Synchronizing Development Teams with Drifting Software Systems: A Decomposability-Based Theory of Evolutionary Mechanisms

Amrit Tiwana (contact author) and Hani Safadi
Terry College of Business
University of Georgia
630 South Lumpkin Street, Athens, GA 30602
tiwana@uga.edu, hanisaf@uga.edu

Author Bios:

Amrit Tiwana is the Fuqua Distinguished Chair of Internet Strategy at the University of Georgia. He has served as a senior editor for *Information Systems Research* and *MIS Quarterly* and serves on the boards of *Strategic Management Journal* and *Journal of Management Information Systems*. His work has appeared in leading MIS, strategy, software engineering, finance, and marketing journals.

Hani Safadi is an Associate Professor at the Terry College of Business, University of Georgia. He is interested in online communities, social media, health IT, information systems development, and the application of computational techniques in management research. His research has been published in prominent outlets, including *MIS Quarterly, Information Systems Research, Journal of Management Information Systems, Journal of the Association for Information Systems, Organization Science,* and *Journal of Medical Internet Research.* He is a Senior Editor for the *Journal of the Association for Information Systems* and a former Associate Editor at *MIS Quarterly*. He is an Arctic Code Vault Contributor, having contributed code to several repositories as part of the 2020 GitHub Archive Program.

Synchronizing Development Teams with Drifting Software Systems: A Decomposability-Based Theory of Evolutionary Mechanisms

Abstract

System evolution challenges persist as scholarship remains entrenched in static models inadequately capturing how systems drift over time, while siloed team and architecture studies obscure their dynamic interplay. This fragmentation obscures the mechanisms—the "how"—through which coevolving team and technical architectures shape evolutionary outcomes. Our theory, building on Simon's decomposability idea, traces systems' evolution to the dynamic syncing of drifting technical and team architectures. We theorize two mechanisms of system evolution—module evolution (internal changes) and architecture evolution (interconnection changes)—each differentially catalyzed by system and team architecture; their syncing simultaneously activates both mechanisms. Econometric analysis of 1,303 open-source systems encompassing 10.1 million evolutionary events across 4.8 billion lines of code supports the proposed ideas. Our contributions are threefold. First, we reconceptualize system and team architectures as temporally dynamic properties, shifting the discourse from static snapshots to dynamic evolutionary trajectories. Second, we introduce module evolution and architecture evolution as mechanistically distinct pathways differentially catalyzed by system and team architecture, enlarging the discourse from magnitude to the content of change. Third, we show how dynamic system-team syncing sustains systems' evolution by continuously restoring alignment as inevitable architectural drift erodes their congruence. Overall, our mechanistic theory underscores the sociotechnical dynamics through which architectural syncing drives systems evolution.

Keywords: software systems evolution, sociotechnical dynamics, syncing, evolutionary mechanisms, architectural drift, decomposability, team architecture, computational archaeology

INTRODUCTION

Software systems that fail to readily evolve trap organizations dependent on them. Approximately 85% of system costs accrue post-deployment [1], with system evolvability crucially impacting organizations' operations and interoperability with newer systems [2, 46]. Gradual divergence between documented design and implemented reality exacerbates such evolutionary challenges, consuming nearly half of organizations' IT budgets [16, 27: 99].

Three critical theoretical gaps persist in our understanding of software systems' evolution. First, the predominance of static empirical snapshots inadequately captures how existing systems inexorably "drift" architecturally over time [8: 208]. The accumulation of undocumented dependencies and technical debt progressively widens the rift between documented system designs and implemented code [2, 33]. This severs correspondence between teams' mental models and systems' current state, complicating evolutionary efforts precisely when teams most need accurate architectural understanding [2, 62]. As the ideal of organizing subteams neatly with modules progressively erodes [25: 854], team organization critically affects the manageability of such drift. Second, although IS scholarship epitomizes bridging social and technical domains, models that simultaneously address the dynamic *coevolution* of teams with systems remain scarce, with research on team structure [62, 66] and technological architecture [2, 23] rarely intersecting. Third, the discourse has centered predominantly on how fast systems evolve while neglecting content—*how* they change [22, 48]. This gap obscures theoretical understanding of the mechanisms—the "how"—through which systems' evolution unfolds.

We theorize two fundamental evolution mechanisms—modifications within modules versus restructuring of module interconnections—that constitute distinct mechanistic pathways through which systems evolve. Different evolutionary imperatives demand distinct mechanisms, determining whether evolution succeeds or stalls. Module-level evolution suffices for incremental innovation: refining user interfaces, optimizing algorithms within existing modules, or fixing bugs—all changes that enhance module internals without restructuring modules' interconnections. Architecture evolution alone enables architectural innovation: migrating from monolithic designs to microservices, integrating third-party APIs through new interface layers, or replacing deprecated technology modules while preserving system functionality—all changes that reconfigure module interconnections without necessarily altering module internals. Radical innovation simultaneously requires both mechanisms: incorporating AI capabilities demands both new module implementations and restructuring module interconnections to enable cross-module data pipelines, while transitioning desktop applications to cloud-native architectures necessitates both module reimplementation and systemwide reorganization. This mechanistic differentiation illustrates why certain evolutionary efforts succeed while others encounter persistent obstacles.

As systems gradually drift from their original design, once-aligned teams become progressively misaligned with the drifted state, compromising evolvability [2, 43]. This dynamic interplay remains theoretically underdeveloped despite its importance to systems evolution. Thus, we lack an integrative, dynamic perspective on how the interplay of system architecture with team architecture shapes systems' evolution. We address this gap through the following research question: ***How* does dynamically syncing team architecture with drifting system architecture shape the evolution of software systems?**

We develop a middle-range theory comprising three interconnected theoretical moves that reconceptualize evolution as a dynamic sociotechnical process of syncing. First, we symmetrically apply Simon's [51] decomposability theory to both technical and organizational architectures to establish a unified theoretical vocabulary for their coevolution. Second, extending insights on residual dependencies [2, 33], we theorize the mechanistic pathways through which technical and team architectures catalyze distinct evolutionary outcomes: system architecture fosters evolution of module interconnections, while team architecture fosters refinement within modules. Third, we theorize how their dynamic synching—wherein team architecture offsets inevitable architectural drift—simultaneously activates both mechanisms.

We test these ideas longitudinal computational analysis of 1,303 open-source systems spanning two decades, collected as part of a decade-long research program on systems evolution [61]. Our archaeological analysis encompasses over 10 million evolutionary events spanning 4.8 billion lines of code to reveal the layered history of their coevolution. We assess system architecture through computational analysis of dependency structures within source code, and infer team architecture from collaborative patterns manifested in code modification histories. Our econometric model disentangles the reciprocal causality inherent in sociotechnical systems, where technical architecture simultaneously shapes and is shaped by team structures. By decomposing evolutionary changes into module-level refinements versus architectural changes, we demarcate the two distinct mechanisms through which systems evolve. This reconceptualizes software systems' evolution as a dynamic sociotechnical process requiring continuous resynchronization between team and technical architectures.

Our novel contributions are threefold. First, we reconceptualize architecture—both technical and organizational—as temporally dynamic properties. Second, we explain how team architectures dynamically compensate for accumulated dependencies [2, 33] and technical debt [46] as shortcuts, patches, and code spanning eras of technologies amalgamate in a system. This bridges the previously-unconnected streams on team design [62, 66] and technical design [2, 46, 60], showing how their nuanced interplay makes them matter more together. Third, we introduce the two fundamental system evolution mechanisms—module evolution and architecture evolution. Our results show that system decomposability fosters evolution of interconnections among a system's modules but not of the modules themselves, while team undecomposability fosters evolution inside modules but not of their interconnections. This mechanistic differentiation directly addresses calls for understanding mechanisms underlying IT evolution [22, 48]. Subsequent sections develop the theoretical logic, methodology, analyses, and implications for theory and practice.

THEORY

Our unit of analysis is the system—encompassing both its codebase and its development team. Construct definitions appear in Table 1. Figure 1 previews our research model, organized around two theoretical ideas. First, system architecture and team architecture trigger *different* evolution mechanisms—the former shapes only modules' interconnections and the latter only modules' internals. Second, dynamically syncing them simultaneously bolsters both mechanisms.

<< INSERT TABLE 1 and INSERT FIGURE 1 HERE >>

Our theory development proceeds as follows. We begin with Simon's theory of system decomposability as our conceptual foundation. We then conceptualize system and team architecture as distinct manifestations of decomposability, followed by the two evolutionary mechanisms—module evolution and architecture evolution. We then theorize how system architecture shapes architecture evolution while team architecture shapes module evolution. Finally, we theorize syncing as dynamic system-team alignment, explaining how it simultaneously enables both evolution mechanisms.

To operationalize decomposability, we will subsequently distinguish three architectural levels: modules (subsystems with defined interfaces), files (implementation units), and classes (object-oriented abstractions). Module evolution captures changes preserving boundaries; architecture evolution changes modules' interconnections. In our empirical setting, Java's structure matches this hierarchy—packages as modules, *.java* files as implementation units, classes as abstractions—enabling precise operationalization of Simon's decomposability through these directly analyzable code structures.[1]

Theoretical Foundation: Simon's Decomposability Theory

Our theoretical foundation is Simon's [51] theory of system decomposability, which describes the fundamental property of independence among and interdependence within a system's modules. Modules are discrete, bounded subsystems encapsulating related functionality. Each module possesses a *boundary* (interfaces defining how it interacts with others), *coupling* (dependencies with other modules), and internal *cohesion* (tight coupling among elements within it) [43].

Decomposability characterizes entire systems—technological or social—through a design philosophy that minimizes cross-module coupling while maximizing internal interdependence. This approach divides a system into smaller, independent subsystems [14: 43]. Severing their connections gives each module the autonomy to change without requiring simultaneous modifications to—even awareness of—other modules [43]. Systems exhibit varying degrees of decomposability along the continuum in Figure 2, ranging from many loosely-coupled small modules (perfectly-decomposable) to one large tightly-coupled module (completely-undecomposable or monolithic). Simon's decomposability applies symmetrically to both system codebases and team structures. Systems can shift along this spectrum over time—a dynamic we term architectural drift.

<< INSERT FIGURE 2 HERE >>

System and Team Architectures as Distinct Decomposability Manifestations

Simon's [51] decomposability instantiates distinctly in technological and social domains. In technical systems, decomposability appears as system architecture, while in social systems it emerges as team architecture. While both share the same theoretical roots, these architectural manifestations vary independently [62, 66]. System decomposability is a property of the IT artifact, measurable through code dependencies and reflecting how cleanly functionality is separated within it. Team undecomposability, by contrast, is inferred from observable collaborative patterns—specifically, developer interactions across module boundaries. System decomposability constrains what dependencies are architecturally possible, while team decomposability reflects how developers coordinate around actual interdependencies, regardless of system architecture.[2]

System decomposability

We define *system decomposability* as the degree to which a system's code comprises loosely-coupled, internally-cohesive modules [60]. The philosophy is organizing system code as smaller, self-contained modules with systematically-severed dependencies among them. Specifically, decomposability requires minimal cross-module dependencies (e.g., API-based interactions) coupled with high intra-module interdependence (e.g., density of class relationships through inheritance and composition within modules). In Java, this maps directly to packages (modules), *.java* files (implementation units), and classes (constructs). System decomposability thus extends beyond modularity by simultaneously capturing decoupling and internal clustering. Monolithic architectures represent the theoretical antipode, wherein the entire system coalesces into a singular, undifferentiated module.

Team decomposability

Although systems are often team-developed [64: 1741], team architecture remains less studied than system architecture. Team architecture represents the formal grouping of members into subteams [7: 1326], wherein a team is a social system with multiple agents, identifiable boundaries, and system-level goals [45: 4]. Software development teams satisfy these criteria—multiple developers, project boundaries, and shared development goals—making them organizational systems amenable to Simon's decomposability lens.

Management theorists emphasize critical distinctions between nascent teams and inherited project structures [47: 2125]—a phenomenon endemic to software maintenance contexts where successive developer cohorts maintain evolving codebases [8: 208]. Temporal drifts in a team's decomposition into subteams can make static representations depict how the team might once have been rather than how it is. Consistent with organizational architecture research using interaction patterns across subgroups [3, 9, 13, 56], we assess team architecture through observable patterns of technical collaboration. We follow Baldwin's [4: 34] recent depiction of team architecture as the scheme by which tasks are divided up among individuals in teams, recognizing that software teams commonly comprise identifiable subteams with different roles [8: 207]. We focus on direct technical collaboration—developers working on shared artifacts in an observation window—to infer coupling through members' module modification behavior. Decomposable team architectures exhibit clustered collaboration within relatively-isolated subteams; undecomposable architectures manifest as pervasive cross-team collaboration.

We define *team undecomposability* as the extent to which team members work jointly on the same parts of the system's codebase, thereby reflecting cohesion within the team across its subteams [10]. Teams also fall along the decomposability continuum in Figure 2, ranging from loosely-coupled, differentiated subteams (decomposable) to a single, tightly-integrated collective (undecomposable). Higher values indicate lower divisibility into discrete subteams. Following Simon [51], we infer team architecture from actual collaborative patterns rather than formal structures—a critical distinction given the persistent divergence between organizational charts and operational reality [27: 211]. This matters particularly in software development's temporally-fluid subteam boundaries [8: 207].

System and team architectures exhibit conceptual independence despite sharing decomposability foundations.[3] They catalyze *different* evolutionary mechanisms and vary independently in practice, influencing outcomes through their dynamic interplay rather than isomorphism. In practice, legacy monolithic systems exemplify technical undecomposability maintained by functionally-specialized, decomposed subteams. In contrast, contemporary open-source systems like Firefox can exhibit high decomposability in both dimensions, with distributed contributors working on architecturally-bounded modules. Such heterogeneity motivates our examination of their dynamic interplay.

Two Fundamental System Evolution Mechanisms

Evolution refers to a system changing over time to meet emerging needs, particularly crucial as new software artifacts are often derived from existing ones [28]. While prior studies conceptualize IT evolution as the overall intensity of change without differentiating the content of change [15, 59], we identify two fundamental mechanisms illustrated in Figure 3: (a) changes in modules' interconnections and (b) changes in module internals.

<< INSERT FIGURE 3 HERE >>

Mechanisms in our theory are structural (intervening steps in the causal chain) rather than process mechanisms (temporal sequences of events), as distinguished in the philosophy of science [36]. The structural approach is the realm of directed acyclic graphs like our research model in Figure 1, as opposed to process sequences. This approach fits our model's focus on explaining how system and team architecture trigger specific system evolution behaviors, rather than describing sequential stages of change.

We define *architecture evolution* as changes that alter modules' interconnections, and *module evolution* as changes within modules' internal implementation that preserve existing interconnections.[4] These mechanisms explain *how* systems exhibit generativity through differential activation of module versus architecture evolution [21]. The two mechanisms *can*—but need not—occur simultaneously.

We characterize them as fundamental because their permutations parsimoniously describe all conceivable forms of evolution in a system. This architecture-versus-module evolution distinction enables understanding how the differential activation of these mechanisms realizes diverse types of innovation. Our two mechanisms map to Henderson and Clark's [26] 2x2 innovation typology in Figure 4: incremental innovation (low architecture, low module evolution); modular innovation (low architecture, high module evolution); architectural innovation (high architecture, low module evolution); and radical innovation (high architecture, high module evolution). This correspondence theoretically links micro-level evolutionary mechanisms to macro-level innovation outcomes.

<< INSERT FIGURE 4 HERE >>

System Architecture and Team Architecture's Evolutionary Effects

System and team architecture differentially influence these two evolution mechanisms via distinct causal pathways. We first examine how system decomposability drives architecture evolution, then how team undecomposability drives module evolution.

