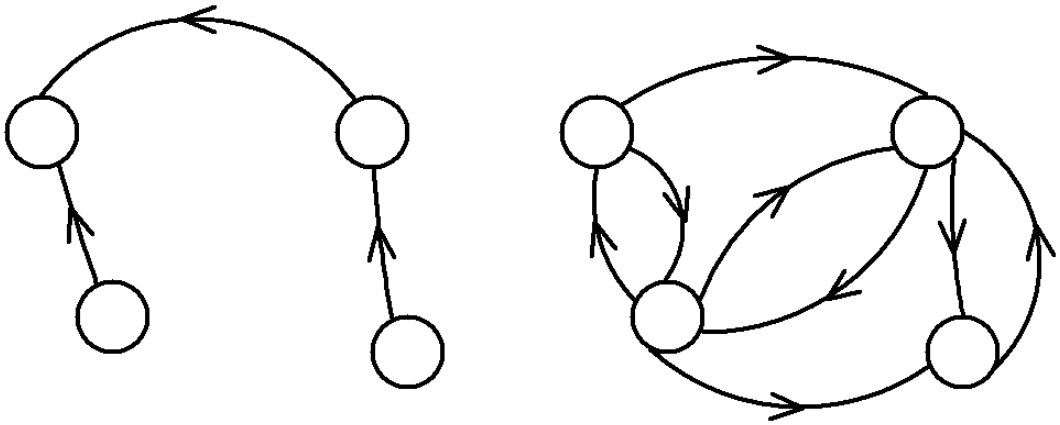

Figure 3: A contrast of a silent (left) versus a chatty (right) system comprised of four modules (○).

