International Journal of
Technology and
Systems
(IJTS)

How Technology Fails Developers: A Qualitative Investigation of Tool, Architecture, and Legacy System Deficiencies in Software Error Introduction

Journal of Technology and Systems
ISSN: 2788-6344 (Online)
Vol. 8, Issue No. 3, pp 19 – 36, 2026
www.carijournals.org

How Technology Fails Developers: A Qualitative Investigation of Tool, Architecture, and Legacy System Deficiencies in Software Error Introduction

[1]*Rahul Azmeera, [2]Dr. James C Hyatt

[1]PhD, Independent Researcher, University of the Cumberlands, United States

[2]Adjunct Professor, University of the Cumberlands, United States

https://orcid.org/0000-0003-4454-1426

Accepted: 9th July, 2026, Received in Revised Form: 23rd July, 2026, Published: 9th August, 2026

ABSTRACT

Purpose: This study investigated the technological deficiencies contributing to software errors in software products, examining how inadequacies in development tools, developer expertise gaps, architectural failures, modern technology adoption challenges, and legacy system constraints create conditions for software errors.

Methodology: A qualitative phenomenological design was employed. Semi-structured interviews were conducted with 12 experienced software developers averaging 14.5 years of experience in U.S.-based organizations across retail, healthcare, technology, and financial sectors. Data were analyzed using reflexive thematic analysis in NVivo 14, anchored in the Technological Context construct of the Technology-Organization-Environment (TOE) framework.

Findings: Five themes emerged from this study, theme 1. Inadequacies in development tools and practices- discovering the challenges with existing tools and developer practices; theme 2. Impact of developer expertise and practices on software quality- discusses how the individual work experience shows bias in software development; theme 3. System architecture and oversight failures- discusses the software errors caused due to poor architecture and error monitoring; theme 4. The role of modern technologies and automation- discusses the impact of AI coding assistants; theme 5. Challenges posed by legacy systems and external integrations- elaborates how legacy systems limit developers' productivity and software innovation. These themes collectively constitute the Technological Error Origin Framework (TEOF).

Unique contribution to theory, practice and policy: This study provides the first developer-experience-grounded, phenomenologically anchored account of software error origins within the TOE framework. Organizations should invest in integrated error-detection toolchains, implement developer upskilling programs spanning both technical and domain knowledge, enforce architecture-first principles with mandatory monitoring, and establish structured legacy modernization roadmaps.

Keywords: *Software Errors, Development Tools, System Architecture, Legacy Systems, AI Code Generation*

1. INTRODUCTION

Software errors are expensive, pervasive, and, in many cases, preventable. The American Consortium for Information and Software Quality estimated that poor software quality cost U.S. organizations over $2.84 trillion in 2018 alone, with a substantial portion attributable to defects that persisted from development into production (Krasner, 2021). Despite decades of investment in testing methodologies, quality frameworks, and development process improvements, errors continue to reach end users at troubling scale. The technological foundations of software development, the tools developers rely on, the knowledge they bring, the architectures they design within, and the systems they inherit remain insufficiently understood as sources of error introduction.

This paper reports on Research Question One (RQ1) from a broader phenomenological dissertation examining software errors and their consequences: What technological deficiencies are contributing to software errors in practice? Positioned within the Technological Context construct of the Technology-Organization-Environment (TOE) framework (Baker, 2011; Horani et al., 2023), the study explores the lived professional experiences of 12 software developers employed in U.S.-based organizations. Their accounts identify five distinct technological failure modes patterns of deficiency that practitioners navigate daily, with direct consequences for software quality.

The technological context in TOE encompasses the characteristics, complexity, compatibility, and functionality of the technologies themselves (Baker, 2011). In software development, this includes the integrated development environments (IDEs) developers use, the programming languages and frameworks they work in, the testing tools available to them, the architectures they design and maintain, and the legacy systems they must accommodate (Li et al., 2022). Deficiencies in any of these dimensions create conditions for errors that testing may fail to catch before deployment.

What distinguishes this study from purely technical analyses of software defects is its grounding in developer-lived experience. Error taxonomies and defect datasets can tell us what kinds of errors occur and how often, but they cannot reveal why tools fail developers at the moment of coding, how expertise gaps translate into specific error-introduction patterns, or how legacy system constraints shape daily development decisions. Those questions require qualitative, interpretive inquiry. The paper proceeds through a literature review, methodology, thematic findings, discussion, and conclusion, with a thematic relational diagram illustrating how the five identified technological failure modes interact.

2. REVIEW OF LITERATURE

2.1 Development Tools and Testing Challenges

The role of development tooling in software quality has attracted sustained research attention. Lopez et al. (2021) documented that developers encounter significant friction when transitioning between tools and frameworks, with inadequate error messaging and limited real-time feedback

mechanisms in IDEs identified as primary contributors to error introduction. Their analysis highlighted that developers who rely heavily on tool-generated feedback are paradoxically more exposed to errors when those tools fail to surface problems at compile time or during active coding sessions.