System decomposability's effects on architecture evolution

Arguments favoring decomposability are well-trodden: managing complexity [5: 66], enabling concurrent work [58], and allowing one module's developers to work without needing knowledge of other modules' innards. System decomposability eases systemwide changes by reducing the effort to add, remove, or swap modules in existing systems. For example, adding files implementing new modules (e.g., facial recognition replacing password authentication) or removing files constituting deprecated modules alters the system's architecture. This conserves scarce cognitive resources needed to preserve modules' mutual interoperation and eases changing connections among modules [32, 58]. While loose coupling reduces ripple effects across module boundaries—theoretically lowering the risk of cross-module changes—this risk reduction primarily facilitates restructuring module connections rather than incentivizing within-module refinement. Reduced cross-module ripple effects make architectural operations safer (adding, removing, or reconfiguring module connections) without simplifying internal module improvements. But module-level refinements still demand deep domain knowledge of the module's functional logic, sustained focus on implementation details, and specialized expertise in the module's technical domain—requirements orthogonal to the systemwide interconnections. Consequently, loose coupling directs developer attention toward restructuring module interconnections rather than toward modules' internal optimization, which requires module-specific expertise regardless. Therefore, system decomposability enhances the changeability of links among a system's modules.

Decomposability's dual aspects facilitate such architecture evolution: decoupling makes module connections malleable, while internal cohesion stabilizes module interfaces.[5] When a module's internal elements exhibit tight functional interdependence, the module's external interfaces become architecturally predictable and stable. This predictability enables developers to confidently restructure how modules interconnect without concern for unexpected ripple effects, accelerating architectural changes by eliminating systemwide impact analysis overhead before each change. Internal cohesion reduces the cognitive overhead of modifying module interconnections. Developers can reason about interface changes knowing that well-defined module boundaries will contain their effects. Thus, decoupling provides flexibility for interface changes while internal cohesion provides stability to facilitate architecture evolution. However, high cohesion can complicate changes *within* modules because modifications to tightly-connected elements require coordinated changes to preserve their functioning. System decomposability thus enables changes in module links but not within modules.[6]

Hypothesis 1a: *System decomposability enhances architecture evolution.*

Team undecomposability's effects on module evolution

Decomposable teams organize individuals into subteams specializing in specific modules, providing cues to manage known interdependencies [7, 12, 22]. Such partitioning guides subteams' interactions and integration of effort [12, 37]. However, such specialization inherently compartmentalizes knowledge around subteams' module-specific activities, creating lags in cross-module awareness that can precipitate iterative rework *within* individual modules. Evolving modules also requires a holistic understanding of the system's intended trajectory—visibility that specialization-induced compartmentalization obscures. Recent IS studies of development teams indeed show that an incongruent understanding of existing systems causes task conflicts and disrupts coordination [62, 66].

Team undecomposability mitigates these coordination challenges through intensive collaborative effort, but this benefit manifests primarily in module-level evolution rather than architectural changes. Cross-module coordination demands therefore create a tension between broad system awareness and deep architectural modification capacity. Team undecomposability mitigates such risks by preempting changes in one module based on obsolete information about other modules. It manifests as either multiple developers working on single modules or individuals working across evolving modules. When multiple developers focus on the same module, they develop a deep, collaborative understanding of module internals that facilitates substantive module evolution. Since team architecture does not affect how the tweaked modules reassemble into the whole system—the province of system architecture—it does not affect architecture evolution. We therefore expect team undecomposability to enhance module evolution but not architecture evolution.

Hypothesis 1b: *Team undecomposability enhances module evolution.*

System-Team Syncing

Our second theoretical idea is that while system architecture and team architecture independently influence different evolutionary mechanisms, their dynamic syncing simultaneously enhances both mechanisms. We distinguish syncing from alignment to capture a temporal nuance: alignment represents a snapshot—a momentary state of architectural congruence at discrete time *t*, measurable through observable system-team configurations. Syncing, by contrast, describes the ongoing dynamic process through which teams continuously restructure themselves as systems architecturally drift. Static alignment cannot persist by definition, as aging systems accumulate technical debt and lose some initial decomposability, any previously-achieved congruence between team structure and system architecture necessarily erodes. Syncing thus captures the dynamic mutual adjustment required to restore momentary alignment as system architecture drifts. System and team architectures rarely drift in lockstep. Maintaining alignment, therefore, requires adapting team architecture. This continuous organizational adaptation to technical drift compensates for accumulating architectural imperfections. Following Venkatraman's [63] fit-as-moderation logic, we theorize team architecture, through direct developer collaboration [62, 66], as compensatory for drift-induced imperfections in system architecture [2].[7]

Architectural drift and residual dependencies

Architecture represents simplified structure [18]—“a miniature representation of reality” [14: 41]—making system decomposability a reflection of how code *appears* to boundedly-rational designers rather than a high-fidelity depiction of its actual implementation. This creates an epistemic gap: perfect initial decomposability would require complete, unambiguous module specifications, yet designs on paper rarely match implemented code [14: 151]. Booch [8: 27] presciently warned decades ago that no perfect abstraction exists, and Parnas [43] recognized that code decomposition rarely creates true independence among developers. This inevitable gap between architectural intention and implementation reality manifests as residual dependencies—unrecognized coupling among ostensibly-independent modules [33].

Residual dependencies—unrecognized coupling that emerges from the gap between documented architecture and current code—constitute a fundamental constraint on systems’ evolution. They emerge from designers’ bounded rationality and incomplete knowledge of emergent module interactions. Scholarly references to unrecorded linkages [5: 56] and unexpected system-level interactions [43] document how architectural abstractions consistently overlook residual dependencies. Such dependencies become most visible during modification, when time-constrained developers employ workarounds that violate module boundaries, bypass APIs, and subvert inheritance hierarchies. Despite Parnas’ [43] prime directive to encapsulate change-prone decisions, these dependencies accrete over time like archaeological strata, creating historical layers recording a system’s evolutionary journey.

This accumulation constitutes architectural drift—the natural structural evolution whereby a system’s organizing logic progressively diverges from its original design as modules and linkages gradually reorganize to accommodate new functionality and interoperate with other systems. Unlike atrophy (irreparable logical degradation [61]), drift (progressive divergence from initial design) represents adaptation to legitimate changing requirements. However, unrecognized drift creates incongruence: teams’ mental models reflect documented architecture while actual implementation has evolved elsewhere. This widening gap between documented and actual dependencies [5: 24] generates evolution-derailing conflicts [2, 62], as static analytical snapshots prevalent in prior research miss this intertemporal shift. Teams can therefore struggle to maintain accurate mental models amid obscured system architecture.

Recent studies show that team behavior and system architecture are interdependent [2, 62], requiring their mutual congruence scarce in prior research. As imperfections accumulate into technical debt, systems become difficult-to-unravel, burdensome-to-maintain contraptions [2, 46, 61], transforming plug-and-play module integration into plug-and-pray. Drift then requires adapting the team to preserve the system's evolvability, without which residual dependencies imperil system longevity [1, 2].

Syncing dynamics and architecture evolution

Team decomposability's advantage of specialization makes residual dependencies particularly problematic. This creates a need for dynamic syncing rather than achieving static alignment. Technical interdependence drives who interacts with whom [56], yet even the initial team organization typically reflects incomplete assumptions about individuals' work interdependencies [7]. When system architecture designates two modules as independent, developers working on them progressively interact less, as module salience shapes the enacted division of labor [4: 21, 47].[8]

These silos deepen as teamwide interactions congeal around modules—a benign pattern only when residual dependencies are truly absent. However, such dependencies lull developers into misplaced technical autonomy, obscuring how their work ripples into other modules until integration phases, when architectural boundaries have largely solidified. This late discovery necessitates expensive rework, disrupting architectural evolution; developers must retrofit dependencies into established module interfaces, creating cascading changes that destabilize boundaries. The temporal lag between dependency emergence and discovery determines whether architectural adjustments remain tractable or become disruptive, making team decomposition's delayed dependency awareness costly [42].

Resolving residual dependencies earlier requires recourse to human interaction across module-centric silos [5: 46, 45: 65], which is where team *undecomposability's* complementarity comes in. Team undecomposability enables earlier dependency discovery through direct collaboration. Dialog among developers working jointly across modules surfaces actual dependencies earlier during active development rather than during post-hoc integration. This helps identify emerging dependencies before module boundaries ossify, and proactive architecture adjustments remain tractable through incremental interface tweaks rather than disruptive wholesale rework. As system architecture progressively drifts from initial design, coordinating across ostensibly-independent modules harboring hidden dependencies becomes essential for architecture evolution.

Decomposable systems provide structural flexibility for evolving module connections; undecomposable teams provide cross-module awareness, enabling implementation before dependencies calcify. Together, they actualize architecture evolution: decomposable systems create the potential, while undecomposable teams provide the coordination. This combination constitutes team-system syncing—a positive interaction effect wherein team undecomposability amplifies system decomposability's facilitation of architecture evolution. We therefore expect team undecomposability to strengthen system decomposability's positive effect on architecture evolution.

Hypothesis 2a: *Team undecomposability strengthens system decomposability's positive effect on architecture evolution.*

Syncing Dynamics and Module Evolution

System decomposability amplifies team undecomposability's advantages in fostering module evolution. By reducing systemwide dependencies among modules, system decomposability confines cross-subteam communication to residual dependencies rather than all dependencies. This increased technical autonomy allows developers to reallocate cognitive effort to module-specific refinements. Team undecomposability preserves this module-focused work because system decomposability makes the system's modules cognitively salient boundaries guiding teamwide division of labor [47]. Cross-module awareness manifests primarily as enhanced coordination around existing dependencies rather than capacity for their elimination. The cognitive load of implementing systemwide architectural changes—maintaining accurate mental models across evolving modules while negotiating with multiple stakeholders—makes dependency elimination prohibitively resource-intensive, scaling nonlinearly with team size and architectural scope.[9] Consequently, team undecomposability's advantages manifest primarily in module-level refinements, where developers' cross-module knowledge enables compatibility-preserving module refinements without overtly requiring systemwide consensus.

Thus, team undecomposability enhances managing residual dependencies (coordinating around constraints) rather than eliminating them (evolving architectural boundaries). Cross-module awareness facilitates working productively within constraint-laden code rather than restructuring the constraints themselves, clarifying why benefits manifest in module evolution rather than architectural evolution. Module-level improvements leverage cross-module knowledge to assess downstream impacts and identify interdependencies with other modules, enabling module evolution without systemwide disruption. This asymmetry—team undecomposability shaping module evolution and system decomposability shaping architectural evolution—reflects distinct mechanistic pathways. In contrast, in undecomposable systems, subteams' overhead managing drifting dependencies among modules would diminish their capacity for focused module-level refinements [2, 62]. We therefore expect system decomposability to strengthen team undecomposability's positive effect on module evolution.

> ***Hypothesis 2b:*** *System decomposability strengthens team undecomposability's positive effect on module evolution.*

METHODS

Research Setting and Data Assembly

Our unit of analysis is the sociotechnical system, encompassing both codebase and development team. We tested the proposed ideas using a random sample of 1,303 open-source Java systems. Our source code derives from the Open Knowledge Relics Archaeology (OKRA) initiative, a decade-long research program studying systems evolution using open-source digital relics with supercomputing-driven induction as a form of computational archaeology [61]. The research program involves an ongoing exploration of multiple facets of systems evolution over time. The raw OKRA source code comprised 10.1 million commits. Over our observation period, we analyzed 4.8 billion lines of code and 27 million classes in code volume. This approximates 97,000 years of *Journal of Management Information Systems*.[10] This study involved excavating historical layers of code to uncover how teams interacted with evolving system artifacts over time. We collected data from January 2011 through December 2016 to capture mature git adoption and precede Java 9's introduction in 2017 (which added modules, fundamentally altering package-level analysis). We focused on Java systems with at least monthly commit frequency, excluding forks and sporadically developed systems. This activity threshold ensured sufficient longitudinal observations for quarterly measurement.[11]

Rationale for focusing on open-source, Java language systems in GitHub

Our research program focused exclusively on Java because its language features enable the precision essential to our approach. Java's explicit package declarations and mature static analysis tools allow precise identification of module boundaries and extraction of dependency relationships from source code.[12] Hierarchically, *systems* comprise *packages* (modules) containing *.java* files (implementation units) defining classes. This structural precision—where files represent atomic developer modification units and packages demarcate module boundaries—is infeasible in dynamically-typed languages. For example, Python's flexible module system and multiple inheritance patterns obscure such relationships; Ruby's and JavaScript's runtime dependency resolution complicates static architectural analysis. Consequently, Java enables the systematic operationalization of Simon's decomposability through directly analyzable code structures.

We focused on open-source systems because they permitted matching ~14,000 individuals responsible for every code change in every system over their development history.[13] When team members collaborate through a version control system, the actual subteam architecture can be inferred by analyzing such interactions [27: 212]. We could then analyze each code change in a system to infer our two mechanisms, which is not possible without access to the source code [29: 225]. Open-source systems typify peer production of IT artifacts [41]. While our empirical focus is open-source, Java's widespread use in proprietary systems strengthens our external validity. In 2025, 150 million developers used GitHub, which hosted 28 million public repositories, making it the largest open-source code repository. Unlike SourceForge or CodePlex, GitHub uses a unified version control system for bug tracking, task management, and continuous integration, thereby eliminating confounding of teamwide interactions by heterogeneous development tools. GitHub's comprehensive API and metadata on commits, pull requests, and issues enabled systematic extraction of repository data, allowing precise reconstruction of team interactions and code change records.

Data assembly and preprocessing

Each code commit contained a unique identifier, timestamp, developer username, and code. We reconstructed system states quarterly by accumulating commits across calendar quarters, with commit data precisely linking developers to source code modifications. The 2011–2016 boundaries preceded Java 9's introduction of new

"Java Modules" in 2017 and eliminated confounding from evolving technological practices around this new additional layer. Our initial sample comprised 1,679 open-source Java systems from GitHub between 2011 quarter 1 (January–March) and 2016 quarter 4 (October–December). Quality assurance procedures ensured parseable commit messages, valid developer identifiers and timestamps, and valid source code files.

We used listwise deletion, excluding systems with incomplete git histories, dependency parsing errors, or insufficient commit activity. Exclusion criteria were: 282 systems with corrupted/incomplete git histories, 43 with dependency parsing errors, and 51 with lifespan < 1 year. No systems required exclusion for missing identifiers. Selection bias assessment (Appendix Table A1) showed negligible effects (Cohen's d: 0–.12), with dependent variables exhibiting trivial differences (module evolution d = .04; architecture evolution d = .02). The largest effect was team undecomposability (d = .12), reflecting excluded systems' slightly more decomposable teams consistent with our sustained-activity criterion naturally filtering sparse networks. These patterns confirm that exclusions reflect data quality constraints rather than systematic theoretical bias. The final sample comprised 1,303 systems generating 23,664 system-quarter observations. All analyses—including main effects, interactions, and robustness checks—used this data. Robust estimation methods mitigated outlier influence; sensitivity analyses confirmed result stability.

Construct Operationalization

Table 1 summarizes the operationalization of our key theoretical constructs. From the raw commit records, we reconstructed each project quarterly and analyzed these quarterly codebases totaling 4.8 billion lines of code to estimate all constructs in a time-series manner. Analyzing our source code took two weeks on a 150-node supercomputing cluster with 2,400 Gigabytes of dynamic memory and 1,200 microprocessor cores. Our approach for extracting code dependencies from analyzing Java source code builds directly on MacCormack et al.'s [34, 35] six-project study.