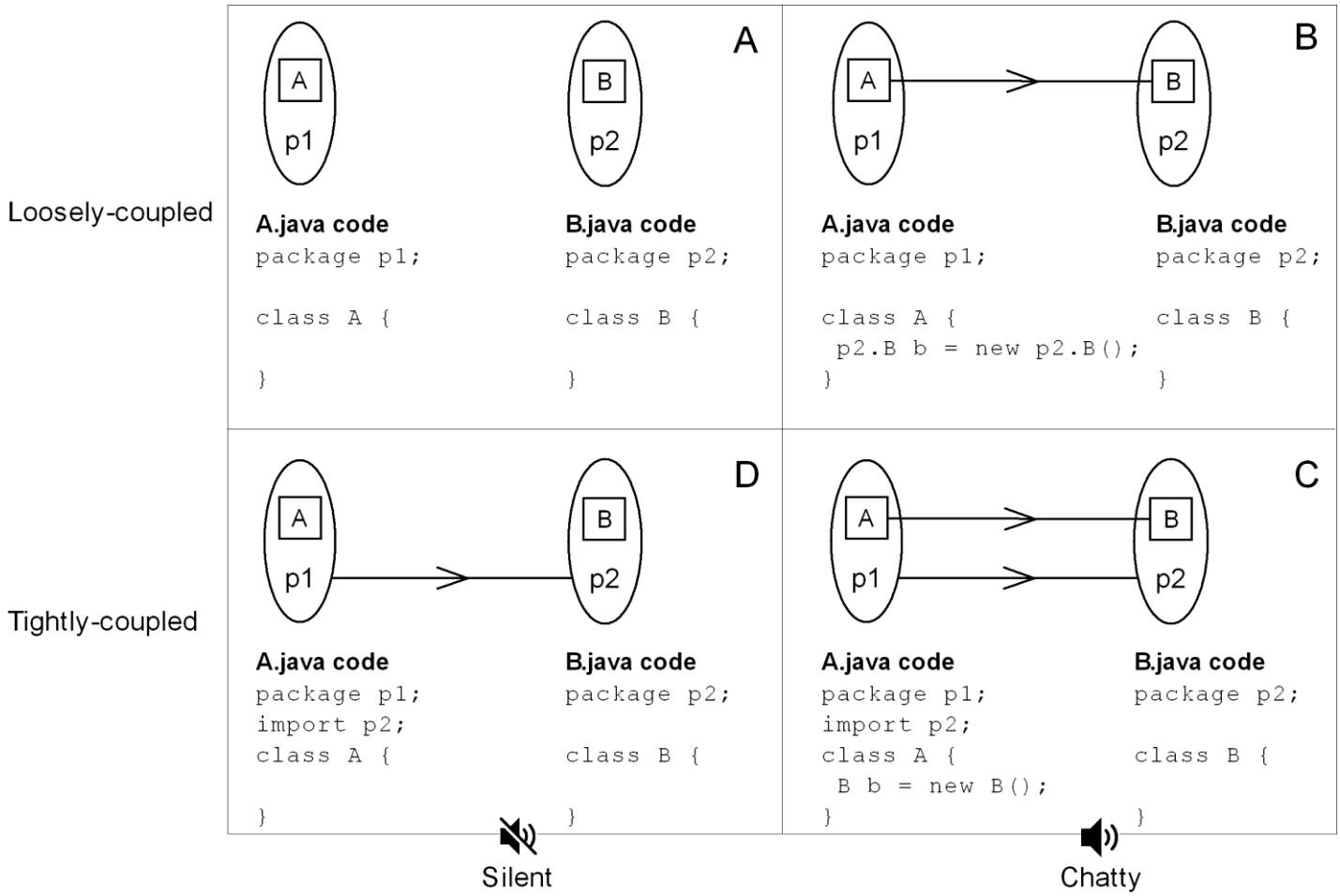

Figure 4: Loosely-coupled modules can be chatty (Cell B), and tightly-coupled ones can be silent (Cell D), although we'd expect loosely-coupled modules to be silent (Cell A) and tightly-coupled ones to be chatty (Cell C).

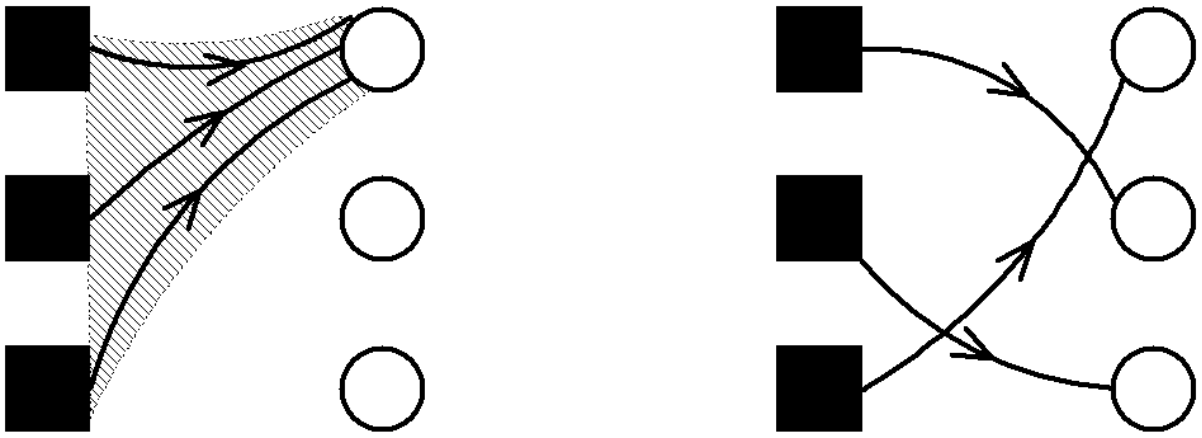

Figure 5: A team with attention funneled (left) versus scattered (right) (○ are modules and ■ are developers).

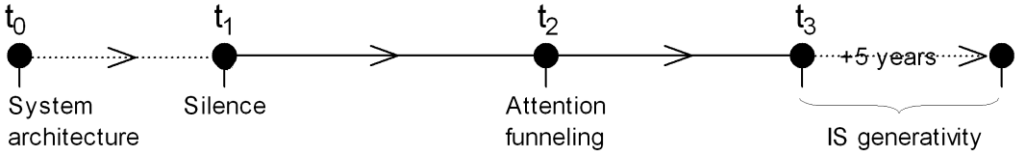

Figure 6: Correspondence of our data timeline with theoretical causal ordering.