Testing presents a related but distinct challenge. Bluemke and Malanowska (2021) and Zardari et al. (2022) identified test case selection as one of the most cognitively demanding tasks in software development, with developers frequently unable to anticipate the full range of edge cases that a complex system might encounter. This challenge is amplified in cloud-based development contexts, where black-box infrastructure services limit developers' ability to reproduce production environments locally for thorough unit testing (Govindaras et al., 2023; Kurian & Thomas, 2023). The growing adoption of third-party cloud services and managed platforms has expanded what Panthi et al. (2022) describe as the unknown testing horizon, the space of potential failure modes that developers cannot directly observe or pre-emptively test.

2.2 Developer Expertise and Knowledge Gaps

The relationship between developer expertise and software error rates has been examined from multiple angles. Lopez et al. (2021) found that developers transitioning between technology stacks frequently carry conceptual residue from prior languages patterns of thought and habitual syntax that interfere with correct implementation in new environments. This cognitive transfer problem is particularly pronounced when the paradigms underlying the new technology differ fundamentally from those the developer has internalized.

Jayakody and Wijayanayake (2021) documented expertise gaps within DevOps-oriented development teams, finding that knowledge silos and uneven familiarity with toolchains contributed to systematic error-introduction patterns that were difficult to detect through standard code review. Khan et al. (2022) extended this analysis, arguing that the increasing pace of framework evolution has created a structural challenge for organizations: the speed at which new tools emerge consistently outpaces the organizational capacity to train developers to use them correctly. The result is a workforce that is perpetually partially competent with at least some of the tools it is required to use.

2.3 System Architecture and Technical Debt

Architecture quality is foundational to software reliability. Taylor (2019) established that architectural deficiencies particularly those involving inadequate error handling, insufficient resilience planning, and poor separation of concerns create compounding failure conditions that manifest as software errors under real-world load and integration scenarios. Tian et al. (2022) documented how communication ambiguities in architecture specifications lead developers to make assumptions that introduce errors, particularly in systems involving multiple integrated components developed by different teams.

The concept of technical debt, formalized by Akbar et al. (2022) as the accumulated cost of deferred quality and security improvements, provides an organizing framework for understanding how architectural oversights compound over time. Systems that begin with manageable debt levels frequently develop into environments where addressing any single error risks introducing cascading failures, creating organizational incentives to defer remediation further a dynamic that researchers have identified as a significant contributor to sustained high error rates in mature software systems (Castro et al., 2023; Nugraha & Talita, 2021).

2.4 Modern Technologies and AI-Assisted Development

The emergence of AI-assisted coding tools has introduced both new opportunities and new error pathways. Xu et al. (2022) found that large language model-based coding assistants could accelerate developer productivity significantly but that the code generated by these tools frequently failed to account for domain-specific business logic nuances, creating errors that were difficult to detect through standard testing because the generated code was syntactically valid. Khaliq et al. (2023) and Soui and Haddad (2023) documented similar limitations in AI-based test case generation, finding that generated tests tended to mirror the assumptions embedded in the code they were testing rather than independently validating expected behavior.

Security implications of AI code generation have received growing attention. Negri-Ribalta et al. (2024) identified that AI-generated code can introduce security vulnerabilities through subtly incorrect database queries and authentication logic errors that may not be detectable through functional testing but that create exploitable conditions in production. These findings underscore the importance of developer understanding of generated code rather than passive adoption.

2.5 Legacy Systems and Integration Complexity

Legacy systems represent one of the most persistent and underexamined sources of software errors. Alexandrova and Rapanotti (2019) characterized legacy system maintenance as a domain in which the normal assumptions of software engineering break down: documentation is absent or inaccurate, the original developers are unavailable, the underlying technology is poorly supported, and the system is too deeply integrated into organizational operations to replace quickly. Jayakody and Wijayanayake (2021) found that developers assigned to legacy system integration tasks consistently reported higher error rates and lower confidence in their work than those operating in greenfield environments.

The challenge is compounded by third-party SDK and library dependencies. Pashchenko et al. (2020) documented how organizational reliance on external libraries whose maintenance has lapsed creates systematic exposure to unresolved errors that the development team cannot directly remediate. The organizational tendency to defer SDK updates to avoid integration risk means that known vulnerabilities and defects in these dependencies frequently persist in production systems for extended periods.

2.6 Research Gap

Existing research on the technological roots of software errors is predominantly quantitative defect density analyses, error taxonomies, testing coverage metrics. What is missing is a developer-centered qualitative account of how these technological deficiencies are experienced in practice: the specific ways in which IDE limitations, expertise gaps, architectural oversights, AI tool misuse, and legacy system constraints combine in the daily experience of working developers to produce software errors. This study addresses that gap directly, contributing a five-theme empirical framework derived from the lived experiences of practitioners who navigate these conditions professionally.

3. METHODOLOGY

3.1 Research Design

A qualitative research design using phenomenological methodology was adopted for this study. Phenomenology is suited to research aiming to understand how individuals make meaning of a specific phenomenon through their lived experiences (Creswell & Creswell, 2018; Neubauer et al., 2019). The research question what technological deficiencies lead to software errors required a method capable of surfacing the experiential texture of developer encounters with these deficiencies, not merely cataloguing them. Phenomenology offered that capacity (Frechette et al., 2020).

