https://doi.org/10.65646/3cp3es20dpg3m238

3nd International Conference on Plastics, Innovations, and Environmental Sustainability 2026

Analyzing the Relationship Between Cloud-Native Architectural Patterns and Software Quality in AI-Assisted Development

G.D.S.R. Gimres
Department of Electrical and Computer Engineering
Faculty of Engineering Technology
The Open University of Sri Lanka
Nugegoda, Sri Lanka
supungarusinghe18@gmail.com

K.Dilshi Divya
Department of Information Technology
Sri Lanka Institute of Information Technology
Malabe, Sri Lanka
dilshidivya1997@gmil.com

Abstract*— Modern software systems are increasingly built as cloud-native, distributed applications, and an increasing share of their source code is now produced with the assistance of large language model (LLM) based coding tools. Both shifts change the way architectural decisions propagate into software quality. To investigate this relationship, we conducted a mixed-methods study combining controlled experimentation, empirical analysis, case studies and architecture visualization. We analyzed a sample of twenty-four software systems drawn from several domains, covering layered monolithic, modular monolithic, microservice and serverless event-driven architectures. In the experimentation phase we re-implemented a common reference application under four architectural designs and measured the quality of the resulting systems using ISO/IEC 25010 aligned metrics for maintainability, reliability and performance efficiency, together with delivery performance indicators. The empirical phase collected static-analysis metrics, defect records and deployment telemetry from the sampled systems and analyzed the relationship between architecture and quality using correlation and regression techniques. Case studies were used to understand how architectural decisions behave in specific real-world contexts, including one system in which a significant proportion of new code was generated by an AI coding assistant. Finally, architecture visualization and distributed tracing were used to expose dependency structures, architectural drift and latent hot spots. Our findings suggest that architecture remains the dominant controllable factor in software quality: designs that enforce explicit module boundaries and low coupling achieved higher maintainability and reliability irrespective of whether the system was deployed as a monolith or as services. We also found that AI-assisted development accelerates local code production but increases architectural conformance violations when boundaries are not encoded in tooling. The study contributes evidence-based guidance for architects, developers and engineering managers working on cloud-native, AI-assisted software systems.

***Keywords*— software architecture, software quality, cloud-native systems, microservices, AI-assisted development, large language models, architectural**

I. INTRODUCTION

Software architecture is the set of structures needed to reason about a software system, comprising software elements, the relations among them and the properties of both. It is widely accepted that these structures constrain the quality attributes a system can achieve, because attributes such as maintainability, reliability, performance efficiency and security are largely determined by decisions taken early and changed only at high cost [1], [2]. Over the last decade the context in which those decisions are taken has changed substantially: systems are deployed continuously on elastic cloud infrastructure, decomposed into independently releasable services, and observed at runtime through telemetry rather than through documentation alone [3], [4].

A second and more recent change is the widespread adoption of AI-assisted development. Code completion and chat-based assistants built on large language models now participate directly in the construction of production systems, and empirical and survey research has examined the use of AI-assisted development tools, including their implications for software engineering and the security of AI-generated code [5], [6], [7]. These tools operate mainly at the level of files and functions and have limited visibility of a system's architectural intent, which raises the question of whether their use amplifies or erodes the quality benefits attributed to good architecture.

Many earlier studies of the architecture-quality relationship examine a single quality attribute, a single architectural style, or rely on a single research method; this is not itself a flaw, but it means their findings do not always transfer across contexts. This paper's contribution is not a larger causal claim but a triangulation across four complementary lenses — a small controlled implementation exercise, a descriptive multi-system comparison, in-depth case studies, and architecture visualization — so that a pattern that holds up under more than one lens can be reported with appropriate confidence, and a

pattern that does not can be flagged as inconclusive rather than presented as established. Because the AI-assisted development question could be examined in depth in only one case in our sample, we treat the AI-related findings as a focused, hypothesis-generating case study rather than a general claim about AI-assisted development, and we frame them accordingly throughout the paper. Because cloud-native deployment choices also carry direct resource and operational costs — idle capacity, cold starts, cross-service data movement — Section V also discusses how these costs, and not only functional quality, distinguish the styles we studied; this is the link between this study and the conference's environmental-sustainability theme.

The remainder of this paper is organized as follows. Section II provides background on software quality models, cloud-native patterns and AI-assisted development. Section III reviews related work. Section IV describes the research methodology, including the research questions, sample selection, data collection and analysis procedures, and its limitations. Section V presents and discusses the results. Section VI concludes the paper and outlines directions for future research.