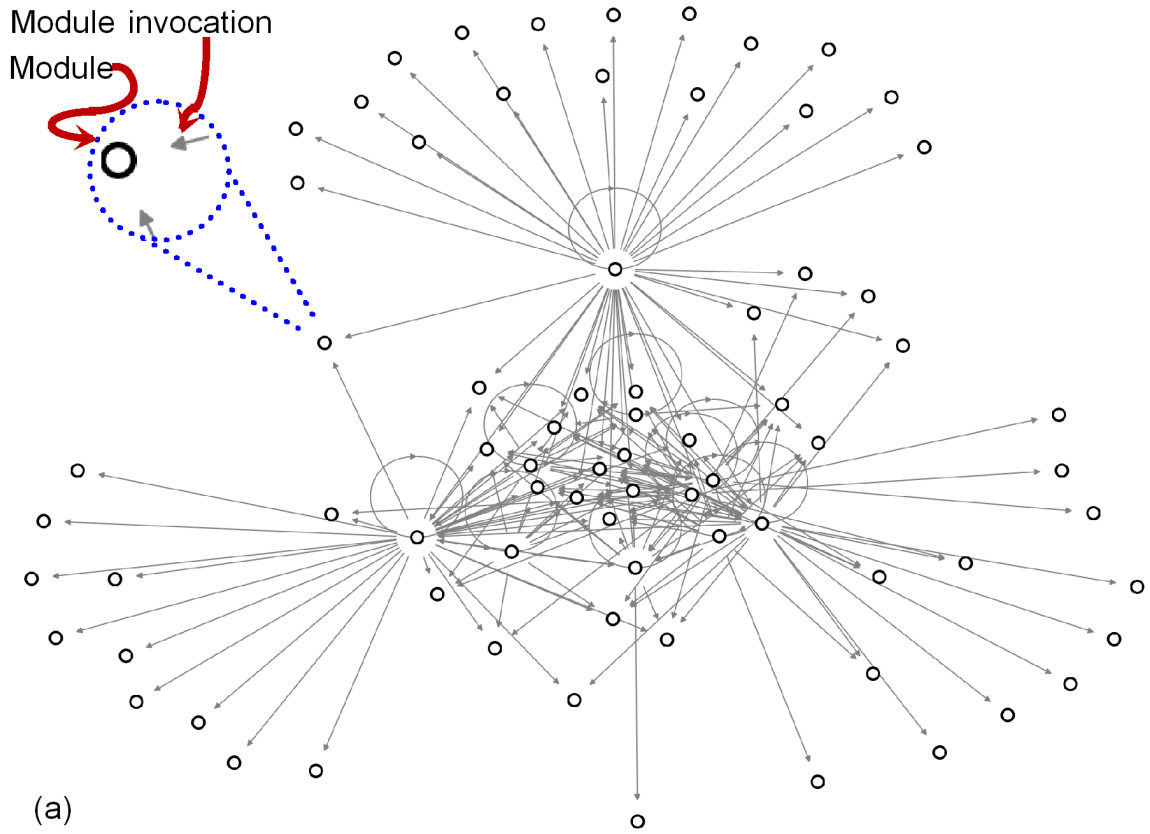

Figure 7a: An illustration of one of 22,534 measurement networks constructed for each quarter for each system to estimate (a) silence among modules (○) in a system.