System decomposability operationalization from source code

We operationalized ***system decomposability*** as the ratio of within-module to cross-module dependencies at time *t*, following Simon's [51] conceptualization and MacCormack [35]. The numerator reflects how well a system encapsulates implementation decisions within modules, while the denominator captures potential information leakage across module boundaries. This ratio captures Simon's notion of decomposability, with module cohesion in the numerator and cross-module dependencies in the denominator. Higher ratios indicate stronger encapsulation [43]. Our measurement strategy exploited three properties of Java codebases, which exhibit the hierarchical structure in Figure 5.

< INSERT FIGURE 5 HERE >

A system's code is organized into packages (*modules*) and .java *files*, with internal cohesion achieved through inheritance, composition, and method invocation. We infer within-module and cross-module dependencies from these patterns. Such cohesion manifests as localized change propagation—modifications within module boundaries necessitate coordinated internal adjustments without external ripple effects [43]. Java's explicit package declarations and access modifiers enable precise identification of these boundaries and dependencies. This technical property of Java makes it especially suitable for operationalizing both Simon's decomposability and Parnas's information hiding notion. Each Java source code file contains one or more *classes* (typically one public class per file). Within a module, classes have unrestricted access through inheritance and composition; across modules, access is restricted through access modifiers. We analyzed two object-oriented dependencies: composition (*has-a*) and inheritance (*is-a*). If a class in a file uses or inherits from another class, a dependency between the files of the two classes exists. We weighted each dependency by the frequency of class use and inheritance between all files in every system's code in each quarter. For example, if file A contains two classes, both used by file B, the dependency weight from B to A equals two. In a perfectly-decomposable system architecture, files would exclusively depend on other files within their own module; deviations from this ideal lower system decomposability.

< INSERT FIGURE 6 HERE >

Figure 6 illustrates this computation for a system with three modules and six classes. Of the six class dependencies, three cross-module dependencies (dotted lines) connect classes from different modules, while solid lines are within-module dependencies. The system decomposability is 50% (= 3 within-module ÷ (3 within-module + 3 cross-module) dependencies). Following Subramanyam et al. [58], we normalized to percentages. The substantive interpretation of this ratio in our sample mean of 39.4% indicates that within-module dependencies constituted approximately 40% of total system dependencies, with 2/3rd dependencies crossing module boundaries, suggesting that the majority of architectural coupling was across module boundaries rather than within them. Pragmatically, 40% decomposability indicates 1.5x as many cross-module than within-module dependencies, signaling high coordination costs for changes; 80% decomposability indicates 4x as many within-module dependencies, enabling largely independent module evolution.[14] This ratio-based operationalization enables comparison across systems of varying scales and complexity by reflecting dependency structure rather than absolute system size, mitigating confounding from heterogeneity in system scale.

Team undecomposability operationalization from code commit records

Our conceptualization of team modules parallels system modules in Figure 3—interdependent developer subgroups working on individual modules. We operationalized ***team undecomposability*** as the logged count of developer pairs modifying identical files within quarterly observation windows, applying Simon's [51: 469] principle of inferring organizational architecture through observed interaction patterns.[15] We conceptualize "teams" as emergent collaborative networks manifested through substantive technical contributions rather than formal designations. We analyzed quarterly artifact-level commit records to map which developers modified each module's files, capturing collaborative actions as they evolved across successive observation windows. Developers modifying shared files demonstrate genuine teamwork, as such modifications require coordination to prevent integration conflicts and maintain code coherence. When multiple developers modify shared files, Git's merge protocols force explicit conflict resolution, code review workflows necessitate mutual awareness of implementation approaches, and continuous integration systems surface incompatibilities requiring coordinated remediation. This creates observable events captured through commit records.

We aggregated quarterly commits, log-transforming counts to address over-dispersion. Our measurement ensures construct validity through four design features: (1) focus on substantive modifications rather than trivial changes, (2) quarterly temporal windows capturing sustained collaborative relationships rather than transient co-occurrences [56], (3) pair-level analysis aligned with Simon's interaction emphasis, and (4) log-transformation addressing collaboration network skewness. This captures actual rather than potential collaboration, distinguishing it from volatile activity fluctuations. Technical interdependence (the degree to which work outputs require integration) drives collaborative patterns more powerfully than formal organizational structures [5: 46].

Further evidence of construct validity includes: (1) a moderate negative correlation with system decomposability (r = -.16), confirming discriminant validity, and (2) theoretical alignment with Simon's decomposability continuum from zero file-sharing pairs to maximal pair density. The measure's modest correlation with module evolution (r = .13) and negligible selection bias (Cohen's d = .04, Appendix Table A1) confirm it captures collaborative structure rather than change magnitude.

Inference of evolution from code changes over time

File additions introduce interface points reconfiguring module interconnections; deletions necessitate dependency redistribution. These structural changes alter which modules interact, distinguishing architectural evolution from internal module refinements. This operationalization maintains construct independence by distinguishing boundary modifications between time periods from architectural dependency structure at discrete time points.

< INSERT FIGURE 7 HERE >

We operationalized both evolution mechanisms by analyzing intertemporal source code changes (t-1 to t), normalized by total unique files across both periods, creating dimensionless measures ranging from 0 to 1. Building on Siggelkow's [50] conceptualization, we capture shrinking/enlarging existing modules and removal/addition of entire modules. Figure 7 illustrates the measurement basis for our two evolutionary mechanisms. We operationalized ***module evolution***—internal changes within modules—as the sum of the percentage of files in the system's codebase that decreased or increased in size between *t-1* and *t*.[16] This encompasses Siggelkow's thickening and thinning. This size-based measure of module evolution matches our theoretical definition because: (a) changes in file size typically represent refinements to internal implementation rather than interface changes, (b) we specifically exclude changes that alter cross-module dependencies, and (c) the measure captures both code additions (e.g., feature enhancement) and reductions (e.g., refactoring), reflecting the entirety of change within modules.[17] This ensured distinctiveness from architecture evolution.[18]

We operationalized ***architecture evolution***—changes in the structural boundaries and interface definitions among modules—as the sum of the percentage of files deleted from the system's codebase and the percentage of files added between *t-1* and *t*. File changes represent module boundary modifications. This file-based operationalization captures architectural evolution because in Java systems, file additions and deletions represent structural boundary modifications. When files are added, new interface points emerge that reconfigure module interconnections. When files are removed, existing interface dependencies must be redistributed among the remaining codebase elements, fundamentally altering the modules' interconnections. Java's conventions reinforce this: interface modifications require new files (defining public classes as interface points), while implementation refinements modify existing files.

File additions adapt architecture by introducing new interface points that modify module boundaries, while file removals deprecate interfaces requiring functionality redistribution. While compilers enforce dependency coherence, our measure captures both incremental interface adjustments (few files) and major architectural reconfigurations (many files), focusing specifically on structure-altering files rather than implementation refinements. For example, adding new modules while removing obsolete ones represents architecture evolution even if net coupling remains unchanged, as the structural topology fundamentally differs.[19]

Construct validation for architecture evolution

To validate our file-based architecture evolution measure, we conducted network-based validation examining actual changes in module interconnections across consecutive intervals. This validation addresses potential concerns that file additions and deletions might conflate interface modifications with internal implementation changes, thereby not capturing genuine architectural change. For each system-quarter observation, we constructed dependency networks wherein nodes represent classes and interfaces, and edges represent composition and inheritance relationships—fundamental Java module interconnection mechanisms. Architectural change equals the symmetric set difference between consecutive quarters' networks, normalized by edge union:

$$\text{Network-based architecture evolution}_{s,t} = |E_{s,t} \triangle E_{s,t-1}| \div |E_{s,t} \cup E_{s,t-1}|$$

where E represents the edge sets (i.e., composition and inheritance relationships) and $\triangle$ denotes symmetric set difference. This measure directly captures changes in actual module interconnections (the theoretical construct underlying architecture evolution) by quantifying the proportion of dependency relationships that appear, disappear, or are restructured between observation intervals.

The strong correlation r = .83 (p < .001, n = 21,364) between file-based and network-based measures corroborates convergent validity. This substantial correlation confirms that our tractable file-based operationalization reliably reflects module interconnection changes. This confirms that file-level boundary modifications predominantly signal genuine architectural evolution rather than superficial code changes.

Descriptive Statistics

< INSERT TABLE 2 HERE >

Table 2 summarizes the key descriptives. Systems averaged 4.8 years (σ 2.98; range 1-19.5) old at the last observation quarter. Mean system size was 181,496 lines of code (σ 299,503) organized into 117 modules (σ 172) comprising 1,019 files (σ 1,405). Systems showed more cross-module dependencies (3,848; σ 12,235) than within-module dependencies (2,880; σ 9,443), yielding mean decomposability of 39.39% (σ 14). Modules averaged 10 files (σ 8), each with 173 lines of code (σ 93). Teams averaged 5.7 developers (σ 6.6), each modifying 59 files (σ 253). An average of 8 developer pairs committed to identical files quarterly (undecomposability 1.96; σ 1.5). Systems underwent 789 commits (σ 2,242) with 79 deleted files (σ 320) and 120 added files (σ 355) quarterly.

RESULTS

Our unit of analysis—a system—corresponds to an existing software system and its development team. Booch [8: 152] might describe our approach as software archaeology. This archaeological research design is inspired by MacCormack et al.'s [35] approach from six systems to over 1,300 systems over two decades.

Analysis Strategy

Choice of observation interval

We used quarterly observation intervals to balance change visibility with confound minimization [39]. Monthly intervals were too sparse, precluding reliable architectural change detection. Our quarterly measurement window provides sufficient temporal separation to observe meaningful changes. We ensured causal ordering by lagging predictors one quarter. Robustness checks confirmed stability across alternative intervals.

Choice of estimation strategy

Unobserved characteristics (domain effects, team composition, and prior collaboration) can bias OLS estimates.[20] Following Kennedy [29: 223], Hausman tests rejected random-effects specifications ($\chi^2_{\text{architecture}}$ 154; χ^2_{module} 104; p<.001). Fixed-effects provide conservative estimates controlling for unobserved time-invariant characteristics while leveraging within-system variation. Diagnostic tests showed serial correlation (Breusch-Pagan $F_{\text{architecture}}$ 158.8; F_{module} 57.4; p<.001) and heteroskedasticity ($\chi^2_{\text{architecture}}$ 8.5e7; χ^2_{module} 2.5e8; both p<.001). We employed Driscoll-Kraay standard errors to address both issues. Multicollinearity diagnostics confirm acceptable VIF (max = 2.68 < 5). This econometric strategy addresses endogeneity beyond fixed-effects, accounting for reciprocal causality and path-dependence.

Endogeneity mitigation strategy

System decomposition represents a deliberate design choice [58], and team architecture reflects managerial coordination-optimizing choices [25]. Therefore our analysis must account for endogeneity due to reciprocal causality, historically-imprinted path-dependence, and time-variant unobserved heterogeneity—none addressable through fixed-effects alone.

First, accounting for reciprocal influence between architectures: Team architecture shapes system architecture [31], while system architecture influences who interacts with whom in a team [20: 11]. We instrumented system architecture with lagged team undecomposability, and vice versa, to account for such reciprocal effects, as informal coordination practices emerge to supplement formal team structure [7, 17]. The architecture of the team within which the search for refinements occurs constrains the system's evolved architecture. Similarly, yesterday's system architecture influenced today's team architecture, which influences tomorrow's system architecture. System architectures thus coevolve to reflect their development environment [8: 207, 34].[21]

Second, system architecture and team architecture at any observation interval are shaped by their own past levels [3]. To account for such path-dependence, we instrumented them with their lagged values. Following Smith's [54] argument, this recognizes that an artifact's structural details embody its accumulated evolutionary history. Our instruments satisfy exclusion restrictions through theoretical mechanisms that channel effects exclusively through endogenous predictors. For example, system age affects evolution through architectural degradation (instrumented system decomposability) rather than direct temporal effects on evolution mechanisms; similarly, team size influences evolution by shaping collaboration (instrumented team undecomposability) rather than directly driving evolution. This theoretical channeling ensures that instruments affect outcomes through their impact on system and team architectures, satisfying the exclusion restriction for causal identification.

Third, seven lagged$_{t-1}$ instruments from prior studies measured quarterly address time-variant system attributes. Table 3 summarizes them, also specifying whether the endogeneity each mitigates is from reciprocal causality (↔), historical path-dependence (■), or unobserved heterogeneity (•). *System age* causes systems' decomposability to deteriorate [57: 4, 58], as Java libraries and external modules available at the time of a system's creation—boundary resources [55]—shift over time. Similarly, *team size* increases integration effort [40], increasing team undecomposability. Larger average modules (*module granularity*) reduce system decomposability [58]; larger average class sizes (*class granularity*) decrease system decomposability (classes are spread over fewer modules) [58]. More extensive code documentation (*design codification*) eases decomposing; absence of slack in development work (*workload tautness*) instigates shortcuts [44], degrading decomposability. *Evolution velocity*—the quantity of code changes per quarter [57: 33]—affects both architecture and team division of labor [39].[22]

<< Insert Table 3 here>>

Model identification

Instrument validity requires correlation with endogenous predictors (relevance) while maintaining orthogonality to error terms (exclusion). Table 2 correlations suggest criterion satisfaction: average correlations with architecture evolution (-0.014) and module evolution (0.024) approach zero. Stock-Wright Lagrange Multiplier tests confirmed instrument strength and orthogonality for both architecture evolution ($\chi^2_{LM\text{-}S}$ 54.39; p<.001) and module evolution ($\chi^2_{LM\text{-}S}$ 54.57; p<.001). The large F-values (>10) of Stage 1 models ($F_{system\_decomposability}$ 989; $F_{team\_undecomposability}$ 633), provide evidence against instrument weakness. Excluded instrument F-tests ($F_{system\_decomposability}$ 429; $F_{team\_undecomposability}$ 411) exceeded the Stock-Yogo 10% threshold of 13.4; and Kleibergen-Paap's [30] underidentification test (χ^2 288; p<.001) rejected the null.

Finally, Sargan-Hansen's J-tests confirmed overidentifying restrictions validity ($\chi^2_{system\_decomposability}$ 1.48; $\chi^2_{team\_undecomposability}$ 2.72; both n.s.). Our first-stage F-values (989, 633) test joint significance of all instruments against the null hypothesis that all coefficients equal zero, with degrees of freedom (k-1, n-k) for k=9 instruments. The excluded instruments F-test (429, 411) exceeded the 10% maximal bias threshold of 13.4. These high F-values, and Kleibergen-Paap and Sargan-Hansen tests, provide assurance against weak instruments.

Our final empirical strategy employed fixed-effects controlling time-invariant characteristics; Driscoll-Kraay standard errors addressing serial correlation; robust standard errors correcting heteroskedasticity; instrumental variables addressing endogeneity; and quarter fixed-effects capturing temporal trends. Simultaneously using system-level controls through fixed-effects, team-level instruments capturing team dynamics, and environment-level variables accounting for technical change made our identification robust.

Analyses

The first stage of our instrumental-variables analyses (Table 4) used instruments to account for endogeneity. The second stage (Table 5) used the predicted values of system decomposability and team undecomposability to test our hypotheses. All variables were appropriately lagged by one quarter.

