Atrophy in Aging Systems: Evidence, Dynamics, and Antidote

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

Mailing address: Terry College of Business, 630 South Lumpkin Drive, Athens, Georgia 30602.

Running subhead: System Atrophy

Abstract

A pervasive, unbroached phenomenon is how once-modern systems age into unmaintainable albatrosses. We conceptualize this phenomenon from first-principles as *system atrophy*. We construct a *trace* dataset from 190 million lines of evolving code in 1,354 systems spanning 25 years to corroborate it. Our middle-range theory introduces system atrophy into the conversation on IS evolution, showing how small increments in modularity slow atrophy but lose potency with age. Atrophy eventually stunts systems, increases bugginess, and disengages developers.

Keywords: system atrophy, code rot, architecture, dynamics of modularity, IS aging, open source, time, IT evolution, generativity, trace data, albatross, data streams, computationally-intensive methods, evolutionary dynamics, code analysis.

1. INTRODUCTION

Information systems do not age gracefully. Their code progressively becomes so convoluted and unmaintainable that it’s often easier to scrap and start over (Arbesman 2017: 2; Booch 2008). They become albatrosses.[1] Delta Airlines’ aged reservation system illustrates this challenge. After Delta could no longer sustain spending $700 million a year with 3,000 IT employees just to maintain it nor adapt it to mobile technologies, it scrapped its innards in a five-year, $1,500 million redo. Such decay in an information system’s logical coherence—a phenomenon we conceptualize here from first principles as *system atrophy*—is largely unbroached in IS research. Practitioners allude to code rotting (Booch 2007), becoming spaghetti-like (Barr 2018: 35), and turning hodgepodge (Arbesman 2017: 131). The software engineering literature—anecdotal and atheoretical (Garcia, Kouroshfar, Ghorbani and Malek 2022)—has long suspected analogous phenomena.

In IS, we don’t yet have a comparable notion for two plausible reasons. First, the recognition that change in systems can be *de*generative is recent (e.g. Benbya, Nan, Tanriverdi and Yoo 2020; Fürstenau and Baiyere 2019), unlike its generative counterpart (e.g., Wareham, Fox and Giner 2014; Yoo, Henfridsson and Lyytinen 2010). IS scholars fear that inattention to the *dynamics* of how IT artifacts evolve invites catastrophe (Benbya et al. 2020: 5; Pentland, Liu, Kremser and Hærem 2020). Second, the IS literature has historically trailed off after the initial segment of systems' lifecycle (Agarwal and Tiwana 2015).[2] This would be analogous to ending medical practice ending at the birth of humans. Today's prominent IT problems—security breaches, crippling maintenance costs, and meltdowns—occur over systems' subsequent lifespans not yet on IS scholars' radar. Recent IS studies identify things that cause this malady (e.g., technical debt, dead code (Fürstenau and Baiyere 2019)) and its symptoms (e.g., brittleness, meltdowns (Benbya et al. 2020; Rolland et al. 2018)), but we don't yet understand the underlying malady itself. Corralling empirical evidence or assessing its progression requires first developing this notion into a theoretically-precise IS phenomenon.

This phenomenon has not been a core part of the IS conversation but it now should be for three reasons. First, system atrophy—as we later show—makes systems difficult to change, costlier to maintain, and buggier. Such business penalties are bread-and-butter for IS. Understanding the phenomenon within an IS-specific nomological context is a precursor to tackling it. Second, cognizance of evolving *existing* systems is rather recent in IS (e.g., Agarwal and Tiwana 2015; Fürstenau and Baiyere 2019; Rolland et al. 2018). Inflexible existing systems can hobble and cannibalize new ones (Steelman, Havakhor, Sabherwal and Sabherwal 2019). Third, IT investments can take years to pay off. Counteracting atrophy—by extending systems' useful life—gives more time to recoup firms' IT investments. Our first research question therefore is: *How does an information system that was once state-of-the-art become an albatross as it ages*?

To address this question, we conceptualize system atrophy from first principles. We then use *trace-datastream* methods to corroborate empirically—for the first time—its existence. Our conception of system atrophy as a dynamic, intertemporally evolving property of an aging information system bridges the nascent IS perspective on technical debt (e.g., Rolland et al. 2018) with IS degenerativity (e.g., Fürstenau and Baiyere 2019; Pentland et al. 2020); and empirically assimilates both. But how can we fend off atrophy that age begets?

A system's changeability depends on its internal anatomy; its architecture (Arthur 2009: 87). It describes the structure of interactions among the modules that constitute a system, *the* defining property of which is a system's modularity (Subramanyam, Ramasubbu and Krishnan 2012; Tiwana 2018). Architecture is cast early in a system's lifespan, when immediate deadlines overshadow distant evolvability concerns. Architectural decisions are therefore often inadvertent (Booch 2008; Ford, Parsons and Kua 2017: 6). Architecture's abstractness also makes it easy to overlook; it can't be observed until a system exists and is usually unnoticed until it has existed for some time (Smith 1981: 382).

Architecture circumscribes a system's changeability (Agarwal and Tiwana 2015), leading IS scholars to presume its immutability. Yet, every little tweak over a system's lifespan can lower or raise its modularity. It is such previously-unexamined *changes* in modularity *over time* that we dissect here for the first time. This leads to our second research question: *How do changes in an information system's modularity over time counteract system atrophy?*

We develop two new ideas to address this second question: Tweaking modularity as a system evolves slows its atrophy, but its potency fades as it ages. Our dynamic modularity↔age interplay is one of the earliest attempts to *theoretically* integrate time, which is empirically-ubiquitous but theoretically-rare in IS theorizing.

We test these ideas in a computationally-intensive econometric model using novel trace-data *streams* from over 1,300 open-source systems spanning 25 years; constructed longitudinally by analyzing about 190 million lines of code, 9 million evolving dependencies over 19 million code changes among their 400,000 modules. A novelty of our approach is that we infer our key constructs by directly analyzing using a supercomputer the IT artifacts' evolving code itself, unlike perceptual or subjective measures in prior IS evolution studies.

We make four novel contributions to research on IS evolution. First, we theoretically develop and provide the earliest large-scale corroboration of system atrophy; while simultaneously assimilating most of its purported sources scattered across adjacent IS evolution perspectives. To the fledgling IS conversation on *de*generativity, this adds system atrophy as a mechanism that bridges nomologically-adjacent IS perspectives on things that cause systems to deteriorate and deterioration's degenerative consequences. Second, we theoretically integrate the passage of time to reveal previously-unseen degenerative dynamics in evolving systems. Third, we contribute novel insights into how tweaking modularity slows atrophy. Our delving into changes in modularity over a system's lifespan is unprecedented, contrasting prior research where such changes have never entered the conversation. Finally, modularity's temporally-diminishing benefits show how it slows but cannot forestall atrophy. To software engineering, we contribute a long-awaited, theoretically-precise way of assessing code deterioration; the first large-scale corroboration; and an evidence-guided approach to changing aging code. For IS practice, our insights are on managing evolution↔atrophy tradeoffs and extracting more useful life from atrophied systems. Subsequent sections develop these ideas, methodology, analyses, and our contributions and implications.

2. THEORY

Figure 1 previews the forthcoming ideas; and the shaded box highlights our study's theoretical focus. Table 1 summarizes the key theoretical concepts. Our unit of analysis is a system; its codebase refers to its uncompiled source code. We first theoretically develop the concept of system atrophy from first principles, articulating its value-added over conceptual proximates. We then theorize the role of time in its unfolding, which lays the foundation for our two subsequent ideas: (a) increments in modularity over time slow atrophy but (b) they lose potency as a system ages. Outside the shaded area in Figure 1 are rival explanations (lower right side), causally-upstream instruments (left), and causally-downstream consequences of atrophy (right). The crux of our middle-range theory is how modularity counteracts atrophy in aging systems.

< INSERT FIGURE 1 AND TABLE 1 HERE >

2.1. System Atrophy

2.1.1. Proximate Concepts

We define *system atrophy* as the intertemporal decay in the internal logical coherence of a system's codebase. Atrophy literally means progressive decline. Its most common usage is in medicine to describe the progressive decline of muscles; their parallel in an information system is its logical structure. Our choice of atrophy over code rot, drift, or entropy heeds Tufte's (2020: 137) caution that old words deform new seeing; atrophy does not have preexisting connotations in prior research.

Although no previous research has explicitly studied system atrophy, proximate concepts—summarized in Table 2—are strewn across IS evolution studies and software engineering studies. IS evolution studies are nomologically-proximate and software engineering studies are conceptually-proximate. Nomological proximity means that are nearby in a cause-effect relationship rather than overlapping in their conceptual meaning. Conceptual proximity means that they overlap in their conceptual meaning. We first describe how they relate to and differ from system atrophy, and the value-added by our theoretical conceptualization.

<< INSERT TABLE 2 HERE >>

a. Technical debt. The closest proximate concept in IS is technical debt (Fürstenau and Baiyere 2019; Rolland et al. 2018). Technical debt—design shortcuts, violation of assumptions, tight coupling with an app's infrastructure, hard-wiring modules, and exceptions—is often incurred to expedite code changes (Kruchten, Nord and Ozkaya 2012). For example, Rolland et al.'s (2018) recent 17-year case study of a system found that the buildup of technical inconsistencies during upgrades, design shortcuts during customization, and tight coupling with infrastructure made its maintenance increasingly challenging.[3] Technical debt is intended to be paid back in the future. It can become delinquent debt if not soon corrected; a widespread problem in practice (Booch 2008). Therefore, *delinquent* technical debt can plausibly lead to atrophy, but it is not atrophy itself. A subtlety in this literature about dead code—evolutionary remnants of once-useful code—merits further attention (Arbesman 2017: 21). Dead code can plausibly decrease a codebase's logical coherence. Akin to junk DNA in humans (Biémont and Vieira 2006), dead code has no discernible purpose that a system's current developers understand, so they often tiptoe around it (Arbesman 2017: 21). It can lead to atrophy but itself is not atrophy. Put differently, delinquent debt and dead code are nomologically-upstream to atrophy but not conceptually-overlapping concepts.

b. Degenerativity. The second proximate concept is the notion of *de*generative evolution that Fürstenau et al. (2019) recently added to the IS conversation historically focused on generative change (e.g., Wareham et al. 2014; Yoo et al. 2010). This is the glass-half-empty side of IS evolution rather than its glass-half-full side. IS scholars using a complexity lens have implored for attention to how IT artifacts evolve over time (Benbya et al. 2020), recognizing that they "drift" (Pentland et al. 2020), become brittle (Rolland et al. 2018), and deteriorate (Subramanyam et al. 2012). They suspect that degenerative evolution balloons their complexity in ways that invite catastrophic IT failures (Benbya et al. 2020).[4] Most system meltdowns can indeed be traced to misbehaving interactions among a system's modules (Clearfield 2019: 22).

In spotlighting the phenomenon, this nascent stream mixes causes with effects. Deteriorative changes in an IT artifact *lead to* poor evolutionary outcomes; they have a cause$\rightarrow$effect relationship. In our terms, atrophy in a system can affect its evolution; but evolution itself is not atrophy. Evolution, whether deteriorative or enhancing, is therefore nomologically-downstream from atrophy. A subtlety in this literature merits attention. Such IS studies emphasize the evolution *of* an entire system as an outcome (e.g., Agarwal and Tiwana 2015; Rolland et al. 2018; Tiwana 2018), but Fürstenau (2019) recently recognized that modules *in* a system might also evolve non-uniformly. Yet mental models of their dependencies might update more lethargically, leading changes based on obsolete information about other modules to aggravate atrophy. This, unlike evolutionary outcomes, is a purported source of atrophy we account for.

In summary, the recent IS evolution literature offers plausible sources and consequences of what we conceptualize as system atrophy, but not what system atrophy is. These IS perspectives are nomologically-proximate but not conceptually-overlapping with our atrophy phenomenon. Debt is nomologically-upstream (i.e., leading to→) and degenerativity is nomologically-downstream (i.e., →caused by) to system atrophy theorized here; our subsequent empirical models therefore account for all of these.

c. Software engineering. In software engineering, the closest conceptually-proximate concept is code rot. Since Lehman (1979), they've argued that software structurally deteriorates unless it is deliberately reversed. Their subsequent studies neither support nor refute it. The idea remains controversial forty years on because its precise *theoretical* meaning remains ambiguous, making empirical studies difficult (Herraiz, Rodriguez, Robles and Gonzalez-Barahona 2013).[5] Anecdotes still predominate (Garcia et al. 2022). Describing code as rotting, moldy, decaying, hodgepodge, and spaghetti-like is evocative but theoretically too imprecise to study empirically (Arbesman 2017: 131; Barr 2018: 212; Booch 2008; Eick, Graves, Karr, Marron and Mockus 2001). Sobhy's (2021) analysis of 30 years (1990-2020) of software evolution studies lamented the absence of theory to understand how architecture evolves, particularly with age (p. 38). The consensus—without hard evidence—in this literature is that code structurally degrades over time. Thus, code rot conceptually overlaps with our atrophy phenomenon, unlike debt and evolution in IS that are nomologically-proximate.

In summary, the thread that systems deteriorate over time crosscuts Table 2. Debt, drift, and code rot are what Minsky (2006: 88) would characterize as "suitcase words"—words whose usage packs multiple meanings into it.[6] Precise theoretical development of what such deterioration in an information system means—a prerequisite for large-scale empiricism—is absent. Our theoretical conceptualization of system atrophy first builds up this missing piece.

2.1.2. Conceptualization of System Atrophy from Theoretical First Principles

Our conceptualization of system atrophy—as intertemporal decay in a system's logical coherence—is grounded in Simon's (1962) idea of growth in a system's entropy. Simon's thinking is the theoretical first principle that also undergirds software engineering's code rot (Booch 2008). Our conceptualization builds on Arthur's (2009: 134) premise that technologies tend to become more complex in their logical structure as they advance.

The Tufte-style (2020) collineation diagram in Figure 2 illustrates the gradual progression (the shaded module and dotted lines) of atrophy in our forthcoming conceptualization over three sequential time periods in a simple five-module system. It illustrates at *t1* the addition of ambiguous, undocumented linkages and accrual of technical debt (dotted links); and at *t2* technical debt from *t1* going delinquent (bold links) and dead code left behind (shaded). This progression over time creates unexpected interactions and unintentional coupling across the codebase, progressively degrading its logical coherence.

< INSERT FIGURE 2 HERE >

To conceptualize decay in logical coherence over time, we must first articulate static logical coherence in a codebase. Three logic structures—sequence, selection, and iteration—underpin all software systems' logic (Barr 2018: 30). As the number of branch points using these operations increases, the code becomes logically more intertwined with code scattered elsewhere in the system (McCabe 1976). Put differently, the code has deeper nested conditional statements. Barr's (2018: 35) spaghetti-code metaphor helps envision this. Understanding such code logic is similar to trying to follow a strand of spaghetti that disappears into unknown places and reappears elsewhere in a bowl.

To assess logical coherence, we represent an *entire* system's codebase as a flowchart and then tally the depth of nested sequence, selection, and iteration logic structures embedded in it. This tally considers only code native to the system, excluding embedded comments and external modules or services that it invokes. We must convert this tally to per-unit-of-code: (a) to make it agnostic to growth in total lines of code as a system's functionality evolves over time, (b) for intertemporal comparability within the same system, and (c) for comparability across systems. Atrophy is an intertemporal *increase* (Δ) in this per-unit-of-code tally. A positive $\Delta_{t0 \rightarrow t1}$ means that a system's logical coherence decreased from time *t0* to *t1*.