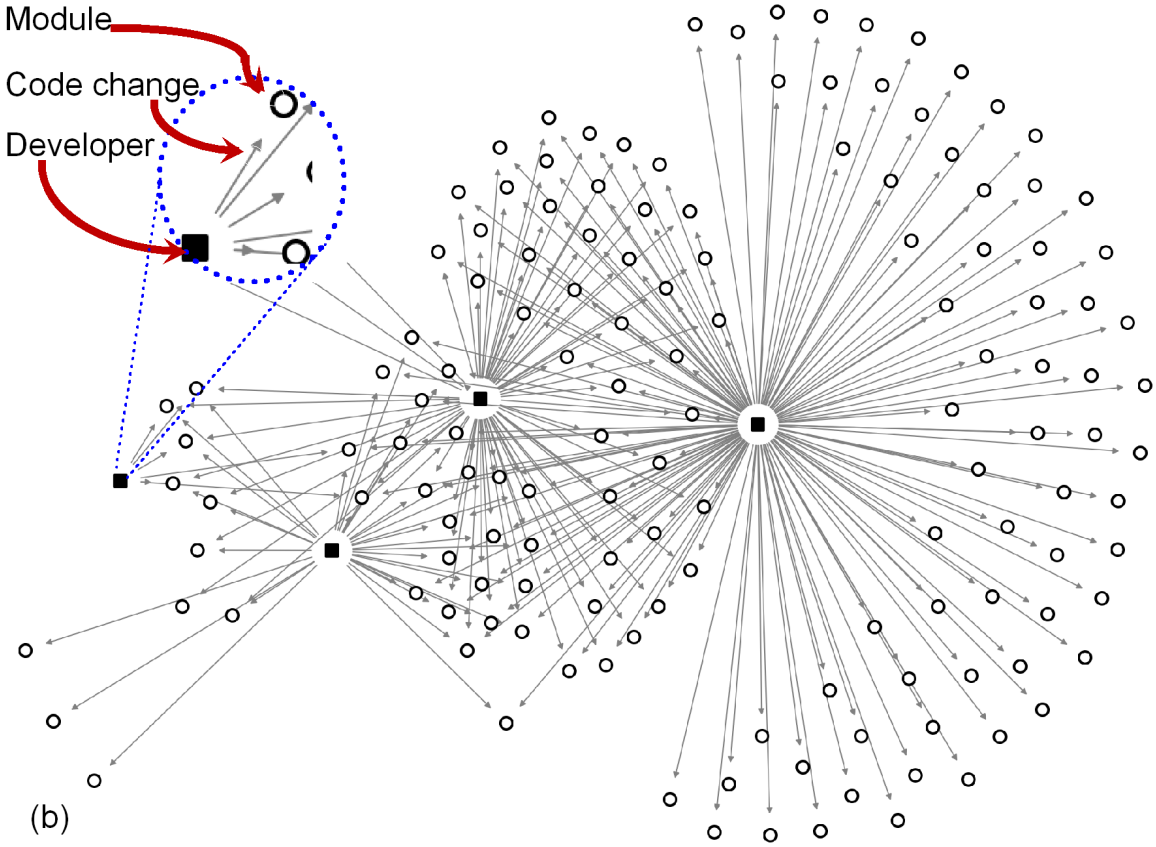

Figure 7b: An illustration of one of 22,534 measurement networks constructed for each quarter for each system to estimate attention funneling of developers (■) in a project team. This is for the same system as Figure 7a but is a developer-to- module (■→○) network.