Stage 1: Endogeneity assessment

$$\begin{aligned}\text{System decomposability}_t \mid \text{Team undecomposability}_t = \alpha_0 + \alpha_1\text{Team undecomposability}_{t-1} + \alpha_2\text{System decomposability}_{t-1}\\ + \alpha_3\text{System age}_t + \alpha_4\text{Team size}_t + \alpha_5\text{Module granularity}_t + \alpha_6\text{Class granularity}_t\\ + \alpha_7\text{Design codification}_t + \alpha_8\text{Workload tautness}_t + \alpha_9\text{Evolution velocity}_t + \varepsilon \ldots\ldots\ldots\ldots(1)\end{aligned}$$

<< Insert Table 4 here>>

The results from Stage 1 in Table 4 (Equation 1) show that both models were significant and revealed some reciprocal causality. Prior system decomposability significantly reduced subsequent team undecomposability (T -3.39, p<.001), suggesting that decomposable systems lead to decomposable teams. However, the relationship between team undecomposability and subsequent system decomposability was nonsignificant. Both also exhibit historical path-dependence as indicated by the significant effects of their immediately-preceding historical levels. The larger coefficient of system architecture suggests that it is stickier over time than team architecture. A negative effect of system age on system decomposability (β = -.27, T-value = -7.20, p<.001) indicates architectural degradation with age. System age also had a negative effect on team undecomposability (β = -.05, T-value = -7.83, p<.001), confirming that teams progressively silo. The opposing effects of team size suggest that keeping both the system and the team decomposable gets harder as a team gets larger. Module granularity enhanced system decomposability (β = .07, T-value = 4.51, p<.001), but class granularity, design codification, and evolution velocity had no effects. This implies that more development activity (higher commit counts) accelerates architecture evolution but not module evolution. Finally, workload tautness's negative effects suggest that the absence of slack in development reduces system decomposability and team undecomposability. This stage explained over 61% of the variance in system decomposability and 50% in team undecomposability.

Stage 2: Hypothesis tests

Stage 2 used predicted ($\hat{y}$) values of system decomposability and team undecomposability from Stage 1 in a hierarchical fixed-effects model (Equation 2, results in Table 5 with hypothesis tests shown in **bold**.)

$$\text{Architecture evolution}_{t+1} \mid \text{Module evolution}_{t+1} = \beta_0 + \beta_1 \hat{y}_{\text{system\_decomposability\_t}} + \beta_2 \hat{y}_{\text{team\_undecomposability\_t}} + \beta_3(\hat{y}_{\text{system\_decomposability\_}t} \times \hat{y}_{\text{team\_undecomposability\_}t}) + \beta_4 \text{System age}_t + \beta_5 \text{Team size}_t + \beta_6 \text{Module granularity}_t + \beta_7 \text{Class granularity}_t + \beta_8 \text{Design codification}_t + \beta_9 \text{Workload tautness}_t + \beta_{10} \text{Evolution velocity}_t + \varepsilon \ldots\ldots (2)$$

<< Insert Table 5 here>>

System decomposability positively influenced architecture evolution (β = .005, T = 6.47, p<.001), supporting Hypothesis 1a, with each percentage point increase yielding a 0.5% greater architectural change. Team undecomposability had a positive effect on module evolution (β = .025, T-value = 4.84, p<.001), supporting Hypothesis 1b. A 1% increase in the number of developers generated a 2.5% increase in module evolution. System decomposability had no effect on module evolution and team undecomposability had no effect on architecture evolution. The interaction term had significant positive effects on both architecture evolution (β = .00, T-value = 2.00, p<.05) and module evolution (β = .001, T-value = 4.39, p<.001), supporting Hypotheses 2a and 2b. Stage 2's lower R^2 reflects instrumental variable allocation to Stage 1 endogeneity correction.

Robustness checks

To validate that file co-modification captures genuine team collaboration, we constructed developer social networks and assessed small-world properties (high clustering, with short path lengths) characteristic of human collaboration networks [65]. Team undecomposability correlated substantially with both clustering coefficient ($r = .706, p < .001$) and average shortest path ($r = .79, p < .001$), confirming our measure reflects actual collaboration patterns. We also corroborated by replicating the analyses with the control function approach. A key difference was the addition in the Stage 2 models (Appendix Table A2) of two residuals (ηs) from Stage 1, capturing deviations from theoretically-predicted levels. The two negative, significant ηs in the module evolution model imply that excess system decomposability or team undecomposability suppresses module evolution, with no such effects for architecture evolution. This suggests that systems benefit from under-decomposed structures paired with over-decomposed teams.

Limitations

a. Temporal and contextual boundaries. Post-hoc interactions with system age revealed no evidence that findings attenuate with system age; neither the system decomposability→architecture evolution relationship (β=-.00, t=-1.20, n.s.) nor the team undecomposability→module evolution relationship (β=-.00, t=-.01, n.s.) exhibited age-dependent effects. Java's explicit package structure enabled our analysis but limits generalizability to dynamically-typed languages where runtime resolution precludes static architectural analysis. Our 2011–2016 window predates microservices and containerization.

b. Sampling and measurement constraints. GitHub open-source systems may underrepresent proprietary enterprise software. Excluded episodically-developed systems [19, 53] limit generalizability to sustained development contexts. Team undecomposability exhibits a mechanical correlation with system size, mitigated through fixed-effects and log-transformation.[23] Small teams (≈4 developers) showed no architecture evolution benefits and attenuated syncing effects. File co-modification may underestimate informal dialog or overestimate temporally-coincident changes.

c. Operationalization limitations. Measuring architecture evolution through file additions and deletions, while tractable at scale, imperfectly captures linkage changes since counting all Java files encompasses private helper classes and test files. File-level changes reliably reflect architectural evolution due to Java's architectural conventions wherein public classes define module interfaces, strong network-based corroboration ($r = .83$, $p < .001$), and system fixed-effects. We refrained from directly measuring module dependency changes as that would introduce endogeneity with our decomposability measure,

d. Causal inference constraints. Our time-series design establishes temporal precedence without definitive causality. While we infer residual dependencies' presence through instruments, corroboration through commit message text analyses would strengthen causal claims. Such efforts would help push the boundaries of what's known about systems' evolution—the final frontier of sociotechnical inquiry. Excluding non-committing contributors (e.g., pair programmers, reviewers) potentially underestimates collaborative intensity.

DISCUSSION

We examined how dynamic team-system architecture syncing shapes evolutionary mechanisms amid architectural drift. Modules rarely align neatly with subteams; residual dependencies persist while system architecture continuously drifts [2, 33]. Recent IS studies of team breakdowns [62, 66], and architectural disconnects [2] underscore why technical and social dimensions merit simultaneous consideration. Our central theoretical contribution—that dynamic syncing between system and team architecture drives evolutionary success—advances this discourse.

We theorized mutual compensation between social and technical architectures for each other's emergent imperfections. This temporal unfolding of IT evolution—previously absent from static models—is crucial for three reasons: (a) evolving existing systems poses persistent challenges despite consuming vast IT resources [2, 22], (b) architectural drift is invisible to static analyses incapable of capturing dynamic interplay between system and team architectures [8: 207, 27: 211, 33], and (c) siloed research on team and technology architecture obscures the mechanistic pathways through which their coevolution unfolds [22, 62, 66].

Using Simon's [51] decomposability as a common theoretical foundation to conceptualize both system architecture and team architecture, we theorized that their syncing predicts two fundamental evolution mechanisms, whose permutations describe *all* forms of innovation in software systems. Conceptualizing systems evolution as the twin structural mechanisms of changes in modules and their interconnections bridges to Henderson and Clark's [26] classic innovation typology. Different evolutionary imperatives demand distinct mechanisms, making their differentiation consequential. System and team architectures exhibit markedly-different mechanistic effects, theorized here for the first time.

We tested these ideas using a social science analog to biologists studying living fossils: Examining historical records over decades, analyzing billions of lines of code across millions of modules in over 1,300 systems. We inferred drifting architecture by analyzing code and team architecture from individual-with-code interactions across 10.1 million changes, resembling a novel form of computational archaeology. By itself, system architecture influences only links among modules and team architecture influences modules' internal changes; their syncing simultaneously enhances both. Our results support the proposed ideas, making three novel contributions.

Contribution #1: Intertemporal Dynamics of System and Team Architectures

Our first contribution is advancing the conceptualization of system and team architectures from static snapshots to dynamic, evolving properties, departing from prevalent *a priori* perspectives [8: 152, 57]. System and team architectures are temporally dynamic properties rather than remaining stable across system lifecycles. Architecture observed at any point in time is a snapshot [54: 70]—a single frame precisely capturing structural state at that discrete interval. But snapshots obscure the continuous evolutionary trajectory between observations. A system degrading from 80% to 37% decomposability *seems* to undergo sudden collapse; yet snapshots mask whether this reflects a gradual drift through technical decisions, punctuated transitions, or compensating team reorganizations. The mechanisms—*how* change unfolds—remain invisible.

Examining quarterly observations spanning decades shifts perspective from snapshots to a movie that reveals such invisible patterns. System decomposability continuously declines while team structures differentially stabilize, and evolutionary mechanisms shift from module-focused early lifecycle to architecture-focused maturation. As technical debt accumulates, systems drift from their original design. Similarly, teams change over time and become misaligned with the system's evolved state. This reframes architecture from static design decisions to dynamic properties drifting along evolutionary trajectories; essentially transitioning from episodic to continuous assessment.

Recent IS work has extensively studied systems' architecture [2, 22, 59], and IS team studies emphasize coordination, communication, and conflict resolution [62, 66]. Yet, the absence of a team architecture conceptualization using theoretical foundations compatible with system architecture has impeded unified sociotechnical theorizing. While as a discipline, we appreciate that teams and systems influence each other, our theoretical conceptualizations of team organization impede grasping their interplay. We redressed this by conceptualizing team architecture using the same Simon-theoretic decomposability lens used to characterize systems architecture, particularly given IT architecture's absence in non-IS organizational architecture studies [3, 13]. System architecture and team architecture *together* constitute IS architecture; their grounding in Simon's decomposability theory provides building blocks for a hitherto-missing unified sociotechnical theory of systems evolution.

The upper half of Figure 8 shows these temporal dynamics in system and team architectures. The uppermost panel shows progressive system degradation; average system decomposability declined from 77% (1995) to 37% (2015), indicating systems became less modular as within-module dependencies decreased from 77% to 37% of total dependencies. This deterioration empirically corroborates architectural drift, wherein systems progressively deviate from initial modular designs through accumulated technical decisions, increasing architectural entanglement. The lower panel shows analogous team drift, where initial near-perfect decomposability deteriorated and then plateaued. These divergent trajectories—system decomposability declining while team decomposability stabilizes—illustrate how architectures drift apart over evolutionary timescales. Thus, when teams neglect resyncing with drifted system architectures, localized changes can cascade unpredictably through dependencies that are neither documented nor understood.

< INSERT FIGURE 8 HERE >

We extend the conceptualization of residual dependencies beyond merely preexisting, undocumented dependencies [2, 33], emphasizing that new ones emerge as systems incorporate technologies spanning multiple technology generations and accumulate technical debt [46]. This matters because, over time, new modules incorporate newer technologies while existing ones deprecate into successive generations of legacy technologies. Aging systems become "contraptions"—practically-useful amalgams of modules based on technologies spanning IT eras—with progressively brittle linkages. Once-good alignment can be lost as systems drift over time.

We cannot fully understand the evolutionary consequences of system architecture or team architecture without accounting for their intertwined histories. Every snapshot of system architecture carries historical baggage, embodying intertemporal accumulations of technical decisions. Our study is among the earliest to account for—before explaining systems' evolution—such reciprocal shaping of system architecture and team architecture, and the path-dependent inertia from the system's own architectural history. Parnas [43] anticipated such inevitability of change fifty years ago, though evidence of reciprocal system-team dynamics is more recent [2]. Our dynamic conceptualization plausibly explains why technical debt spirals despite sound initial design, and why system longevity depends on continuous organizational adaptation to technical drift. Econometrically accounting for reciprocal causality ensured cognizance of systems' history. Without correcting for endogeneity, Table A4 shows that we would overestimate main effects but fail to detect the benefits syncing them.

These patterns also surface system architecture's temporal stickiness relative to teams, making initial choices crucial as they create persistent evolutionary constraints. Initial architectural choices embed constraints into code that cannot be reversed; teams must continuously realign to accommodate technical trajectories established in earlier system lifecycle eras. Yesterday's team architecture shapes today's system, and today's system architecture shapes tomorrow's team. Thus, early-stage architectural design effort yields disproportionate returns for system longevity compared to later-stage team reorganization.

Contribution #2: Mechanisms

Our second contribution introduces two fundamental mechanisms—module evolution and architecture evolution—that parsimoniously characterize *all* forms of system evolution. The permutations of these mechanistic pathways map to established innovation typologies, providing a unified framework for understanding diverse evolutionary patterns. Prior conceptualizations emphasize rapidity [1, 15, 59] and ease [22, 48] of how a system changes *as a whole*, but without articulating the content of the changes. Rather than treating evolution coarsely as aggregate change, our mechanistic pathways directly answer IS scholars' calls for precise evolution pathways [22, 48].

Our results show that architectural properties activate mechanisms through mechanism-specific pathways. System decomposability does not reduce the cognitive demands of within-module refinement, which requires deep domain expertise regardless of coupling; conversely, team collaboration cannot substitute for architectural flexibility when restructuring interconnections, which demands reduced cross-module dependencies. This specificity—that system architecture catalyzes only architecture evolution while team architecture catalyzes only module evolution—explains why certain evolutionary efforts can stall despite good architecture or skilled teams. A project may possess optimal team structure for module evolution yet have a monolithic system architecture that prohibits architectural innovation, or conversely, a decomposable system undermined by siloed teams unable to navigate residual dependencies.

This mechanistic distinction has substantial implications as different types of business moves require activating one mechanism or the other. Appreciating them can guide evolution efforts based on a system's short-term operational needs versus long-term evolvability needs. Module evolution enables rapidly responding to immediate business needs—enhancing features, fixing bugs, and fine-tuning performance—directly impacting operational reliability. Architecture evolution, in contrast, serves a more strategic role by creating the structural flexibility to incorporate new technologies, prevent technical obsolescence, and adjust existing systems to interoperate with newer generations of interoperating systems.

The sample-wide sparklines in the lower half of Figure 8 show how these mechanisms differentially unfold, with system evolution shifting from predominantly module-focused to architecture-focused evolution over time. The variance in architecture evolution progressively attenuated in amplitude; module evolution also plateaued. Further, variance in new module additions gradually declined, with changes concentrating on fewer parts of a system's codebase. This intertemporal shift from module evolution to architecture evolution reflects a maturation process. Early evolution in a system's lifecycle perfects individual modules' implementation, while later evolution reconfigures their connections to incorporate new technologies and systemwide capabilities. Module implementations solidifying over time reduce the need for internal module changes, while new technical requirements demand changing interconnections among modules rather than modifications to discrete modules. Aging legacy modules become harder to modify later but can be leveraged further through new interfaces.

This pattern of systems evolution, gradually shifting from module- to architecture-focused evolution, also suggests a theoretical rationale for why technical debt accumulates differently in mature versus new systems. In early stages, module evolution creates localized technical debt, while in later stages, evolution through architectural changes creates more systemwide technical debt that's harder to address [46]. This lifecycle progression emerges because once-evolvable modules become prohibitively risky to modify internally due to accumulated technical debt. It implies that organizational interventions should shift focus from module optimization early on to architectural innovation later, with corresponding adjustments to team structure.

This trajectory helps explain why older systems often maintain stable core modules while evolving through new interface layers—a pattern evident in how legacy mainframe systems integrate with modern technologies. The Federal Aviation Administration's air traffic control systems illustrate this pattern, where core COBOL modules from the 1970s remain largely unchanged due to modification risks, while multiple modern interface layers have been added to connect these legacy systems to contemporary technologies like pilots' tablets and airlines' reservation systems.