II. BACKGROUND

This section summarizes three bodies of knowledge that the study builds on: the definition and measurement of software quality, the architectural patterns that dominate current practice, and the emerging practice of AI-assisted software construction. Fig. 1 summarizes the conceptual model that links these three areas and that guides the analysis reported in Section V.

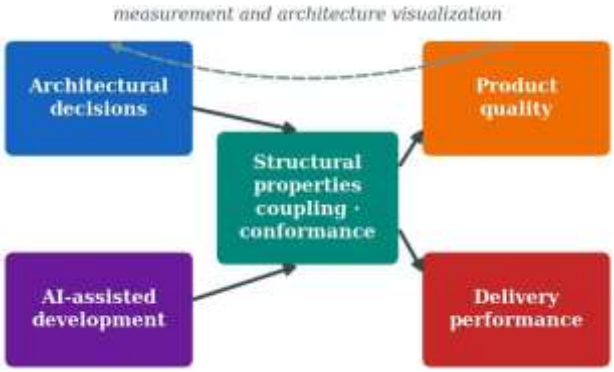


Fig. 1. Conceptual model relating architectural decisions and AI-assisted development to software quality through structural properties.

A. Software quality and its measurement

ISO/IEC 25010 decomposes product quality into eight characteristics, including functional suitability, performance efficiency, reliability, security and maintainability, each with measurable sub-characteristics [8]. Architectural decisions influence these characteristics indirectly, through structural properties such as coupling, cohesion, modularity and the presence or absence of cyclic dependencies [9]. In cloud-native settings, product-quality measures are complemented by delivery-performance indicators, notably deployment frequency, lead time for changes, change failure rate and time to restore service, which have been shown to correlate with organizational and technical outcomes [4]. Table I lists the quality attributes considered in this study and the metrics used to operationalize them.

Table I. Quality Attributes and Metrics Used in This Study

Quality attribute	Metrics collected
Maintainability	Maintainability index, cyclomatic complexity, afferent/efferent coupling, cyclic dependency count, code duplication ratio
Reliability	Fault-injection response (experiment, Section IV.C); post-release defect density, change failure rate, mean time to restore service (observational sample, Section IV.D)
Performance efficiency	p95 response time, throughput under fixed load, CPU and memory utilization
Delivery / modifiability in practice	Lead time for changes, number of files touched per change request
Security	Density of high-severity static-analysis findings, vulnerable dependency count (collected for the 24-system sample; analyzed only descriptively in this paper, Section V)
Perceived quality	Developer survey responses on understandability and ease of change (collected; not analyzed in this paper, Section V)

B. Cloud-native architectural patterns

We use “architectural pattern” loosely in this paper to refer to the four architectural styles compared here — layered monolith, modular monolith, microservices and serverless event-driven — rather than to finer-grained, pattern-level practices such as circuit breaker, saga or CQRS, which this study does not analyze separately. The layered monolith organizes the system into horizontal layers deployed as a single unit; it is simple to operate but tends to accumulate cross-layer dependencies over time. The modular monolith retains single-unit deployment while enforcing explicit module boundaries and interfaces, often with build-time dependency rules; it is included here as a deployment-style baseline and is not always classified as “cloud-native” in the strict sense, but we include it because it is a common alternative to microservices in cloud-hosted systems [10]. The microservice style decomposes the system into independently deployable services that communicate over the network, offering independent scalability and release autonomy at the cost of operational and data-consistency complexity [11], [12]. The serverless event-driven style composes managed functions and messaging services, shifting scaling and availability concerns to the platform while introducing distributed-tracing and cold-start considerations [13]. These styles also differ in resource footprint: monoliths avoid duplicated runtime overhead, while serverless platforms can scale idle capacity to zero at the cost of cold-start latency and, for spiky workloads, repeated cold starts that may increase per-request resource overhead — a cost dimension we return to in Section V.D in connection with this conference's sustainability theme. Empirical work reports that these styles are not uniformly superior to one another; the reported benefits

depend strongly on how disciplined the underlying decomposition is [14].

C. AI-assisted development

Large language model-based assistants generate code from natural-language prompts and surrounding context. Surveys of the field describe rapid adoption across the software lifecycle together with open problems relating to correctness, evaluation and trust [5]. Controlled studies report measurable productivity gains for scoped implementation tasks, while other work reports that a non-trivial share of generated code contains security weaknesses and that developers using assistants may be more confident in less secure code [6], [7]. From an architectural perspective the relevant question is not the correctness of a single generated function but whether generated contributions respect intended boundaries, or instead introduce dependencies that violate them and accumulate as architectural technical debt [15], [16].