Our theoretical conceptualization of system atrophy is qualitatively compatible yet distinct from software engineering's code rot in two substantive ways. First, it is theoretically precise rather than descriptive, yet it encompasses code rot's idea of logical degradation. This makes it amenable to subsequent empirical operationalization. Second, it is an intertemporally evolving property of a system rather than a snapshot as in software engineering. This assimilates the passage of time into its conceptualization.

Our conceptualization's unique value-added over prior research is two-fold. First, it is objectively measurable by *directly* examining the software code itself. In contrast, prior empiricism used inherently-subjective manual coding of historical change logs. Directly analyzing the evolving IT artifact itself affords us our unprecedented use of trace datastreams to corroborate—for the first time—even the existence of system atrophy. Second, it explicitly accounts for the intertemporal *dynamics*—acceleration and deceleration—of atrophy.

2.2. Time

The passage of time is central to the emergence of system atrophy. However, time is often implicit in much theorizing (Kelly and McGrath 1988: 39). Most IS theorizing talks *around*—but not *about*—time. For example, time undergirds IT lifecycle models, lags in IT payoffs, phases in IT implementation, punctuated equilibrium in IT projects, IT path dependencies, evolutionary competition in platforms, and IT first-mover advantages. Time appears in their empirical specifications but rarely in their theorizing. Such theories where time plays a tangential role are called *holochronic*. Subtle references to the passage of time similarly pervade prior studies of IS evolution, even though time does not directly enter the theorizing. Notions such as the *gradual* erosion of architecture (Ford et al. 2017: 6); *once*-elegant systems turning into messy kludges (Arbesman 2017: 36); a system's deterioration *over time* (Subramanyam et al. 2012: 800); code handoff *across time* (Barr 2018: 63); and maintenance challenges attributed to the context's evolution *with the passage of time* (Kruchten et al. 2012) illustrate this.

Technical debt delinquency, dead code, and uneven module evolution described in §2.1 manifest only with the passage of time, as a system ages. Technical debt becomes delinquent only after remaining unpaid over a prolonged time. Accumulation of *delinquent* debt *over time* creates atrophy-inducing connections among supposedly-independent modules (Barr 2018: 125). The intertemporally-forgotten, undocumented linkage in the *t1*$\rightarrow$*t2* transition in Figure 2 illustrates this. Similarly, innocuous dead code turns malignant only after it accumulates over time. For example, Windows' codebase grew ten-fold over a decade leading up to Windows XP (Arbesman 2017: 35), some of it plausibly dead.[7] Even newer systems are often built atop legacy—including dead—code (Ghanbari et al. 2018: 17). This pattern of new layers accumulating on top of old over time exists across broad swaths of evolving technologies (Arthur 2009: 189). The accumulation over time of dead code—as the shaded module at *t2* in Figure 2 illustrates—decreases the codebase's logical coherence. Similarly, decay in logical coherence manifests from the differential evolution of a system's modules over time. Over time, as atrophy gradually metastasizes across a codebase, unanticipated interactions within it degrade its logical coherence.

Delinquent debt, dead code, and differential evolution of a system's modules scattered across prior IS studies are therefore purported plausible alternative explanations—beyond system age—for system atrophy. Put differently, system age predicts system atrophy even after accounting for these purported explanations. (We subsequently empirically account for all three.) Therefore, our baseline theoretical assertion—which we subsequently test—is that system atrophy worsens as a system ages.

2.3. An Antidote for System Atrophy

A system's architecture refers to its internal anatomy (Arthur 2009: 87). Simon (1962) defines architecture as the structure of the interactions among the subsystems that constitute a system. *The* defining property of an IT artifact's architecture is its modularity, which describes how its modules interact (Subramanyam et al. 2012). Most other properties of IT architecture—many empirically accounted for in our subsequent model—are pathways towards or away from modularity.[8] We define a system's *modularity* as the degree to which a system's modules are loosely-coupled through standardized interfaces (Subramanyam et al. 2012; Tiwana 2018). A system can range along a continuum of being perfectly monolithic where the entire system is one module and perfectly modular with numerous independent modules.

We focus for two reasons on changes over time in codebase modularity, unlike its static snapshots that predominate prior IS research (e.g., Subramanyam et al. 2012; Tiwana 2018). First, as a system evolves, adding new code to existing modules and replacing existing modules, refactoring, or correcting earlier imperfections can decrease or increase its modularity (Barr 2018: 211; Booch 2007). Having only observed snapshots of an *evolving* property in prior studies should not mislead us to assume that modularity is a static property. Second, focusing on changes in modularity over time disentangles existing technical debt paid off or left unrepaid during changes to the codebase.

Increasing modularity is an antidote to atrophy. An antidote is a remedy that counteracts a particular poison. It derives from the Latin *antidotum* meaning countermeasure—but not a cure—for something undesirable.[9] Increasing codebase modularity reapportions developers' focus more on module-specific changes than on their connections, which makes them lesser likely to introduce new cross-module dependencies. Doing so hides internal implementation details of modules from each other by codifying their dependencies into a system's structure. This lowers the risk of changes in one module inadvertently introducing new logical dependencies among modules. Even complicated revisions within one module are then lesser likely to have a ripple effect elsewhere in the codebase. Otherwise, stabilizing such ripple effects under time pressure encourages shortcuts, cross-module patchwork, and leaving untouched ambiguous, comatose and dead code. An increase in codebase modularity over time thus discourages developer behaviors that exacerbate system atrophy. Therefore, an increase in modularity slows down system atrophy. This leads to our first hypothesis.

> ***Hypothesis 1:*** *An intertemporal increase in the modularity of a system's codebase slows down system atrophy.*

2.3.1. The Interplay of Systems' Age with Modularity over Time

The atrophy-reducing advantages of modularity fade as a system ages because of our inherent tendency to pay more attention to the near-term than to the longer-term (Wittmann 2017: 5). Whatever is experienced in temporal proximity influences decisions more (Wittmann 2017: 11). Team members involved in a system's initial development likely focus more on its successful initial completion rather than the system's subsequent, post-completion life. Ghanbari's (2018) review spanning nearly 50 years (1968-2015) of the software quality literature implicated developers focused on short-term deliverables for structure-degrading shortcuts. Therefore, earlier in a system's lifespan, immediate development pressures likely overshadow architectural decisions.

Further, some initial design decisions are held in the original developers' collective "tribal memory" rather than being explicitly documented (Booch 2007; Ghanbari et al. 2018: 19). As time passes, the original developers might depart and different developers take over the codebase (Narayanan et al. 2009). As more time passes, more of this collective memory is lost. As a system ages, understanding code written by others then impedes changes to it (Barr 2018: 58). Some evidence for this tendency exists in the IS maintenance literature. For example, Banker and Slaughter (2000) found that developers of mainframe COBOL systems struggled to anticipate the long-term maintainability implications of their implementation choices. Similarly, during the Covid pandemic, New Jersey's governor made televised appeals seeking volunteer COBOL programmers to help adapt its mainframe financial system to deal with a surge of unemployment claims.[10] The state's existing IT staff could neither understand the aged system nor code in then-defunct COBOL.

As a team of current developers adapt an existing codebase, they are also lesser likely to consider the temporally-distant consequences of *their* own choices. As a system ages, its current developers are then increasingly more likely to encounter unanticipated code changes from earlier developers in the codebase's history. Further, improvements in modularity provide little recourse against exogenous changes in a system's surrounding infrastructure (which are external to a codebase, thus not atrophy *within* it[11]). As a system ages, such changes are more likely to occur. We therefore expect that systems' aging weakens the extent to which increasing modularity slows atrophy.

Hypothesis 2: *The atrophy-slowing advantages of modularity fade as a system ages.*

3. METHODS

3.1. Trace-Data Collection Strategy

We used longitudinal *trace*-data on software systems collected as part of a larger, multiyear research program on IS evolution. Trace data is process-oriented data that is a byproduct of developers' everyday actions, which leave a residual trace (Salganik 2017: 13, 154). Experiments and cross-sectional designs are weaker tests given our theory's temporal nature, which instead requires an accretive historical perspective that trace data better provides. We used the *record linkages* strategy for our trace-data collection (Salganik 2017: 26), which is the process of assembling sequential records over the lifespan of each information system. It is akin to assembling sequenced snapshot-like pages into a book; one for each system—our unit of analysis—in our study.

Our sampling frame was all Java-language systems in Github, the largest open-source repository with about 375 million (28 million public) projects and 100 million users. We randomly sampled 1,354 systems with the prerequisites that they be actively developed (i.e., at least a monthly commit over a five-year period) and be *de novo* (i.e., not forked from other systems).[12] Focusing solely on Github mitigated confounding due to variance in source-code management tools such as version control, bug tracking, and documentation. Focusing solely on open-source systems permitted our using source code to create multidecade trace datastreams across over a thousand systems implausible with proprietary systems. Our novel use of datastreams—rather than data points—is analogous to switching from a Polaroid camera to a video recorder.

Focusing solely on Java had three advantages. First, it is one of *the* most widely used programming languages, aiding generalizability. Our sample—as Figure 3 illustrates—spanned diverse mobile, desktop, and enterprise domains. Second, it permitted constructing our trace dataset by directly analyzing the source code, without cross-language differences confounding comparability across systems. Third, as our measurement strategy subsequently describes, the Java language's uniformity of interface standardization among modules allowed us to measure systems' modularity over time by directly analyzing source code. The PRISMA diagram in the online appendix summarizes the data collection process.

< INSERT FIGURE 3 HERE >

We locally cloned every system's source code over its lifespan along with code commit records for every change ever made. This included timestamps, developer identifiers, the changed code itself, modules and files to which these changes were made, and who made them. Following Salganik's (2017: 26) record linkages strategy, we assembled these snapshots into 1,354 "books" of sequenced trace-data; one for each system. Our sample's source code comprises 1,354 systems spanning 20 years (1996-2016).[13] We lagged data on our downstream degenerative consequences by another five years, for a total timespan of a quarter century. We observed each system over a sufficient period (median 14 quarters; 3rd/ 1st quartile 22/10 quarters) involving 7,195 active developers. To construct our trace dataset, we analyzed 189 million lines of code spread over 1,458,126 files in 176,221 modules, about 9 million dependencies, over 19 million code commit records, 63 million lines of documentation, and 215,151 external modules that these systems invoked. The total corpus of source code analyzed was about 365 Gigabytes; this equals 88,000 copies of the Bible.

3.2. Construct Measurement Using Trace-Data

All our constructs used *trace* measures, which refer to accretions made by the behaviors that are unknown to the actors enacting them (Kelly and McGrath 1988: 77). We analyzed on a supercomputer cluster the source code of each system in every quarter over its lifespan to tally various system properties (e.g., modules, files associated with them, dependencies among modules, code commits, the invocation of external modules, documentation line counts, logic pathways in the code, and individuals who made each of the 19 million changes). We computed our theoretical constructs from these. Table 3 summarizes the trace measures of the theoretical constructs, instruments, and controls. We next describe the concept↔measure↔code mapping for our key constructs.

< INSERT TABLE 3 HERE >

3.2.1. System Atrophy

No prior measure exists for system atrophy. We operationalized it as the intertemporal decay ($\Delta_{t1\rightarrow t2}$) in a codebase's internal logical coherence. We assessed logical coherence using McCabe's (1976) graph-theory-based quantification of the count of linearly independent paths in the source code's logic.[14] Pentland et al. (2020) explicitly encouraged using McCabe's (1976) approach to assess "drift" in information systems. Higher values of McCabe's metric indicate greater logical *in*coherence of the codebase (Ford et al. 2017: 18). The word complexity in this metric's colloquial "cyclomatic complexity" label is a misnomer because it measures a system's *static* logical *in*coherence rather than volumetric complexity or size. It is also for an entire system, rather than per-unit-of-code. System atrophy is how it *increased* ($\Delta_{t1 \rightarrow t2}$) per-unit-of-code during each observation interval. We measured system atrophy as Δ(McCabe's metric ÷ #lines of code) $_{t1 \rightarrow t2.}$ This converts McCabe's metric (= E-N+2P) from a system-level metric to one that is per-unit-of-code before estimating the intertemporal Δ, where:

E = number of #transfers of control ("edges")
N = number of statements with only one transfer of control ("nodes")
P = number of disconnected parts in the system's flowchart

In operational terms, E this is the number of source code file lines excluding comments, P is the number of branch-entering[15] lines, and N is the number of branch-exiting[16] lines. The emphasis of our atrophy measure is intertemporal *changes* in logical coherence *per-unit-of-code*, which is conceptually distinct from both a system's static, snapshot logical coherence and from sheer system complexity (often measured by counting function points or lines of code that we also empirically account for). An intertemporal increase in our measure indicates that the depth of conditional statements in the code has progressively increased between two observation intervals. This measurement approach also circumvents atrophy from bleeding into our modularity predictor.[17]

For each observation interval, we analyzed line-by-line each source code file in every system's codebase. We counted the lines of *code* (excluding documentation) having branching or control paths and aggregated them to create a system-level count for each system. McCabe's (1976) original metric is language-independent. It required us to translate his generic notions of sequence, selection, and iteration to the Java language-specific equivalents. In Java, such branching is indicated by `if-else statements, ?:; while, do, for; catch, switch, case statements; &&;` and || statements. Our code analyzer counts the number of programming constructs—including loops, branching, and assertions—that increase the logic complexity of the code.[18] The code specifically considers two types of these constructs: (1) entry tokens and (2) exit tokens. Entry tokens indicate the beginning of these constructs (if, while, for) and exit tokens indicate branching out to other modules (return, break).

McCabe's (1976) original measure is for the *entire* system, so we divided it by the total lines in the codebase for each period. System atrophy is the *change* ($\triangle_{t0 \rightarrow t1}$) in this percentage during each observation window. An increase in the proportion of such branching—a positive △—indicates that the codebase's logical coherence is progressively decreasing. An increase in our metric therefore indicates an increase in atrophy in a codebase. Figure A1 in the APPENDIX illustrates using Java code the intertemporal emergence of atrophy in a METHOD snippet as it underwent two major changes.

3.2.2. Modularity

Most prior IS studies have measured modularity using perceptual psychometric Likert scales (e.g., Tiwana 2018). They are all snapshot, static measures.[19] In contrast, we objectively infer codebase modularity by directly examining the software code of the IT artifact itself. Our measurement approach replicates MacCormack et al.'s (2006) approach of counting dependencies by examining source code, but on a larger scale and for many time periods; theirs was a static count for six systems. Our computationally-intensive, trace-measurement approach assesses for the first time changes in modularity over time.

We exploited two unique properties of the Java language to estimate modularity. A Java system's codebase is organized as modules, each of which has *.java* files in its own source code directory. Dependencies between two files in different directories therefore represent dependencies across modules (called *packages* in Java). We measured *modularity* as the ratio of the number of modules to the count of dependencies among the modules *native* to the system. Native modules' dependencies therefore exclude imports from the Java library and external modules and services that they invoke. Our source code analyzer custom-developed for this study scanned the modules in the project codebase. It then scanned all source code files in these modules to construct a dependency graph.[20] In doing so, it separately tallied dependencies among a codebase's native modules and to external modules. We only counted the former in computing codebase modularity, consistent with our definition. We repeated this counting procedure for every quarter.