The study was anchored in the Technological Context construct of the TOE framework (Baker, 2011; Horani et al., 2023), which provided structural coherence to the interview protocol and analytical focus. Positioning RQ1 within TOE's technological construct ensured that analytical attention remained trained on technology-level factors tools, languages, architectures, automation, legacy constraints rather than drifting into organizational or environmental dimensions addressed by the broader dissertation.

3.2 Participants and Sampling

Criterion-based purposive sampling was used to identify participants capable of providing substantive data on the research question (Hennink & Kaiser, 2022). Inclusion criteria required at minimum five years of software development experience in mobile or web application development, a bachelor's degree or higher in computer science, information technology, or a related field, and current employment in a U.S.-based organization. Participants were identified through LinkedIn, with initial contact via direct message and informed consent documentation distributed by email.

Purposive criterion sampling yielded an initial pool of approximately 35 LinkedIn contacts who met the inclusion criteria. Of these, 17 responded to the initial outreach. Following exclusions for non-response to subsequent communication and voluntary withdrawal during the consent process, 12 participants proceeded to interview. The sample comprised nine males and three females with an average age of 39.25 years and average professional experience of 14.5 years across retail,

convenience, technology, healthcare, and asset management sectors. Table 1 presents full demographic data.

Table 1: Participant Demographic Characteristics (N = 12)

Code	Education	Experience	Age	Gender	Sector	Duration
CD01	Master's	13 yrs	38	Male	Retail	37 min
CD02	Bachelor's	17 yrs	41	Male	Convenience	27 min
CD03	Master's	8 yrs	31	Male	Convenience	26 min
CD04	Bachelor's	13 yrs	38	Male	Technology	40 min
CD05	Bachelor's	15 yrs	45	Female	Convenience	24 min
CD06	Master's	12 yrs	36	Male	Technology	24 min
CD07	Master's	24 yrs	50	Male	Health Care	33 min
CD08	Master's	7 yrs	30	Male	Technology	28 min
CD09	Master's	11 yrs	35	Female	Technology	34 min
CD10	Bachelor's	20 yrs	44	Male	Technology	29 min
CD11	Bachelor's	19 yrs	43	Female	Technology	36 min
CD12	Master's	15 yrs	40	Male	Asset Mgmt	33 min
Average	—	14.5 yrs	39.25	9M / 3F	Diverse	30.9 min

Note. *Participants recruited via LinkedIn; identified by coded designations to preserve confidentiality. Average interview duration: 30.9 minutes.*

3.3 Data Collection

Semi-structured individual interviews were conducted via Zoom, enabling simultaneous audio-video recording and live transcription. The interview protocol comprised four questions aligned with the TOE framework constructs; the question relevant to RQ1 asked participants to elaborate on what technological improvements, if available, might have reduced the software errors they had encountered. Sessions averaged 30.91 minutes, ranging from 24 to 40 minutes. Informed consent was obtained before each recording, and all data were stored in password-protected folders with personally identifiable information removed.

3.4 Data Analysis

Thematic analysis followed Braun and Clarke's (2022) six-phase reflexive approach, managed in NVivo 14. Analysis proceeded from data familiarization through initial coding, theme clustering and refinement, and final thematic representation (Byrne, 2022). The process was iterative coding categories were revised multiple times as patterns tested against the full dataset. Theoretical saturation was reached at the twelfth interview; no new substantive codes emerged after the eleventh participant.

3.5 Trustworthiness

Credibility was supported by anchoring every theme in direct participant quotation, triangulating findings against existing literature, and maintaining active researcher bracketing throughout analysis. Dependability was addressed through detailed procedural documentation enabling replication. Transferability was supported by sector diversity in the sample. Confirmability was maintained through reflexive journaling. IRB approval was secured prior to recruitment, and all participation was fully voluntary (Karcher et al., 2024; Lincoln & Guba, 1985).

4. FINDINGS

Thematic analysis of the twelve interview transcripts generated five themes capturing the technological deficiencies that, in developer-lived experience, contribute to software errors. Table 2 presents the thematic structure, key codes, participant representation, and supporting literature.

Table 2: Thematic Structure: Technological Deficiencies Leading to Software Errors