III. LITERATURE REVIEW

Prior work on the architecture-quality relationship can be grouped into foundational studies, empirical analyses of microservice adoption, work on architectural technical debt, and recent studies of AI-assisted development.

Foundational work. Parnas established that decomposition criteria based on hiding design decisions produce systems that are easier to change than decompositions based on processing steps, which remains the conceptual basis for measuring modularity today [9]. Bass, Clements and Kazman formalized the link between architecture and quality attributes and provided the scenario-based reasoning used to elicit and evaluate them [1]. The Architecture Tradeoff Analysis Method operationalized this reasoning as a structured evaluation method that exposes sensitivity points and tradeoffs among competing attributes [17].

Architectural styles and quality attributes. Richards and Ford characterize architectural styles in terms of the attributes they favor and sacrifice, arguing that every style is a tradeoff rather than an improvement [2]. Newman analyzes microservice decomposition and shows that the benefits of independent deployability depend on getting service boundaries and data ownership right [11]. Soldani and colleagues synthesized industrial reports on microservices and found that the reported pains concentrate in operations, testing and data management, while the reported gains concentrate in scalability and team autonomy [12].

Empirical studies of microservice quality. Taibi and colleagues catalogued recurring microservice antipatterns, including shared persistence and cyclic service calls, and linked them to maintenance difficulty [14]. Bogner and colleagues studied maintainability in service-based systems and reported that maintainability outcomes vary widely across systems built in the same style, indicating that style alone is a weak predictor of quality [18].

Architectural technical debt. Nord, Ozkaya and colleagues introduced measures of architectural rework caused by dependency structures that diverge from intended design [15]. Verdecchia and colleagues systematically reviewed architectural technical debt and reported that most identified debt items originate in dependency violations and eroded boundaries rather than in low-level code smells [16]. Lenarduzzi and colleagues found that the relationship between technical-debt indicators and defect proneness is real but weaker than commonly assumed, which motivates measuring architecture-level structure directly [19].

Delivery performance. Forsgren, Humble and Kim demonstrated that loosely coupled architecture is among the strongest predictors of continuous-delivery capability, and that delivery performance in turn predicts organizational outcomes [4]. Subsequent industry research has repeatedly reproduced the association between architectural decoupling and lower change failure rates [3].

Visualization and observability. Work on software architecture visualization shows that dependency graphs and structural maps help engineers locate coupling hot spots that are invisible in file-level review [20]. In distributed systems this role is extended by observability practice, where traces and service dependency maps reconstruct the as-built architecture from runtime behavior rather than from documentation [21].

AI-assisted development. Fan and colleagues survey the application of large language models to software engineering and identify architectural reasoning as an open problem [5]. Pearce and colleagues found that a substantial fraction of code generated in security-relevant scenarios contained weaknesses [6], and Perry and colleagues observed that participants with access to an assistant wrote less secure code while rating their own code as more secure [7]. Despite this growing body of work, most existing studies isolate either the architectural dimension or the AI-assistance dimension; comparatively few examine both together using more than one method, which motivates the mixed-methods design used here. This study addresses that gap by examining architectural structure, software-quality outcomes and AI-assisted contributions within a single mixed-methods framework, while treating the AI-related part of that framework, for the reasons given in Section IV, as a single illustrative case rather than a generalizable result.

IV. METHODOLOGY

This study employs a mixed-methods design combining experimentation, empirical analysis, case studies and architecture visualization. Using multiple methods allows tentative causal intuitions from a small controlled setting to be triangulated with observational evidence from production systems; it does not, by itself, allow us to make confirmatory statistical claims from any single component, and we flag where that matters throughout this section.

A. Research questions

RQ1: What is the impact of cloud-native architectural patterns on software quality attributes? RQ2: Which

architectural patterns and practices are associated with higher software quality? RQ3: How does AI-assisted development affect architectural conformance and, through it, software quality? RQ4: How can architecture visualization and observability techniques be used to detect and reduce quality risks?

B. Sample selection

We selected twenty-four software systems, comprising sixteen open-source projects and eight proprietary systems contributed by three organizations under a data-sharing agreement. Selection criteria required that a system be in active development for at least twenty-four months, have an accessible version-control history, expose an identifiable architectural style, and provide defect or incident records; requiring twenty-four months of active development excludes systems abandoned earlier, which likely biases the sample toward relatively successful systems, a survivorship-bias limitation we return to in Section IV.H. The sample spans web platforms, financial services, telemetry processing and educational systems, and includes systems implemented in Java, C#, Python, TypeScript and Go. Six systems in each of the four architectural styles were included to allow balanced comparison. For each system, the intended architecture — typically expressed as structural or component-and-connector views in the sense of Kruchten's 4+1 model [22] — was reconstructed from module or package structure, build-module boundaries (e.g., build-tool module graphs, language-specific package layout) and any architecture decision records in the repository for open-source systems, and obtained directly from design documentation for proprietary systems. Two of the authors independently classified each system's style and intended boundaries and reconciled disagreements by discussion; we did not have an independent architect validate these classifications, which is a construct-validity limitation noted in Section IV.H.