The numerator in our computation taps into how extensively a system is decomposed into modules; a system's span in Simon's parlance. The denominator taps into the degree of interdependence among the system's modules. Jointly, they characterize the degree to which a system is composed of relatively-independent modules. Both parts can change over time, so we recomputed them for every observation interval. For modularity's interface standardization facet, the theoretical benefit of concentrating on a single programming language is that it eliminates variability in interface standardization among modules within each system. All Java-based systems use a uniform, prespecified set of interface standards (e.g., the Java API and an extensive set of core, class, and integration libraries) that prespecify how one module may interact with another (e.g., only using public methods). These define the kinds of calls or requests that a module can make, their data formats, and conventions. These cross-module interface standards are frozen, remain backward compatible, and only the release of a major revision of the programming language (which our sampling strategy eliminates) deprecates them.[21] Therefore, focusing exclusively on systems coded in Java eliminates the interface standardization dimension of modularity and reduces it solely to the looseness of coupling among modules. This makes the changes in modularity over time amenable to estimation by analyzing each system's source code over its multi-year evolution. The larger this metric, the higher the modularity of a system. Figure A2 in the APPENDIX uses a Java-specific code snippet to illustrate this modularity inference strategy.

3.3. Descriptive Statistics and Dynamics over Time

The average system in our study was 4.76 years (σ 2.96) old at the completion of our study. It had 118 (σ 172) modules containing 1,017 (σ 1,404) files containing 132,055 (σ 213,032) lines of code and 49,001 (σ 93,120) lines of documentation on average, with 5,786 (σ 8,545) dependencies among modules. The average granularity of modules was .15 (σ .12). On average, a system had 27,144 (σ 50,403) lines of code with branching or conditional logic and invoked 134 (σ 126) external modules. Table 4 summarizes the construct correlations, with the instruments in the shaded columns.

< INSERT TABLE 4 AND FIGURE 4 HERE >

Figure 4 shows the cross-correlogram between atrophy and modularity over time. It represents the correlation between two time-series. We can start lagging one series to ask how changes in modularity yesterday correlate with atrophy today using their time-series' correlation across a multidecade timespan. It reveals two patterns. First, that previous modularity is associated with lesser atrophy in the future; after passing stationarity tests.[22] Second, the zero values of lead correlations show that previous atrophy is not associated with future loss of modularity. This underscores that modularity and system atrophy are distinct, empirically-separable properties of a system. Figure 5 shows that modularity itself degraded over time, both on average and across quartiles. As these shifts occurred, on average 5.6 (σ 6.5) developers committed 785 (σ 2,208) changes to a system every quarter.

< INSERT FIGURE 5 HERE >

4. ANALYSIS

Our unit of analysis is a system. We constructed our trace dataset by analyzing every change made to any of the 189 million lines of source code across 1,354 systems' 400,000+ modules and 1.4+ million files. Our analysis is therefore at a higher, system level of aggregation than our microdata. This directly responds to Benbya et al.'s (2020) call to examine interactions among lower-level entities (e.g., modules) in the internal structure of IT artifacts. Since system age is a chronological conception of time, we organized our data using objective clock time (Kelly and McGrath 1988: 65).

We used the quarter as our observation interval to balance a delicate tradeoff. Our window had to be long enough for atrophy to manifest yet short enough to curb confounds. Too long an observation interval risks theoretically-unknown confounds entering the analyses. Too short an observation interval risks being unable to observe atrophy. It would be like trying to observe sunrise every six hours. An observation interval of six hours would miss finding evidence for a phenomenon whose existence interval is 24 hours. Similarly, less than a quarter is shorter than the *existence interval* of the atrophy phenomenon that we are trying to observe. (Less than 70% of our codebases had more than one monthly change, from which atrophy would be challenging to observe.) Guided by Kelly and McGrath's (1988: 87) recommendation to use a long series of observations at short intervals, we used a quarterly observation interval; complemented by robustness checks in §4.6 for doubled and quadrupled intervals.

4.1. Overview of Analysis Strategy

To isolate the influence of system age and modularity on system atrophy, our analyses must: (a) account for the endogeneity of modularity and (b) control for time-invariant and time-variant rival explanations (§4.4) of system atrophy. A system's modularity over time is not an exogenous property but rather a deliberate choice (Arthur 2009: 100; Subramanyam et al. 2012); our model accounts for its endogeneity.

We ruled out a random-effects model after a Hausman test ($\chi 2$ 38.99; $p<.001$) rejected the null of a non-systematic difference between fixed-effects and random-effects estimators. Using fixed-effects eliminates bias from unobserved time-invariant, system-specific factors influencing atrophy, which a random-effects model cannot. This strategy balances both unobserved and observed factors (Salganik 2017: 206).

Our data was heteroscedastic (χ^2 2.4e33; $p<.001$; rejected the null of homoskedasticity), was expectedly autocorrelated ($F_{\text{Wooldridge}}$ -191.9; $p<.001$; rejected the null of no autocorrelation) but was not multicollinear (average VIF was 1.11; max 1.31 <10). We used robust standard errors clustered on the system to mitigate heteroskedasticity. Our theory focuses on changes in modularity over time and atrophy itself is *change* in systems' logical coherence; mirroring the logic where a *change* in X leads to a *change* in Y. We therefore used a first-difference estimator, which is also more efficient than fixed-effects under heteroskedasticity and autocorrelation. This approach—particularly applicable to theories of change—fits our model in which both X and Y change over time; and our data has the prerequisite multiple assessments. Our focus on within- rather than between-system differences is appropriate because atrophy and changes in modularity occur over time *within* a system.[23]

4.2. Stage 1: Endogeneity Correction

The primary source of endogeneity in our model is omitted variables i.e., many known factors that influence changes in a system's modularity. The potentially-long list of plausible omitted variables is challenging to empirically control in a conventional regression model. Examples include application domain (Herraiz et al. 2013), commercial sponsorship, multi-platform support, country of origin, initial investment size, or a system's originating architects' skill level. Fortunately, most of them are time-invariant, which system fixed-effects eliminate. But time-variant sources must separately be mitigated. Given the absence of empirical precedent for system atrophy models, we used a few, strong, theoretically-guided instruments. The conceptual requirement is that they influence modularity but not system atrophy.

Our two instruments capture intertemporal dynamics, consistent with our use of a first-difference model. The first instrument is the intertemporal change (Δ) in *external module invocation,* which refers to the extent to which the system uses third-party services or modules (Ford et al. 2017: 115); measured as the external module percentage of a system's total module count. A system might leverage outside capabilities by combining preexisting external modules and building the rest. This typically involves using code within the system to invoke functionality from external modules using API calls; and this can change over time. Notwithstanding its immediate benefits of rapidly expanding a system's functionality, it simultaneously introduces dependencies outside developers' control. An intertemporal increase in external module invocation can therefore lower a system's modularity. However, since the code within an invoked external module does not count as code native to the system, we do not expect it to directly affect atrophy. The second instrument *module granularity* growth, is defined as the intertemporal changes (Δ) in the fine-graininess of a system's modules (Subramanyam et al. 2012); measured as the ratio of modules native to the system to the number of files in the codebase. A system can be built with any combination of coarse- or fine-grained modules (Subramanyam et al. 2012). The system's modularity is likely to increase if a system's modules become finer-grained between two observation intervals (e.g., by the addition of new files or removing existing modules). However, it is unlikely to affect system atrophy by itself because this does not directly affect the system's logical coherence. We used a first-difference estimator with two-stage instrumental variables (*xtivreg2* in Stata 16); the first stage equation being equation 1.

$$\Delta\text{Modularity}_{t1} = \alpha_0 + \alpha_1\Delta\text{external module invocation}_{t0} + \alpha_2\Delta\text{module granularity}_{t0} + \{\text{controls}\} + \varepsilon \ldots\ldots(1)$$

The results from Stage 1 in Table 5 show that both instruments (the shaded rows) were significant and in the expected direction; Stage 2 controls are customary covariates.

<< INSERT TABLE 5 HERE >>

4.2.1. Identification and Instrument Diagnostics

Both instruments satisfied the *relevance criterion* of being correlated with modularity ($r_{\text{module granularity}} = .73$; $p<.001$ and $r_{\text{external module invocation}} = -.16$; $p<.001$) and the *exclusion restriction* of not being correlated with atrophy ($r_{\text{external module invocation}} = .01$; ns $p=.35$ and $r_{\text{module granularity}} = .002$; n.s; $p=.63$). Our Stage 1 model's large F-value (21>10; p<.001) and high explanatory power (R^2 was 52%) provided evidence against instrument weakness. The nonsignificant F-value of the Anderson-Rubin exogeneity test ($F_{\text{Wald}} = 2.15$, p=.11; ns) signaled endogeneity. Further, (a) rejection of Kleibergen-Paap test's (χ^2 21.26; p<.001) underidentification null; (b) rejection of Cragg-Donald's weak identification null ($F_{\text{Wald}} = 8{,}749$; p<.001); (c) Kleibergen-Paap ($F_{\text{Wald}} = 21.26$) statistic exceeding the Stock-Yogo 10% threshold of 19.93; and (d) failure to reject the Sargan-Hansen J-test's null (χ^2 1.58; p=.20; ns) that the over-identifying restrictions are valid collectively indicated that our model was overidentified and our instruments valid.

4.3. Stage 2: Model Estimation

Stage 2 (equation 2) used a hierarchical first-difference model to test the hypotheses; all controls are the conceptual interpretations of the Δs produced by the first-difference estimator of their underlying static variable. (The steps appear as hierarchical models 1→5 in Table 6.) We sequentially added the first cluster ($\subseteq$1) of controls (model 1; controls from the prior IS literature in §4.4), the second cluster ($\subseteq$2) of controls (model 2; the three purported sources of atrophy in §2.1), then added cumulative system age (model 3) to assess the baseline existence of atrophy, then modularity (model 4; Hypothesis 1), and finally the interaction of modularity with system age (model 5; Hypothesis 2). As explained in §4.3.1, the constructs in equation 2 are the first-differenced Δs of the underlying variables.

$$\begin{aligned}\text{System atrophy}_{t2} = \{&\beta_1\text{Development burstiness}_{t2} + \beta_2\text{Team turnover}_{t2} \\ &+ \beta_3\text{Evolution intensification}_{t2} + \beta_4\text{Code volume growth}_{t2}\}\end{aligned}$$

$$\{+\beta_5\text{Technical debt delinquency}_{t2} + \beta_6\text{Differential evolution}_{t2} + \beta_7\text{Code staleness}_{t2}\} + \beta_8\text{System age}_{t2} + \beta_9\Delta\text{Modularity}_{t1\rightarrow t2} + \beta_{10}(\text{System age}_{t2} * \Delta\text{Modularity}_{t1\rightarrow t2}) + \varepsilon \quad \ldots\ldots(2)$$

4.4. Controls for Time-variant Rival Explanations of System Atrophy

Using fixed-effects in our first-difference model controls for plausible time-invariant rival explanations of atrophy but not for time-variant ones. We controlled for the latter in two clusters comprised of seven time-*variant* rival explanations, computed objectively by analyzing systems' source code in every quarter. Each accounts for a *dynamic* construct, estimated by our first-difference model as a $\Delta_{t1\rightarrow t2}$ of an underlying static variable. Table 3 summarizes their measures.

Cluster 1 – apriori from prior IS studies: Four controls (subset $\subseteq$ 1) derived from prior IS studies are: (a) development burstiness, (b) team turnover, (c) evolution intensification, and (d) code volume growth. First, the number of individuals working on the system—team size in prior IS studies—post-release is likely to vacillate over time. More developers active during an observation interval lowers manpower constraints that can lead to atrophy-inducing design shortcuts and leftover junk code. Similarly, aggressive refactoring attempts and efforts to pay down existing technical debt can involve a temporary burst of developers. We account for this by controlling for *development burstiness*, measured as the intertemporal change (Δ) in the count of active developers who committed code during each consecutive quarter.[24] Second, over time, new generations of programmers might come on board and deal with the codebase that they inherited. Such personnel turnover and code handoff across time (Barr 2018: 63; Narayanan et al. 2009) dilutes undocumented "tribal memory" described in §2, exacerbating atrophy. We thus control for such turnover using a newly-developed *team turnover* measure. We used each developer's unique Github identifier to create a pooled list of developers who committed code in every quarter to each codebase. We then calculated the *set* difference between the pools for each consecutive quarter (t1$\rightarrow$t2) for every system. In set theory notation $\{x : x \in \text{Team}_{t1} \text{ AND } x \notin \text{Team}_{t0}\}$. This reflects contributing developers who did not contribute in the previous quarter. Third, *evolution intensification* can reduce atrophy by organizing code changes into smaller rather than larger chunks of development work. We measured it as the intertemporal change (Δ) in codebase evolution rate, measured as the logged count of code commits during each quarter. Fourth, a system's code volume (Subramanyam et al. 2012) can grow, shrink, or remain flat in each observation interval as new code is added, old code is removed, or existing code is changed. Growth in such absolute system size is likely to increase atrophy by introducing new logical dependencies in the codebase (Ford et al. 2017: 74). We therefore, controlled for *code volume growth*, measured as the intertemporal change (Δ) in code volume, measured following Subramanyam et al. (2012) as the logged codebase line count. As Table 6 shows, three of the four controls in this cluster were significant and in the expected direction.

***Cluster 2 – time-variant, purported atrophy sources*:** The three $\subseteq$ 2 controls are the purported, time-variant sources of system atrophy scattered across prior IS evolution studies from §2: (a) technical debt delinquency, (b) differential evolution of modules, and (c) code staleness. Although most exist in recent qualitative studies, this cluster empirically assimilates all of them for the first time as rival explanations beyond system age. (They don't pass inclusion and exclusion restrictions as instruments.) First, *technical debt delinquency* is measured *inversely* as the intertemporal increase (Δ) in documentation coverage (the ratio of lines of documentation to lines of code). The logic is that as more unrepaid technical debt accumulates, the poorer a codebase's documentation will become. To invert into delinquency, we computed the difference in this ratio vis-à-vis the highest coverage ratio observed in the sampled population of systems before first-differencing it. Second, *differential evolution* across a system's modules was measured as the intertemporal shifts in the imbalance in the concentration of commits across codebase files in each observation interval. We used the Freeman centralization measure, a higher value indicating more differential evolution within a system's codebase.[25] Third, we proxied dead code as the intertemporal swings (Δ) in the proportion of codebase files that received code commits in each quarter. The more refreshed the files, the lesser stale is the codebase. Software artifacts are challenging to pronounce dead, comatose, or alive (Tiwana 2016); our measure therefore captures the code's proximity to death as *code staleness*. The staler the code, the larger the shadow of death looming over it. Table 4 shows that these three constructs were positively, two significantly, but none strongly correlated with system age or with each other, implying that they are separable sources of atrophy. This is empirically reinforced by an average VIF of 1.013 when using these three controls and age to predict atrophy.[26] Table 6 shows that both technical debt delinquency and differential evolution had significant positive effects on atrophy but code staleness did not. This provides the earliest empirical support for two of these purported sources of atrophy.

4.4.1. Hypothesis Tests

Table 6 summarizes the results. The significant positive effect of system age on atrophy (β = .013, T-value = 2.31, p<.05) provides evidence for our baseline idea that system atrophy increases as a system ages. ΔModularity had a negative effect on system atrophy (β = -.11, T-value = -2.09, p<.05), supporting Hypothesis 1. Its interaction with system age had a positive effect on system atrophy ($\beta = .12$, T-value = 2.04, $p<.05$), supporting Hypothesis 2. Thus, increasing codebase modularity over time decreases atrophy but this relationship weakens as a system ages. The results in Table 6 imply that one unit increase in modularity results in a 0.11 unit decrease in atrophy.[27] Halving dependencies among a system's modules thus reduces cyclomatic complexity by about 11%.