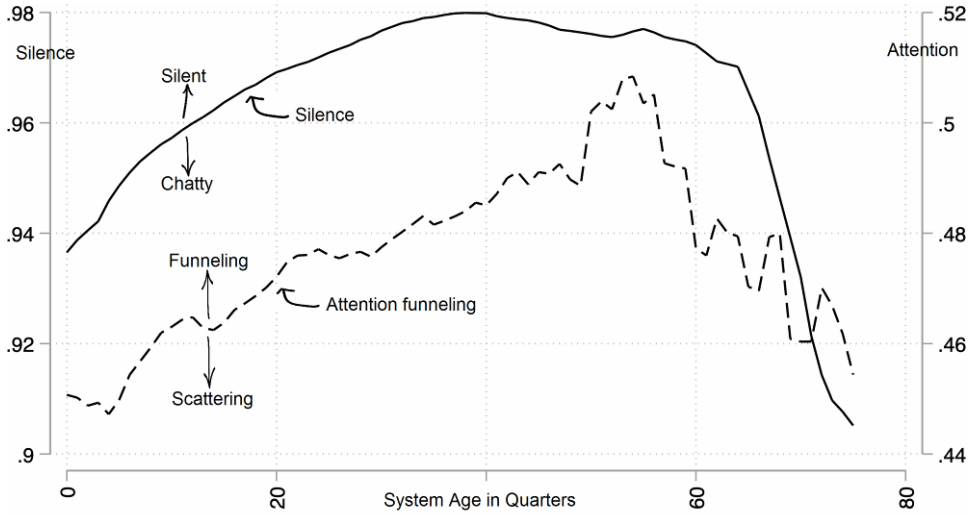

Figure 8: Shifts over time in silence and attention funneling.

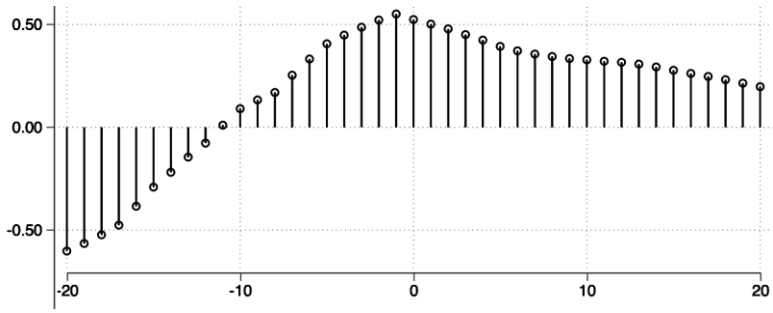

Figure 9: Cross-correlogram of the association of attention funneling and lagged silence over time.

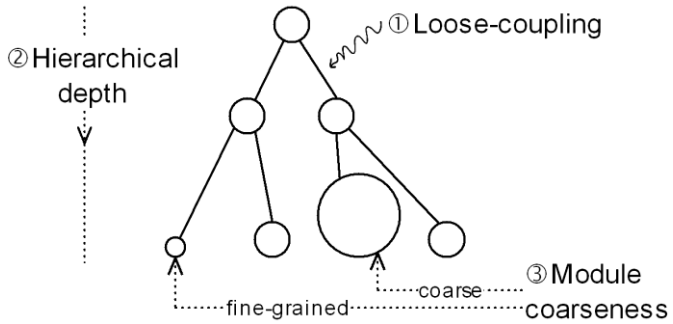

Figure 10: Three attributes of system architecture from prior research—① loose-coupling, ② hierarchical depth, and ③ module coarseness—can endogenously shape silence inside a system (○ are modules).

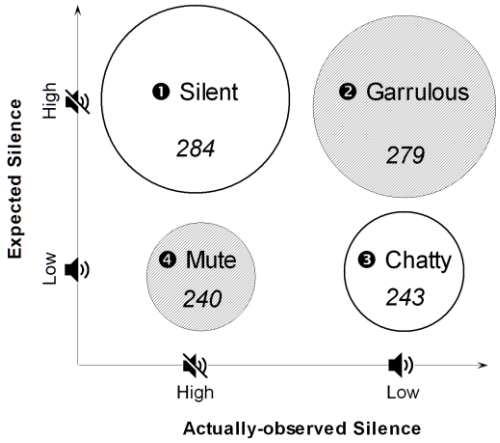

Figure 11: Generativity in systems whose actually-observed behavior matches (❶ ❸) and mismatches (❷ ❹) architecturally-driven expectations of silence.

