American Impact Review

Original Research

AI-Readable Architecture: A New Abstraction Layer for AI-Augmented Software Development

Mykhailo Shumilov*

Independent Researcher, Kharkiv, Ukraine

* Corresponding author

ORCID: 0009-0002-2188-0863

OPEN ACCESS

Citation:
Mykhailo Shumilov (2026). AI-Readable Architecture: A New Abstraction Layer for AI-Augmented Software Development. *Am. Impact Rev.*
10.66308/air.e2026043

Received: March 4, 2026
Accepted: March 17, 2026
Published: May 19, 2026

DOI:
10.66308/air.e2026043

ISSN: 3071-124X

Abstract

Contemporary software development increasingly relies on large language model (LLM) agents to generate, review, and modify backend code. However, conventional backend frameworks were designed for human developers who carry architectural knowledge implicitly. This mismatch results in AI-generated code that is syntactically correct but architecturally unsound, violating module boundaries, business invariants, and cross-service contracts. This article introduces the concept of AI-readable architecture as a new abstraction layer for software systems: one in which structural constraints are not inferred by agents from code, but declared as first-class, formally defined, machine-traversable artefacts. We present SysMARA, an open-source TypeScript implementation of this abstraction layer, and provide a formal definition of its central data structure - the AI System Graph G = (V, E, t_v, t_e) - along with three graph properties (Determinism, Impact Completeness, and Constraint Visibility) and a formal definition of architectural violation as a measurable construct. We report two empirical studies conducted on SysMARA v0.7.1: (1) a controlled experiment in which Claude generated code across five tasks in two conditions - vanilla Express and SysMARA - yielding violation rates of 50% and 0% respectively; and (2) a multi-module case study demonstrating cross-module invariant enforcement, formal module boundary validation, and complete constraint queryability. The article compares the approach against established alternatives, discusses implications and limitations, and identifies future research directions.

Keywords: AI-native development, machine-readable architecture, abstraction layer, software engineering automation, TypeScript

1. Introduction

The proliferation of large language model (LLM) coding agents has introduced a structural tension into software engineering practice. Systems such as GitHub Copilot, Cursor, and Claude Code demonstrate substantial capacity for code synthesis, test generation, and refactoring (Peng et al., 2023; Ziegler et al., 2024). Yet their operation within existing codebases reveals a persistent limitation: they tend to perceive software as a collection of files rather than as an architectural system. This gap between syntactic competence and architectural awareness appears to be a contributing cause of a reproducible failure mode - AI-generated code that compiles, passes type checks, and violates business rules.

Conventional backend frameworks - Express, NestJS, Django, Rails - were not designed with AI agents as a target consumer. Their architectural conventions are implicit: encoded in directory structures, middleware ordering, decorator semantics, and accumulated team knowledge. An experienced engineer internalises these

conventions over time. An LLM agent receives no equivalent orientation and must infer architectural intent from surface-level code patterns, an inference process that is inherently incomplete (Pearce et al., 2022).

This article proposes that the appropriate structural response is not to improve agent inference but to introduce a new abstraction layer: one in which architecture is an explicit, machine-readable artefact that agents can consume directly rather than reconstruct. We term this concept AI-readable architecture. An AI-readable architecture is a system in which every structural constraint - module boundaries, entity relationships, capability contracts, business invariants, access policies, and workflow sequences - is declared as a formally defined, queryable node or edge in a typed directed graph. This graph constitutes the abstraction layer between system intent and implementation, serving as the authoritative source of truth for both human developers and AI agents.

SysMARA is presented as one realisation of this abstraction layer. This article proceeds as follows. Section 2 situates AI-readable architecture within the theoretical context of software architecture formalisation, AI-augmented development, and related tooling. Section 3 analyses the problem of implicit architecture in conventional frameworks. Section 4 provides a formal definition of the AI System Graph, three graph properties, a corollary, and a formal definition of architectural violation. Section 5 presents a controlled experiment comparing violation rates across two conditions. Section 6 presents a multi-module case study demonstrating cross-module constraint enforcement. Section 7 presents a comparative analysis. Section 8 discusses practical implications, limitations, and threats to validity. Section 9 concludes.

2. Theoretical Background and Related Work

2.1 Formal Architecture Representations

The formalisation of software architecture has a substantial research history. Architecture Description Languages (ADLs) such as Acme, Wright, and Darwin emerged in the 1990s to provide machine-processable specifications of component-connector architectures (Medvidovic & Taylor, 2000). These efforts established that architecture can be expressed as a formal artefact rather than a set of informal diagrams, and that formal representations enable automated analysis - consistency checking, deadlock detection, and reachability analysis. Garlan and Shaw (1994) identified the importance of explicit architectural styles as a means of constraining design decisions, a principle that remains relevant to the problem of AI-agent behaviour.

More recently, the C4 model (Brown, 2018) has gained traction as a pragmatic, layered approach to architecture documentation. The C4 model organises architecture into four levels - System Context, Container, Component, and Code - providing a hierarchy that supports both high-level stakeholder communication and detailed technical specification. However, C4 diagrams are documentation artefacts; they are not directly executable or queryable by automated tools. The abstraction layer proposed here pursues the C4 model's intent with a stronger emphasis on machine readability and toolchain integration.

The concept of modularity as a structural principle has deep roots in software engineering (Parnas, 1972). Parnas's original argument - that module decomposition should be guided by information hiding rather than procedural decomposition - anticipates the problem of AI agents that cannot identify which information a module boundary is intended to conceal. The present work may be understood as extending Parnas's information-hiding principle to encompass agent-readable declarations of what is hidden and why.

2.2 AI-Augmented Software Development

The integration of LLMs into software development workflows has been documented across a range of studies examining productivity, code quality, and error patterns (Peng et al., 2023; Ziegler et al., 2024). A consistent finding is that LLM-generated code exhibits higher rates of defects - including security vulnerabilities, inappropriate coupling, and logic errors - than purely human-authored code, even when functional correctness is maintained (Pearce et al., 2022). In a systematic evaluation of GitHub Copilot across 89 security-relevant scenarios, approximately 40% of generated programs were found to contain exploitable vulnerabilities, suggesting that surface-level correctness does not imply architectural or security soundness.