<< INSERT TABLE 6 AND FIGURE 7 HERE >>

The interaction plot in Figure 7 shows that modularity reduces atrophy in young systems but not in aged systems. The latter pattern implies that maintenance obligations incurred over incremental code changes emphasized by Rolland (2018) by themselves are not evolvability-stunting; but rather their buildup over time. Such reversal with age suggests modularity's diminishing returns. Modularity introduced very early in a system's lifespan appears to pay off disproportionately. Later on, developers must consider the cost of modularity-improving maintenance vis-à-vis scrapping and redeveloping the system from scratch.

The model explains only 11% of the variance in system atrophy, which is not surprising because (a) no prior empirical studies of system atrophy exist to build atop and (b) the first-difference approach removes any explanatory contribution of any time-invariant system and team characteristics that represent the bulk of prior IS development studies.

4.5. Atrophy's Downstream, Degenerative Consequences

Nomologically-downstream from our theoretical focus—i.e., beyond the shaded part of Figure 1—are atrophy's consequences discussed in §2.1. We examined post-hoc four such empirically-underexplored degenerative consequences that interest IS scholars. These are atrophy's causal effects on a system's: (a) Evolution stunting, (b) worsening bugginess, (c) developer attention attrition, and (d) codebase adoption losses. Data on three of them is cumulative through five years after the data used for the theorized model, consistent with them being causally-downstream, longer-term consequences of system atrophy. We used four Poisson models since most are count variables. The results appear in Table 7 (controls are omitted for brevity). Figure 6 illustrates the patterns of atrophy's effects on them. Our subsequent analysis shows that system atrophy retards IS evolution, increases bugginess, causes attrition of developers' attention, and discourages codebase adoption.

<< INSERT TABLE 7 AND FIGURE 6 HERE >>

First, we expect atrophy to stunt a system's evolution (e.g., Agarwal and Tiwana 2015; Pentland et al. 2020) because a more logically-convoluted codebase is less comprehensible to developers maintaining it (Arbesman

2017: 44; Barr 2018: 58; Clearfield 2019: 174; Rolland et al. 2018). Atrophy negatively influenced a system's *lifetime evolution rate* (β = -.003, T-value = -15.6, p<.001), measured as its logged cumulative lifetime frequency of commits to the codebase.[28] Each 10 unit increase in atrophy results in about 50 fewer yearly code commits. Second, atrophy also makes a system buggier and more failure-prone, symptomatic of IS degenerativity (e.g., Benbya et al. 2020; Fürstenau and Baiyere 2019). Using Subramanyam et al.'s (2012) strategy of counting in-process defects, we measured *bugginess* as the objective count of unresolved open technical issues with a five-year lag (2021). Its significant, positive relationship (β = .008, T-value = 14.6, p<.001) confirms this. Third, atrophy-induced code incomprehensibility would also cause developers to lose interest in it. Following Borges and Valente (2018), we proxied *developer attention* by using a cumulative objective count with a five-year lag (2021) of other Github developers who "follow" a system's codebase. A negative and significant (β = -.099, T-value = -250.3, p<.001) relationship shows that atrophy leads to the attrition of developers' attention. Finally, an atrophied codebase is unattractive to other developers to use as a foundation for future systems, imperiling generativity that leverages existing code to spawn new systems (Wareham et al. 2014). We proxied *codebase adoption* using a five-year lagged objective cumulative count of the number of times the codebase had been forked into a spinoff system. A negative relationship (β = -.14, T-value = 44.83, p<.001) tentatively suggests that atrophy discourages other developers from building on a codebase. Atrophy significantly mediated the effect of system age of all four (Sobel $T_{evolution}$ 2.29, $T_{bugginess}$ 2.28, $T_{attention}$ 2.31, T_{forks} 2.31; all p<.05).

4.6. Boundary Conditions and Robustness Tests

Six sets of robustness tests draw boundary conditions around the ownership, size, application domains of systems; panel vector estimation; and ruling out reverse causality. (The full results appear in the online appendix; with the relevant rows highlighted.) First, commercially-sponsored open-source systems are no less vulnerable to atrophy. Second, systems larger than those in our sample are likely to suffer more from atrophy. Third, we found no evidence that systems in some domains are more or less vulnerable to atrophy as they age. Fourth, the results were robust to reduced observational interval granularity i.e., when we doubled and quadrupled our quarterly observation window. Fifth, the results were robust using a panel vector autoregression model with modularity, atrophy, and other control variables as input time-series with a prediction lag of one and instrument lags of four. Finally, a Granger causality test ruled out reverse causality by showing that modularity can forecast atrophy (Ψ^2 64.5, p<.001) but atrophy cannot forecast modularity (Ψ^2 1.32, ns).

4.7. Five Limitations

First, findings from our open-source sample cannot readily be generalized to proprietary, closed-source systems. The inaccessibility of a large sample of the source code of proprietary systems spanning multiple decades make similar analyses implausible. Second, even the use of time-series trace-data and confirmation of Granger causality cannot establish true causality. Third, our use of the Java language to mitigate confounds warrants caution in generalizing to other languages. Fourth, this initial foray into the atrophy phenomenon could only examine a subset of moving parts in the theoretical model. Since atrophy can be precipitated by the changing infrastructural environment in which every single system instance is deployed, plausibly in thousands of organizations, that contextual interplay merits future research attention. That entails studying the source code of a single system deployed across thousands of organizations (e.g., Apache server). Finally, competitive intensity in a system's market environment—beyond its market segment controlled here—can pressure developers to take atrophy-inducing shortcuts to hasten releases. For example, competitive intensity in mobile apps plausibly exceeds desktop apps; which exceeds server and mainframe apps. Similarly, commercial apps likely face worse competitive pressures than open-source ones. Fully grasping these nuances requires access to proprietary source code.

5. DISCUSSION

We studied how once-modern systems age into unmaintainable, buggy albatrosses. The mechanism of system atrophy—a progressive decay in aging systems' internal logical coherence—introduced here is pervasive but scantly researched. Although scholarly recognition of degenerative IS change is nascent, the phenomenon has bread-and-butter consequences for IS. Our conceptualization of system atrophy from theoretical first principles and its first-ever corroboration springboarded our subsequent ideas.

Our use of trace data*streams* spanning a quarter century directly examined the evolving IT artifact itself. They spanned 19 million code changes, almost 200 million lines of code, and 9 million dependencies in 1,354 open-source systems as they evolved over decades. Our computationally-intensive approach exemplifies how open-source traces coupled with supercomputing horsepower permit IS scholars to address previously-infeasible questions. Our first-difference model shows that incrementing modularity slows atrophy but loses potency as a system ages.

5.1. Contributions and Theoretical Implications

Our substantive contributions—primarily to the IS evolution literature—are four-fold.

5.1.1. Contribution #1: System Atrophy from Theoretical First Principles

Our first contribution is theoretically developing atrophy as a mechanism to explain the unfolding over time of degenerative change in systems. This expands the nascent IS conversation on *how* IT artifacts degenerate and drift (Fürstenau and Baiyere 2019; Pentland et al. 2020), intertemporal changes within which get scant attention (Benbya et al. 2020). Headline-making IT meltdowns are on this nascent degenerative side of IS evolution, in contrast with its generative side that's historically dominated IS conversations (e.g., Wareham et al. 2014; Yoo et al. 2010). IS evolution is by definition a dynamic phenomenon, yet explanations where time is theoretically-integral have been surprisingly scarce. Our new conceptualization of system atrophy from theoretical first principles as intertemporal decay in a system's logical coherence is an initial step towards understanding such dynamics of IT artifacts' evolution. This mechanism demystifies how imperfect tweaks, hacks, patchwork, and workarounds progressively create an albatross. Evolution ironically then becomes its own showstopper: Atrophy emerges as a system evolves, which straitjackets further evolution.

Atrophy adds a theoretical connective tissue between the nomologically-adjacent IS evolution perspectives; a mechanism that bridges nomologically-upstream things that cause atrophy (○→🐦) with nomologically-downstream things that atrophy causes (🐦→○) in Table 2. On the upstream side, our model integrates for the first time its purported sources—debt, some modules evolving faster than others, and dying code—scattered across disparate IS evolution perspectives. Theoretically assimilating them into an aging-centric logic and empirically as dynamic, rival explanations synthesizes them into a parsimonious middle-range theory of system atrophy.

On the downstream side, we showed atrophy's long-term degenerative consequences that worry IS scholars (e.g., Benbya et al. 2020; Fürstenau and Baiyere 2019; Pentland et al. 2020). Specifically, atrophy stunting systems' evolution handicaps organizations that rely on them. Atrophy-induced bugginess increases their vulnerability to IT meltdowns. Atrophy-induced inflexibility in existing systems can handicap firms' implementation of new systems—by jeopardizing integration and their exorbitant maintenance costs cannibalizing budgets—in previously-unrecognized ways. Atrophy also discourages other developers from building atop a codebase. For firms open-sourcing development (Lindberg, Berente, Gaskin and Lyytinen 2016), atrophy thus depletes engagement from the broader developer community. Finally, an atrophied codebase dissuades other developers from using it to develop new systems. This jeopardizes the potential of code reuse increasingly proffered to improve software quality. Further, since new systems are often built atop old code (Arthur 2009: 189; Ghanbari et al. 2018), this introduces an unrecognized route for atrophy to infest new systems. Mediation by atrophy of aging's effects on all four of these degenerative consequences further underscores its role as a central mechanism.

5.1.2. Contribution #2: The Passage of Time

Second, bringing system age to the foreground revealed previously-unseen degenerative dynamics in evolving systems. In the IS evolution literature where time is empirically-ubiquitous yet theoretically-invisible (e.g., Fürstenau and Baiyere 2019; Rolland et al. 2018; Tiwana 2018), time served to *theoretically* integrate into our middle-range theory purported sources of degenerative change scattered across various IS evolution studies. It is the culmination *over time* of unrepaid technical debt, differentially-evolved modules, and accumulation of comatose and dead code that exacerbates atrophy. Complementing such time-centric theorizing with trace datastreams empirically assimilated them to show that age ravages systems even after accounting for these as competing explanations. First-differencing permitted modeling intertemporal changes in these purported sources, thus isolating their cumulation that would otherwise be correlated with system age. This underscores how time can be a powerful, underappreciated theorizing device alongside trace datastreams.

Figure 8 illustrates two further degenerative microdynamics. First, systems increased their reliance on external modules (the --- line) more than adding native modules (the — line). Second, as native modules increased, so did the dependencies among them (the … line). The shrinking gap between the … and the — lines is circumstantial evidence for attempts such as refactoring to deescalate atrophy in aging systems.

< INSERT FIGURE 8 HERE >

5.1.3. Contribution #3: Modularity as an Antidote for Atrophy

Our third contribution is showing how modularity increments over the course of a system's evolution slow its atrophy. To the IS conversation that implicitly assumes modularity to be immutable (e.g., Agarwal and Tiwana 2015; Subramanyam et al. 2012; Tiwana 2018), this recognizes it as an evolving property. Capitalizing on Benbya et al.'s (2020: 9) recent call to understand changes in the *internal* structure of an IT artifact by zooming into interactions among its parts, we add to that conversation novel insight into how *changes* in modularity over time allow a system to age more gracefully and perhaps even prolong its useful life.

Modularity's advantages thus stretch far beyond IS inception known in prior studies and into their subsequent upkeep. Bolstering modularity early in a system's lifespan can make it more atrophy-resilient, yet initial architecture has never been part of the conversation on IS implementation. Since initial modularity circumscribes modularity's lifelong mutability, attention to it at the outset likely makes systems more atrophy-resistant, maintainable, and evolvable. We anticipate two reasons why this is easier said than done. First, it is hard to observe modularity before a system exists. Smith (1981: 382) presaged decades ago that an artifact's form cannot be observed until it exists, and it is usually not noticed until after it has existed for a while. Second, the concerted time and effort that modularization demands will compete with upfront project completion pressures. Nonetheless, trying to modularize ex-post a monolithic codebase is like slapping Band-Aid on a deep wound.

5.1.4. Contribution #4: Aging as the Scourge of IS Degenerativity

Our final contribution is showing that modularity increments lose potency as an antidote as a system ages. In software engineering has long considered code degradation an entropy problem that can be stopped as long as enough effort is dedicated to maintaining and refactoring code (Coleman 1994). Our findings show that such maintenance slows but cannot forestall atrophy. An aged system's code (see Figure 7) might have become so logically incoherent that attempts at increasing its modularity might break more unknown dependencies among modules than they fix, thus worsening atrophy relative to simply leaving it alone. Plausibly, its codebase's logical degradation has made it unfixably "brittle" (Rolland et al. 2018). Teams' tendency to prioritize what's temporally-closer and ignore what's farther out likely further neutralizes this antidote's potency in the long run.

5.1.5. Secondary Contributions to Software Engineering

To software engineering, where the absence of theory has hamstrung empirical scrutiny of code rot for decades (Garcia et al. 2022; Herraiz et al. 2013), our contributions are three-fold. First, we contribute the first, long-awaited (Garcia et al. 2022) large-scale empirical corroboration of this phenomenon. Our theoretically-grounded way to assess atrophy by analyzing code provides an elusive "maintainability index" (Herraiz et al. 2013: 17). Second, we contribute an atrophy-reducing, "anti-regressive" intervention to guide code changes in aging systems (Herraiz et al. 2013: 5; Sobhy et al. 2021). Third, we substantiate atrophy's downstream consequences. Atrophy-causing bugginess corroborates Garcia et al.'s (2022) suspicion that low "architectural quality" makes code defect-prone; and it retarding evolution corroborates that degraded codebases are harder to maintain (Garcia et al. 2022). Yet-unconsidered consequences include: (a) losing developer engagement, which impairs code quality and (b) discouraging forking, which imperils code reuse.

5.2. Three Implications for Practice

First, IS managers must not assume that faster evolution is better. Faster evolution exacerbates opportunities to perpetuate atrophy, requiring cognizance of an evolution↔atrophy tradeoff. Managers can slow atrophy by stabilizing a system's modularity after major code changes and avoiding tardiness in repaying technical debt. Second, upfront modularization bakes in resilience to atrophy. Not shortchanging initial modularization even under schedule pressures preempts lifelong costlier maintenance, poor evolvability, and bugginess. Third, an atrophied but functional system can still have some useful life left in it. It requires forbidding changes within its codebase and instead invoking its salvageable functionality via API calls, code wrappers, and interface frontends. An example of this strategy is the US Air Traffic Control system that still uses WWII-era radar technology, is coded in long-defunct ADA-Pascal, and has repeatedly proved impossible to modernize. The FAA subsequently left the native system code untouched and built APIs around it to allow it to interface with modern iPads and external applications.

5.3. Four Questions for Future Research

First, how can atrophy in one system metastasize to other interacting systems? Can the absence of interaction among systems—silence—slow it? Second, what are atrophy's longer-term business consequences; such as system death? Third, how does *inter*system modularity (e.g., across a firm's IT portfolio)—rather than within-system modularity studied here—affect atrophy? Fourth, what other mechanisms—plausible hidden pathways—can explain IS degenerativity? Three other mechanisms beyond system atrophy in this initial foray merit attention. First, exogenous changes over decades in the external *infrastructural environment* within which an application operates (e.g., microservices, cloud infrastructure, and APIs). Second, changes over time in the paradigmatic mindset and development *philosophy* such as a shift from object-orientation to microservices and containerization (e.g., Docker). Third, in platform ecosystems, an application "platform-sources" some functionality through API calls. An intriguing mechanism there is the *mode* through which an application evolves i.e., whether using *de novo* code written from scratch or via reuse (e.g., platform sourcing and code imports).