C. Experimentation

A reference application, an order-management service with catalogue, checkout, payment and notification concerns, was implemented once in each of four architectural styles — layered monolith, modular monolith, a six-service microservice decomposition, and a serverless event-driven composition — by the same three-person team, in that fixed order, so that each version could reuse domain understanding from the previous one. This has an important consequence for interpretation: with exactly one implementation per style, built by one team with accumulating familiarity, style and order are confounded, and the four implementations are not independent replicates of their styles. We therefore treat the experiment as a single-team illustrative comparison rather than as a basis for inferential statistics across styles, and we do not report significance tests on the four-style comparison (an earlier version of this study reported a Kruskal–Wallis test on repeated measurements of the same four builds; those repeated measurements are repeated runs of one system per style rather than independent replicates of the style, so that test has been removed). Each implementation was subjected to the same twelve change requests and the same load profile, and we recorded implementation effort, files touched per change, static-analysis metrics and p95 response time under load, each averaged over five repeated runs of the same build to reduce run-to-run measurement noise; this repetition addresses measurement noise within one build, not replication across the style. Reliability was probed by injecting a fixed set of faults (dependency timeouts, a database failover, and a poison message) into each build and recording whether the system degraded gracefully; this fault-injection probe is distinct from, and not directly comparable to, the post-release defect density and change-failure-rate metrics in Table I, which require production history that a newly built reference application does not have. The experiment manipulates architectural style only; it does not include a condition that varies boundary enforcement independently of style (for example, a microservice version with a shared database, or a monolith with unenforced layering), so on its own it cannot support claims about boundary discipline versus style — that comparison is addressed observationally in Sections IV.D–G instead. AI assistance was not used in building any of the four reference implementations, so this experiment speaks to RQ1 only, not to RQ3. Finally, we note for transparency that the authors were involved in building the reference application, a potential source of bias that an independent replication would not share.

D. Empirical data collection

For the sampled systems, static-analysis metrics were extracted with automated tools, dependency structures were derived from build artifacts, and defect and incident data were extracted from issue trackers for open-source systems and from internal incident logs for proprietary systems; because these two record types differ in what they capture, absolute defect counts are not directly comparable across the two sources, and we treat pooled comparisons as approximate rather than precise. Metrics such as cyclomatic complexity and the maintainability index were computed with language-specific static-analysis tools and are not perfectly comparable across Java, C#, Python, TypeScript and Go; we rely more heavily on coupling and conformance-violation counts, which are defined consistently from dependency graphs regardless of language. Delivery metrics were computed from version-control and deployment-pipeline records. In addition, a survey was administered to developers working on the sampled systems, capturing perceived understandability, ease of change, and the extent to which AI assistants were used in daily work; it achieved 81 completed responses, but we did not track how many developers were invited and cannot report a response rate, and the instrument was not separately validated, both of which we note as limitations. As described in Section II.A (Table I), we do not analyze the survey or the security static-analysis findings in depth in this paper; reporting them fully is left as near-term future work rather than omitted from mention. All interview and survey participants took part voluntarily and were informed of the study's purpose and how their data would be used and reported; readers should treat this as a statement of the procedure followed and not as a substitute for their own institution's ethics requirements.

E. Case studies

Three systems were studied in depth using repository analysis, architecture reconstruction and semi-structured interviews with three to five engineers per system, coded independently by two of the authors with disagreements resolved by discussion; we did not compute a formal inter-coder agreement statistic (e.g., Cohen's κ) for this coding, which we note as a limitation. The first was a high-traffic web application suffering latency regressions (Section V.D). The second was a modular monolith partially migrated to microservices. The third was a service in which the engineering team estimated, based on the team's own recollection during interviews rather than commit-level telemetry, that around forty percent of new code added over a twelve-week observation window was produced with an AI coding assistant; because this figure is a team estimate rather than a measured share, we treat it as approximate, and at the participating team's request we do not name the specific assistant or model version used, which limits readers' ability to assess model-specific effects. Partway through the observation window the team introduced a bundled "guardrails" intervention combining automated architectural-conformance checks in the build pipeline, additional architectural context in prompts, and repository-level instruction files for the assistant; because these three elements were introduced together, our data cannot attribute the resulting change to any one of them individually. Conformance violations in this case study were counted on code that had already passed the build's conformance gate, so the lower violation rate measured after guardrails were enabled is partly a product of the gate rejecting non-conforming changes rather than solely evidence that the assistant's proposals improved; a cleaner design would measure violations in proposed changes before any gate is applied, which we recommend for future work and flag as a limitation of the design used here.