Our results show that these mechanisms respond to different architectural influences: System decomposability fosters the evolution of the architecture linking modules but not the modules themselves, while team undecomposability fosters internal module evolution but not their architectural links. The theoretical prowess of these mechanisms in their parsimony; permutations of these two mechanisms encompass the entire spectrum of Henderson and Clark's [26] innovation typology (Figure 9)—incremental, modular, architectural, and radical—bridging IS evolution theory with classic innovation scholarship.[24]

< INSERT FIGURE 9 HERE >

- *Incremental innovation* (low architecture evolution + low module evolution) is associated with decomposable teams
- *Modular innovation* (low architecture evolution + high module evolution) is associated with decomposable teams and moderate system decomposability
- *Architectural innovation* (high architecture evolution + low module evolution) is associated with decomposable systems

– *Radical innovation* (high architecture evolution + high module evolution) is associated with tight-knit teams and intermediate system decomposability

Table A3 illustrates the theoretical loss from not differentiating these two fundamental mechanisms. It replicates our model simply aggregating all quarterly changes (*t-1→t;* percentage). The significant main effects and interaction would consistently infer their interplay without discerning the specific mechanistic pathways through which system and team architecture catalyze systems evolution.

Contribution #3: Syncing Team Architecture with System Architecture

Our third contribution is that the syncing of team architecture with system architecture amplifies their effects—they matter more together. Greater team undecomposability amplifies the benefits of system decomposability in enhancing architecture evolution. Greater system decomposability amplifies the benefits of tight-knit teams in enhancing the system's modules. Booch [8: 79] presciently advocated 30 years ago—counterintuitively for his time—for tight-knit teams for object-oriented (modular) systems. Our findings offer a mechanistic account of his intuition. When architectural properties operate in tension—one constrained while the other remains flexible—their syncing preserves evolvability that neither alone provides. Momentary alignment snapshots reveal which configurations sustain both mechanisms at discrete intervals; syncing explains how teams maintain responsiveness as systems drift away from alignment.

An evolving system perfectly aligned at quarter *t* might exhibit measurable misalignment by quarter t+5 due to architectural drift. Syncing describes the dynamic process of teams restoring alignment despite inevitable erosion. Specifically, team architecture offsets drifting system architecture by facilitating early detection of residual dependencies through cross-module collaboration. Undocumented coupling remains invisible when developers work in module-specific silos, becoming visible only when direct interaction across module boundaries surfaces unexpected dependencies. System decomposability mitigates tight-knit team coordination overhead by confining interdependencies to residual coupling, freeing cognitive resources for focused module refinement. Team architectures—which determine what dependencies remain hidden until later—therefore cannot be evaluated in isolation from system architecture; they must coevolve to compensate for inevitable drift. Simon's decomposability, applied symmetrically to system architecture and team architecture, provides a common vocabulary spanning both technical and social domains, and a foundation for theorizing dynamics of their coevolution. This symmetric approach is generalizable to any sociotechnical systems beyond software development, where technical and organizational structures jointly shape what information surfaces and when.

<<INSERT FIGURE 10 HERE >>

The interaction plot in Figure 10 illustrates this. The left panel shows that system decomposability's influence on architecture evolution is stronger with undecomposable teams (the steeper solid line) than decomposable (the dotted line). Conversely, the right panel shows team undecomposability's effect on module evolution is stronger with decomposable systems (the steeper solid line) than undecomposable (the dotted line).

< INSERT FIGURE 11 HERE >

Figure 11's heatmaps further show how different system-team combinations influence both evolution mechanisms. Contrary to IT folklore, isomorphic architectures produce suboptimal outcomes—decomposable systems with decomposable teams (cell A) underperform across both mechanisms while monolithic systems with tight-knit teams (cell B) excel at module evolution but are terrible at architectural evolution. Optimizing solely for system architecture (cell D) or team architecture (cell B) maximizes one evolutionary mechanism while subduing the other. A system architecture-focused approach (cell D) generates the highest architectural evolution but inhibits module evolution. Only one configuration—decomposable system architectures paired with tight-knit teams (cell C)—simultaneously activates both mechanisms, achieving both architectural and module evolution without maximizing either.

Implications for Practice

Software development managers and technical leads should determine target innovation types—incremental, modular, architectural, or radical (Figure 9)—to guide architectural choices. The 2x2 in Figure 12 translates our results into actionable managerial insights. Managers should assess evolutionary intent—module refinement (y-axis) versus architectural reconfiguration (x-axis)—positioning the system in one of four cells in Figure 12, each indicating what aspects of architecture demand attention. If they intend to evolve by tweaking a system's modules (cell A), then they must not divide the team up into discrete subteams. If the intent is to evolve links among modules (cell C), they must increase decomposability of the system's architecture. If they intend to do both (cell B), they must simultaneously increase the system's decomposability and make the team tight-knit. Since team architecture is less hamstrung by its history relative to system architecture, adjusting teams provides managers a powerful lever within their control.

<< INSERT FIGURE 12 HERE >>

Ideas for future research

First, when do upfront decomposability costs outweigh evolutionary advantages, particularly for legacy monolithic systems with disproportionately-large central modules? Second, how does architectural drift differ between open-source and proprietary systems [11]? Third, future studies should attempt to capture both intended and emergent architectural properties. Fourth, Simon's [52: 598] concept of *allometric compatibility*—how different aspects of systems must grow proportionally to maintain alignment—can help understand how technical and social architectures scale together. This lens can help explain why certain IT configurations destabilize as systems scale, analogous to how biological systems face structural constraints as they grow larger. Just as biological growth requires coordinated development of multiple subsystems, system evolution amid technical change might require simultaneous changes in both system and team architecture. Finally, while we show how sociotechnical syncing matters, the theoretical mechanisms underlying its *initial* emergence merit further conceptual development.

CONCLUSION

We broadened systems evolution from a predominantly artifact-centric discourse to a dynamic, sociotechnical process of syncing teams with drifting systems. Using Simon's decomposability to envision the architectures of both systems and teams, we showed the distinct mechanistic pathways through which each catalyzes systems evolution. Our unearthing of previously-invisible patterns using novel computational archaeology is an initial step towards understanding their coevolution. More broadly, we advocate attention to how their coevolution helps systems live long and prosper.

REFERENCES

1. Agarwal, R. and Tiwana, A. Evolvable Systems: Through the Looking Glass of IS. *Information Systems Research*, 26, 3 (2015), 473-479.
2. Akbari, K., Fürstenau, D., and Winkler, T. Governance and Longevity of Architecturally Embedded Applications. *Journal of Management Information Systems*, 41, 1 (2024), 266-296.
3. Albert, D. Organizational Module Design and Architectural Inertia: Evidence from Structural Recombination of Business Divisions. *Organization Science*, 29, 5 (2018), 890-911.

4. Baldwin, C. *Design Rules 2: How Technology Shapes Organizations*. Cambridge, MA: MIT Press, 2024.
5. Baldwin, C. and Clark, K. *Design Rules: The Power of Modularity*. Cambridge: MIT Press, 2000.
6. Bartling, B., Fehr, E., and Schmidt, K. Screening, Competition, and Job Design. *American Economic Review*, 102, 2 (2012), 834-64.
7. Ben-Menahem, S., Von Krogh, G., Erden, Z., and Schneider, A. Coordinating Knowledge Creation in Multidisciplinary Teams. *Academy of Management Journal*, 59, 4 (2016), 1308-1338.
8. Booch, G. *Object Solutions*. Reading, MA: Addison-Wesley, 1996.
9. Brusoni, S. and Prencipe, A. Unpacking the Black Box of Modularity: Technologies, Products and Organizations. *Industrial and Corporate Change*, 10, 1 (2001), 179-205.
10. Brusoni, S. and Prencipe, A. Making Design Rules. *Organization Science*, 17, 2 (2006), 179-189.
11. Cheng, H.K., Liu, Y., and Tang, Q.C. The Impact of Network Externalities on the Competition between Open Source and Proprietary Software. *Journal of Management Information Systems*, 27, 4 (2011), 201 - 230.
12. Clement, J. and Puranam, P. Searching for Structure. *Management Science*, 64, 8 (2018), 3879-3895.
13. Crilly, D. and Sloan, P. Autonomy or Control? Organizational Architecture and Corporate Attention to Stakeholders. *Organization Science*, 25, 2 (2014), 339-355.
14. DeMarco, T. *Controlling Software Projects*. New York: Yourdon Press, 1982.
15. Eaton, B., Elaluf-Calderwood, S., Sørensen, C., and Yoo, Y. Distributed Tuning of Boundary Resources. *MIS Quarterly*, 39, 1 (2015), 217-243.
16. Edberg, D.T., Ivanova, P., and Kuechler, W. Methodology Mashups: An Exploration of Processes Used to Maintain Software. *Journal of Management Information Systems*, 28, 4 (2012), 271 - 304.
17. Espinosa, J.A., Slaughter, S.A., Kraut, R.E., and Herbsleb, J.D. Team Knowledge and Coordination in Geographically Distributed Software Development. *Journal of Management Information Systems*, 24, 1 (2007), 135-169.
18. Ethiraj, S. and Levinthal, D. Modularity and Innovation in Complex Systems. *Management Science*, 50, 2 (2004), 159-173.
19. Fang, Y. and Neufeld, D. Understanding Sustained Participation in Open Source Software Projects. *Journal of Management Information Systems*, 25, 4 (2009), 9-50.
20. Ford, N., Parsons, R., and Kua, P. *Building Evolutionary Architectures.* CA: O'Reilly, 2017.
21. Fürstenau, D., Baiyere, A., Schewina, K., Schulte-Althoff, M., and Rothe, H. Extended Generativity Theory on Digital Platforms. *Information Systems Research*, 34, 4 (2023), 1686–1710.
22. Fürstenau, D. and Baiyere, A.K., N. A Dynamic Model of Embeddedness in Digital Infrastructures. *Information Systems Research*, 30, 4 (2019), 1319-1342.
23. Gopalakrishnan, S., Matta, M., and Cavusoglu, H. The Dark Side of Technological Modularity. *Information Systems Research*, 33, 3 (2022), 1072-1092.
24. Hansen, M. Knowledge Networks. *Organization Science*, 13, 3 (2002), 232-248.
25. Harris, M. and Raviv, A. Organization Design. *Management Science*, 48, 7 (2002), 852-865.
26. Henderson, R. and Clark, K. Architectural Innovation: The Reconfiguration of Existing Product Technologies and the Failure of Established Firms. *Administrative Science Quarterly*, 35,(1990), 9-30.
27. Hohpe, G. *The Software Architect Elevator: Redefining the Architect's Role in the Digital Enterprise*. Sebastopol, CA: O'Reilly, 2020.
28. Huang, J., Henfridsson, O., and Liu, M. Extending Digital Ventures through Templating. *Information Systems Research*, 33, 1 (2022), 285-310.
29. Kennedy, P. *A Guide to Econometrics*. Cambridge, MA: MIT Press, 1992.
30. Kleibergen, F. and Paap, R. Generalized Reduced Rank Tests Using the Singular Value Decomposition. *Journal of Econometrics*, 133, 1 (2006), 97-126.
31. Leonardi, P., Bailey, D., and Pierce, C. The Coevolution of Objects and Boundaries over Time. *Information Systems Research*, 30, 3 (2019), 665-686.
32. Leong, C., Lin, S., Tan, F., and Yu, J. Coordination in a Digital Platform Organization. *Information Systems Research*, 35, 1 (2024), 363-393.

33. Lindberg, A., Berente, N., Gaskin, J., and Lyytinen, K. Coordinating Interdependencies in Online Communities. *Information Systems Research*, 27, 4 (2016), 751-772.
34. MacCormack, A., Baldwin, C., and Rusnak, J. Exploring the Duality between Product and Organizational Architectures. *Research Policy*, 41, 8 (2012), 1309-1324.
35. MacCormack, A., Rusnak, J., and Baldwin, C. Exploring the Structure of Complex Software Designs. *Management Science*, 52, 7 (2006), 1015-1030.
36. Machamer, P., Darden, L., and Craver, C. Thinking About Mechanisms. *Philosophy of Science*, 67, 1 (2000), 1-25.
37. Mehta, N. and Bharadwaj, A. Knowledge Integration in Outsourced Software Development. *Journal of Management Information Systems*, 32, 1 (2015), 82-115.
38. Mihm, J., Loch, C., and Huchzermeier, A. Problem-Solving Oscillations in Complex Projects. *Management Science*, 49, 6 (2003), 733-750.
39. Mitchell, T. and James, L. Building Better Theory: Time and the Specification of When Things Happen. *Academy Of Management Review*, 26, 4 (2001), 530-547.
40. Nidumolu, S. The Effect of Coordination and Uncertainty on Software Project Performance. *Information Systems Research*, 6, 3 (1995), 191-219.
41. O'Riordan, S., Emerson, B., Feller, J., and Kiely, G. The Road to Open News: A Theory of Social Signaling in an Open News Production Community. *Journal of Management Information Systems*, 40, 1 (2023), 130-162.
42. Ozer, M. and Vogel, D. Contextualized Relationship between Knowledge Sharing and Performance in Software Development. *Journal of Management Information Systems*, 32, 2 (2015), 134-161.
43. Parnas, D. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15, 9 (1972), 1053-1058.
44. Perlow, L. The Time Famine: Toward a Sociology of Work Time. *Administrative Science Quarterly*, 44, 1 (1999), 57-81.
45. Puranam, P. *The Microstructure of Organizations*. Oxford: Oxford University Press, 2018.
46. Ramasubbu, N. and Kemerer, C. Controlling Technical Debt Remediation in Outsourced Enterprise Systems Maintenance. *Journal of Management Information Systems*, 38, 1 (2021), 4-28.
47. Raveendran, M., Puranam, P., and Warglien, M. Object Salience in the Division of Labor: Experimental Evidence. *Management Science*, 62, 7 (2016), 2110-2128.
48. Rolland, K., Mathiassen, L., and Rai, A. Managing Digital Platforms in User Organizations. *Information Systems Research*, 29, 2 (2018), 419-443.
49. Schilling, M. Toward a General Modular Systems Theory and its Application to Interfirm Product Modularity. *Academy Of Management Review*, 25, 2 (2000), 312-334.
50. Siggelkow, N. Evolution toward Fit. *Administrative Science Quarterly*, 47, 1 (2002), 125-159.
51. Simon, H. The Architecture of Complexity. *Proceedings of the American Philosophical Society*, 106, 6 (1962), 467-482.
52. Simon, H. Near Decomposability and the Speed of Evolution. *Industrial and Corporate Change*, 11, 3 (2002), 587-599.
53. Singh, P.V. and Tan, Y. Developer Heterogeneity and Formation of Communication Networks in Open Source Software Projects. *Journal of Management Information Systems*, 27, 3 (2011), 179 - 210.
54. Smith, C. *A Search for Structure*. Cambridge, MA: MIT Press, 1981.
55. Soh, F. and Grover, V. Leveraging Platform Boundary Resources: The Role of Distributed Sensemaking. *Journal of Management Information Systems*, 39, 2 (2022), 366-394.
56. Sosa, M., Eppinger, S., and Rowles, C. The Misalignment of Product Architecture and Organizational Structure in Complex Product Development. *Management Science*, 50, 12 (2004), 1674-1689.
57. Sterling, C. *Managing Software Debt*. Boston: Addison-Wesley, 2010.
58. Subramanyam, R., Ramasubbu, N., and Krishnan, M. In Search of Efficient Flexibility. *Information Systems Research*, 23, 3 (2012), 787-803.
59. Tiwana, A. Evolutionary Competition in Platform Ecosystems. *Information Systems Research*, 26, 2 (2015), 266-281.
60. Tiwana, A. Platform Synergy. *Information Systems Research*, 29, 4 (2018), 829-848.