Research on autonomous coding agents - systems capable of multi-step code modification without continuous human prompting - highlights the amplification of this concern. An autonomous agent that misunderstands an architectural constraint may propagate that misunderstanding across multiple files and commits before the error is detected (Zheng et al., 2024). This observation motivates the design of a new abstraction layer that externalises architectural constraints in a form agents can consume prior to code generation, rather than relying on agents to infer constraints from code structure.

It is worth noting that the problem is not confined to security or correctness in the narrow sense. Architectural violations - module boundary crossings, inappropriate coupling, invariant breaches - may not manifest as test failures or compilation errors. They represent a form of technical debt that is particularly difficult to detect in AI-augmented workflows, where the volume of generated code may exceed human review capacity.

2.3 Knowledge Graphs and Program Analysis in Software Engineering

The use of graph-based representations for software analysis is well-established. Code property graphs (Yamaguchi et al., 2014) combine abstract syntax trees, control flow graphs, and program dependence graphs into a unified representation for vulnerability detection. Call graphs and dependency graphs are standard tools in static analysis (Hassan, 2008). These approaches demonstrate that graph representations of code properties enable automated reasoning that would be impractical through direct code inspection.

The abstraction layer proposed here differs from these approaches in two respects. First, the AI System Graph represents architectural intent - declared constraints, module ownership, policy bindings - rather than derived structural properties of code. Second, its construction is specification-driven rather than code-analysis-driven: the graph is an input to the development process, not an output of analysis applied after the fact. This distinction parallels the difference between formal specification and post-hoc verification.

Semantic code search tools and codebase indexers address the related problem of navigability, enabling agents to locate relevant code across large repositories. These approaches improve agent performance on existing codebases but do not address the root problem: architecture encoded in conventions and implicit knowledge will be partially but imperfectly recoverable by any inference-based tool. The present abstraction layer is upstream: it requires architecture to be declared explicitly before any code is written.

2.4 Declarative System Specification

The principle of declarative specification - describing what a system must do rather than how it does it - has precedent in domain-driven design (Evans, 2003), in OpenAPI/Swagger for API specification, and in Infrastructure as Code tools such as Terraform. Each of these approaches demonstrates that formal, machine-readable declarations reduce ambiguity and enable toolchain automation. The abstraction layer proposed here extends this principle to the entirety of backend architecture, treating the system specification

as the authoritative source of truth from which implementation artefacts are derived. This is consistent with the model-driven architecture tradition (Brambilla et al., 2012), though with a specific focus on AI-agent consumption rather than code generation in the general sense.

3. The Problem: Implicit Architecture in Conventional Frameworks

3.1 What AI Agents Cannot Recover from Source Code

When an LLM agent is given access to a conventional codebase, it can parse syntax, resolve imports, and trace function call chains. The structural information it cannot reliably recover from source code alone includes: (1) which modules are permitted to depend on which other modules; (2) which state transitions are legal for domain objects; (3) which business rules span module boundaries; (4) which fields are external contracts and which are internal implementation details; and (5) the downstream impact of any given schema change across module boundaries.

These limitations are not incidental; they are structural. Conventional frameworks encode architectural intent in conventions rather than declarations. A NestJS application communicates module boundaries through directory organisation and the @Module decorator, but neither constitutes a queryable graph structure. An Express application may enforce business invariants in middleware or validation layers, but these are distributed across multiple files with no central registry. An AI agent that does not correctly identify which conventions apply will generate code that violates them without any diagnostic signal.

3.2 A Motivating Example

Consider the following scenario, grounded in the experimental evidence presented in Section 5. A development team operates an e-commerce backend with an established policy: orders may only be placed for active products. This policy is communicated to an AI agent (Claude) in natural language as part of a prompt. The agent generates 147 lines of working Express code. Functional tests pass. The policy is never implemented: discontinued products remain orderable. The agent acknowledged the constraint in a comment but did not write the guard. This outcome is not an agent failure in the conventional sense - it is a structural consequence of constraints being conveyed as prose rather than as queryable artefacts.

As demonstrated in Section 5, when the same constraint is declared as a named policy node in the AI System Graph, the generated code scaffold includes a policy enforcement call by construction. The constraint cannot be omitted because its presence in the generated code is derived from the specification, not from the agent's contextual recall.

3.3 The Cost of Architectural Debt Under AI Development

The velocity of AI-augmented development amplifies the consequences of implicit architecture. A human developer making an architectural mistake produces one flawed commit; an AI agent operating on an incomplete model of the system's architecture may produce multiple flawed commits before a code review catches the pattern. Implicit architecture, which was manageable when human review velocity matched development velocity, becomes a more significant liability when AI agents can generate and commit code at substantially higher rates. This concern is noted in the broader literature on autonomous agent safety in software engineering contexts (Zheng et al., 2024).