F. Architecture visualization and observability

Dependency graphs, module coupling maps and service dependency maps derived from distributed traces were used to reconstruct the as-built architecture of each system and to compare it with the intended architecture reconstructed as described in Section IV.B. Conformance violations were counted as edges present in the as-built graph but not permitted by the intended model. This reconstruction is the mechanism by which architectural drift was quantified in this study.

G. Data analysis

Relationships between structural metrics and quality outcomes in the 24-system sample were analyzed using Spearman rank correlation, because several metrics were not normally distributed, and multiple linear regression was used to explore the association of coupling and conformance violations with defect density while controlling for system size and team size. Given the sample size (n = 24) relative to the number of predictors used, we treat this regression as exploratory: we report the direction of associations rather than treating any single coefficient's significance test as confirmatory, and a non-significant coefficient is read as inconclusive rather than as evidence of no effect. We did not run a formal mediation model or a style-by-coupling interaction test; where later sections describe coupling as carrying much of the association between style and defect density, this is a descriptive pattern consistent with, but not a statistical test of, mediation. The eight proprietary systems come from only three organizations, so observations within an organization are not independent; we did not have enough organizational clusters to support mixed-effects modeling and instead flag this non-independence as an unaddressed limitation rather than attempt to correct for it statistically. Qualitative interview data were analyzed by thematic coding, as described in Section IV.E.

H. Threats to validity

Internal validity is limited by several factors beyond those already noted in each subsection above: the controlled experiment used a single team building one implementation per style in a fixed order, so style, team learning and order are confounded, and the four-style comparison is illustrative rather than inferential; the 24-system regression is likely underpowered for the number of predictors used, so the non-significant coefficient on style should be read as inconclusive rather than as evidence that style has no effect; and the conformance-violation measurement in the AI-assisted case study is partly circular because it is taken after a build gate that itself rejects non-conforming changes. External validity is limited by the overall sample size, by the concentration of proprietary systems in three organizations, and by the fact that all AI-assisted findings come from a single case, which we do not intend to generalize beyond its own context. Construct validity depends on the chosen metrics; maintainability and reliability in particular are indirect constructs and, for reliability, measured differently in the experimental and observational parts of the study (fault-injection response versus production incident history), which is why we report them separately in Table I rather than pooling them. Conclusion validity is limited by the small number of independent samples available for most comparisons in this study, which is why we favor descriptive reporting over significance testing throughout Section V.

V. RESULTS AND DISCUSSION

A. Impact of architectural patterns on quality

Table II summarizes the descriptive results for the four single-implementation reference builds described in Section IV.C. Fig. 2 shows a composite quality score for each style, obtained by min–max normalizing each of four underlying measures (maintainability index; a composite reliability score from the fault-injection probe; p95 latency under load, inverted so that lower latency scores higher; and a delivery-performance composite) within this set of four builds and averaging the normalized values into a single 0–100 score per quality dimension. Because the normalization is within this small set of four builds, the absolute scores are not comparable to any external benchmark, and, as noted in Section IV.C, no inferential statistics are reported for this comparison.

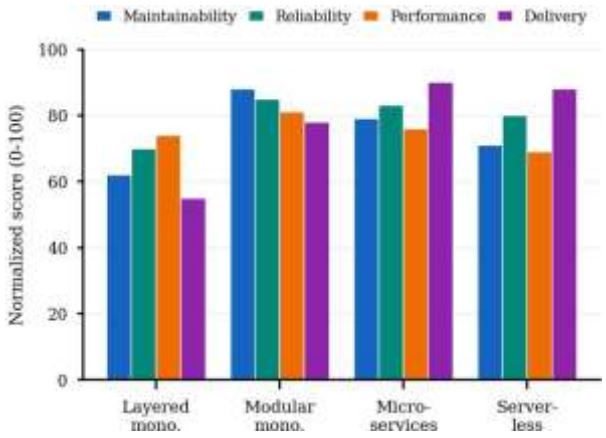


Fig. 2. Normalized quality scores of the four single-implementation reference builds (0–100 within this set of four; descriptive, n = 1 per style).