61. Tiwana, A. and Safadi, H. Atrophy in Aging Systems: Evidence, Dynamics, and Antidote. *Information Systems Research*, 35, 1 (2024), 66-86.
62. Tsai, J., Jiang, J., Klein, G., and Hung, S. Task Conflict Resolution in Designing Legacy Replacement Systems. *Journal of Management Information Systems*, 40, 3 (2023), 1009-1034.
63. Venkatraman, N. The Concept of Fit in Strategy Research: Toward Verbal and Statistical Correspondence. *Academy Of Management Review*, 14, 3 (1989), 423-444.
64. Vial, G. A Complex Adaptive Systems Perspective of Software Reuse in the Digital Age. *Information Systems Research*, 34, 4 (2023), 1728-1743.
65. Watts, D. and Strogatz, S. Collective Dynamics of Small-World Networks. *Nature*, 393, 6684 (1998), 440-442.
66. Wong, S., Zhang, L., Černe, M., and Moe, N. Influence of Digital Communication Configuration in Virtual Teams: A Faultline Perspective. *Journal of Management Information Systems*, 41, 4 (2024), 1111-1141.

Endnotes

[1] Java's explicit package declarations, access modifiers (public/private/protected), and import statements enable precise identification of module boundaries and separating within-module dependencies (classes accessing others within the same package) from cross-module dependencies (classes accessing others across package boundaries). We used frequency-weighted measures rather than binary dependency to capture coupling intensity beyond mere presence or absence of architectural connections. This granular approach is crucial in operationalizing decomposability; two systems might exhibit identical module structures yet differ substantially in within-module cohesion strength and cross-module entanglement severity. Frequency weighting allows comparative analysis across systems of varying scales and technological maturity without conflating architectural topology with dependency intensity.

[2] Scholars use various terms synonymously for undecomposable systems e.g., coarse-grained [58], monolithic [60], or integral [49]. Modules are conceptually synonymous with subsystems and components. Team decomposability refers to relatively-independent subteams; its inverse (team undecomposability) reflects organizational designs requiring extensive integration and coordination [7, 45]. Just as classes interact through technical mechanisms like inheritance and composition, team interactions manifest in developers modifying the same modules—activities that parallel how classes share implementation and coordinate functionality

[3] The distinction between within-module and among-module dependencies reflects Parnas' [43] fundamental design principles: within-module dependencies encapsulate implementation decisions, while dependencies among modules propagate changes across boundaries, requiring developers to coordinate across multiple subsystems. This explains why decomposability minimizes dependencies among modules while accepting dependencies within them.

[4] Our two mechanisms mirror Ramasubbu et al.'s [46] system-level adjustments versus module refinements distinction; Albert's [3] module-specific and systemwide distinction; and Ethiraj's [18] localized versus system-level adaptation. This distinction matters theoretically as technical debt-laden systems handicap organizations [46], and architectural misalignment imperils system longevity [2]. Module-level evolution has received scant scholarly attention despite small changes driving systems' evolution [20: viii], especially when experimenting with new ideas in IT artifacts [28].

[5] Minimal cross-module dependencies facilitate architectural changes by reducing cascading effects when interfaces are modified. However, loose coupling doesn't ease within-module changes, which depend on internal properties. Loosely-coupled modules with high cohesion remain difficult to modify internally despite minimal external dependencies, as tightly-connected elements require coordinated changes. Conversely, loosely-coupled modules with low cohesion are easier to modify internally but lack stable boundaries for architectural restructuring. Thus, system decomposability—combining loose coupling with high cohesion—enables architectural evolution but not module evolution.

[6] System decomposability (dependency structure at time t) and architecture evolution (boundary changes between t and $t+1$) are empirically distinct, with a 0.1 correlation. Decomposable systems may remain static, while entangled systems may undergo restructuring; the constructs vary independently.

[7] Fit-as-moderation is appropriate when alignment is theoretically anchored on specific dependent variables (module and architecture evolution), has high theoretical specificity, and depends on predictor interactions. Conceptually, fit-as-interaction is identical to the notion of complementarities, in which an increase in one predictor increases the value of the other [6]. Baldwin [4: 12] describes complements in a technological system on similar lines.

[8] Prior studies have observed this propensity to act locally within teams. Individuals working on a particular component in product development teams tend to work without much concern for other components [38]. Similarly, individuals responsible for module maintenance for existing enterprise systems do not always consider system-level design. Metaphorically, focusing on the trees loses sight of the forest.

[9] This coordination overhead differs qualitatively from within-module coordination. Developers working on different classes within the same module face coordination challenges, but these occur among elements sharing functional cohesion and unified purpose, requiring substantially less cognitive load than cross-module coordination managing dependencies between functionally-distinct architectural units (modules) with separate interfaces and potentially conflicting design philosophies.
[10] *JMIS* publishes ~1,200 pages a year, with 530 words per page. The average English word is about 5 characters, which rises to six characters per word to account for spaces and punctuation. This translates into about 3,726 KB of plaintext published annually. Our source code translates into (346 GB ÷ 3,726 KB) = 97,361 years of the journal.
[11] At least monthly commits ensured ongoing development activity; excluding forks and artificially condensed repositories. From GitHub, we collected each system's source code and lifetime commit history through 2016. We halted data collection in 2016 because Java 9's introduction of modules in 2017 would fundamentally alter our analysis. Java 9 introduced the new concept of "module" which allows for further grouping and bundling of packages. Since the code in this study predates this change in the Java language, we can safely consider a package to be the main modular unit in Java systems.
[12] When we refer to adding a module, we mean adding files that create a new package; module evolution means changes *within* existing files; architecture evolution means file additions/deletions. This precise mapping enables empirics-to-theory alignment: packages instantiate modules, files instantiate implementation units, classes instantiate object-oriented constructs. Java's explicit structural conventions enable this precision. Package declarations define module boundaries; access modifiers (public/private/protected) control cross-boundary visibility. Classes declared public are visible across packages; package-private classes (the default) are visible only *within* their package. *Within-module* dependencies leverage package-private declarations and internal inheritance; cross-module interactions pass through public interfaces enforced by the compiler. This architectural constraint—where files define package boundaries and file-level changes restructure module interconnections—enables reliable static dependency analysis unavailable in dynamically-typed languages (e.g., Python's implicit imports, JavaScript's flexible scoping). A package is any Java package, including nested packages. A class in a subpackage belongs only to that subpackage; it cannot access the parent package's classes without explicit imports. For example:
```
com
     └── example
          └── outer
                   ├── OuterClass.java
                   └── inner
                              └── InnerClass.java
```
OuterClass belongs only to com.example.outer; *InnerClass* belongs only to com.example.outer.inner; neither can access each other without proper imports.
[13] This is the approximate count of unique GitHub usernames matched to code commits across our systems. This count excludes anonymous commits, organizational bot accounts executing automated deployment scripts, and redundant accounts where we identified developers using multiple usernames through GitHub's user unification mechanisms. Each commit record contains a unique developer identifier enabling reconstruction of collaborative patterns through file co-modification analysis across our 10.1 million commits. The developer count is conservative estimate focusing on identifiable human contributors engaged in *substantive* code development rather than peripheral activities.
[14] Values approaching 100% represent highly modular architectures where dependencies remain largely contained within module boundaries, enabling isolated module modifications without systemwide coordination. Conversely, values approaching 0% indicate monolithic structures characterized by extensive cross-module coupling that complicates isolated modifications. We often observe this pattern in legacy systems with eroded architectural boundaries. We handle boundary conditions by setting decomposability to 100% when cross-module dependencies equal zero (representing perfect decomposition) and to 0% when within-module dependencies equal zero (indicating complete architectural entanglement).
[15] Our measure captures coordination through shared file modifications. This excludes coordination between developers on separate but interdependent modules (which required unavailable communication data) but conservatively emphasizes *observable* technical interdependence. File co-modifications require coordination via Git merge protocols, code review, and continuous integration, rendering coordination reliably observable. Developers working on architecturally-distant modules contribute to team undecomposability when they modify the same files, reflecting genuine cross-module coupling requiring coordination. This captures frequency and depth of interaction patterns, degree to which developers work across multiple modules, and pattern of code modifications across subteam boundaries through observable technical collaboration. While other forms of developer interaction exist, our focus on shared file modifications provides a measurable, dynamic indicator that can be consistently tracked across large codebases. This

differs fundamentally from developers working on separate files within the same module, where independent work proceeds without integration overhead. Perfect team decomposition would have individual developers working on separate modules, while perfect undecomposability would entail all developers working on all modules.

[16] Architecture and module evolution measures can exceed 1.0 when systems undergo major restructuring (e.g., deleting 60% of files while adding 60% new files = 1.2 value). Our data show maximum of 1.0 for both measures, indicating at least one system underwent complete architectural replacement during observation. This range distinguishes incremental adjustments (0.1–0.3) from transformational changes (≈1.0).

[17] We aggregated file-level changes across the entire system rather than calculating module-level evolution then averaging across modules. This approach aligns with our theoretical focus on system-level evolution rather than module-specific evolution. Module-level averaging would require weighting schemes to address heterogeneity in module size, complexity, and functional centrality that would introduce additional atheoretical measurement assumptions. Our system-level aggregation, combined with fixed-effects accounting for time-invariant system characteristics including module structure, treats all file-level changes as contributing to overall system evolution regardless of specific module location.

[18] Architecture evolution is operationally distinct from system decomposability; decomposability measures dependencies at time t (static structure), while architecture evolution measures boundary changes between t and $t+1$ (dynamic change). This distinction avoids tautology and maintains construct independence. Decomposable systems can remain architecturally static despite structural flexibility; monolithic systems may undergo restructuring attempts. These constructs vary independently both conceptually and empirically ($r = .10$). Our measure excludes *within-module* dependency changes because these represent internal refinement rather than architectural restructuring. Links within modules constitute cohesive relationships among elements serving unified purposes; modifications to these relationships refine module internals without altering the module's architectural boundaries or interface relationships with other modules. Architecture evolution specifically captures changes in module interconnections—addition, removal, or reconfiguration of inter-module dependencies that restructure the system's overall topology. Including within-module changes would conflate our theoretically-distinct mechanisms: system decomposability facilitating boundary restructuring versus team undecomposability facilitating internal refinement. This distinction separates dynamic architectural change from static architectural structure, enabling precise measurement of evolutionary mechanisms.

[19] Our normalized ratio of within-module to cross-module dependencies treats changes asymmetrically at boundary extremes (near 0% or 100%).

[20] For example, the system might be in a domain with widespread availability of software components or microservices; might have had a luminary architect (e.g., Linux); or might have been sponsored by a major firm (e.g., Android); or individuals working on it might have apriori collaborative history.

[21] Ben-Menahem et al.'s [7] case study shows subteam structures adjusting as new dependencies are discovered. A similar temporal pattern appears semesterwide in a typical undergraduate MIS class. It begins as an undecomposed class, divides into smaller groups shrinking or expanding as the add-drop period ends, and disbands at the end of the semester.

[22] Evolution velocity (code commit count per quarter) was uncorrelated with both architecture evolution ($r = .15$, Table 2) and module evolution ($r = .09$), supporting its use as an exogenous variable.

[23] *Mechanical relationship mitigation*: Larger systems with more files create more developer pair opportunities, potentially confounding team architecture with system scale. We mitigated this through: (1) system fixed-effects absorbing time-invariant size, leveraging within-system rather than cross-system variation; (2) log-transformation compressing scale effects so doubling system size increases the measure incrementally rather than proportionally; (3) controls for team size, lines of code, module counts, and granularity; and (4) within-system estimation comparing each system's collaborative intensity to its own temporal baseline. *Small team boundary conditions*: Median-split analyses on teams ≈4 developers revealed boundary conditions; tight-knit small teams showed no architecture evolution benefits from system decomposability ($\beta=-.01$, $t=-0.43$, n.s.) and attenuated alignment effects on module evolution ($\beta=.0004$, $t=0.38$, n.s.), suggesting our mechanistic theory operates in team contexts with sufficient scale for meaningful subteam differentiation. Our normalized ratio of within-module to cross-module dependencies treat changes asymmetrically and require careful interpretation at boundary extremes, suggesting alternative operationalizations merit exploration.

[24] Incremental innovation alters barely changes modules or their links; radical changes both; architectural changes only links; modular changes only modules. The 1/3rd percentile-split, dichotomized high versus low levels of architecture and module evolution correspond to Henderson-Clark's [26] notion of changed versus unchanged links and components, respectively. Each cell depicts system decomposability (dotted box) and team undecomposability (solid box). Relative comparisons show that tight-knit teams with intermediate system decomposability exhibit radical innovation (cell C), the highest system decomposability fosters architectural innovation (cell D), and the highest team decomposability—lowest undecomposability—fosters incremental innovation (cell A).

FIGURES AND TABLES

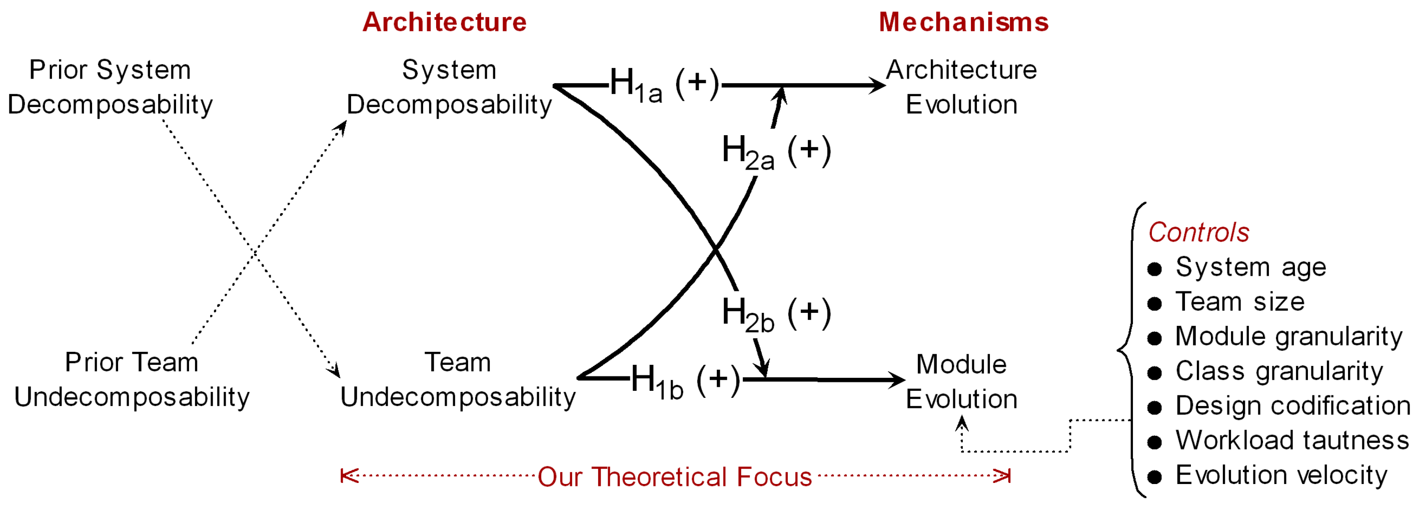

Figure 1: The research model situated in its broader nomological network of prior IS studies.