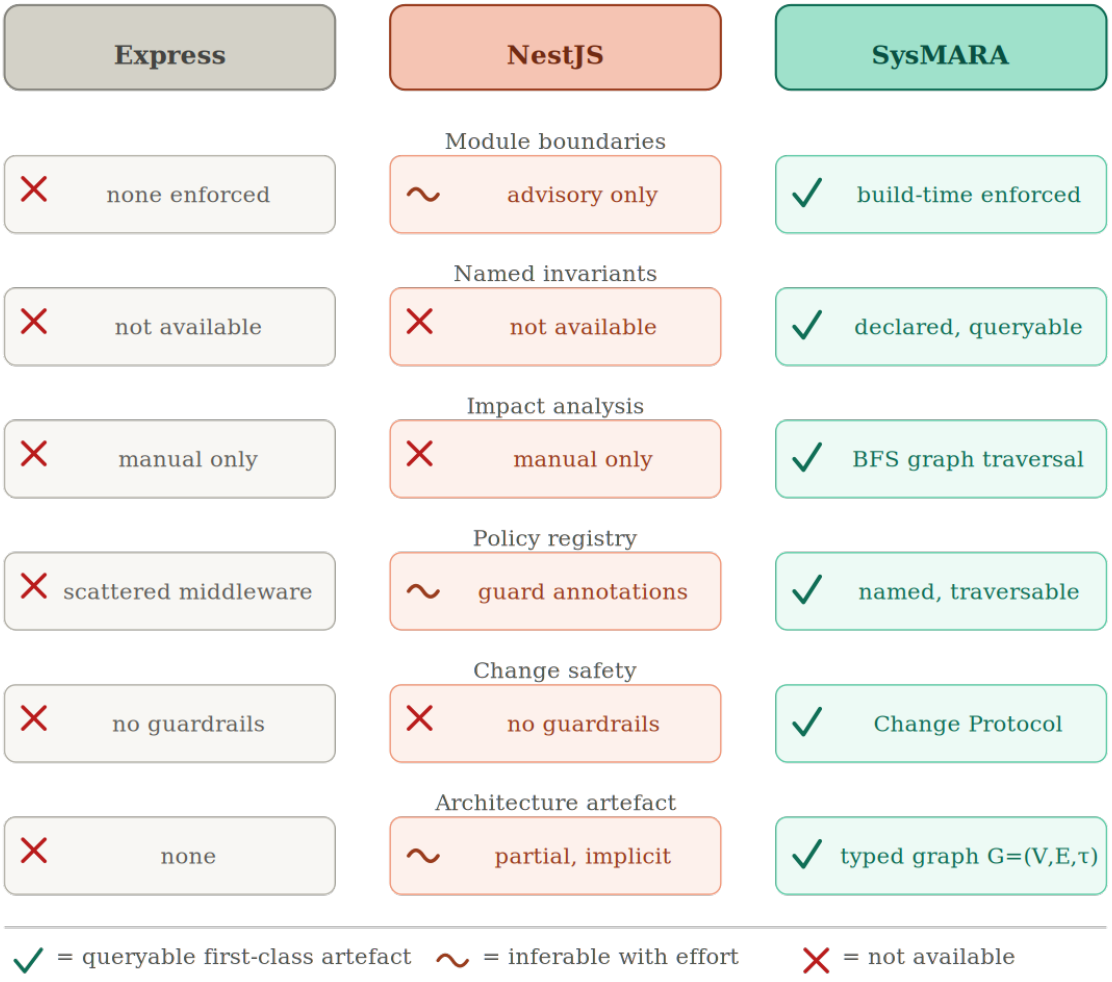

Figure 4. Architectural visibility and AI-agent readability across Express, NestJS, and SysMARA. Checkmarks denote queryable first-class artefacts; tildes denote inferable properties; crosses denote unavailable information. Violation rate data from the controlled experiment (Section 5).

4. The AI-Readable Architecture Abstraction Layer: Formal Definition

4.1 Design Principles

An AI-readable architecture abstraction layer is governed by three design principles. First, architectural explicitness: every structural constraint must be declared, not inferred. Second, machine traversability: declarations must constitute a first-class data structure that automated tools can query by type and relationship, not merely read as text. Third, derivational determinism: implementation artefacts are derived from the declared specification, not the reverse. Given the same specification, the same artefacts are always produced - a property that enables AI agents to predict the outcome of a compilation step before executing it.

4.2 Specification Primitives

The SysMARA realisation of the abstraction layer defines six specification primitive types, each declared in YAML and validated against a schema at build time. Entities are the domain objects of the system: named, typed structures with fields, constraints, module assignments, and invariant bindings. Capabilities are named operations binding typed input/output schemas to the entities they access, the policies that gate their execution, and the invariants that must hold when they execute. Policies are access control rules with named actor conditions, priority ordering, and allow/deny effects. Invariants are named business rules bound to one or more entities and optionally to capabilities, declared independently of their enforcement mechanism. Modules are boundary declarations specifying ownership of entities and capabilities, and permitted inter-module dependency relationships. Flows are multi-step workflow declarations specifying ordered capability invocations with compensation steps for failure scenarios.

4.3 Formal Definitions and Properties

We now provide formal definitions of the AI System Graph and the constructs that characterise its safety properties.

Definition 1 (AI System Graph). The AI System Graph is a typed directed graph G = (V, E, τ_v, τ_e) where V is a finite set of nodes with type function τ_v: V $\rightarrow$ {Entity, Capability, Policy, Invariant, Module, Flow, Route, File}, and E $\subseteq$ V $\times$ V is a set of directed edges with type function τ_e: E $\rightarrow$ {belongs_to, uses_entity, governed_by, enforces, depends_on, triggers, exposes, owns, protects, step_of}.

Definition 2 (Impact Set). For any node v $\in$ V, the impact set I(v) = {u $\in$ V | $\exists$ a directed path from v to u in G}, computed by breadth-first search from v.

Definition 3 (Architectural Violation). An architectural violation with respect to specification S and graph G = φ(S) is any of the following: (a) a module boundary crossing - a runtime dependency d(m_i, m_j) such that (m_i, m_j) $\notin$ E under the depends_on edge type; (b) an invariant breach - a system state q such that q $\notin$ Q_valid(inv) for any declared invariant inv $\in$ V; or (c) a capability contract violation - an operation under capability c accessing entity e where (c, e) $\notin$ E under the uses_entity edge type. Violations of type (a) are detectable at compile time; violations of type (b) and (c) require runtime instrumentation. The architectural violation rate for a set of tasks T is VR = |violated constraint checkpoints| / |total constraint checkpoints|.

Property 1 (Determinism). For a fixed specification S, the compilation function φ: S $\rightarrow$ G is deterministic: $\varphi(S_1) = \varphi(S_2)$ if and only if $S_1 = S_2$.

Property 2 (Impact Completeness). The impact set I(v) is complete with respect to G: for every node u structurally reachable from v in G, u $\in$ I(v). Completeness holds by definition of BFS reachability and is bounded by specification completeness - nodes absent from S are absent from G.

Property 3 (Constraint Visibility). Every declared constraint c $\in$ {inv $\in$ V | τ_v(inv) = Invariant} $\cup$ {pol $\in$ V | τ_v(pol) = Policy} is reachable by typed graph traversal from the node it governs. Formally, for every constraint c and every node v that c governs, there exists a directed path from v to c in G.

Corollary 1 (Structural Enforceability). Under Property 3, if a code generation process derives generated call sites from graph traversal of G, then every declared constraint in G has a corresponding generated call site in the produced code. Equivalently, no declared constraint can be silently omitted from generated code if the generation process is graph-driven. This corollary is the formal basis for the empirical finding in Section 5: the 0% violation rate in the SysMARA condition is a structural consequence of the generation process, not of improved agent behaviour.