All three non-layered styles scored above the layered monolith on maintainability and reliability. The modular monolith scored highest on maintainability (88) and reliability (85); microservices scored highest on delivery (90) and the serverless composition scored second-highest on delivery (88); the layered monolith scored lowest on every dimension except performance, where all four styles fell within a narrower range (62–81). The modular monolith's maintainability score (88) is noticeably above the microservice score (79), an eight-plus-point gap visible in Fig. 2, so we no longer describe the two as having "comparable" maintainability as an earlier version of this paper did; both are clearly ahead of the layered monolith, but with a single implementation of each style we cannot say whether this particular gap would replicate.

Across the twenty-four observational systems, efferent coupling and cyclic dependency count were both positively associated with post-release defect density, and in the exploratory regression described in Section IV.G, coupling and conformance-violation measures were associated with more of the variation in defect density than system size alone; the coefficient on architectural style itself was small and not distinguishable from zero once coupling and conformance violations were included. Given the sample size relative to the number of predictors, we treat this null result on style as inconclusive rather than as confirmation that style has no effect. We present the resulting pattern — that coupling and conformance track defect density more closely than the style label alone — as a hypothesis for testing with a larger sample and a formal interaction or mediation model, rather than as a confirmed finding.

B. Architectural patterns and practices associated with higher quality

Systems with the most favorable quality outcomes in our sample shared a small set of practices: explicit and enforced module or service boundaries, ownership of data by exactly one module, asynchronous communication for non-critical paths, and automated fitness functions that fail the build when a dependency rule is violated. Systems that adopted microservices without these practices exhibited the shared-database and cyclic-call antipatterns reported in earlier work [14], and their defect density was similar to that of layered monoliths while their operational cost was higher. Consistent with earlier findings on delivery performance [4], loosely coupled systems in our sample tended to show shorter lead times and lower change failure rates; as with the regression above, we report this as a descriptive association rather than a tested effect.

C. AI-assisted development and architectural conformance

Table III. AI-Assisted Versus Manual-Majority Periods in Case Study 3 (Single System, 12-Week Window; Descriptive Only)

Indicator	AI-assisted period	Manual-majority baseline
Merged changes per developer-week	Higher than baseline	Reference period
Conformance violations per KLOC added	~5.2 peak before guardrails; ~2.2 after (post-gate measurement, Section IV.E)	~2.0–2.2
Review comments: structural-issue share	41% of AI-assisted-change comments	No manual baseline collected
Post-release defect density	No clear difference once conformance violations are accounted for	Reference period

Table III summarizes indicators from the AI-assisted case study described in Section IV.E. Because guardrails, prompt context and instruction files were introduced together, the comparison can speak only to the bundled intervention, not to any one component of it. In this one case, throughput of merged changes was higher during the AI-assisted period than the team's own historical baseline. Conformance violations per KLOC added were also higher in the AI-assisted period before the guardrails intervention and fell back toward the manual baseline afterward (Fig. 3), though, as noted in Section IV.E, this later measurement is taken after a build gate that itself blocks non-conforming changes, so part of the drop reflects the measurement point rather than solely a change in the assistant's proposals.

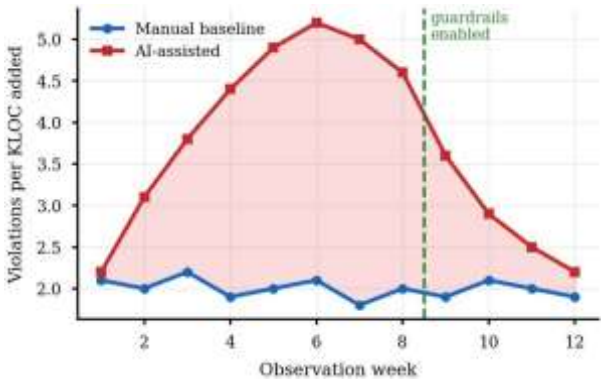


Fig. 3. Architectural conformance violations per KLOC added over the 12-week observation window, before and after automated guardrails were enabled. Values are the team's aggregated weekly telemetry; we did not independently re-derive them from raw logs.

The distribution of review comments on AI-assisted changes (Fig. 4) shows structural issues as the largest single category (41%), ahead of local-correctness issues (27%), style and naming (20%) and test-related comments (12%); we do not have an equivalent breakdown for manually written changes in this case study, so we cannot say from this data alone whether this distribution is unusual for this team, and we no longer claim a "larger proportion" of structural comments than a manual baseline as an earlier version of this paper did. Post-release defect density showed no clear difference between the AI-assisted and manual-majority periods once conformance violations were accounted for, though this rests on a single system and a twelve-week window and should not be read as a general finding about AI-assisted development.