Table 1: Key constructs and their measures

Construct	Definition ≈ measure	Role	Representative reference
Silence$_t$	Sparsity of module-to-module interactions within a system. ≈ (1–density of codebase-wide module-to-module network)$_t$	Predictor	(Richards and Ford 2020: 127; Simon 2002)
Attention funneling$_t$	Concentration of code changes on fewer modules. ≈ Centralization of codebase-wide developer-to-file linkages$_t$	Mediator	(Hansen and Haas 2001)
Codebase generativity$^{⌘}$	Derivative systems spawned from the original codebase ≈ Cumulative number of codebase forks	Criterion	(Fürstenau et al. 2023)
Degenerativity$_t$	The number of linearly independent paths in codebase ≈ Cyclomatically complexity	Criterion	(Chidamber and Kemerer 1994)
Evolution rate$_t$	Rate of change in a system's codebase ≈ Quarterly count of code commits to a codebase$_t$	Criterion	(Simon 1962: 470)
Release cadence$^{⌘}$	Count of upgraded versions of the system released. ≈ Cumulative lifetime number of releases	Criterion	(Richards and Ford 2020: 116)
Loose-coupling$_t$	Looseness of coupling among system's modules ≈ 1- (#module invocations$_t$ ÷ #possible module invocations$_t$)	Instrument	(Chidamber and Kemerer 1994)
Hierarchical depth$_t$	#layers deep module-to-subordinate module relationships run ≈ Average shortest path length in the file-to-module network constructed for the entire system's codebase	Instrument	(Simon 1962: 468)
Module coarseness$_t$	Degree of coarseness of a system's modules ≈ Centralization of codebase-wide network of file-to-module dependencies$_t$	Instrument	(Subramanyam et al. 2012)
Code volume$_t$	Number of lines of code ≈ count of lines of code in a system's codebase$_t$	Control	(Subramanyam et al. 2012)
Module count$_t$	Number of modules ≈ Number of Java "packages" in a system's codebase	Control	(Simon 1962)
Age	Chronological system age ≈ Years lapsed since a system's initial release	Control	—
Team size$_t$	Number of active developers ≈ #developers who committed code$_t$	Control	—
Architecture refactoring$_t$	Ratio of files that differ from the previous quarter ≈ files changed since last quarter ÷ all files in this and previous quarter	Control	(Aiomar et al. 2024)

Notes: ⌘ constructs are cumulative through t_3+5 years; all others are panel data measured each quarter$_t$.

Table 2: Silence versus three adjacent concepts

Concept	SE	IS	Idea
Loose-coupling	●	●	Modules independent of others' internal implementation
Accidental coupling	●	○	Discrepancy between design intentions and implementation actions
Architectural drift	○	●	Behavior diverges from architectural expectations

● extensively studied; ○ somewhat studied.

Table 3: Construct correlations and psychometric properties

	$\bar{x}$	σ	1	2	3	4	5	6	7	8	9	10	11	12	13
1. Silence	.93	.10													
2. Attention funneling	.42	.15	.09*												
3. Degenerativity	27140.4	50400	.23*	.14*											
4. Evolution rate	784.9	2208	.11*	.03*	.2*										
5. Release cadence⌘	7.60	11.7	-.12*	.00	-.07*	-.02*									
6. Codebase generativity⌘	365.20	1376.3	.01	.06*	.01	.03*	.21*								
7. Loose-coupling	.98	.05	.38*	.15*	.16*	.08*	-.044*	.02*							
8. Hierarchical depth	3.61	.91	.54*	.13*	.23*	.09*	-.14*	-.02*	.43*						
9. Module coarseness	.57	.23	.48*	.08*	.24*	.08*	-.13*	.01	.31*	.51*					
10. Code volume (million LOC)	6.49m	11.14m	.26*	.17*	.93*	.21*	-.09*	.01	.18*	.28*	.28*				
11. Module count	118.7	172.9	.34*	.2*	.57*	.21*	-.13*	.01*	.2*	.47*	.36*	.7*			
12. System age (years)	3.26	2.89	.21*	.12*	.26*	-.01*	-.02*	.04*	.17*	.16*	.20*	.27*	.27*		
13. Team size	5.56	6.54	.21*	.21*	.32*	.28*	-.00	.22*	.15*	.22*	.20*	.37*	.43*	.11*	
14. Architecture refactoring	.36	.32	-.15*	-.02*	-.18*	.18*	.07*	.03*	-.15*	-.12*	-.16*	-.2*	-.19*	-.33*	.02*

*p<.01; all variables are panel data on a system-quarter basis, except for ⌘ that are cumulative through t_3+5 years; shaded rows are instruments.