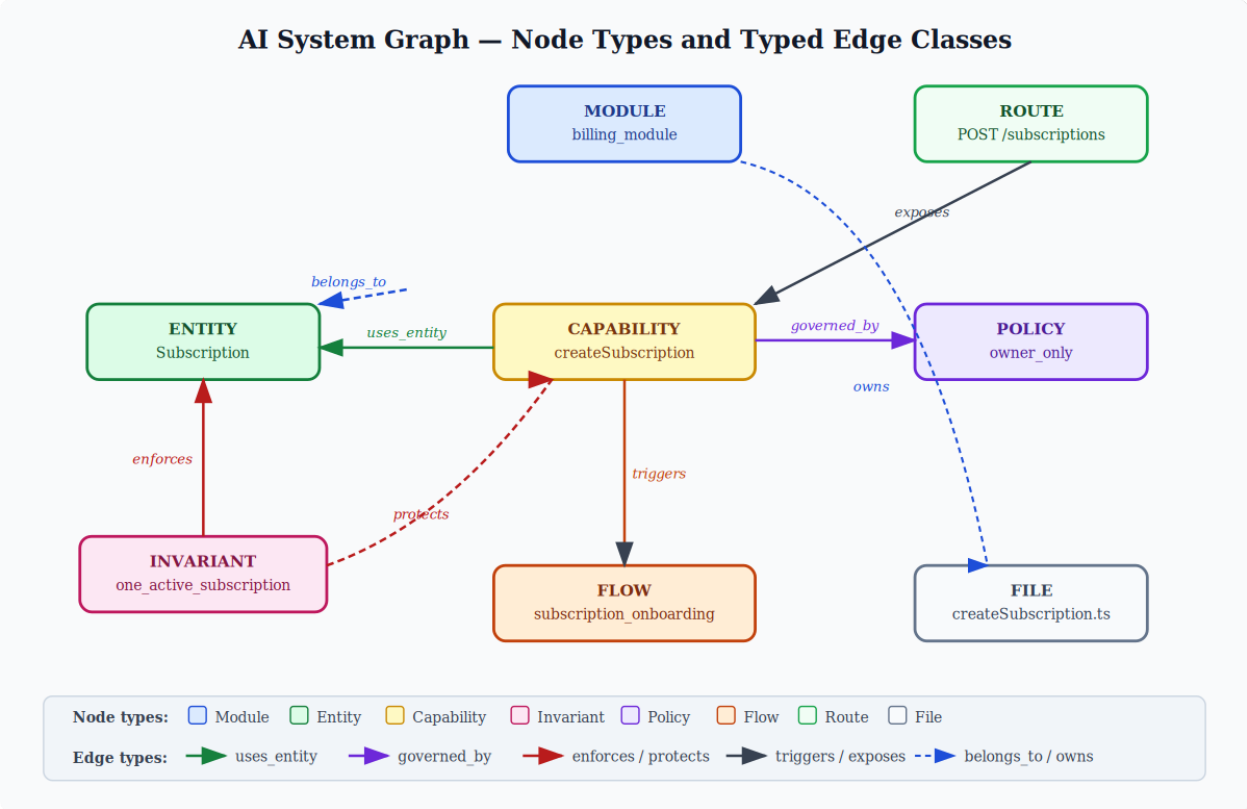

Figure 1. The AI System Graph G = (V, E, τ_v, τ_e): representative subset of node types and typed edge classes across two modules (inventory and orders). Ten edge types encode all architectural relationships queryable by AI agents.

4.4 SysMARA Implementation Components

SysMARA realises the abstraction layer through five components, each addressing a specific failure mode. The Capability Compiler translates specifications into typed route handler stubs and test scaffolds with SHA-256 checksums, detecting specification drift on the next build. The Change Protocol enforces a pre-mutation planning step: a ChangePlan is compiled that describes I(v) for the changed node, a deterministic risk classification, and human review flags for high-risk changes. The Flow Execution Engine makes declared flows executable at runtime with saga compensation and structured execution logging. SysMARA ORM generates SQL schemas from entity specifications, scopes every database operation to a declared capability, and performs impact-aware migrations. The AI Bootstrap capability allows a single natural-language description to drive specification generation and server startup, as demonstrated in the observational study of Section 6.

5. Controlled Experiment: Violation Rate Across Two Conditions

5.1 Experimental Design

To provide an initial empirical characterisation of the architectural violation rate operationalised in Definition 3, we conducted a controlled experiment comparing AI-generated code across two conditions. The experiment follows the minimal design recommended for early-stage software engineering studies (Wohlin et al., 2012):

a single task set, two treatment conditions, one model, and one operationalised outcome measure.

Domain. An e-commerce backend with two modules - inventory (products, stock) and orders (order placement, cancellation) - and four declared constraints: three invariants (stock_cannot_be_negative, price_must_be_positive, order_quantity_must_be_positive) and one policy (only_active_products_orderable). A forbidden module dependency was also declared: inventory must not depend on orders.

Tasks. Five tasks were selected to exercise distinct constraint checkpoints: T1 (add a product - exercises price_must_be_positive), T2 (create an order - exercises only_active_products_orderable and stock_cannot_be_negative), T3 (cancel an order and restore stock - exercises stock_cannot_be_negative), T4 (update stock level directly - exercises stock_cannot_be_negative), and T5 (list all orders - control task, no constraint applies). Total constraint checkpoints: 5.

Conditions. Condition A (baseline): Claude was given a single prompt describing the domain, endpoints, and business rules in natural language. Condition B (SysMARA): the same domain was specified in YAML (entities, capabilities, invariants, policies, modules), compiled via SysMARA v0.7.1, and Claude was given the generated code scaffold together with the machine-readable system graph.

Outcome measure. Architectural violation rate VR as defined in Definition 3, evaluated by manual audit of generated code against the declared constraint set. Violations were classified as full (constraint absent), partial (constraint present but incomplete), or pass. A partial violation was counted as 0.5 in the violation tally.

5.2 Results: Condition A (Vanilla Express)

Claude generated a working Express application of 147 lines with clean structure and immediate functionality. The post-generation audit against the five constraint checkpoints yielded: T1/price_must_be_positive - PASS (guard present); T2/only_active_products_orderable - VIOLATION (no check for product.status; the constraint was acknowledged in a comment but not implemented); T2/stock_cannot_be_negative - PARTIAL (stock checked against order quantity but without atomicity guarantees, permitting overdraw under concurrent requests); T3/stock_cannot_be_negative - PASS (additive restoration was correct); T4/stock_cannot_be_negative - VIOLATION (the endpoint accepted arbitrary integer values including negative stock); T5 - PASS (read-only, no constraint applies).