Theme	Key Codes / Sub-Themes	Interpretive Description	Frequency	Supporting Literature
Theme 1: Inadequacies in Development Tools & Practices	IDE limitations; Black-box cloud frameworks; Unit testing gaps; Rapid tool churn	Development tools fail to catch errors in real-time, lack auto-testing capabilities, and restrict local debugging. Cloud black-box frameworks limit developers' ability to fully test systems before deployment.	12/12 participants; 70 references	Lopez et al. (2021); Bluemke & Malanowska (2021); Zardari et al. (2022)
Theme 2: Developer Expertise & Practice Gaps	Experience-level disparities; Business knowledge gaps; Technology transition friction; Code responsibility	Knowledge gaps between senior and junior developers, limited business domain understanding, and resistance to adopting new technology stacks generate systematic error-introduction patterns.	12/12 participants (multiple refs)	Jayakody & Wijayanayake (2021); Lopez et al. (2021); Khan et al. (2022)
Theme 3: System Architecture & Oversight Failures	Poor architecture design; Absent error monitoring; Technical debt accumulation; Type-safety issues	Flawed system architecture blueprints, absent error monitoring and logging systems, and unhandled exceptions create compounding failure conditions that escalate into production errors.	All 12 participants; 15 references	Taylor (2019); Tian et al. (2022); Nugraha & Talita (2021)
Theme 4: Modern Technologies & Automation Challenges	AI code generation risks; Automated testing gaps; Copilot adoption issues; Automation benefits	AI-generated code introduces errors when developers lack understanding of generated output. Absence of automated code health checks allows errors to reach production undetected.	All 12 participants; 12 references	Xu et al. (2022); Khaliq et al. (2023); Negri-Ribalta et al. (2024)
Theme 5: Legacy Systems & External Integration Challenges	Outdated technology stacks; Documentation failures; SDK dependency issues; Migration complexity	Legacy systems constrain technology adoption, create integration bottlenecks, and demand specialized knowledge that is scarce. Third-party SDK inconsistencies compound these challenges.	All 12 participants; 15 references	Alexandrova & Rapanotti (2019); Jayakody & Wijayanayake (2021)

Note. Reference counts reflect frequency of theme-related expressions coded across all 12 participant interviews in NVivo 14. IDE = Integrated Development Environment; SDK = Software Development Kit.

4.1 Theme 1: Inadequacies in Development Tools and Practices

Every participant all twelve addressed inadequacies in development tools, generating 70 coded references, the highest frequency of any theme in the dataset. The sheer volume of this theme reflects how centrally tooling sits in developers' daily experience of error. Three distinct inadequacy patterns emerged: IDEs that fail to detect errors at coding time, cloud-based infrastructure tools that restrict testing visibility, and the error-introduction risk associated with rapidly evolving frameworks that development teams have not had time to master.

On IDE limitations, participants described a gap between what current development environments offer and what developers need to write clean code reliably. The problem is not merely that IDEs miss errors it is that developers come to rely on them as a primary error-detection mechanism, so when they fail, errors advance unchallenged through the development pipeline:

> *"You could use a better IDE that catches errors in real-time, which is right in line. Tools will not teach you how to write error-free code, and completely relying on these tools is also a negative."*
>
> **- CD01**

Cloud infrastructure tools introduced a qualitatively different problem. As organizations migrate workloads to managed cloud services Snowflake for data processing, AWS Lambda for serverless functions developers lose direct access to the runtime environment they are coding against. Testing becomes, in the words of one participant, a black-box exercise:

> *"As we move more and more to the cloud, we depend more on black-box frameworks that we may not have access to test locally. These could include systems like Snowflake for large data storage or services like Lambda functions, where the code runs on remote servers. While I can be confident about what platform it is operating on, I may not be able to reproduce that environment in a unit test beforehand."*
>
> **- CD07**

The third inadequacy pattern concerned the pace of tooling change. When organizations adopt new frameworks before adequate testing infrastructure exists for those frameworks, developers are forced to build in environments they cannot fully validate:

> *"When I started with the Golang, I had to learn everything again. As new tools were being introduced, we did not have adequate tooling for unit testing or end-to-end testing because most people were unfamiliar with them. When we build an API, once it was completed, there were no incremental improvements, such as adding unit test cases to ensure that the new code did not break existing functionality."*

- CD08

These findings align with Lopez et al.'s (2021) documentation of tool transition friction and Zardari et al.'s (2022) analysis of test case selection challenges. What the participant data contributes is a practitioner-level account of the mechanisms: not tool inadequacy as an abstract deficiency, but the specific ways in which real-time feedback gaps, cloud opacity, and framework immaturity combine to create error pathways in daily development work.

4.2 Theme 2: Impact of Developer Expertise and Practices on Software Quality

Developer expertise emerged as the second major technological root of software errors not merely in the sense that inexperienced developers make more errors (though they do), but more significantly in the sense that expertise gaps are structural and organizational rather than purely individual. Organizations consistently deploy developers against technology stacks they have not fully mastered, under time constraints that prevent adequate learning, and without the business domain knowledge necessary to implement correct logic.

Business knowledge gaps were identified as a particularly underappreciated source of error. Developers working in domains they do not understand cannot anticipate the edge cases that domain logic requires:

> *"We receive limited communication and very little information about the functionality. We may not be experts in that particular area, may lack business knowledge, and may not fully understand the business rules. This leads to errors in day-to-day development when working on a product."*
>
> **- CD02**

The experience-level differential between senior and junior developers was described as a meaningful error-rate driver:

> *"If you are a senior developer, you are likely proficient enough to catch up within a couple of months. However, in terms of technical improvements, junior or mid-level developers may require additional coaching or guidance from senior developers to catch up. This support helps them understand the best paradigms and practices used in a particular programming language."*
>
> **- CD08**

Code ownership and responsibility were raised as a practice dimension with direct quality implications:

> *"At the end of the day, the developer is the one putting their signature on the code and committing it. Therefore, the responsibility of the specific code written falls on the developer. However, you should never commit code without fully understanding what you are writing."*

- CD01

These findings corroborate Jayakody and Wijayanayake's (2021) documentation of expertise gaps in DevOps teams and extend them by identifying business domain knowledge not only technical knowledge as a critical expertise dimension. The implication for organizations is that developer training investments need to span both technical tooling and domain understanding to be effective at reducing error rates.