In conclusion, age is the scourge that makes atrophy inescapable in aging systems. This study is a baby step in deconstructing this pervasive phenomenon. Modularity is an antidote that can prolong systems' productive life. In the long run, resistance is futile. Death is inevitable.

Acknowledgments: We are grateful for inputs from Joshua Garcia, Hadi Ghanbari, Rick Kazman, Gregorio Robles, and Mikko Siponen. Safadi acknowledges support from the Terry-Sanford Research Award. Tiwana dedicates this article to the memory of Dick Poist, his former department chair.

REFERENCES

Agarwal, R., and Tiwana, A. 2015. Evolvable Systems: Through the Looking Glass of IS. *Information Systems Research*. **26**(3) 473-479.
Arbesman, S. 2017. *Overcomplicated: Technology at the Limits of Comprehension*. Portfolio, New York.
Arthur, B. 2009. *The Nature of Technology*. Free Press, New York.
Banker, R. and Slaughter, S. 2000. The Moderating Effects of Structure on Volatility And Complexity in Software Enhancement. *Information Systems Research*. **11**(3) 219-240.
Barr, A. 2018. *The Problem with Software: Why Smart Engineers Write Bad Code*. MIT Press, Cambridge, MA.
Benbya, H., Nan, N., Tanriverdi, H., and Yoo, Y. 2020. Complexity and Information Systems Research in the Emerging Digital World. *MIS Quarterly*. **44**(1) 1-17.
Biémont, C., and Vieira, C. 2006. Junk DNA as an Evolutionary Force. *Nature*. **443**(7111) 521-524.
Booch, G. 2007. The Economics of Architecture-First. *IEEE Software*. **24**(5) 18-20.
Booch, G. 2008. Measuring Architectural Complexity. *IEEE Software*. **25**(4) 14-15.
Borges, H., and Valente, M. 2018. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software*. **146** 112-129.
Clearfield, C. 2019. *Meltdown: What Plane Crashes, Oil Spills, and Dumb Business Decisions Can Teach Us About How to Succeed at Work and at Home*. Penguin.
Coleman, D. 1994, Using Metrics to Evaluate Software System Maintainability. *IEEE Computer*. 27(8) (1994), 44-49.
Coleridge, S. 1798 *The Rime of the Ancyent Marinere*. in: Lyrical Ballads (gutenberg.org/ebooks/9622), J. & A. Arch, London.
Eick, S., Graves, T., Karr, A., Marron, J., and Mockus, A. 2001. Does Code Decay? Assessing Evidence from Change Management Data. *IEEE Transactions on Software Engineering*. **27**(1) 1-12.
Ford, N., Parsons, R., and Kua, P. 2017. *Building Evolutionary Architectures: Support Constant Change.* O'Reilly, CA.
Fürstenau, D., and Baiyere, A.K., N. 2019. A Dynamic Model of Embeddedness in Digital Infrastructures. *Information Systems Research*. **30**(4) 1319-1342.
Garcia, J., Kouroshfar, E., Ghorbani, N., and Malek, S. 2022. Forecasting Architectural Decay from Evolutionary History. *IEEE Transactions on Software Engineering*. 48(7), 2439-2454.
Ghanbari, H., Vartiainen, T., and Siponen, M. 2018. Omission of Quality Software Development Practices: A Systematic Literature Review. *ACM Computing Surveys*. **51**(2) 1-27.
Herraiz, I., Rodriguez, D., Robles, G., and Gonzalez-Barahona, J. 2013. The Evolution of the Laws of Software Evolution: A Discussion Based On a Systematic Literature Review. *ACM Computing Surveys*. **46**(2) 1-28.
Kelly, J., and McGrath, J. 1988. *On Time and Method*. Sage, Thousand Oaks CA.
Kruchten, P., Nord, R., and Ozkaya, I. 2012. Technical Debt: From Metaphor to Theory and Practice. *IEEE Software*. **29**(6) 18-21.
Lehman, M. 1979. On Understanding Laws, Evolution, and Conservation in the Large-program Life Cycle. *Journal of Systems and Software*. **1** 213-221.
Lindberg, A., Berente, N., Gaskin, J., and Lyytinen, K. 2016. Coordinating Interdependencies in Online Communities. *Information Systems Research*. **27**(4) 751-772.

MacCormack, A., Rusnak, J., and Baldwin, C. 2006. Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code. *Management Science*. **52**(7) 1015-1030.

McCabe, T. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering*. **2**(4) 308–320.

Minsky, M. 2006. *The Emotion Machine*. Simon & Schuster, New York.

Narayanan, S., Balasubramanian, S., and Swaminathan, J. 2009. A Matter of Balance: Specialization, Task Variety, and Individual Learning in a Software Maintenance Environment. *Management Science*. **55**(11) 1861-1876.

Pentland, B., Liu, P., Kremser, W., and Hærem, T. 2020. The Dynamics of Drift in Digitized Processes. *MIS Quarterly*. **44**(1) 19-47.

Rolland, K., Mathiassen, L., and Rai, A. 2018. Managing Digital Platforms in User Organizations: The Interactions Between Digital Options and Digital Debt. *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.

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

Sobhy, D., Bahsoon, R., Minku, L., and Kazman, R. 2021. Evaluation of Software Architectures under Uncertainty: A Systematic Literature Review. *ACM Transactions on Software Engineering and Methodology*. **30**(4) 1-50.

Steelman, Z., Havakhor, T., Sabherwal, R., and Sabherwal, S. 2019. Performance Consequences of Information Technology Investments: Implications of Emphasizing New or Current Information Technologies. *Information Systems Research*. **30**(1) 204-218.

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. 2016. Platform Desertion by App Developers. *Journal of Management Information Systems*. **32**(4) 40-77.

Tiwana, A. 2018. Platform Synergy: Architectural Origins and Competitive Consequences. *Information Systems Research*. **24**(9) 829–848.

Tufte, E. 2020. *Seeing with Different Eyes: Meaning, Space, Data, Truth*. Graphics Press, Cheshire, CT.

Wareham, J., Fox, P., and Giner, J. 2014. Technology Ecosystem Governance. *Organization Science*. **25**(4) 1195-1215.

Wittmann, M. 2017. *Felt Time: The Science of How We Experience Time*. MIT Press, Boston.

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

Endnotes

[1] Albatross as an idiomatic expression for something that causes persistent anxiety originated from Samuel Coleridge's (1798) composition in the late 1700s.

[2] So has software engineering (a recent literature review appears in Ghanbari, Vartiainen and Siponen 2018; Sobhy, Bahsoon, Minku and Kazman 2021). A handful of IS maintenance studies are exceptions; but they are single-system or small-sample studies or of pre-Internet mainframe systems (e.g., Narayanan, Balasubramanian and Swaminathan 2009; Rolland, Mathiassen and Rai 2018).

[3] The software quality literature in software engineering also documents this tendency to take quick-and-dirty shortcuts under time pressure (for a recent review, see Ghanbari et al. 2018).

[4] This literature follows the tradition of the systems dynamics literature, where the closest analogous concept is system fragility. That work focuses on broad, generic classes of systems spanning a gamut of scales, including mechanical systems, biological organisms, organizations, and even entire economies. Fragility arises when a system's parts begin interacting in ways that its initial design did not anticipate. A change in one part can then create ripple effects on others, requiring time-consuming iterations to resolve. Even when an opportunity arises to improve one inadequate part of the system, the unknown vulnerability to a complete system breakdown deters change. Using analytical models rather than empiricism, this literature focuses more on systems dynamics-relevant consequences such as oscillations, snowball effects, and meltdowns rather than the phenomenon of system fragility. Notwithstanding the powerful insight that unanticipated interactions between its parts can destabilize an entire system, systems dynamics models study generic systems using analytical models. The scale-agnostic nature of generic systems makes it conceptually challenging to draw a boundary around *a* system to be able to empirically study it.

[5] Recent software engineering reviews confirm that code rot has not evolved beyond an atheoretical, descriptive metaphor (e.g., Garcia et al. 2022; Herraiz et al. 2013).

[6] Minsky's examples of *suitcase words* include conscience, consciousness, thinking, morality, right, and wrong. Even the word rot's meaning in the prior software engineering ranges from software code decaying in its logical coherence, digital information losing integrity over time, storage media becoming obsolete, and installs getting corrupted.

[7] Un examen attentif de cette figure révélera deux œufs de Pâques.

[8] For example, service oriented architectures, centralized versus decentralized IT architectures, and monolithic designs are simply a more or less modular ways of organizing an IT system. Other architecture properties empirically accounted for by our model include module granularity, invocation of external modules, code volume, technical debt.

[9] Antidote ≠ vaccine; antidotes counteract, vaccinces prevent.

[10] https://www.cnbc.com/2020/04/06/new-jersey-seeks-cobol-programmers-to-fix-unemployment-system.html

[11] Such IT infrastructure change is external to a system's codebase itself, thus not central to atrophy *within* a codebase. Any system depends on assumptions about its surrounding technological environment such as IT infrastructure (e.g., hardware it is running on, operating system services available, bandwidth, network protocols, and display devices), which is also susceptible to changes over long-time scales. Such IT infrastructure evolution, consolidation, and fragmentation are often oblivious to a particular system (Fürstenau and Baiyere 2019), which means that a system might inhabit an environment different from the one in which it originated. This system↔environment mismatch can invalidate a system's original design assumptions. Such changes can aggravate the *consequences* of system atrophy even without changes in a system's native code e.g., a new version of a code library, a compiler update, or a patch to the operating system in which it runs or its demise. For example, Blackberry OS8→10 and Windows XP→Vista made *all* existing apps for those environments defunct. Even the IT system↔infrastructure boundary can evolve so that functionality previously built into the codebase might subsequently be infrastructurally-provisioned (e.g., microservices APIs and cloud storage in the 2020s) (Agarwal and Tiwana 2015).

[12] Although Github itself was launched in 2008, many projects ported their git repositories with full commit histories to Github.

[13] We stopped at 2016 because of a major change in the Java programming language in 2017.

[14] Prior IS research has used a static McCabe's measure as a snapshot proxy for coordinative complexity and decision density.

[15] In Java code, these are lines matching the following patterns: `"else", r"for\s+\(.*\)", r"if\s+\(.*\)", r"case\s+\w+:", "default:", r"while\s+\(.*\)"`

[16] In Java code, these are lines containing the following tokens: `"assert", "break", "continue", "return"`

[17] Modern metrics in software engineering build on object-oriented concepts, thus tap into architecture more than code quality; where McCabe is the most popular metric for assessing code quality.

[18] We used the *gitinspector* tool; source code available at `github.com/ejwa/gitinspector/blob/master/gitinspector/metrics.py`

[19] In contrast, Subramanyam (2012) did not directly measure modularity; their unit of analysis was the module (they called it a component) in their study of a single system's 89 modules. They instead measured the independence dimension of module granularity, which was a count of a module's internal interfaces.

[20] A module depends on another module when one file from the first module imports the second module.

[21] https://www.oracle.com/java/technologies/javase/8-compatibility-guide.html

[22] To test stationarity, we used using Stata's *xtunitroot* procedure to run a series of Fisher-type tests with lags of 1 (using residuals from model 4 in Table 6). This approach is appropriate in unbalanced panels like ours. The test rejected the null hypothesis that all panels contain unit roots; the alternative hypothesis is that at least one panel is stationary.

[23] Between-group differences using static modularity yielded consistent main effects but a nonsignificant interaction effect.

[24] Burstiness in a statistical sense refers to the intermittent increases or decreases in a variable.

[25] This is a centralization—not centrality—measure. Our "network" is conceived as commits to files; therefore higher centralization means few files are receiving most commits, hence differential evolution.

[26] These three controls are inappropriate instruments for Δmodularity because they are correlated with system atrophy. As a robustness check, we also created a composite product term of these three sources; the interaction of this composite with system age was also positive and significant.

[27] To assess whether—independent of system age—modularity improvements over time have diminishing benefits, we added post-hoc to our model the squared term of Δmodularity. The squared term was negative and significant ($\beta = -.19$, T-value = -4.86, $p<.001$), suggesting that improving modularity exponentially slows atrophy.

[28] Unlike our control for intertemporal swings (Δs) in evolution rate, this is cumulative over the system's lifespan.

Figures and Tables

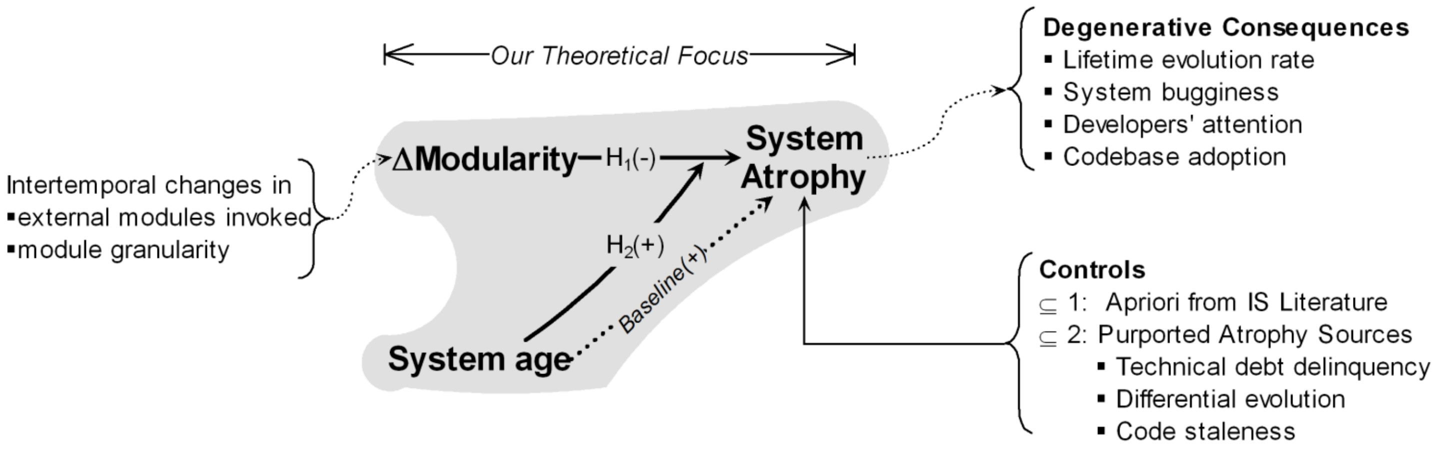


Figure 1: Our research model in its broader IS nomological context.