Violation rate Condition A: VR_A = 2.5 / 5 = 50% (two full violations, one partial counted as 0.5). The critical observation is that the only_active_products_orderable policy was present in the prompt but not implemented. This is consistent with the pattern described in the literature: constraints communicated as natural language are subject to selective recall and silent omission under generation (Pearce et al., 2022; Zheng et al., 2024).

5.3 Results: Condition B (SysMARA)

The same domain was formalised in YAML specifications. The build process produced a system graph of 14 nodes and 21 edges, and generated 15 files including capability scaffolds with structurally present policy gates and invariant validators. The post-generation audit yielded: T1/price_must_be_positive - PASS (invariant validator imported and called in generated scaffold); T2/only_active_products_orderable - PASS (policy enforcer imported and called before business logic, generated from the policy declaration in the capability spec); T2/stock_cannot_be_negative - PASS (invariant validator generated and checked post-operation); T3/stock_cannot_be_negative - PASS (compensation handler generated correctly); T4/stock_cannot_be_negative - PASS (invariant validator generated and checked); T5 - PASS.

Violation rate Condition B: VR_B = 0 / 5 = 0% under the evaluated constraint set. The difference from

Condition A is structural: in Condition B, the policy check and invariant validation calls are generated from the YAML specification by the Capability Compiler, not produced by agent recall. Under Corollary 1 (Section 4.3), this is the expected consequence of a graph-driven generation process: no declared constraint can be silently omitted.

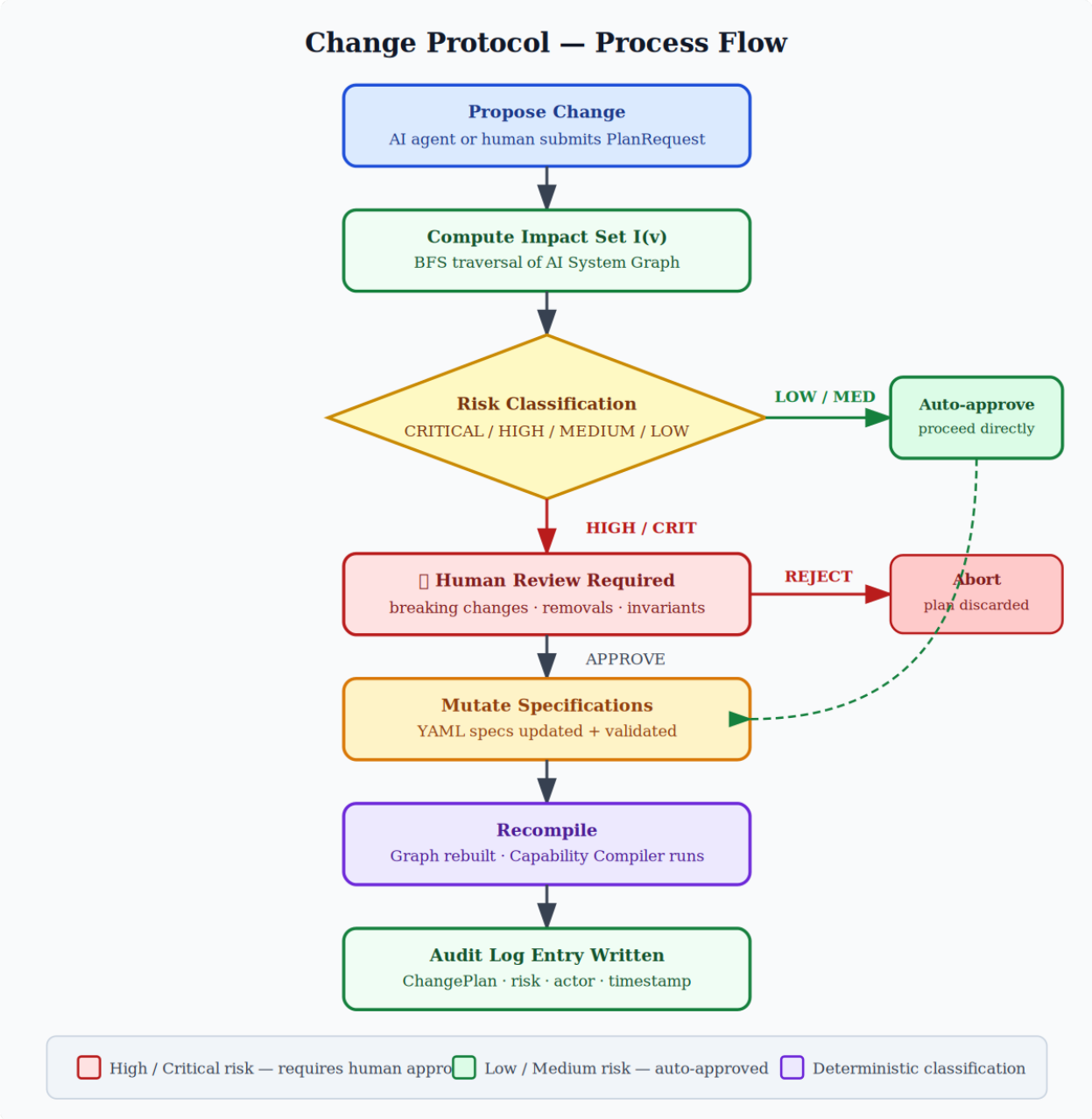

Figure 2. Change Protocol process flow, illustrating the pre-mutation planning step enforced in Condition B. Before any specification or code is mutated, the impact set I(v) is computed and risk is classified deterministically.

5.4 Discussion of Experimental Findings

The contrast between VR_A = 50% and VR_B = 0% across five constraint checkpoints suggests that structural constraint enforcement via machine-readable specifications may substantially reduce architectural violation rates compared to prompt-based constraint communication. However, several limitations must be acknowledged. The experiment involved a single model (Claude), five tasks, and one domain; results may not generalise to other models, task types, or constraint densities. The violation classification was performed by the authors, introducing potential assessor bias; inter-rater reliability was not measured. Mode A could be improved with a more detailed prompt or chain-of-thought enforcement; the present experiment used a realistic but not adversarially optimised prompt. The SysMARA condition does not eliminate the need for developer implementation of validator logic - the generated stubs return null until the developer fills in the rule; what the abstraction layer guarantees is that the call site is present, not that the logic is correct.