4.3 Theme 3: System Architecture and Oversight Failures

Poor system architecture was identified as a third technological root of software errors, with 15 coded references across participants. What distinguishes this theme analytically is its focus on the upstream conditions that determine how errors arise in the first place. Architecture documents serve as the blueprint for development; when those blueprints are flawed, the errors they produce are systematic and compounding rather than incidental.

Participants identified system complexity particularly the involvement of multiple languages, frontend-backend integrations, and diverse toolsets as a primary architectural error driver:

> *"There are several factors contributing to software errors on the technology side, primarily due to the complexity of systems. As systems become more complex, the number of errors increases. The involvement of multiple technologies, including different programming languages, frontend and backend components, and various tools, further adds to this complexity."*
>
> **- CD06**

The absence of error monitoring infrastructure emerged as a distinct architectural oversight:

> *"It can be any type of software error that may lead to other issues. When you see an error in a product, you must address it as a developer. This happens effectively only when proper monitoring, alerting, and logging are in place."*
>
> **- CD03**
>
> *"Errors will occur; developers must have some kind of visibility to fix those errors. If you do not have visibility on the monitoring and logging to get those details, you will not be able to address that error, so you need to have those tools in place."*
>
> **- CD05**

Technical debt the accumulated consequence of deferred architectural improvements was raised as an error-amplifying condition:

"I mean, undefined data, null data types, different data types, checking these properly can help. If we do error handling properly, we can avoid causing the errors, at least avoid the server errors, application crash errors, and null checking problems at the API level."

- CD09

These findings support Taylor's (2019) analysis of architectural deficiency as a primary error source and Tian et al.'s (2022) documentation of how specification ambiguities produce developer-assumption errors. The study adds a practitioner-level account of how absent monitoring systems and accumulated technical debt amplify the consequences of architectural inadequacy.

4.4 Theme 4: The Role of Modern Technologies and Automation

Modern AI-assisted development tools, particularly code generation tools like GitHub Copilot emerged in participant accounts as a source of both productivity gains and new error risks. All participants addressed this theme, generating 12 coded references. The dominant pattern was not that AI tools are universally harmful or beneficial, but that their impact on error rates depends substantially on how developers employ them: applied with critical understanding, they reduce errors; applied uncritically, they introduce new categories of error that standard testing processes are not designed to detect.

"They gave senior developers a trial license for GitHub Copilot, asking us to use it and provide feedback on productivity improvements. However, they instructed us not to include the generated code in the project. Our leaders warned us against using the generated code, as it could contain errors."

- CD08

"When we ask AI to write a database query for retrieving users from XYZ where the class equals a specific value, it provides the code. But leaders have advised against using it directly. Instead, they recommended thoroughly reviewing the AI-generated code to ensure that no major bugs or side effects are being introduced with AI code in production."

- CD08

The counterpoint that automation, implemented correctly, reduces errors was articulated with equal conviction:

"We also need to implement automation. Once everything is automated, it will reduce many things. Initially, it will be difficult, but going forward, if we use the automation and integrate AI or other controls for these tasks, it will help. For instance, when you commit the code, it will automatically check for errors."

- CD06

4.5 Theme 5: Challenges Posed by Legacy Systems and External Integrations

Legacy systems were described by participants as among the most challenging and persistent sources of technological error not because they are inherently more error-prone than modern systems, but because the conditions surrounding them make errors both more likely and harder to address. Fifteen coded references across participants addressed this theme. The characteristic features of legacy system environments in participant accounts were absent or inaccurate documentation, scarce specialist knowledge, and deep organizational dependency that resists replacement.

> *"All those transactions are currently sitting on a store system where the store is polling all the transactions. It is a kind of a legacy application, written probably in the year 2000 or even before. The problem is, once we start migrating the legacy application into a newer one, the real challenge begins. To migrate, you first need to know the end-to-end understanding of that system."*
>
> **- CD03**
>
> *"Some of the things we use are legacy systems, which were built using old technology. Depending on how we need them to work, we rely on those language skills and address the kinds of errors that need fixing now. Only those specific skilled persons who have worked on the system can fix the issues."*
>
> **- CD06**
>
> *"They are not willing to address technical debt. Instead of modifying the existing legacy system, they prefer to leave it untouched and build new systems from scratch."*
>
> **- CD03**

These findings align closely with Alexandrova and Rapanotti's (2019) characterization of legacy maintenance as a domain where standard software engineering assumptions break down, and with Pashchenko et al.'s (2020) analysis of third-party library dependency risks. What participant accounts add is an organizational dimension: legacy system difficulties persist not only because of technical complexity but because organizational incentives consistently favor avoidance over remediation.