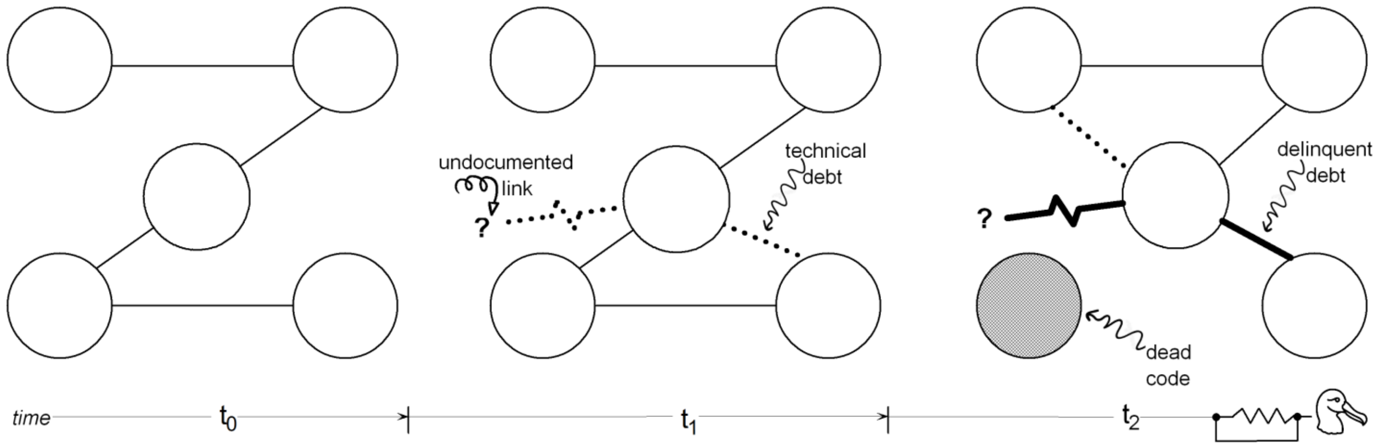


Figure 2: A collineation diagram illustrating the progression of atrophy in a five-module (○) system over three time periods; t_1 shows the appearance of undocumented linkages and technical debt (…); and t_2 of dead code (shaded ○) and delinquency of existing technical debt (—).

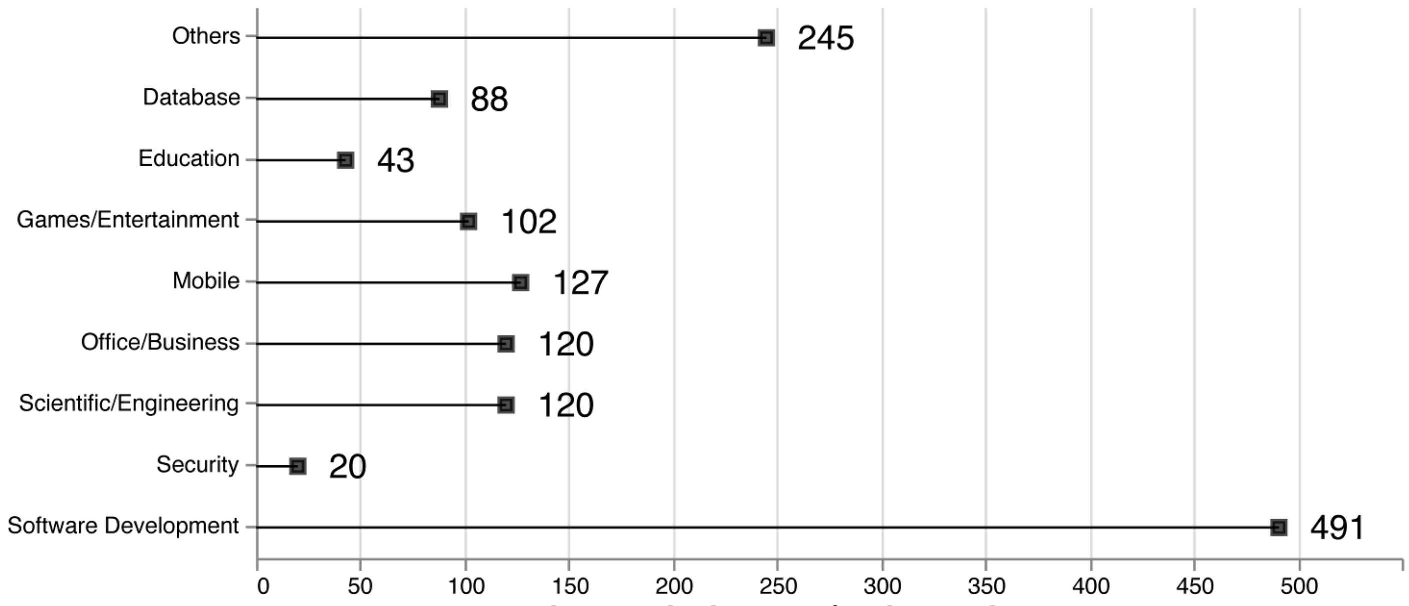


Figure 3: Our sample spanned a diversity of application domains.

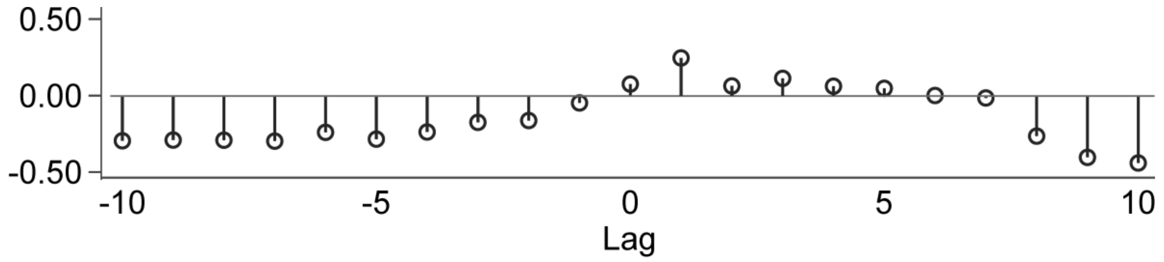


Figure 4: Cross-correlogram between system atrophy and changes in modularity over time.

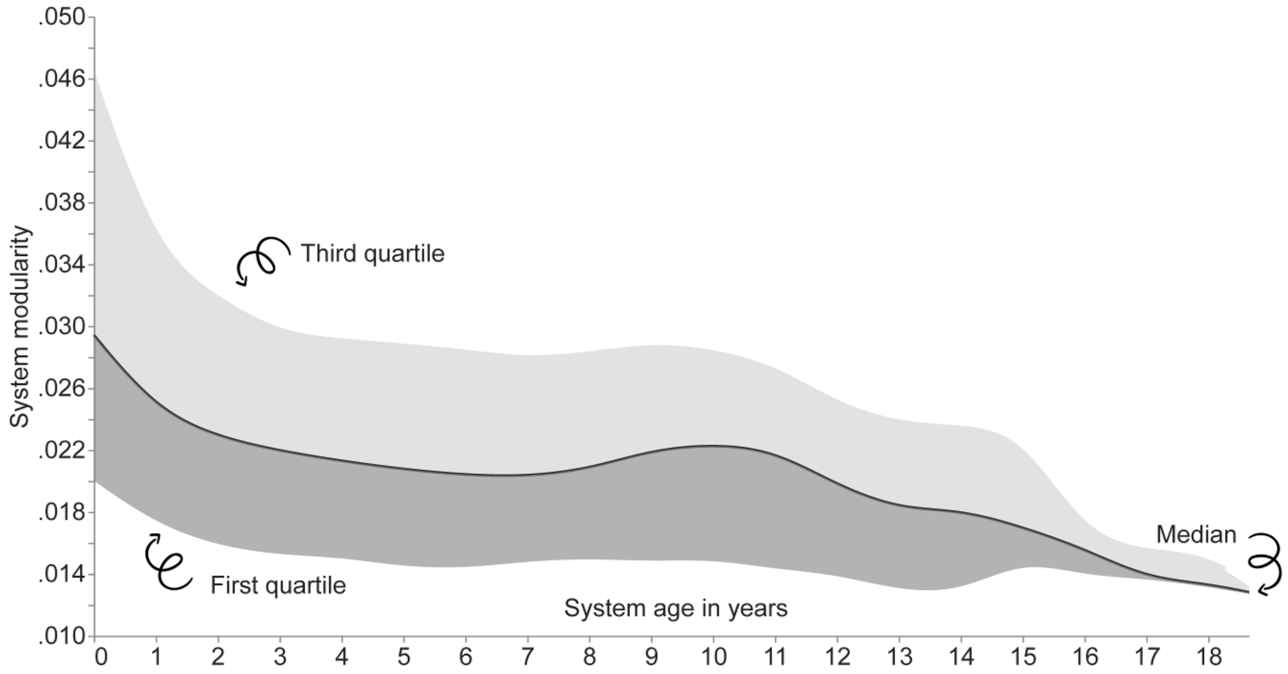


Figure 5: Sample-wide modularity degraded over time.

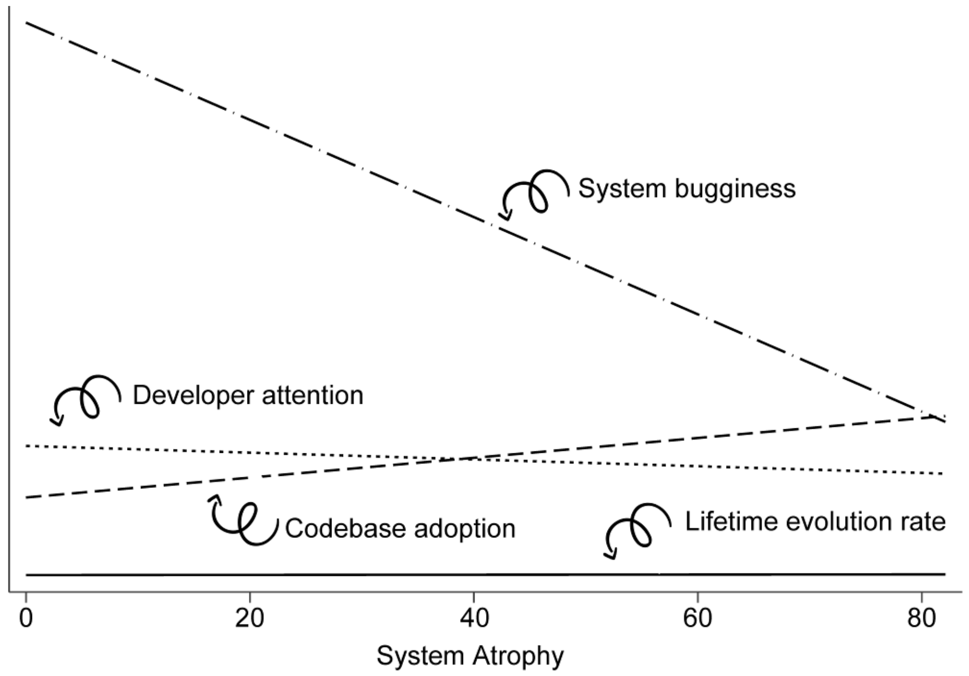


Figure 6: Patterns of four degenerative consequences of system atrophy (decomposition in online appendix).

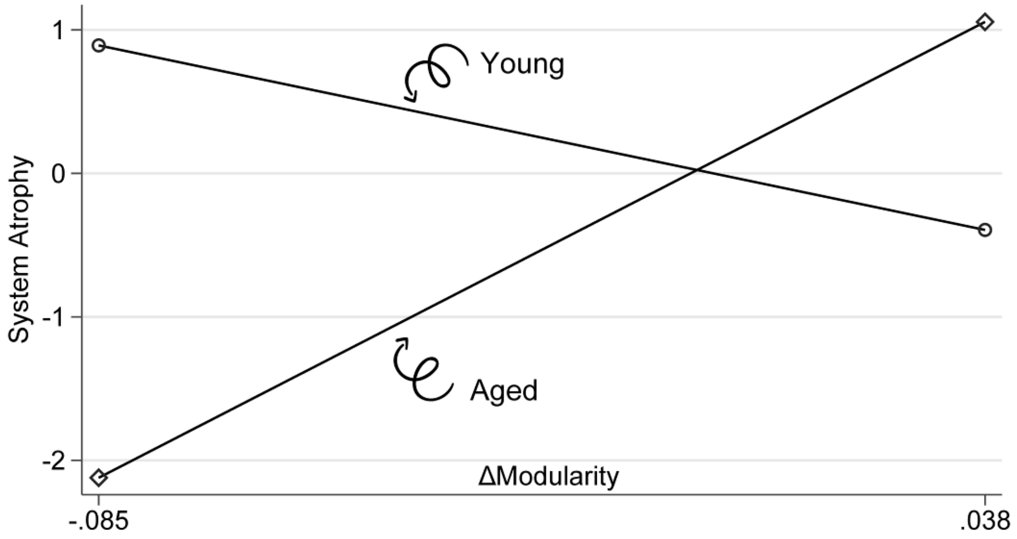


Figure 7: Effects of changes in modularity over time on atrophy for young and aged systems (±1sd).

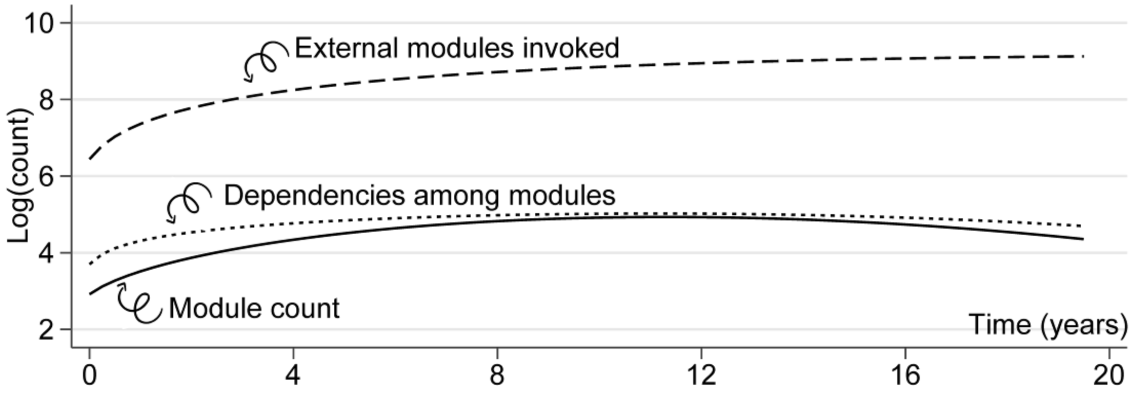


Figure 8: Systems evolved more by invoking external modules rather than adding modules to their own codebase. (The Y-axes are the logarithms of the number of external modules invoked, dependencies among modules, and module counts.)

Table 1: Construct definitions.

Core constructs	Definition	Guiding references
System atrophy	Intertemporal decay in the internal logical coherence of a system's codebase	Booch (2007); Fürstenau and Baiyere (2019); Simon (1962)
System age	Chronological age of an information system since its inception	—
Modularity	The degree to which a system's modules are loosely-coupled through standardized interfaces	Subramanyam et al. (2012); Tiwana (2018)
Instruments[a]		
External module invocation	The extent to which a system uses third-party modules	Ford (2017: 115)
Module granularity	The fine-graininess of a system's modules	Subramanyam et al. (2012)
Controls (cluster ⊆ 1: apriori)		
Development burstiness	Intertemporal change in the number of individuals working on a system's codebase	Narayanan et al. (2009)
Team turnover	Intertemporal turnover of developers working on the codebase	Faraj et al. (2011)
Evolution rate	Count of code commits in an observation interval; Δ equals evolution intensification	Tiwana (2018)
Code volume	A system's codebase's line count	Subramanyam et al. (2012)
Controls (cluster ⊆ 2: purported atrophy sources)		
Technical debt delinquency	Incurred technical debt that has not been repaid	Rolland et al. (2018)
Differential evolution	The extent to which a system's modules evolve at different rates	Fürstenau and Baiyere (2019)
Code staleness	Accumulation of dying or dead code in the codebase	Arbesman (2017: 21)

[a]Δ*s* are used to assess intertemporal *dynamics* of both instruments.