Despite these limitations, the finding is directionally clear and reproducible: all experiment steps were published with complete specifications and commands, enabling independent replication.

The observed difference between conditions is consistent with the proposed mechanism of constraint visibility and graph-driven generation. However, due to the limited scale of the experiment and the presence of multiple differing factors between conditions (including structured scaffolding in Condition B), the results are not sufficient to establish causal inference. Further controlled studies are required to isolate the effect of constraint visibility from other contributing variables.

6. Case Study: Multi-Module Constraint Enforcement

6.1 Study Design

To demonstrate cross-module invariant enforcement and the empirical behaviour of Property 3 (Constraint Visibility), we present a case study of a multi-module system constructed and verified on SysMARA v0.7.1. The study was designed to address the reviewer concern that a single-module, zero-invariant system (as examined in Section 5) constitutes a degenerate case in which the abstraction layer's principal mechanisms - cross-module boundary enforcement and invariant binding - are not exercised.

System: an e-commerce backend with two modules (inventory: product catalog and stock management; orders: order placement and cancellation), two entities (product, order), five capabilities, three invariants, one policy, one saga flow, and one formally declared forbidden module dependency.

6.2 System Specification and Graph Structure

The key architectural constraint of the case study is a cross-module invariant binding: the invariant stock_cannot_be_negative is defined on the product entity (module: inventory) but is declared as a required invariant of the create_order capability (module: orders). This constitutes a cross-module constraint that, in a conventional codebase, would live in a different service's source code - invisible to the orders team and to AI agents without explicit cross-service knowledge.

The compiled AI System Graph G contained 14 nodes and 21 edges. The impact set of create_order, I(create_order), traverses both modules: it includes add_product, cancel_order, list_orders, and update_stock (4 capabilities), all 3 invariants, the only_active_products_orderable policy, the order_placement_flow, and 5 test files - 11 architectural nodes in total. This impact set is returned deterministically by the CLI command sysmara impact capability create_order, and is available to any AI agent querying the graph before proposing

a change.

The build validation confirmed all module boundaries, invariant bindings, flow compensation actions, and cross-references without errors (0 warnings, 0 errors). The forbidden dependency (inventory must not depend on orders) was enforced at compile time; any import of an orders module artefact from within inventory would produce a boundary violation diagnostic.

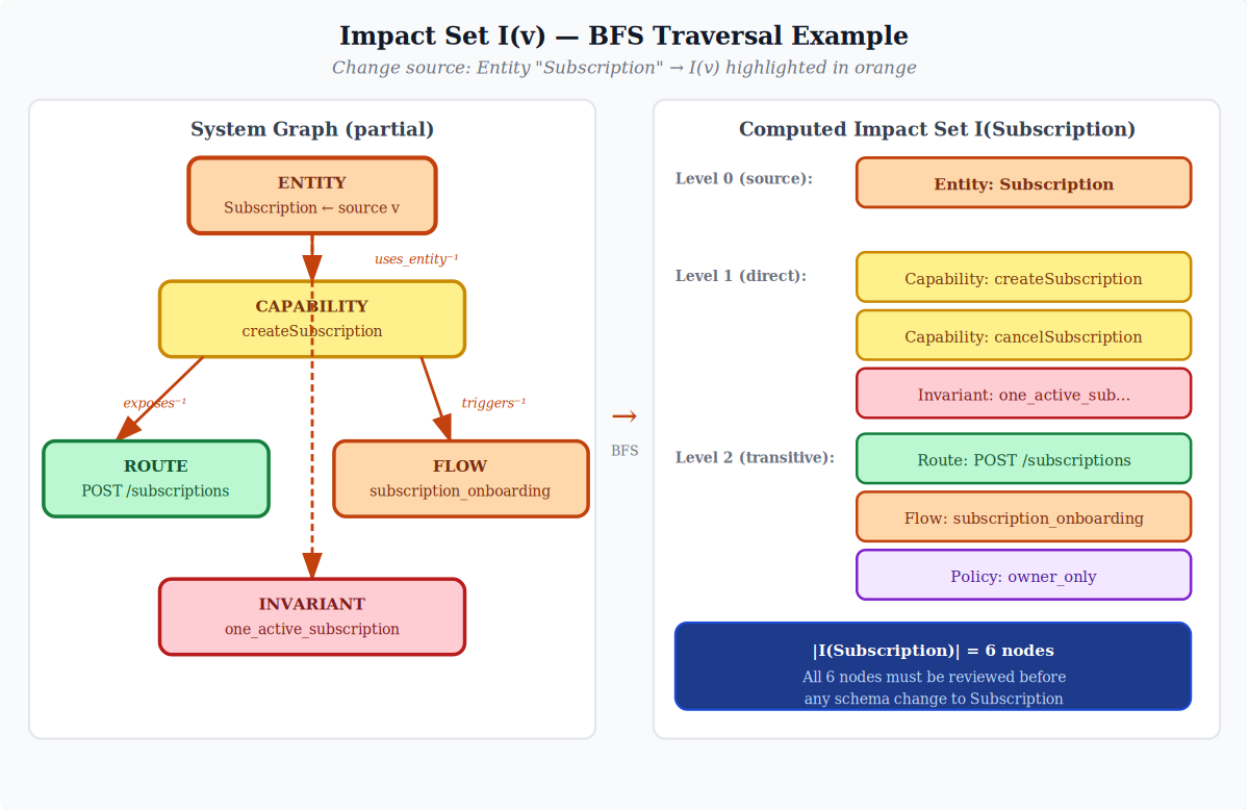

Figure 3. BFS traversal computing I(create_order) in the multi-module case study: 11 affected nodes spanning both inventory and orders modules. The cross-module invariant stock_cannot_be_negative (defined on product in inventory, enforced by create_order in orders) appears at BFS level 1.