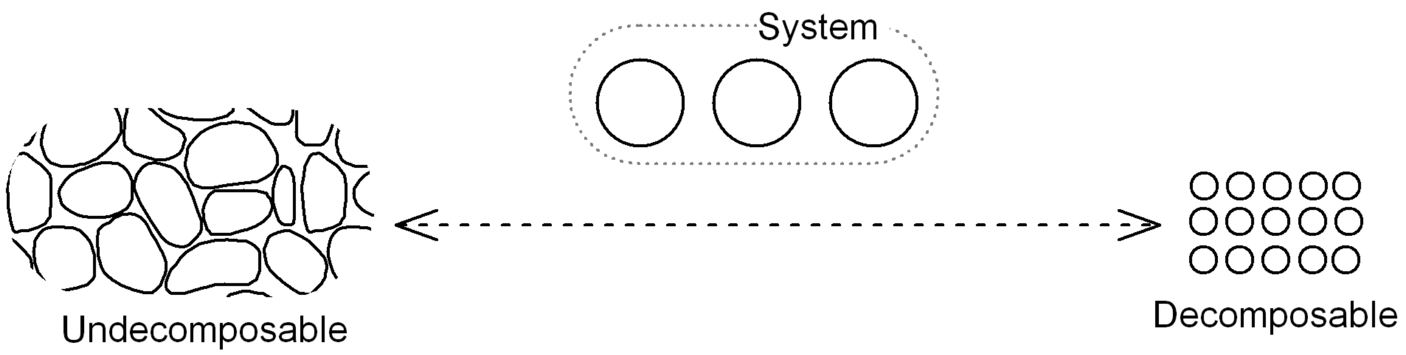

Figure 2: Any system—technological or organizational—can reside anywhere on the undecomposable-to-decomposable continuum and also drift over time.

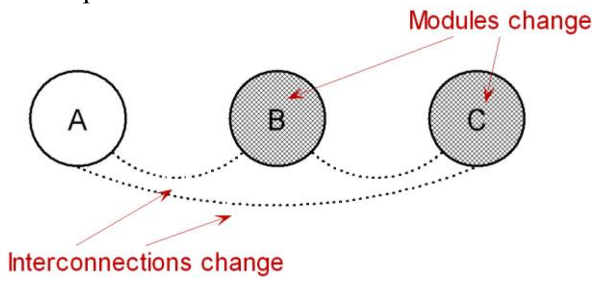

Figure 3: The two fundamental system evolution mechanisms: Architecture evolution alters modules' interconnections (dotted lines) and module evolution alters modules' internals (hatched areas).

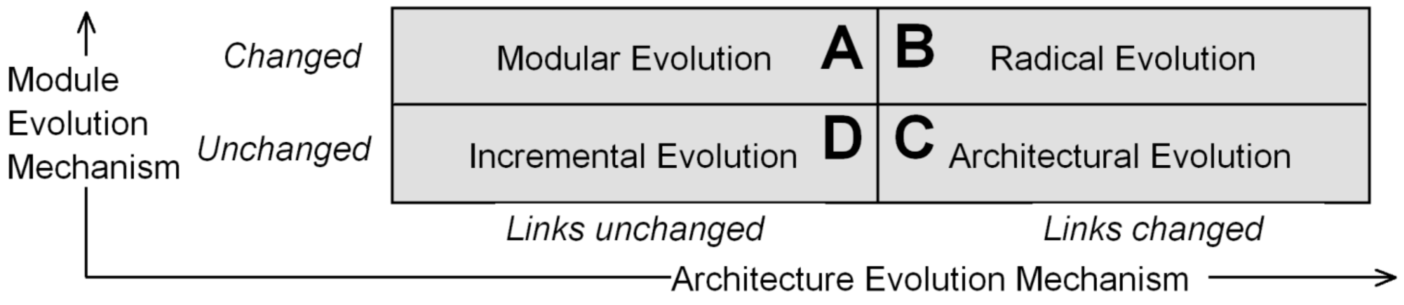

Figure 4: The two fundamental mechanisms—architecture evolution and module evolution—jointly encompass all classic four modes through which innovation occurs.

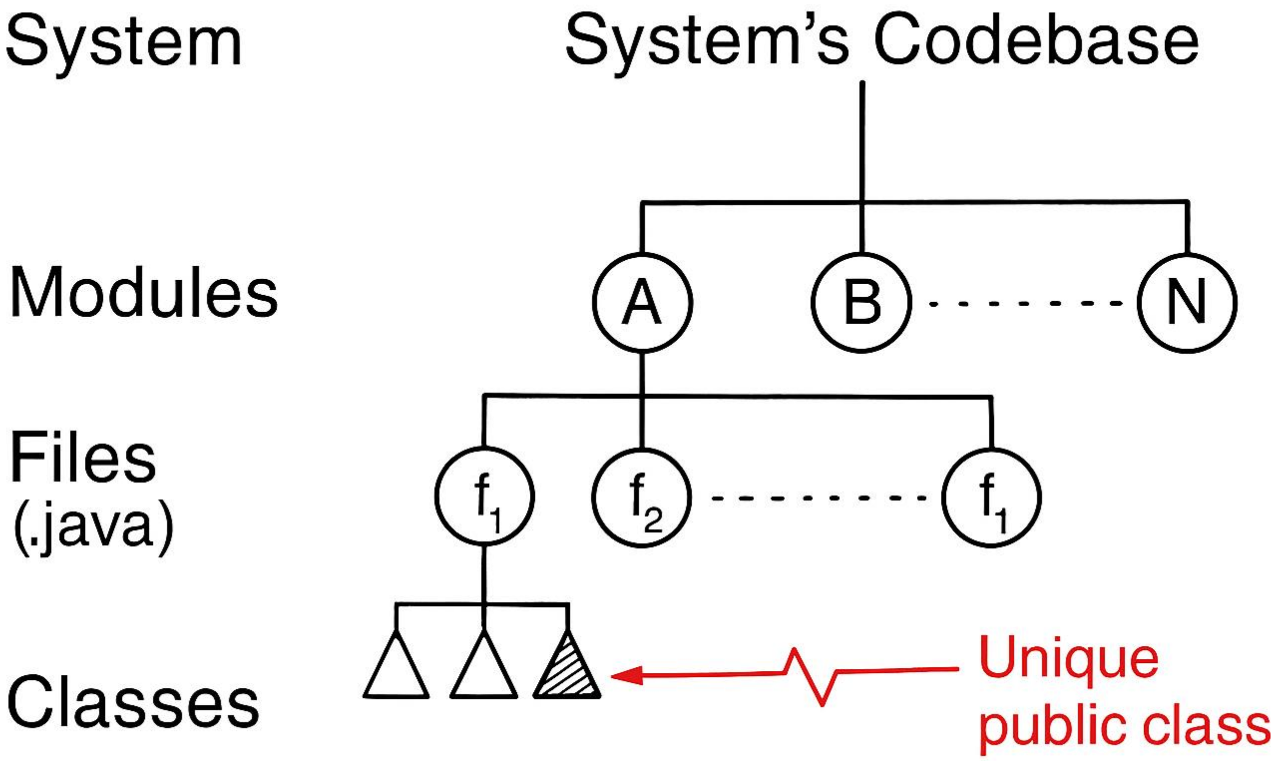

Figure 5: The archetypal hierarchical structure of a Java system's codebase used to infer system decomposability.

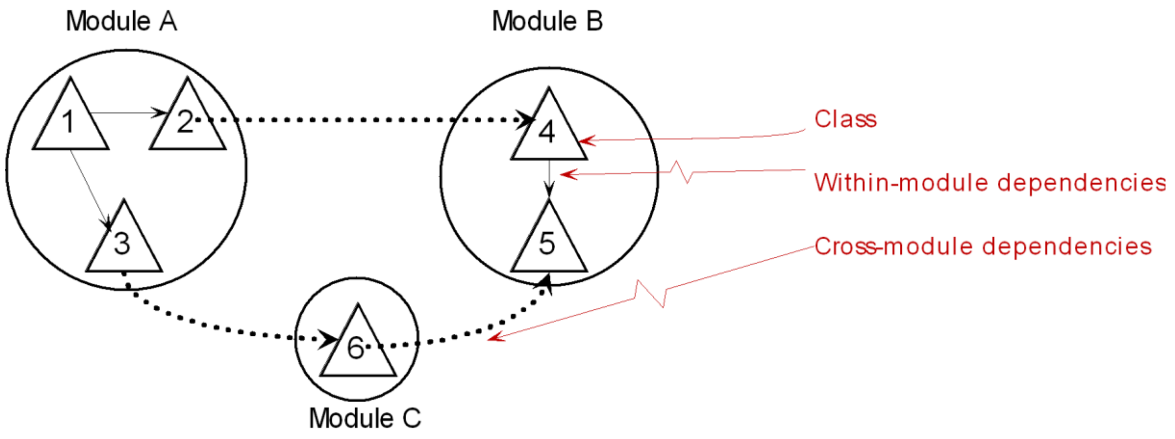

Figure 6: An illustration of the estimation of system decomposability from a system's code.

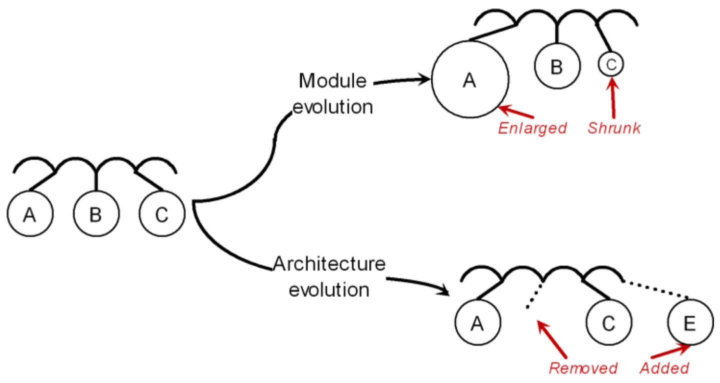

Figure 7: Architecture evolution is inferred from changes in links among a system's modules and module evolution from changes in modules' internal implementation.

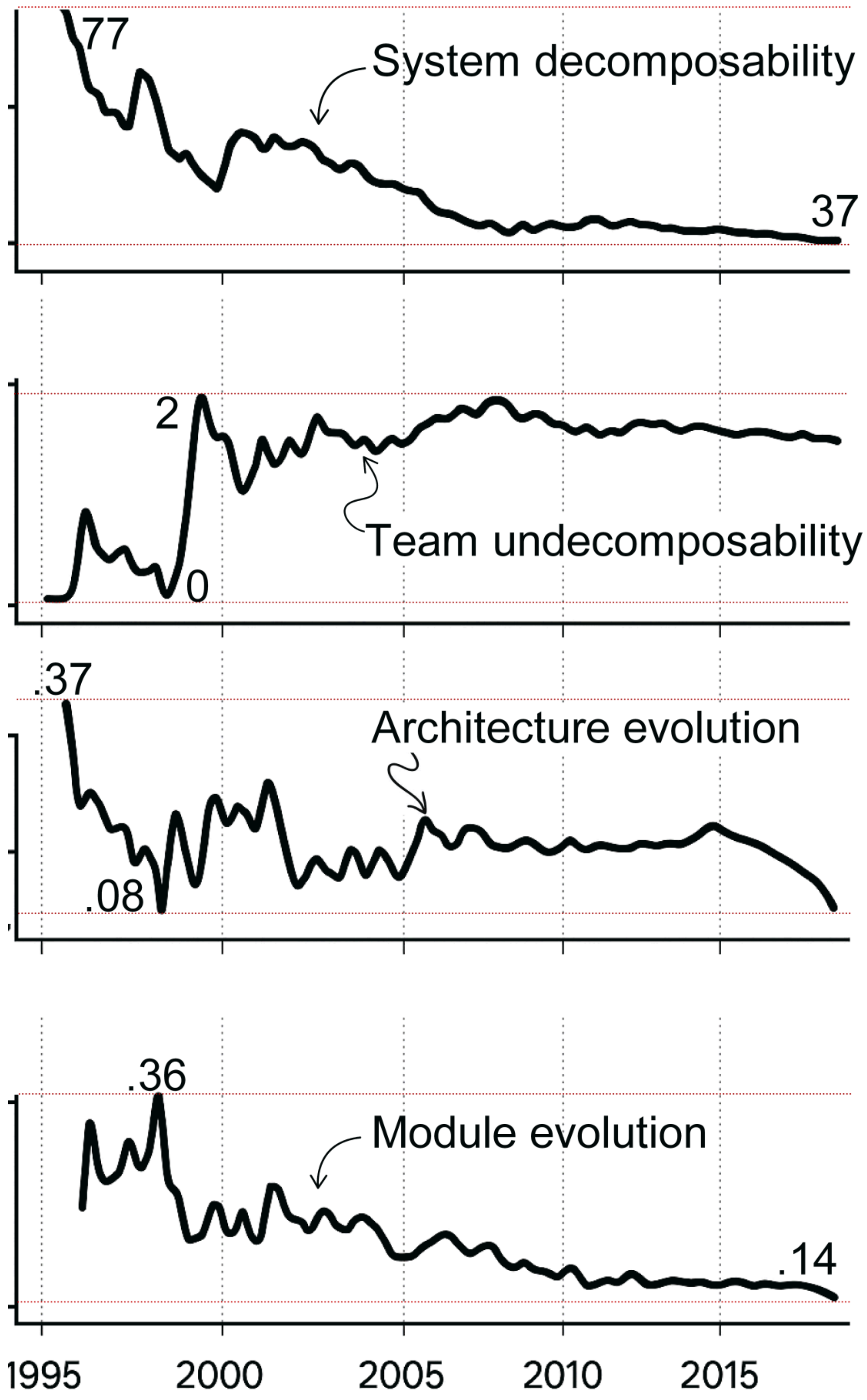

Figure 8: Sparklines showing dynamics in system and team architecture (upper half) and in architecture and module evolution (lower half) over two decades.

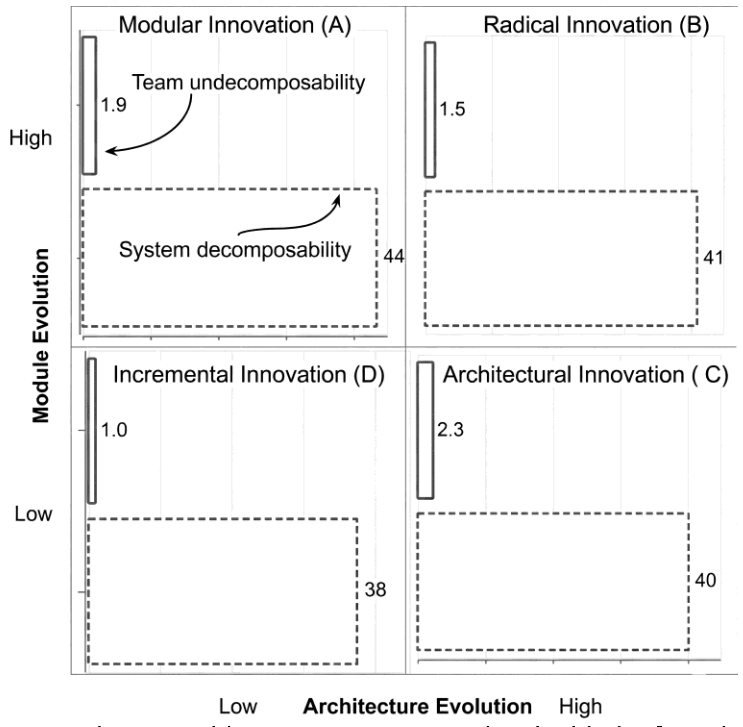

Figure 9: System architecture and team architecture patterns associated with the four classic modes of innovation.