Table 2: Concepts proximate to system atrophy in prior IS evolution and software engineering studies.

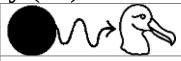
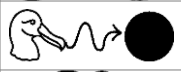
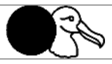
Concept (●)	**Proximity to atrophy ( )**		**Representative references**
Technical debt	Nomologically-upstream		Rolland et al. (2018); Narayanan et al. (2009)
Degenerativity	Nomologically-downstream		Pentland et al. (2020); Benbya et al. (2020); Fürstenau and Baiyere (2019)
Code rot	Conceptually-proximate		Booch (2008); Eick et al. (2001); Lehman (1979)

Table 3: Trace measures used for all constructs.

Constructs	**Measure**
System atrophy$_{t1}$	Δ(cyclomatic complexity ÷ #lines of code)$_{t0\rightarrow t1}$
System age$_{t1}$	#Quarters since the inception of the system
ΔModularity$_{t0\rightarrow t1}$	Δ(#modules ÷ #dependencies among modules)$_{t0\rightarrow t1}$
Instruments	
External module invocation intensification	Δ(#External modules ÷ (#Native + #External modules))$_{t0\rightarrow t1}$
Module granularity growth	Δ(#Modules ÷ #Files)$_{t0\rightarrow t1}$
Controls	
Development burstiness	Δ(Log(#Developers active in each quarter))$_{t0\rightarrow t1}$
Team turnover	Δ{x:x∈Team$_{t1}$ AND x∉Team$_{t0}$}
Evolution intensification	Δ(Log(# Code commits in each quarter))$_{t0\rightarrow t1}$
Code volume growth	Δ(Log(#Lines in the codebase))$_{t0\rightarrow t1}$
Technical debt delinquency	Δ(#Lines of documentation ÷ #Lines of code)$_{t0\rightarrow t1}$
Differential evolution of modules	Δ{Max$_{population}$(#Lines of documentation ÷ #Lines of code) – (#Lines of documentation ÷ #Lines of code)}$_{t0\rightarrow t1}$
Code staleness	Δ(#Files in codebase ÷ #changed files in codebase)$_{t0\rightarrow t1}$

Table 4: Construct correlations and descriptive statistics.

Construct[†]	$\bar{x}$	σ	IQR	1	2	3	4	5	6	7	8	9	10	11
1. System atrophy	.05	1.95	.32											
2. System age (quarters)	13.02	11.58	13.00	-.03***										
3. Modularity	.03	.05	.02	.00	-.11***									
4. External module invocation	58.23	19.70	26.53	.01	-.11***	-.16***								
5. Module granularity	.15	.12	.10	.00	-.18***	.73***	-.06***							
6. Active developer count[a]	1.27	.93	1.25	-.02***	.15***	-.11***	-.16***	-.12***						
7. Team turnover (non-Δ)	.45	.37	.75	.00	.10***	-.08***	.09***	-.07***	.29***					
8. Evolution rate[b]	5.62	1.65	2.03	-.00	-.05***	-.16***	-.26***	-.24***	.46***	.01				
9. Code volume[c]	11.19	1.52	1.90	-.01*	.43***	-.41***	-.46***	-.52***	.43***	.10***	.42***			
10. Documentation coverage[d]	2.67	.21	.24	.01	-.12***	.00	.27***	.08***	-.03***	.00	-.04***	-.24***		
11. Differential evolution	.35	.11	.13	.00	.11***	-.11***	-.10***	-.12***	.08***	.05***	.20***	.19***	-.05***	
12. Code staleness	.35	2.07	.31	.01*	-.00	.01	.01*	.03***	-.00	.05***	-.02***	-.03***	.01	-.02***

[†]Δ in our first-difference model estimates: [a]development burstiness; [b]evolution intensification; [c]code volume growth; [d]technical debt delinquency

$^{*}p < .05$, $^{**}p < .01$, $^{***}p < .001$; 1,354 systems, 23,407 observations; instruments are in the shaded rows.

Table 5: Stage 1 instrumental variables model.

	Change in Modularity
System age (cumulative)	-.01(-.96)
ΔExternal module invocation	-.24**(-3.10)
ΔModule granularity	.58***(5.74)
Development burstiness	.01(1.40)
Team turnover	-.00(-.31)
Evolution intensification	.00(.62)
Code volume growth	-.18*(-2.57)
Technical debt delinquency	-.07(-1.41)
Differential evolution of modules	-.01(-1.44)
Code staleness	-.01(-.72)
N	23407
R^2(model-F)	52%(21.26)
Log-Likelihood	61,580

$\beta(t)$; $^{*}p < .05$, $^{**}p < .01$, $^{***}p < .001$; 1,354 systems, 23,407 observations; instruments are shaded.

Table 6: First-difference model predicting system atrophy (Stage 2; hypothesis tests).

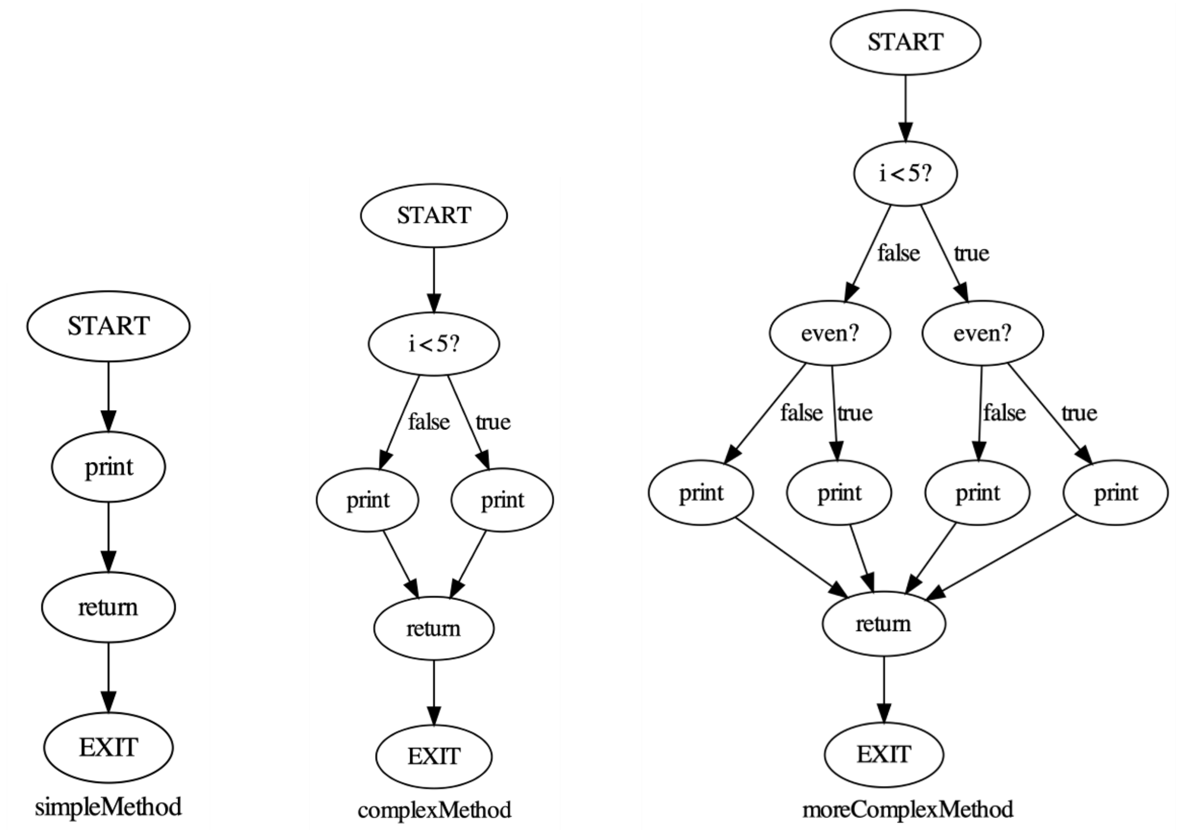
	Model 1	Model 2	Model 3	Model 4	Model 5
Predictors	Controls $\subseteq$ 1	Controls $\subseteq$ 2	System age	ΔModularity	Interaction
System age			.01*(2.31)	.01*(2.36)	.02**(3.28)
ΔModularity				-.11*(-2.09)	-.21**(-2.97)
System Age*ΔModularity					.12*(2.04)
Controls $\subseteq$ 1: Apriori from IS literature					
Development burstiness	-.05***(-3.47)	-.05***(-3.49)	-.05***(-3.46)	-.05***(-3.35)	-.05***(-3.32)
Team turnover	.01(1.36)	.01(1.05)	.01(1.14)	.01(.74)	.00(.32)
Evolution intensification	-.03**(-2.59)	-.03**(-2.97)	-.03**(-2.99)	-.03***(-3.30)	-.03***(-3.63)
Code volume growth	.29***(7.41)	.30***(7.76)	.31***(7.69)	.26***(5.65)	.27***(5.92)
Controls $\subseteq$ 2: Purported atrophy sources					
Technical debt delinquency		.10**(2.89)	.10**(2.90)	.09**(2.86)	.09**(3.04)
Differential evolution		.01*(1.76)	.01*(1.75)	.01(1.36)	.01(1.65)
Code staleness		.03(.90)	.03(.91)	.03(.86)	.02(.97)
R^2(model-F)	8%***(16.5)	9%***(11.6)	9%***(12.6)	10%***(12.9)	11%***(11.9)
F-change		11.47***	12.65***	12.97***	11.88***
LR-test χ^2		298.95***	4.69*	266.80***	94.92***
Log-Likelihood	-48,259	-48,110	-48,107	-47,974	-47,927

$\beta(t)$; $^{*}p < .05$, $^{**}p < .01$, $^{***}p < .001$; 1,354 systems, 23,407 observations; hypothesis tests highlighted.

Table 7: System atrophy's ( ) detrimental effect on four downstream outcomes.

Org Outcome		*System atrophy's effect*	*Model*	*N*	*5-year lag*	*Measure*
Lifetime evolution rate	—	-.003***(-15.59)	Poisson	1,345 systems	●	Log(#Lifetime commits ÷ system age)
System bugginess	+	.008***(14.60)	Poisson	1,345 systems	●	#unresolved open issues 5 years later
Developers' attention	—	-.099***(-250.28)	Poisson	21,452 system-quarters		Cumulative count of codebase watchers
Codebase adoption	—	-.014***(-44.83)	Poisson	1,345 systems	●	Lifetime codebase fork count 5 years later

Appendix 1: Illustration of inference of system atrophy from Java code changes

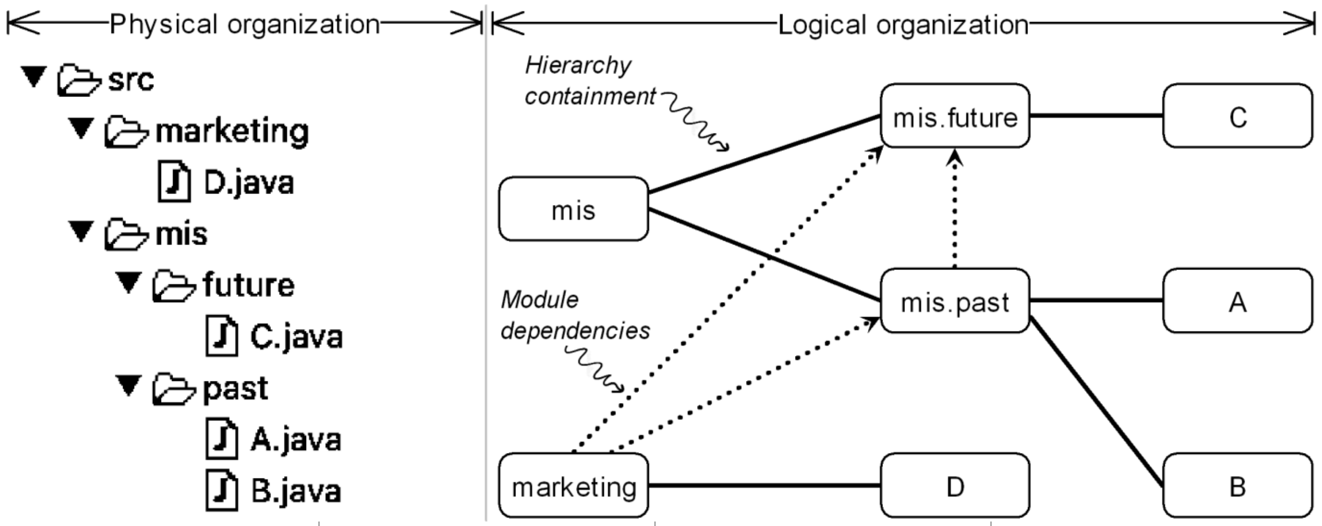


```
public static void simpleMethod(int i) {
        System.out.println(i);
}
public static void complexMethod(int i) {
        if(i < 5) {
                System.out.println(i + " is less than 5");

        } else {
                System.out.println(i + " is more or equal to 5");
        }
}
public static void moreComplexMethod(int i) {
        if(i < 5) {
                if (i % 2 == 0)
                        System.out.println(i + " is less than 5 and is even");
                else
                        System.out.println(i + " is less than 5 and is odd");
        } else {
                if (i % 2 == 0)
                        System.out.println(i + " is more or equal to 5 and is even");
                else
                        System.out.println(i + " is more or equal to 5 and is odd");
        }
}
```

Figure A1: An illustration of atrophy over two changes in a Java method code snippet. Envisioning a different code structure with two methods each taking charge of one behavior would have mitigated system atrophy.

Appendix 2: Illustration of inference of modularity from Java code structure



Source code of A.java
```
package mis.past;

import mis.future.*;

public class A {
        private B b;
        private C c;
}
```

Source code of B.java
```
package mis.past;

public class B {

}
```

Source code of C.java
```
package mis.future;

public class C {

}
```

Source code of D.java
```
package marketing;

import mis.past.*;
import mis.future.*;

public class D {
        private C c;
        private A a;
}
```

Figure A2: An illustration using Java code snippets of the correspondence between the physical (left) and logical (right) organization of modules A-D that we used to infer modularity by analyzing source code. Packages are Java's nomenclature for modules.

Atrophy in Aging Systems: Evidence, Dynamics, and Antidote
Online Appendix: Robustness Checks and Boundary Conditions

A: Generalization across firm-sponsored versus communal systems

Since firm ownership is a time-invariant dummy variable, it is unusable as a predictor in a fixed-effects model but can interact in it. (We used a first-difference model with system fixed effects predicting system atrophy.) The interaction of this dummy with (Δmodularity×system age) was nonsignificant (β = .056, T-value = .956, n.s.), suggesting that the results are robust to whether a system is firm-sponsored or community-driven. This dummy's interaction with system age was also nonsignificant (β = -.007, T-value = -1.92, n.s.), suggesting that firm-sponsored open-source systems are no less vulnerable to atrophy than those built entirely by unpaid volunteers.