Table 4: Results

Model →	Silence	Attention	Generativity (incidence response ratios)			
	Stage 1	Stage 2	Stage 3			
Instruments	Silence$_{t1}$	Attention funneling$_{t2}$	Codebase generativity⌘	Degenerativity$_{t3}$	Evolution rate$_{t3}$	Release cadence⌘
Loose-coupling$_{t0}$	.13**(3.19)					
Hierarchical depth$_{t0}$	.02***(15.71)					
Module coarseness$_{t0}$	.05***(16.46)					
Endogenous predictor						
Silence $_{\hat{y}\text{-}t1}$		.48***(8.00)	.06(-1.85)	9.21***(22.67)	1.22*(2.10)	.20(-1.93)
Mediator						
Attention funneling$_{\hat{y}\text{-}t2}$			616.87***(6.36)	.50***(-4.85)	1.98***(5.08)	3.68*(2.08)
Controls						
Code volume$_{t2}$			1.00(.22)	1.00***(24.16)	1.00(-.94)	1.00(-.80)
Module count$_{t2}$			1.00**(-3.15)	1.00***(12.50)	1.00(-.01)	1.00**(-2.78)
System age$_{t2}$			1.10**(3.26)	1.08***(66.48)	.97***(-12.26)	1.01(.43)
Team size$_{t2}$			1.11***(7.56)	1.01***(24.28)	1.03***(28.41)	1.02**(2.82)
Architecture refactoring$_{t2}$			.69(-1.26)	.79***(-22.51)	1.65***(26.21)	1.18(.84)
N	22,534	22,534	1,180	21,299	20,805	1,180
χ^2			150.39	24348.02	2255.89	37.94
Model-F (Log-likelihood)	286.45***	63.96***	(-6668.61)	(-198555.86)	(-137909.21)	(-3002.08)

Notes: Stage 1 and 2 are 2SLS linear regression coefficients; stage 3 is incidence response ratios for NBR model; T-statistics in parentheses; *p<.05, **p<0.01; ⌘ are cumulative through t_3 + 5 years; shaded cells are hypothesis tests.

Appendix: An illustration of inference of systemwide silence by examining Java code