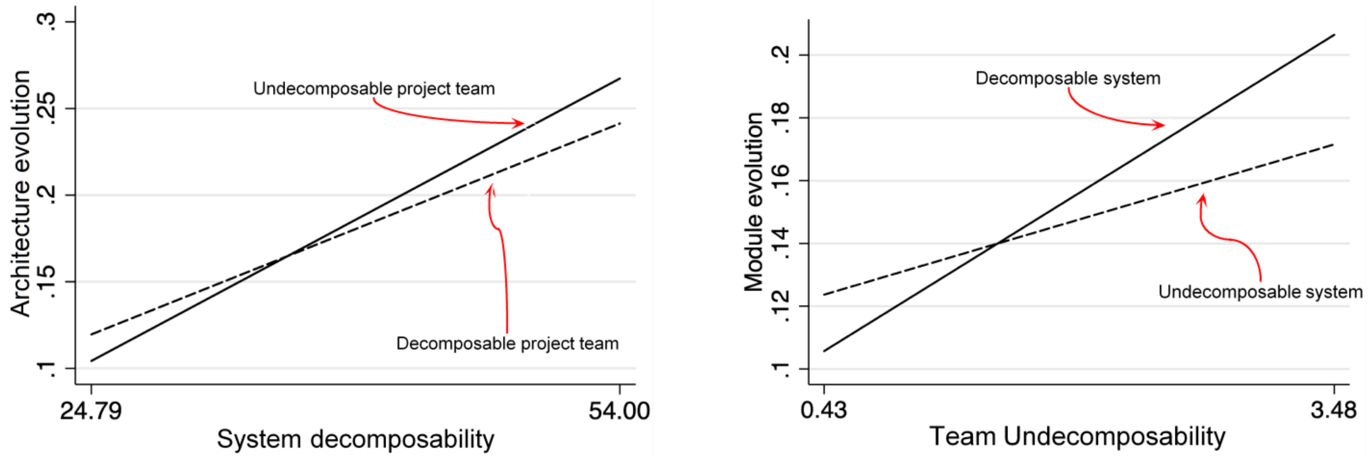

Figure 10: System—team syncing strengthens the evolutionary payoff of (left) team undecomposability on the system decomposability→architecture evolution relationship and (right) system decomposability on the team undecomposability→module evolution relationship. (High and low are $\pm 1\sigma$.)

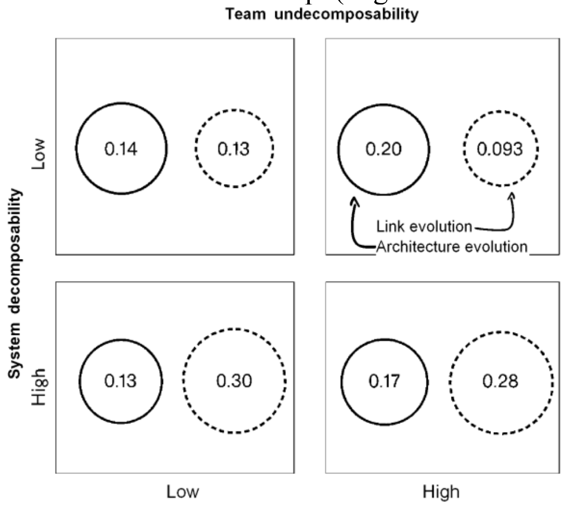

Figure 11: A system's module and architecture evolution associated with four possible combinations of system decomposability and team undecomposability (using 1/3rd percentiles).

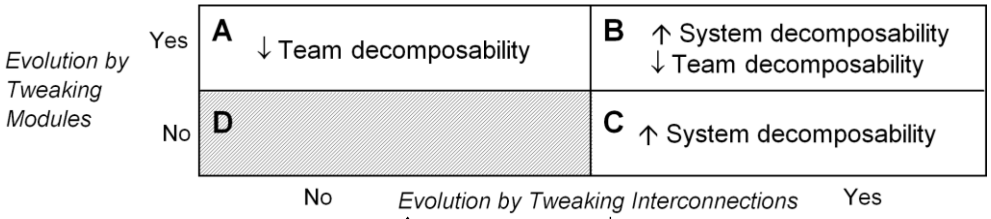
Evolution by Tweaking Modules
Yes
No
A ↓ Team decomposability
B ↑ System decomposability ↓ Team decomposability
D
C ↑ System decomposability
No
Evolution by Tweaking Interconnections
Yes

Figure 12: Implications: Parameters to increase (↑) and decrease (↓) to optimize the two mechanisms of evolution.

Table 1: Definitions and Measures for the Central Theoretical Constructs

Construct	Definition	Measure	Unit of measurement
System decomposability	Degree of a system codebase's decomposition into modules with high within-module dependencies and low cross-module dependencies	%(ΣWithin-module dependencies÷Σ(cross-module + within-module dependencies in codebase)$_t$	System-quarter
Team undecomposability	Degree team members directly collaborate through working on the same code	Log(#pairs of developers who changed the same codebase files)$_{t-1 \to t}$	System-quarter
Architecture evolution	Changes in module boundaries and interface structures	%existing codebase files deleted$_{t \to t+1}$ + %new codebase files added$_{t \to t+1}$	System-quarter
Module evolution	Changes in modules' *internal* implementation	% of codebase files that either increased or decreased in size$_{t \to t+1}$	System-quarter

Table 2: Construct Correlations and Descriptive Properties

	1	2	3	4	5	6	7	8	9	10	11
1. Architecture evolution											
2. Module evolution	-.12***										
3. System decomposability	.10***	.00									
4. Team undecomposability	.09***	.13***	-.16***								
5. System age	-.28***	-.17***	-.11***	.04***							
6. Team size	-.00	.04***	-.14***	.78***	.09***						
7. Module granularity	-.12***	.03***	.23***	-.03***	.10***	-.01					
8. Class granularity	-.13***	.07***	.06***	.01	.15***	.02**	.17***				
9. Design codification	-.07***	-.08***	.04***	.03***	.16***	.02***	.03***	.18***			
10. Workload tautness	.13***	.10***	-.09***	-.15***	.01*	-.12***	.18***	-.01	.07***		
11. Evolution velocity	.15***	.09***	-.11***	.26***	-.01*	.28***	.04***	.01*	.02***	.26***	
Mean	.22	.15	39.39	1.96	3.50	5.77	10.10	173.28	.23	82.53	789.24
Median	0.09	0.11	36.49	1.95	2.75	4.00	8.15	151.87	0.23	35.5	345
σ	.29	.16	14.60	1.52	2.91	6.65	8.26	93.64	.10	205.93	2242.67
Min	0.00	0.00	0.00	0.00	0.25	1.00	1.00	17.53	0.00	0.25	1.00
Max	1.00	1.00	99.94	8.08	19.50	144	241	1433.43	.64	12481	192476

$^*p < .05$, $^{**}p < .01$, $^{***}p < .001$

Table 3: Instrument Measures and Source of Endogeneity Addressed

Instrument	Mitigates*	Measure	Exclusion restriction justification	Reference
Team undecomposability	■ ↔	—	Past team structure affects evolution through its influence on current system architecture rather than directly determining change capacity independent of resulting system design	[31]
System decomposability$_{t-1}$	■ ↔	—	Historical system architecture influences evolution through its effect on current team structure rather than directly enabling change capacity independent of resulting team organization	[34]
System age	•	Number of years since the system's	Affects decomposability through architectural decay but influences evolution through current architectural	[57: 4]

		inception	state, not temporal passage per se	
Team size	•	Number of developers involved between t-1 and t	Determines collaboration complexity affecting team structure but influences evolution through realized team architecture, not size directly	[40]
Module granularity	•	Number of public classes ÷ number of modules in system	Shapes decomposability through structural constraints but affects evolution via architectural properties, not granularity itself	[58]
Class granularity	•	Total lines of code ÷ number of public classes in system	Influences system organization affecting decomposability but impacts evolution through realized architecture, not class structure directly	[58]
Design codification	•	Comment lines ÷ lines of code in system's codebase	Affects both through documentation quality but influences evolution via architectural clarity, not documentation volume	[24]
Workload tautness	•	Number of key classes in codebase ÷ number of developers	Shapes architectural choices under resource constraints but affects evolution through resulting architectural properties	[44]
Evolution velocity	•	Number of code commits during the quarter$_{t-1\rightarrow t}$	Reflects historical change intensity affecting current architectures but influences future evolution through architectural changes	[52]

**Source of endogeneity mitigated:* ↔ reciprocal causality; ■ historical path-dependence; • time-variant unobserved heterogeneity.

Table 4: Stage 1 - Endogeneity Assessment

Instrumental variables	**System decomposability**	**Team undecomposability**
Team undecomposability	-.05(-1.34)	.22***(9.31)
System decomposability	.67***(27.73)	-.00**(-3.39)
System age	-.27***(-7.20)	-.05***(-7.83)
Team size	-.07***(-4.41)	.14***(14.54)
Module granularity	.07***(4.51)	.00(1.11)
Class granularity	-.00(-1.56)	.00(.57)
Design codification	2.46(1.55)	-.08(-.46)
Workload tautness	-.00***(-7.05)	-.00*(-2.14)
Evolution velocity	-.00(-1.06)	.00(.87)
Constant	13.47***(9.53)	1.05***(12.75)
Model-F(R^2_w)	989(.61)	633(.50)

β(t); $^{*}p < .05$, $^{**}p < .01$, $^{***}p < .001$

Table 5: Stage 2 - System Decomposability and Team Undecomposability's Influence on Architecture Evolution and Module Evolution

	Architecture evolution$_{t+1}$		**Module evolution$_{t+1}$**	
	Main Effects	Interaction	Main Effects	Interaction
$\hat{y}_{system\_decomposability\_t}$	**.005***(6.47)**	.004***(4.54)	.000(.56)	-.001*(-2.07)
$\hat{y}_{team\_undecomposability\_t}$	.003(.18)	-.017(-.92)	**.025***(4.84)**	.001(.13)
$\hat{y}_{system\_decomposability\_t} \times \hat{y}_{team\_undecomposability\_t}$		**.000*(2.00)**		**.001***(4.39)**
System age	-.016***(-9.14)	-.016***(-9.24)	-.013***(-17.14)	-.013***(-17.09)
Team size	-.002(-1.12)	-.002(-.83)	-.001(-1.58)	-.001(-.67)
Module granularity	-.003***(-6.44)	-.003***(-6.33)	-.001(-1.71)	-.001(-1.65)
Class granularity	-.000(-1.40)	-.000(-1.41)	.000(.04)	-.000(-.03)
Design codification	-.057(-.85)	-.056(-.83)	-.252***(-9.04)	-.251***(-8.98)
Workload tautness	.000***(3.44)	.000**(3.28)	.000*(2.39)	.000*(2.27)
Evolution velocity	.000*(2.07)	.000*(2.32)	.000(1.33)	.000(1.63)
Constant	.125**(2.77)	.157**(3.10)	.210***(12.80)	.251***(13.34)
Model-F(R^2_w)	45.3(.053)	41.3(.053)	91.6(.051)	85.6(.052)

β(t); $^{*}p < .05$, $^{**}p < .01$, $^{***}p < .001$. Hypothesis tests are **bolded**.

Appendix: Sensitivity Analyses, Robustness Checks, and Causal Inference Hazards

Table A1: Missing Data Sensitivity Analyses comparing Sample before and after Observation Exclusions

Variable	**Sample after exclusions (n=23,719)**	**Sample before exclusions (n varies)**	**Difference**	**t-stat**	**p-value**	**Cohen's d**
Dependent Variables						
Module evolution	.153 (.157)	.146 (.159)	.007	-4.71	<.001	.04
Architecture evolution	.217 (.290)	.211 (.292)	.006	-2.23	.026	.02
Predictors						
Team undecomposability	1.956 (1.52)	1.768 (1.55)	.189	-13.83	<.001	.12
System decomposability	39.40 (14.6)	40.95 (16.8)	-1.55	11.03	<.001	.10
Controls						
System age (years)	3.49 (2.91)	3.26 (2.90)	.23	-8.93	<.001	.08
Team size	5.82 (6.71)	5.65 (6.58)	.17	-	-	-
Module granularity	10.10 (8.26)	10.67 (18.4)	-.57	4.56	<.001	.04
Class granularity	173.2 (93.6)	172.2 (113.7)	1.03	-1.11	.266	.01
Design codification	.233 (.104)	.232 (.106)	.001	-1.22	.222	.01
Workload tautness	82.6 (205.8)	86.0 (214.3)	-3.44	1.81	.070	.02
Evolution velocity	788.9 (2240)	784.8 (2206)	4.05	-0.20	.840	.00

Notes: Values are means (SDs). Differences calculated as included minus excluded. *t*-statistics from two-sample tests assuming unequal variances. Cohen's *d* is calculated as standardized mean difference. The included sample comprises 23,719 system-quarter observations retained after listwise deletion; Excluded observations were for incomplete git histories (282 systems), dependency parsing errors (43 systems), or insufficient commit activity (51 systems).

Table A2: Robustness Check with Control Function Estimation and Consequences of Too Much and Too Little Decomposability

	Architecture evolution	**Module evolution**
System decomposability$_{\hat{y}}$	.005***(6.34)	.000(.54)
$\eta_{\text{system decomposability}}$	-.001(-1.30)	**-.002***(-4.39)**
Team undecomposability$_{\hat{y}}$	.003(.26)	.027***(4.91)
$\eta_{\text{team undecomposability}}$	.006(.51)	**-.012*(-2.10)**
System age	-.016***(-9.25)	-.013***(-16.57)
Team size	-.002(-1.28)	-.002(-1.88)
Module granularity	-.003***(-6.56)	-.001(-1.69)
Class granularity	-.000(-1.36)	.000(.02)
Design codification	-.061(-.94)	-.252***(-9.20)
Workload tautness	.000***(3.67)	.000*(2.42)
Evolution velocity	.000*(2.24)	.000*(2.01)
Constant	.121**(2.81)	.209***(12.69)
Model-F(R^2_w)	53.8(.057)	232.9(.058)

$^{*}p < .05$, $^{**}p < .01$, $^{***}p < .001$

Table A3: Coarse-grained Evolution Masks How System and Team Architecture Differentially Influence the Two System Evolution Mechanisms

	Coarse Evolution	
$\hat{y}_{\text{system decomposability}}$	**.005***(7.69)**	.003***(3.45)
$\hat{y}_{\text{team undecomposability}}$	**.028*(2.02)**	-.016(-.78)
$\hat{y}_{\text{system decomposability}} * \hat{y}_{\text{team undecomposability}}$		**.001***(3.61)**
System age	-.029***(-15.80)	-.029***(-16.04)
Team size	-.003(-1.83)	-.002(-1.13)
Module granularity	-.004***(-6.33)	-.004***(-6.15)
Class granularity	-.000(-1.56)	-.000(-1.60)

Design codification	-.310***(-5.37)	-.307***(-5.22)
Workload tautness	.000***(4.49)	.000***(4.29)
Evolution velocity	.000(1.70)	.000*(2.00)
Constant	.335***(7.38)	.408***(7.60)
Model-F(R^2_w)	63.7(.097)	57.5(.098)

$^*p < .05$, $^{**}p < .01$, $^{***}p < .001$

Table A4: Neglecting Endogeneity Infers Absence of System—Team Syncing

System decomposability	.005***(11.10)	.004***(8.25)
Team undecomposability	.024***(9.37)	.014*(2.26)
System decomposability*Team undecomposability		**.000(1.47)**
System age	-.032***(-18.40)	-.033***(-18.32)
Team size	-.003***(-4.95)	-.003***(-5.01)
Module granularity	-.003***(-4.23)	-.003***(-4.30)
Class granularity	-.000**(-2.69)	-.000**(-2.74)
Design codification	-.336***(-5.87)	-.336***(-5.87)
Workload tautness	.000***(4.41)	.000***(4.20)
Evolution velocity	.000*(2.04)	.000*(2.00)
Constant	.382***(15.30)	.398***(14.15)
Model-F(R^2)	108.6(.123)	99.5(.123)

$^*p < .05$, $^{**}p < .01$, $^{***}p < .001$