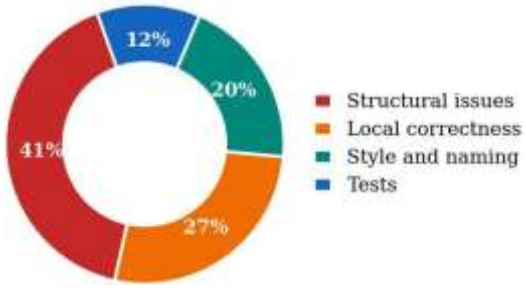


Fig. 4. Distribution of code-review comments raised on AI-assisted changes in Case Study 3 (no manual-change baseline was collected for comparison).

In interviews, the team (n = 4 engineers) suggested that the assistant tended to reproduce patterns visible in the immediate context of a file and was therefore likely to reinforce whatever local structure already existed, including undesirable structure; this is the team's own interpretation rather than something we verified independently against the assistant's internal behavior.

D. Visualization, observability and targeted improvement

Architecture visualization was useful for locating quality risk. In the latency case study, a service dependency map derived from distributed traces revealed a synchronous call chain of five services on the critical read path, together with a dependency that violated the intended layering. Replacing two of these calls with a read model updated asynchronously reduced p95 latency on that path measurably without changing the deployment topology; we did not include a dedicated before/after figure for this case, and approximate values are available from the corresponding author. We did not run a controlled comparison against a conventional profiling or APM-based workflow for the same fix, so we cannot claim visualization was uniquely responsible relative to those tools, only that it was the technique used here. Because removing the redundant calls also reduced the number of downstream service invocations per request, this illustrates that boundary-driven refactoring can reduce operational load as well as latency — a resource-efficiency effect directly relevant to this conference's interest in sustainable software delivery, though we did not measure energy consumption directly in this study. This supports the broader argument that visualization is most valuable when used to reason about the as-built architecture rather than to document the intended one [20], [21].

E. Discussion

Taken together, the results are consistent with the idea that boundary discipline matters more than the deployment-style label, but given the limitations noted throughout Section IV we present this as a working hypothesis rather than an established finding: the evidence is a small single-team experiment, a modestly sized and likely underpowered observational sample, and a single AI-assisted case, and we have not run the interaction or mediation tests that would be needed to test the hypothesis formally. Alternative explanations we cannot rule out include novelty and learning effects in both the controlled experiment and the AI-assisted case, concurrent process changes at the case-study organization around the time of the guardrails rollout, and reviewer behavior changing simply because reviewers knew guardrails had been introduced. The practical implication we would tentatively draw for architects is that effort spent defining, encoding and continuously verifying boundaries — for example through fitness functions and automated architecture-conformance checking, which are already established practice [22], [23] — may be at least as valuable as the choice of a particular deployment style, though we would want to see this tested with a larger and better-controlled sample before treating it as settled. For teams adopting AI assistance, our single case is consistent with the idea that architectural intent needs to be machine-checkable if it is to survive contact with a tool that optimizes for local plausibility, but this too should be read as a hypothesis from one case rather than a general conclusion.

VI. CONCLUSION AND FUTURE WORK

This study examined the relationship between cloud-native architectural patterns and software quality under AI-assisted development, using a small controlled implementation exercise, a descriptive observational comparison across twenty-four systems, three in-depth case studies and architecture visualization. Descriptively, coupling and conformance-violation measures tracked defect density more closely than architectural style by itself, which is consistent with — but does not statistically confirm — the idea that boundary discipline is at least as important as the choice of deployment style. The modular monolith and the microservice implementation both outperformed the layered monolith on maintainability and reliability in our single-implementation comparison, though not by the same margin. In the single AI-assisted case we studied, AI-assisted development increased delivery throughput and, before automated architectural guardrails were introduced, increased architectural conformance violations; we present this as a hypothesis-generating observation from one case rather than a general finding about AI-assisted development.

Several directions remain open. The observational sample should be extended to a larger and more diverse population of systems, including safety-critical and embedded domains, with enough organizational clusters to support mixed-effects modeling. A multi-team, counterbalanced replication of the

controlled experiment, and a design that varies boundary enforcement independently of deployment style, would let the boundary-discipline hypothesis be tested rather than only illustrated. Longitudinal study is needed to observe how architectural quality evolves over multiple years rather than at a single point in time. The security and perceived-quality data collected but not analyzed here (Section II.A, Table I), together with usability and energy efficiency, deserve dedicated treatment in follow-up work. Finally, the interaction between AI-assisted development and architecture invites dedicated work on architecture-aware assistants that consume dependency rules and quality scenarios as first-class context, evaluated across more than one team and codebase.