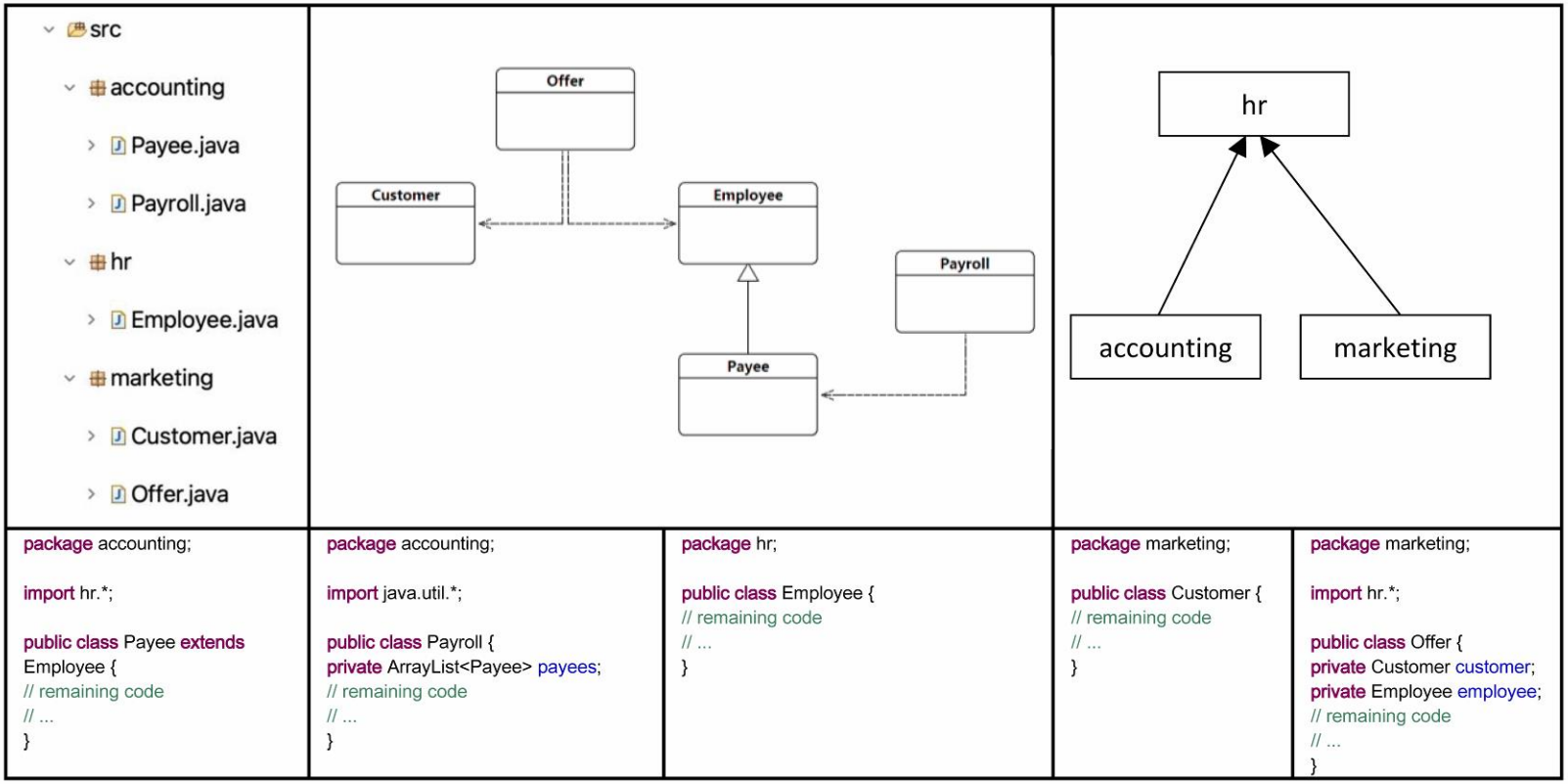

Figure A: Inference of MODULE-MODULE interaction behaviors using class composition and inheritance.

In Figure A, an archetypical enterprise resource planning application is structured with three Java packages (~modules): ACCOUNTING, HR, and MARKETING, with corresponding code shown in the lower half of the figure. The HR module contains the class *Employee*, which encapsulates employee data. Within the MARKETING module, two classes exist: *Customer* (representing company customers) and *Offer* (representing offers to customers). Since an offer requires both a customer and a supervising employee, the *Offer* class has a composition relationship with classes. In UML lingo, *Offer* has-a *Customer* and *Offer* has-a *Employee*. Because *Employee* resides in a different module, the *Offer* class must import it via an import statement—creating a silence-reducing coupling between the two modules (MARKETING-import-HR).

The *accounting* module contains two classes, *Payroll* and *Payee*. A payroll comprises multiple payees. In UML lingo, Payroll has-a Payee. No import statements are required for this relationship since both classes reside within the same module. Though *Payroll* imports java.util module to access the class *ArrayList*, this import is excluded from our silence measurements as it's an external, non-native module. Finally, the class *Payee* is a subclass of *Employee*. A *Payee* is an employee with extra information like the bank account and tax withholdings. In UML lingo, *Payee* is-a *Employee*, represented with the Java extends keyword. Because the two classes *Payee* and *Employee* are in different modules, an import statement is needed and thus accounting-import- HR. This example illustrates how two fundamental object-oriented coupling relationships—inheritance and composition—create import relationships between modules, affecting system silence.