5. DISCUSSION

5.1 The Technological Error Origin Framework (TEOF)

The five themes identified in this study form an empirically grounded framework for understanding how technological deficiencies produce software errors in what this paper terms the Technological Error Origin Framework (TEOF). The TEOF is not a list of independent error causes; it is a relational architecture in which each theme interacts with and amplifies the others.

Theme 1 (Tool Inadequacies) and Theme 2 (Expertise Gaps) operate as proximate error origins the immediate conditions under which developers introduce errors at the coding stage. Theme 3 (Architecture Failures) functions as both an enabler of Themes 1 and 2 and an independent error source: poorly designed systems create environments where tool limitations and expertise gaps cause more damage than they would in well-structured codebases. Theme 4 (Modern Technology Adoption) intersects with Theme 1 in a specific way new tools that lack mature testing infrastructure compound the inadequacy patterns of Theme 1, while AI code generation creates novel error categories that Theme 2 expertise gaps fail to catch. Theme 5 (Legacy Systems) operates as a persistent background condition that constrains all other themes: legacy constraints limit the tools developers can use, demand expertise that is increasingly scarce, resist architectural improvement, and create integration targets that confound modern toolchains.

Figure 1: Technological Error Origin Framework (TEOF) Relational Map

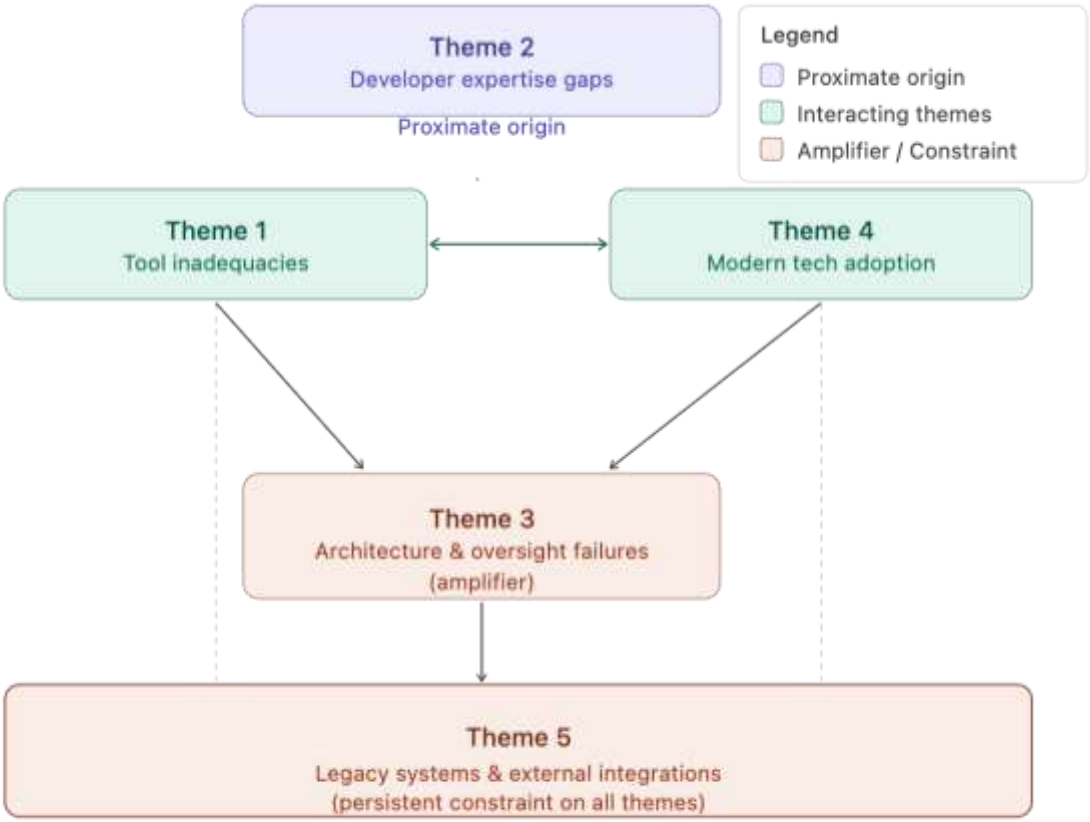

Figure 1. Relational architecture of the Technological Error Origin Framework (TEOF) showing how five technological deficiency themes interact to produce software errors.

Figure 1. Relational architecture of five themes illustrating how technological deficiencies interact to produce software errors.

5.2 Alignment with and Contribution Beyond Existing Literature

The TEOF corroborates existing literature across multiple dimensions while contributing insights that quantitative defect analyses cannot provide. Lopez et al.'s (2021) documentation of tool transition friction, Bluemke and Malanowska's (2021) analysis of testing challenges, Taylor's (2019) architectural deficiency framework, and Alexandrova and Rapanotti's (2019) legacy system characterization all find support in the participant accounts.

First, the study identifies business domain knowledge as an underappreciated expertise dimension. Existing literature on developer expertise focuses predominantly on technical skills programming language proficiency, framework familiarity, debugging capability. Participant accounts highlight that errors frequently originate not in technical ignorance but in domain ignorance: developers implementing business logic they do not understand well enough to anticipate the edge cases it requires.