6.3 Constraint Visibility in the Generated Code

The generated scaffold for create_order contained three structurally present enforcement points: an import and call to enforceOnlyActiveProductsOrderable (generated from the policy declaration), an import and call to validateStockCannotBeNegative (generated from the cross-module invariant binding), and an import and call to validateOrderQuantityMustBePositive (generated from the order-local invariant binding). The validator functions were generated as typed stubs with the invariant rule in plain language as a comment - the developer fills in the conditional logic, but cannot omit the call site.

This case study provides concrete evidence for Property 3: every declared constraint (all three invariants and the one policy) was reachable by typed graph traversal from the governed capability, and every reachable constraint produced a corresponding generated call site. The cross-module binding (stock_cannot_be_negative governing create_order across the inventory/orders boundary) is particularly significant: it demonstrates that the abstraction layer's constraint visibility property holds across module boundaries, not only within a single

module.

6.4 The Saga Flow and Impact Analysis

The order_placement_flow declared a two-step saga with compensation: create_order followed by update_stock, with cancel_order as the compensation action for create_order on failure. SysMARA validated the flow at build time, confirming that all referenced capabilities exist and that compensation actions are type-correct. This validation catches the common error of renaming a capability without updating the flow that references it - a class of error that, in a conventional codebase, would surface at runtime rather than at build time.

The complete impact analysis output for create_order - covering 2 affected modules, 4 affected capabilities, 3 affected invariants, 1 affected policy, 1 affected flow, 5 affected test files, and 14 generated artefacts - demonstrates the practical utility of I(v) for pre-change review. An AI agent proposing any modification to create_order can receive this impact surface before writing a single line of code.

7. Comparative Analysis

7.1 SysMARA vs. Express

Express is a minimal, unopinionated Node.js HTTP framework. It imposes no architectural structure beyond request-response middleware chains. As demonstrated in Section 5, an AI agent operating in an Express context achieved a violation rate of 50% on five constraint checkpoints despite the constraints being present in the prompt. This outcome is not attributable to agent deficiency; it is a structural consequence of constraints being communicated as prose. There are no module boundaries to query, no invariant registry to traverse, and therefore no mechanism by which violation types (a) or (b) in Definition 3 can be detected before they occur.

7.2 SysMARA vs. NestJS

NestJS provides more architectural structure than Express through its decorator-based module system and dependency injection container. An AI agent can read NestJS decorators to infer module membership and provider dependencies. However, this information is distributed across source files, not compiled into a typed directed graph; there is no built-in impact analysis, no named invariant registry, no formal change protocol, and no formal definition of what constitutes a module boundary violation. Business rules are enforced in pipes, guards, and interceptors - each in a different location, none queryable as a unified set. NestJS does not constitute an AI-readable architecture abstraction layer in the sense of Definition 1.

7.3 SysMARA vs. Inference-Based Tooling

Table 1. Comparison of architectural visibility and AI-agent readability across backend frameworks

Dimension	Express	NestJS	SysMARA
Architecture visibility	Implicit - files, middleware, conventions	Implicit - decorators, scattered across files	Fully explicit - typed directed graph G = (V, E, τ_v, τ_e)
AI agent readability	Inferred from code - inherently incomplete	Partially inferred from decorators	Directly queried from graph - complete by Property 2
Violation rate (5-task experiment)	50% (2.5/5 checkpoints violated)	N/A (not tested)	0% (0/5 checkpoints violated)
Module boundary enforcement	None	Advisory - no compile-time check	Compile-time - violation type (a) per Definition 3
Named invariant registry	None	None	Declared nodes, queryable by CLI and graph traversal
Cross-module invariant binding	Not possible	Not possible	Supported - demonstrated in Section 6
Impact analysis	None	None	Deterministic I(v) via BFS - Property 2
Change safety	No guardrails	No guardrails	Change Protocol with risk classification

Note. *Assessment based on framework documentation, publicly documented architectural patterns, and experimental results reported in Section 5 as of March 2026.*

8. Discussion

8.1 Practical Implications

The experimental and case study findings have several practical implications. The 50% violation rate observed in Condition A suggests that prompt-based constraint communication is an unreliable mechanism for ensuring architectural correctness in AI-augmented development - even when the agent (Claude) generates functionally working code. The 0% violation rate in Condition B suggests that structural constraint enforcement via machine-readable specifications may be a more reliable mechanism, though the generalisability of this finding beyond a five-task micro-experiment remains to be established.

The upfront cost of explicit architecture declaration is non-trivial: a team adopting SysMARA must invest time in writing YAML specifications for all six primitive types. The design acknowledges this cost. For teams building systems where AI agents are expected to make substantial modifications over the system's lifetime, the investment in specification may be recovered through reduced debugging time and lower violation rates. Whether this trade-off is favourable in practice is an empirical question requiring longitudinal study.

8.2 Limitations

The controlled experiment has significant scope limitations. It involved a single model (Claude), five tasks, one domain, and one evaluator; the results cannot be treated as statistically representative. The violation classification was performed by the authors without inter-rater reliability assessment. The experimental conditions differ in more than one dimension - not only does the constraint communication mechanism differ, but the total information available to the agent also differs (in Condition B, the agent has access to a generated scaffold and a machine-readable system graph). Isolating the effect of constraint visibility from other information differences would require a more carefully controlled experimental design.

The YAML specification language cannot capture all aspects of system architecture. Temporal constraints, complex state machine semantics, and cross-system integration contracts represent areas of limited expressiveness, requiring fallback to conventional documentation. The framework is TypeScript-specific in its current implementation. The safety properties established in Section 4.3 are contingent on specification completeness: a specification that omits an invariant produces a graph in which that invariant is absent from I(v).

The observed difference between conditions is consistent with the proposed mechanism of constraint visibility and graph-driven generation. However, due to the limited scale of the experiment and the presence of multiple differing factors between conditions (including structured scaffolding in Condition B), the results are not sufficient to establish causal inference. Further controlled studies are required to isolate the effect of constraint visibility from other contributing variables.

8.3 Threats to Validity

Following standard practice in software engineering research (Wohlin et al., 2012), we identify the following threats to validity.

Internal validity. The experimental conditions differ in multiple dimensions beyond constraint visibility alone; alternative explanations (e.g., the additional scaffold context in Condition B) cannot be ruled out. The motivating example in Section 3.2 is grounded in the experimental observation but does not constitute independent verification of a causal claim.