	Step (1)		Step (2)		Step (3)	
	β	T-value	β	T-value	β	T-value
ΔModularity	-.210**	(-2.972)	-.210**	(-2.972)	-.209**	(-3.022)
Development burstiness	-.049***	(-3.317)	-.049***	(-3.316)	-.049***	(-3.307)
Evolution intensification	-.034***	(-3.629)	-.034***	(-3.630)	-.034***	(-3.654)
Code volume growth	.272***	(5.915)	.272***	(5.916)	.273***	(5.972)
Team turnover	.002	(.319)	.002	(.322)	.002	(.316)
Technical debt delinquency	.091**	(3.039)	.091**	(3.040)	.090**	(2.969)
Differential evolution	.011	(1.649)	.011	(1.650)	.011	(1.646)
Code staleness	.017	(.969)	.017	(.969)	.016	(.999)
System age	.019**	(3.284)	.024***	(3.719)	.023***	(3.643)
Δmodularity×system age	.119*	(2.041)	.119*	(2.041)	.071	(1.340)
Firm_owned_dummy×System age			-.007	(-1.918)	-.006	(-1.452)
Firm_owned_dummy×(Δmodularity×system age)					.056	(.956)

B: Smaller versus larger systems

To empirically assess whether our results hold for larger systems, we interacted the systems' code volume with the focal two-way interaction term. The three-way interaction was nonsignificant (β = -.019, T-value = -.75, n.s.), thus failing to provide evidence that our results would fail to generalize to larger systems, where system size is code volume. However, a T-test for differences among the means of system atrophy for smaller ($\bar{x}$ .023) versus larger ($\bar{x}$ .075) systems (median-split subgroups) was significant (t = 2.07; $p < .05$). This indicates that systems larger than those in our sample are likely to suffer more from atrophy.

	Step (1)		Step (2)		Step (3)	
	β	T-value	β	T-value	β	T-value
ΔModularity	-0.210**	(-2.972)	-0.161*	(-2.111)	-0.166*	(-2.247)
Development burstiness	-0.049***	(-3.317)	-0.048**	(-3.260)	-0.048**	(-3.246)
Evolution intensification	-0.034***	(-3.629)	-0.036***	(-3.794)	-0.036***	(-3.795)
Code volume growth	0.272***	(5.915)	0.320***	(5.047)	0.317***	(5.134)
Team turnover	0.002	(0.319)	0.001	(0.118)	0.001	(0.106)
Technical debt delinquency	0.091**	(3.039)	0.090**	(3.005)	0.092**	(3.043)
Differential evolution	0.011	(1.649)	0.012	(1.871)	0.012	(1.851)
Code staleness	0.017	(0.969)	0.016	(1.008)	0.014	(0.961)
System age	0.019**	(3.284)	0.024**	(3.183)	0.024**	(3.216)
Δmodularity×system age	0.119*	(2.041)	0.083	(1.373)	0.094	(1.668)
Code volume×system age			-0.063	(-1.016)	-0.059	(-0.971)
Code volume×(Δmodularity×system age)					-0.019	(-0.754)

C: Boundary conditions around application domains of systems

To assess whether our results differed by the system's application domain, we used its category (Figure 3) information in our model. Our interest was only in whether category makes a difference; following the software engineering literature's emphasis on application domains (Herraiz et al. 2013). Since category is time-invariant, we interacted a Guttmann summation of the nine category dummies with our focal (Δmodularity×system age) term. The interaction was nonsignificant (β = -.034, T-value = -.21, n.s.), providing no evidence against generalization across application domains. Its interaction with system age was also nonsignificant (β = 0.00, T-value = .062, n.s.), providing no evidence that systems in some domains are more or less vulnerable to atrophy as they age.

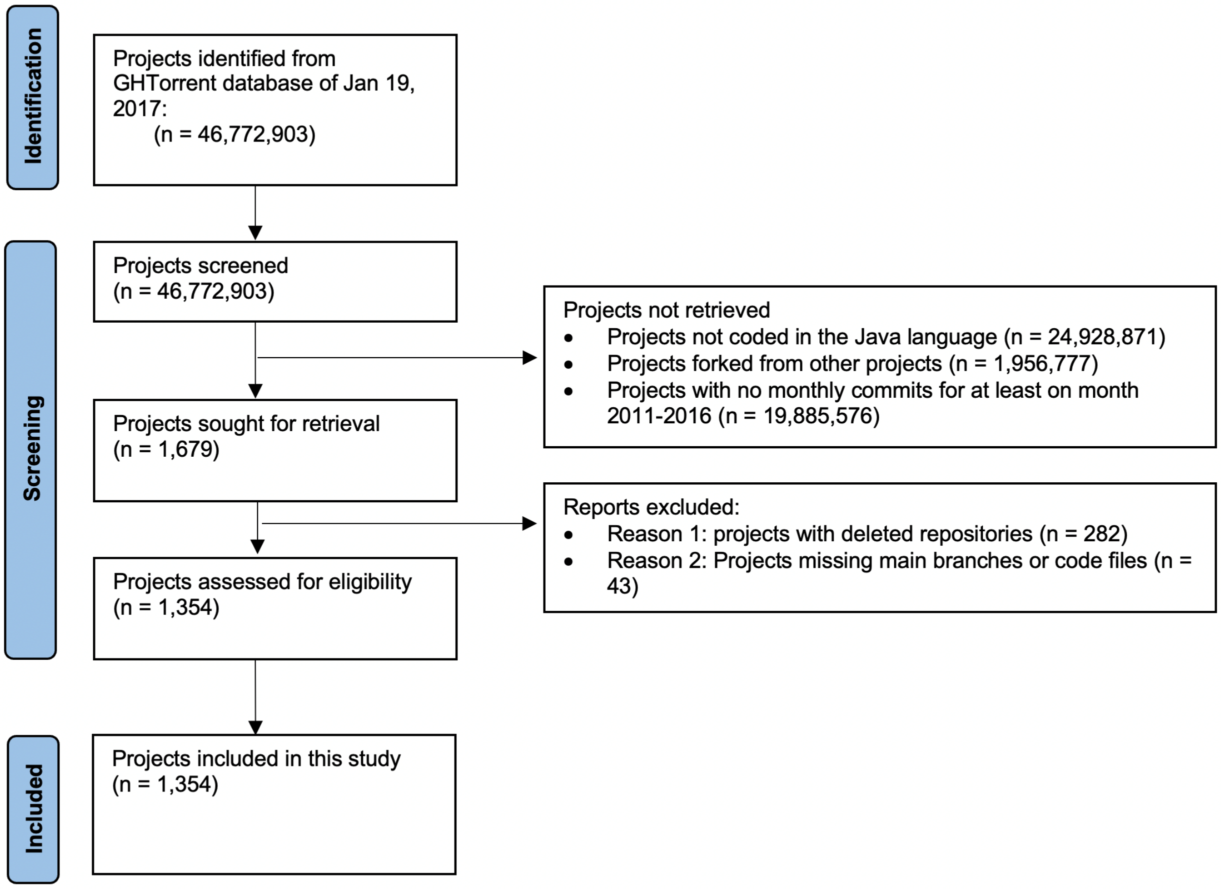
	Step (1)		Step (2)		Step (3)	
	β	T-value	β	T-value	β	T-value
ΔModularity	-.210**	(-2.972)	-.210**	(-2.972)	-.211**	(-3.010)
Development burstiness	-.049***	(-3.317)	-.049***	(-3.317)	-.049***	(-3.315)
Evolution intensification	-.034***	(-3.629)	-.034***	(-3.629)	-.034***	(-3.628)
Code volume growth	.272***	(5.915)	.272***	(5.915)	.273***	(5.817)
Team turnover	.002	(.319)	.002	(.319)	.002	(.309)
Technical debt delinquency	.091**	(3.039)	.091**	(3.039)	.092**	(2.980)
Differential evolution	.011	(1.649)	.011	(1.649)	.011	(1.638)
Code staleness	.017	(.969)	.017	(.969)	.016	(.972)
System age	.019**	(3.284)	.019**	(2.611)	.020*	(2.310)
ΔModularity×System age	.119*	(2.041)	.119*	(2.041)	.153	(.808)
Category of system×System age			.000	(.062)	-.001	(-.103)
Category of system×(Δmodularity×system age)					-.034	(-.209)

D: Main effects remain robust when the observation interval is doubled and quadrupled

To empirically assess the relevance of picking the just-right quarterly observation window, we reestimated our model with a doubled and quadrupled observation window. Since this collapses more granular data, the N dropped to 11,928 and 5,820. The baseline of system atrophy with age consistently showed up significant in both the models with the two-quarter window (β = .026, T-value = 1.97, $p < .05$) and the yearly window (β = .025, T-value = 2.85, $p < .01$). The effect slightly weakened as we enlarged the observation interval. The modularity$\rightarrow$atrophy main effect was also consistently significant in both the two-quarter window (β = -.127, T-value = -8.23, $p < .001$) and the yearly window (β = -.133, T-value = -5.74, $p < .001$). Here, the effect slightly strengthened as we enlarged the observation window. This directional contrast suggests a tension in the size of the existence interval vis-à-vis the observation interval to simultaneously observe the effects of aging and modularity on system atrophy. Their interaction failed to appear in either the two-quarter window (β = -.016, T-value = -1.30, n.s.) or the yearly window (β = -.014, T-value = -.701, n.s.), implying that too long an observation interval likely increases susceptibility to yet-unknown confounds (Kelly and McGrath 1988: 91) and obscures the visibility of their interplay vis-à-vis at a more granular quarterly level. Further affirming this interpretation is the resurfacing of a significant interaction effect (β = -.127, T-value = -8.23, $p < .001$) when we take the maximum level rather than the averaged level across quarters in collapsing the quarterly data.

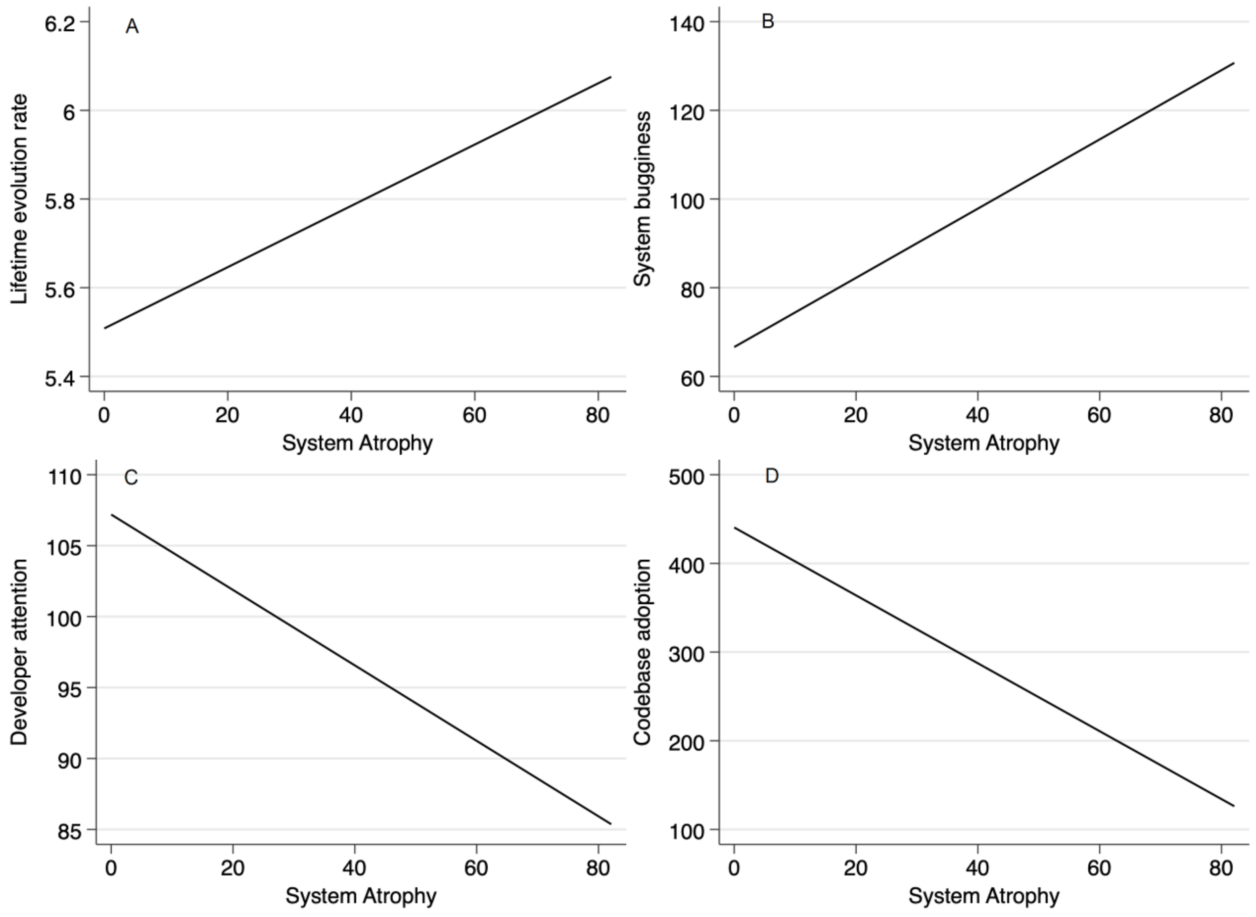
	Lengthened Observation Interval							
	Doubled (Two quarters)				Quadrupled (One year)			
	Step 1		Step 2		Step 1		Step 2	
	β	T-value	β	T-value	β	T-value	β	T-value
ΔModularity	-.127***	(-8.277)	-.117***	(-6.069)	-.133***	(-5.737)	-.121***	(-3.799)
Development burstiness	-.039***	(-4.013)	-.039***	(-4.009)	-.022	(-1.452)	-.022	(-1.463)
Team turnover	.015	(1.645)	.015	(1.692)	-.009	(-.715)	-.009	(-.676)
Evolution intensification	-.046***	(-4.561)	-.046***	(-4.516)	-.057***	(-3.659)	-.057***	(-3.638)
Code volume growth	.270***	(22.622)	.269***	(22.833)	.259***	(14.205)	.259***	(14.498)
Technical debt delinquency	.089***	(10.210)	.088***	(10.163)	.062***	(4.992)	.062***	(4.942)
Differential evolution	-.007	(-.842)	-.008	(-.873)	-.013	(-1.014)	-.013	(-1.012)
Code staleness	.025**	(2.907)	.026**	(2.996)	.012	(.982)	.013	(1.044)
System age	.025**	(2.846)	.025**	(2.748)	.026*	(1.973)	.025	(1.896)
ΔModularity×System age			-.016	(-1.297)			-.014	(-.701)
N	11928		11928		5820		5820	
R^2	.111		.111		.096		.097	

E: PRISMA diagram for sampling


Identification
Projects identified from GHTorrent database of Jan 19, 2017: (n = 46,772,903)
Screening
Projects screened (n = 46,772,903)
Projects not retrieved
• Projects not coded in the Java language (n = 24,928,871)
• Projects forked from other projects (n = 1,956,777)
• Projects with no monthly commits for at least on month 2011-2016 (n = 19,885,576)
Projects sought for retrieval (n = 1,679)
Reports excluded:
• Reason 1: projects with deleted repositories (n = 282)
• Reason 2: Projects missing main branches or code files (n = 43)
Projects assessed for eligibility (n = 1,354)
Included
Projects included in this study (n = 1,354)

GHTorrent database available at http://ghtorrent-downloads.ewi.tudelft.nl/mysql/mysql-2017-01-19.tar.gz Prisma chart template from: Page MJ, McKenzie JE, Bossuyt PM, Boutron I, Hoffmann TC, Mulrow CD, et al. The PRISMA 2020 statement: an updated guideline for reporting systematic reviews. BMJ 2021; 372:n71. doi: 10.1136/bmj.n71

F: Decomposition from Figure 6 of the four downstream consequences of system atrophy


A
Lifetime evolution rate
System Atrophy
B
System bugginess
System Atrophy
C
Developer attention
System Atrophy
D
Codebase adoption
System Atrophy