Second, the study captures the AI code generation paradox empirically. Prior research has documented AI tool capabilities and limitations in controlled settings; participant accounts describe how the adoption of AI tools plays out in actual organizational development environments, including the role of organizational governance in mediating the error-introduction risk.

Third, the study identifies organizational avoidance as a mechanism that perpetuates legacy system error risks. Participants described organizations that systematically choose not to address legacy technical debt preferring to build around old systems rather than modernize them. This avoidance behavior extends the period during which legacy constraints compound errors in all other themes.

5.3 Implications for Practice

For technology leaders and software architects, the TEOF suggests a set of prioritized interventions. Tool investment should be directed toward IDEs with robust real-time error detection, testing frameworks that support cloud-native environments, and code health monitoring systems that operate continuously rather than as point-in-time checks. The black-box testing problem associated with cloud infrastructure tools is a structural issue that requires organizational workarounds dedicated cloud test environments, investment in infrastructure-as-code testing frameworks, and explicit testing protocols for third-party service integrations.

Developer upskilling programs should address both technical and domain dimensions. The expertise gap documented in this study is not adequately addressed by technical training alone; developers who lack business domain understanding will continue to introduce logic errors regardless of how proficient they become with their tools. Architecture governance warrants investment proportional to its error-amplification role. Mandatory architecture review processes, error monitoring as a non-negotiable deployment requirement, and technical debt tracking systems that make deferral visible to leadership are the structural interventions the architectural theme most directly calls for.

5.4 Limitations

Several limitations merit acknowledgement. The sample is restricted to U.S.-based developers in mobile and web application development, limiting applicability to embedded systems, operating system development, and non-U.S. organizational contexts. LinkedIn recruitment may skew toward professionally active developers, potentially underrepresenting those in more constrained or specialized roles. The experiential nature of the data means findings represent developer interpretations of technological conditions rather than objective measurements of error rates.

6. CONCLUSION

This study set out to understand the technological conditions that, in the daily professional experience of working developers, produce software errors. The accounts of twelve highly experienced practitioners averaging 14.5 years of professional experience and drawn from diverse organizational sectors converge on a consistent and actionable picture. The Technological Error Origin Framework advanced here identifies tool inadequacies, developer expertise gaps, architectural failures, modern technology adoption risks, and legacy system constraints as the five principal technological roots of software errors. These are not independent problems; they interact and amplify each other in ways that make any single-point intervention insufficient.

Three contributions stand out. It offers the first developer experience grounded, phenomenologically anchored account of the technological roots of software errors within the TOE framework, enriching a literature dominated by quantitative defect analyses; it identifies business domain knowledge as a distinct expertise dimension that technical training programs systematically neglect; and it documents organizational avoidance of legacy technical debt as a behavior pattern with systemic consequences for software error rates, offering a practitioner-level diagnosis that governance researchers and technology leaders can act on. Future research should pursue longitudinal studies tracking error rate changes through tool transitions, mixed-methods designs quantifying the organizational cost of the identified expertise gaps, sector-specific investigations of legacy system modernization strategies, and empirical assessments of AI code generation reliability in enterprise production environments. The last of which is especially urgent, since the evidence here suggests organizational governance of AI-assisted coding is currently underdeveloped relative to the error risks these tools introduce when used without sufficient critical understanding. Ultimately, software errors are not eliminated by any single technological improvement but are produced by the interaction of tools, expertise, architecture, technology adoption patterns, and legacy constraints; the TEOF offers an empirically grounded framework for understanding this relational system and directing organizational intervention where it is most needed.

REFERENCES

Akbar, M. A., Smidt, H., Shafiq, M., Alwageed, H., Pitafi, A. H., & Humayun, M. (2022). An empirical investigation of software technical debt. Journal of Software: Evolution and Process, 34(6), e2454. https://doi.org/10.1002/smr.2454

Alexandrova, T., & Rapanotti, L. (2019). Legacy system replacement: Understanding the risks. Information Systems Management, 36(1), 54–67. https://doi.org/10.1080/10580530.2018.1537496

Baker, J. (2011). The technology-organization-environment framework. In Y. Dwivedi, M. Wade, & S. Schneberger (Eds.), Information systems theory (Vol. 28, pp. 231–245). Springer. https://doi.org/10.1007/978-1-4419-6108-2_12

Bluemke, I., & Malanowska, A. (2021). On the difficulty of test case selection. Symmetry, 13(11), 2193. https://doi.org/10.3390/sym13112193

Braun, V., & Clarke, V. (2022). Thematic analysis: A practical guide. SAGE Publications.

Byrne, D. (2022). A worked example of Braun and Clarke's approach to reflexive thematic analysis. Quality & Quantity, 56, 1391–1412. https://doi.org/10.1007/s11135-021-01182-y

Castro, L., Galster, M., & Goedicke, M. (2023). Technical debt in software architecture: A systematic mapping study. Journal of Systems and Software, 197, 111601. https://doi.org/10.1016/j.jss.2023.111601

Creswell, J. W., & Creswell, J. D. (2018). Research design: Qualitative, quantitative, and mixed methods approaches (5th ed.). SAGE Publications.