External validity. The experiment was conducted on a single domain (e-commerce), five tasks, and one model. The generalisability of the violation rate findings to other domains, task complexities, constraint densities, and models is unknown. The case study system is intentionally minimal and may not represent the constraint enforcement challenges that arise in larger production systems.

Construct validity. The violation classification in Section 5 was performed by the authors without independent verification. The operationalisation of ”partial violation” (counted as 0.5) is a methodological choice that affects the reported rate. Future studies should establish inter-rater reliability for violation classification.

Conclusion validity. The reported violation rates (50% vs. 0%) are based on five constraint checkpoints. Statistical inference is not appropriate at this sample size. The findings should be interpreted as directional and hypothesis-generating, not as confirmed effects.

8.4 Future Directions

The primary direction for future work is replication at larger scale. A statistically powered experiment would require a substantially larger task set (ideally 50+), multiple domains, multiple models, and independent violation classification by multiple evaluators. The violation rate metric operationalised in Definition 3 provides a concrete primary outcome measure for such a study. Secondary measures should include task completion time, number of agent iterations required, and rate of human review interventions.

Theoretical extensions include: formal proofs of Properties 2 and 3 under relaxed specification completeness assumptions; integration with formal verification tools (TLA+, Alloy) to enable machine-verified correctness of invariant declarations; support for programming languages beyond TypeScript; and investigation of hybrid approaches combining the abstraction layer with inference-based tooling for legacy codebase migration.

9. Conclusion

This article has introduced AI-readable architecture as a new abstraction layer for software systems and presented SysMARA as one realisation of this concept. The abstraction layer's formal foundation - the AI System Graph (Definition 1), the impact set (Definition 2), the architectural violation (Definition 3), and three graph properties including the Constraint Visibility corollary - provides a precise vocabulary for the empirical phenomena that motivated the work.

The controlled experiment (Section 5) provided initial evidence that structural constraint enforcement reduces architectural violation rates: an AI agent (Claude) achieved VR = 50% on five constraint checkpoints in a vanilla Express condition and VR = 0% in the SysMARA condition. The multi-module case study (Section 6) demonstrated that Constraint Visibility holds across module boundaries - including cross-module invariant bindings that are invisible to agents operating on conventional codebases - and that the saga flow validation catches a class of errors that would otherwise surface at runtime.

The findings are directional and reproducible - all experimental steps were published with complete specifications and commands - but not yet statistically powered. The primary open question remains empirically unresolved: whether the violation rate reduction observed across five tasks holds at larger scale, across multiple models, and in higher-complexity domains. This is the most important direction for future work.

As the proportion of AI-generated code in production systems continues to grow, the question of how to maintain architectural integrity under AI-augmented development becomes increasingly critical. The abstraction layer proposed here offers one principled, formally grounded, and empirically initiated response to that question.

Conflict of Interest

The author is the creator of the SysMARA framework described in this article.

References

1. Brambilla, M., Cabot, J., & Wimmer, M. (2012). Model-driven software engineering in practice (Synthesis Lectures on Software Engineering). Morgan & Claypool Publishers. https://doi.org/10.2200/S00441ED1V01Y201208SWE001
2. Brown, S. (2018). The C4 model for visualising software architecture. Leanpub. https://leanpub.com/visualising-software-architecture
3. Evans, E. (2003). Domain-driven design: Tackling complexity in the heart of software. Addison-Wesley.
4. Garlan, D., & Shaw, M. (1994). An introduction to software architecture. In V. Ambriola & G. Tortora (Eds.), Advances in software engineering and knowledge engineering (Vol. 1, pp. 1 - 39). World Scientific.
5. Hassan, A. E. (2008). The road ahead for mining software repositories. In Proceedings of the 2008 Frontiers of Software Maintenance (pp. 48 - 57). IEEE. https://doi.org/10.1109/FOSM.2008.4659248
6. Medvidovic, N., & Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering, 26(1), 70 - 93. https://doi.org/10.1109/32.825767

7. Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(12), 1053 - 1058. https://doi.org/10.1145/361598.361623

8. Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022). Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. In Proceedings of the 43rd IEEE Symposium on Security and Privacy (pp. 754 - 768). IEEE. https://doi.org/10.1109/SP46214.2022.9833571

9. Peng, S., Kalliamvakou, E., Cihon, P., & Demirer, M. (2023). The impact of AI on developer productivity: Evidence from GitHub Copilot. arXiv preprint arXiv:2302.06590. https://arxiv.org/abs/2302.06590

10. SysMARA Project. (2026). Five tasks, two modes, one model: A controlled experiment. https://sysmara.com/blog/five-tasks-two-modes-one-model/

11. SysMARA Project. (2026). Constraint visibility in practice: A multi-module case study. https://sysmara.com/blog/constraint-visibility-in-practice/

12. SysMARA Project. (2026). Building a TODO API with MySQL from scratch. https://sysmara.com/blog/building-a-todo-api-with-mysql/

13. SysMARA Project. (2026). SysMARA documentation: AI-native TypeScript backend framework (v0.7.1). https://sysmara.com/docs

14. SysMARA Project. (2026). AI System Graph: Machine-readable architecture. https://sysmara.com/docs/concepts/ai-system-graph/

15. SysMARA Project. (2026). Change Protocol: Safe AI-driven code changes. https://sysmara.com/docs/concepts/change-protocol/

16. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). Experimentation in software engineering. Springer. https://doi.org/10.1007/978-3-642-29044-2

17. Yamaguchi, F., Golde, N., Arp, D., & Rieck, K. (2014). Modeling and discovering vulnerabilities with code property graphs. In Proceedings of the 2014 IEEE Symposium on Security and Privacy (pp. 590 - 604). IEEE. https://doi.org/10.1109/SP.2014.44

18. Ziegler, A., Kalliamvakou, E., Li, X. A., Rice, A., Rifkin, D., Simister, S., Sittampalam, G., & Aftandilian, E. (2024). Measuring GitHub Copilot's impact on productivity. Communications of the ACM, 67(3). https://doi.org/10.1145/3626253

19. Zheng, R., Liang, Y., Qin, W., & Hassan, A. E. (2024). Rethinking autonomy: Preventing failures in AI-driven software engineering. arXiv preprint arXiv:2508.11824. https://arxiv.org/abs/2508.11824