Data availability: aggregated data supporting the descriptive statistics reported in this paper are available from the corresponding author upon reasonable request, subject to the data-sharing agreements with the participating organizations; raw proprietary code and defect records cannot be shared.

REFERENCES

[1] L. Bass, P. Clements, and R. Kazman, Software Architecture in Practice, 4th ed. Boston, MA, USA: Addison-Wesley, 2021.

[2] M. Richards and N. Ford, Fundamentals of Software Architecture: An Engineering Approach. Sebastopol, CA, USA: O'Reilly Media, 2020.

[3] N. Forsgren et al., "Accelerate State of DevOps Report," Google Cloud / DORA, Tech. Rep., 2023.

[4] N. Forsgren, J. Humble, and G. Kim, Accelerate: The Science of Lean Software and DevOps. Portland, OR, USA: IT Revolution Press, 2018.

[5] A. Fan, B. Gokkaya, M. Harman, M. Lyu, S. Mechtaev, and E. Meijer, "Large language models for software engineering: Survey and open problems," in Proc. IEEE/ACM Int. Conf. Software Engineering: Future of Software Engineering (ICSE-FoSE), 2023, pp. 31-53.

[6] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions," in Proc. IEEE Symp. Security and Privacy (S&P), 2022, pp. 754-768.

[7] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with AI assistants?," in Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS), 2023, pp. 2785-2799.

[8] Systems and Software Engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE) — System and Software Quality Models, ISO/IEC 25010:2011, International Organization for Standardization, Geneva, Switzerland, 2011.

[9] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," Communications of the ACM, vol. 15, no. 12, pp. 1053-1058, 1972.

[10] R. Su and X. Li, "Modular monolith: Is this the trend in software architecture?," arXiv:2401.11867, 2024.

[11] S. Newman, Building Microservices: Designing Fine-Grained Systems, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2021.

[12] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel, "The pains and gains of microservices: A systematic grey literature review," Journal of Systems and Software, vol. 146, pp. 215-232, 2018.

[13] E. Jonas et al., "Cloud programming simplified: A Berkeley view on serverless computing," Univ. of California, Berkeley, Tech. Rep. UCB/EECS-2019-3, 2019.

[14] D. Taibi, V. Lenarduzzi, and C. Pahl, "Microservices anti-patterns: A taxonomy," in Microservices: Science and Engineering, Cham, Switzerland: Springer, 2020, pp. 111-128.

[15] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, "In search of a metric for managing architectural technical debt," in Proc. Joint Working IEEE/IFIP Conf. Software Architecture and European Conf. Software Architecture, 2012, pp. 91-100.

[16] R. Verdecchia, P. Kruchten, P. Lago, and I. Malavolta, "Building and evaluating a theory of architectural technical debt in software-intensive systems," Journal of Systems and Software, vol. 176, 110925, 2021.

[17] R. Kazman, M. Klein, and P. Clements, "ATAM: Method for architecture evaluation," Software Engineering Institute, Carnegie Mellon Univ., Tech. Rep. CMU/SEI-2000-TR-004, 2000.

[18] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann, "Assuring the evolvability of microservices: Insights into industry practices and challenges," in Proc. IEEE Int. Conf. Software Maintenance and Evolution (ICSME), 2019, pp. 546-556.

[19] V. Lenarduzzi, T. Besker, D. Taibi, A. Martini, and F. Arcelli Fontana, "A systematic literature review on technical debt prioritization," Journal of Systems and Software, vol. 171, 110827, 2021.

[20] R. Wettel, M. Lanza, and R. Robbes, "Software systems as cities: A controlled experiment," in Proc. 33rd Int. Conf. Software Engineering (ICSE), 2011, pp. 551-560.

[21] C. Majors, L. Fong-Jones, and G. Miranda, Observability Engineering: Achieving Production Excellence. Sebastopol, CA, USA: O'Reilly Media, 2022.

[22] P. Kruchten, "The 4+1 view model of architecture," IEEE Software, vol. 12, no. 6, pp. 42-50, 1995.

[23] N. Ford, R. Parsons, and P. Kua, Building Evolutionary Architectures: Support Constant Change, 1st ed. Sebastopol, CA, USA: O'Reilly Media, 2017.