Frechette, J., Bitzas, V., Aubry, M., Kilpatrick, K., & Lavoie-Tremblay, M. (2020). Capturing lived experience: Methodological considerations for interpretive phenomenological inquiry. International Journal of Qualitative Methods, 19. https://doi.org/10.1177/1609406920907254

Govindaras, B., Hussain, A., Tengku Wook, T. S. M., & Ahmad, W. F. W. (2023). Challenges in software testing for complex integrated systems. IEEE Access, 11, 45621–45639. https://doi.org/10.1109/ACCESS.2023.3274189

Hennink, M., & Kaiser, B. N. (2022). Sample sizes for saturation in qualitative research: A systematic review of empirical tests. Social Science & Medicine, 292, 114523. https://doi.org/10.1016/j.socscimed.2021.114523

Horani, O., Suleiman, H., & Tal, I. (2023). A systematic review of TOE framework for enterprise digital adoption. IEEE Access, 11, 31922–31949. https://doi.org/10.1109/ACCESS.2023.3261867

Jayakody, J. A. S. K., & Wijayanayake, W. M. J. I. (2021). Developer-perceived challenges in adopting DevOps toolchains: A qualitative study. Journal of Systems and Software, 178, 110982. https://doi.org/10.1016/j.jss.2021.110982

Karcher, S., Kirilova, D., Andreata, A., & Sherif, N. (2024). Reflexivity in qualitative research. International Journal of Qualitative Methods, 23. https://doi.org/10.1177/16094069241238049

Khaliq, A., Shahzad, B., & Saleem, M. A. (2023). Limitations of AI-based test case generation in enterprise software. IEEE Transactions on Software Engineering, 49(4), 2241–2258. https://doi.org/10.1109/TSE.2022.3225442

Khan, M. S., Fernandez-Crehuet, J. M., & Dabbagh, M. (2022). Perceived ease of use and developer tool adoption: An empirical study. Journal of Software: Evolution and Process, 34(9), e2465. https://doi.org/10.1002/smr.2465

Krasner, H. (2021). The cost of poor software quality in the US: A 2020 report. Consortium for Information & Software Quality (CISQ). https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2020-report/

Kurian, J., & Thomas, A. (2023). Testing challenges in microservices-based software architectures. ACM Computing Surveys, 55(14), 1–36. https://doi.org/10.1145/3568024

Li, F., Liang, P., & Avgeriou, P. (2022). A systematic mapping study on technical debt and its management. Journal of Systems and Software, 193, 111477. https://doi.org/10.1016/j.jss.2022.111477

Lincoln, Y. S., & Guba, E. G. (1985). Naturalistic inquiry. SAGE Publications.

Lopez, M., Medina, L., & Morales-Salazar, E. (2021). Developer challenges in tool and framework adoption: An empirical study. IEEE Access, 9, 82167–82184. https://doi.org/10.1109/ACCESS.2021.3086549

Negri-Ribalta, C., Noei, E., & Adams, B. (2024). Security risks in AI-generated code: An empirical study. In Proceedings of the 46th International Conference on Software Engineering (ICSE), pp. 1–12. https://doi.org/10.1145/3597503.3639177

Neubauer, B. E., Witkop, C. T., & Varpio, L. (2019). How phenomenology can help us learn from the experiences of others. Perspectives on Medical Education, 8, 90–97. https://doi.org/10.1007/s40037-019-0509-2

Nugraha, F., & Talita, A. S. (2021). Scalability challenges in modern distributed software systems. Journal of Physics: Conference Series, 1803, 012003. https://doi.org/10.1088/1742-6596/1803/1/012003

Panthi, V., Mohapatra, D. P., & Mall, R. (2022). Software testing: A survey of testing practices and challenges. International Journal of Software Engineering and Knowledge Engineering, 32(3), 391–428. https://doi.org/10.1142/S0218194022500127

Pashchenko, I., Vu, D. L., & Massacci, F. (2020). A qualitative study of dependency management and its security implications. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (pp. 1117–1130). https://doi.org/10.1145/3372297.3417232

Soui, M., & Haddad, R. (2023). Automated GUI test case generation: Challenges and opportunities. ACM Computing Surveys, 55(12), 1–42. https://doi.org/10.1145/3565472

Taylor, R. N. (2019). Software architecture: Foundations, theory, and practice. Wiley.

Tian, Y., Zhang, F., & Hassan, A. E. (2022). Software architecture smells: A systematic mapping study. Journal of Systems and Software, 193, 111481. https://doi.org/10.1016/j.jss.2022.111481

Xu, F. F., Alon, U., Neubig, G., & Hellendoorn, J. (2022). A systematic evaluation of large language models of code. In Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (pp. 1–10). https://doi.org/10.1145/3520312.3534862

Zardari, S., Bhutto, A., & Rehman, M. (2022). Test case selection and prioritization: A systematic review. IEEE Access, 10, 32987–33004. https://doi.org/10.1109/ACCESS.2022.3161234.