MIS · TECHNICAL RESEARCH PAPER

Engineering Secure, Maintainable and AI-Ready Software

A technical research paper on modern programming, secure coding, software quality, code automation and AI-assisted development

Software engineering · source-code quality · secure coding · software supply chain · AI-assisted development

FIELD	ENTRY
Research domain	Software — Programming and Coding
Organisation	Al Moammar Information Systems Co. (MIS), Kingdom of Saudi Arabia
Author of record	Shaker Alzoubi
Document type	Technical research paper / white paper
Version	1.1
Publication date	24/07/2026
Classification	Public
Permalink	Public

ABSTRACT

Software has moved from relatively self-contained codebases to interconnected systems assembled from application services, programming interfaces, open-source dependencies, cloud-native components, automated delivery pipelines and, increasingly, code generated or assisted by artificial intelligence.

That change has raised development speed and technical capability. It has also introduced new engineering problems in software security, maintainability, dependency governance and traceability.

This paper examines one question: how can an organisation develop software rapidly while preserving security, maintainability, quality and control across the whole software lifecycle? The study applies a structured qualitative engineering method drawing on recognised software-quality, secure-development, application-security and software-supply-chain guidance, together with the national cybersecurity control environment of the Kingdom of Saudi Arabia.

Eight engineering dimensions are analysed: coding architecture, secure coding, code quality, automated verification, testing, dependency and supply-chain governance, controlled delivery, and AI-assisted development.

The analysis indicates that software quality cannot be established by whether code executes correctly or satisfies its immediate functional requirements. Software that is expected to last must stay understandable and testable, and must absorb change without disproportionate engineering risk.

The paper proposes a Secure and Sustainable Software Engineering Framework organised on a nine-stage lifecycle, intended as a technology-neutral model for enterprise and government software environments.

define → design → code → verify → secure → integrate → release → observe → improve

KEYWORDS *software engineering; programming; source code; secure coding; software quality; DevSecOps; software supply chain; software bill of materials; code review; automated testing; AI-assisted development.*

01 Introduction

Programming is the process by which architecture and business requirements are turned into executable digital capability. Modern programming, however, no longer consists mainly of writing instructions in a programming language.

A production system in an enterprise or government environment typically combines internally developed source code, open-source libraries, commercial third-party packages, programming interfaces, databases, cloud services, containers, infrastructure defined as code, automated build pipelines, security controls and — increasingly — machine-generated code.

The delivered software is therefore a network of code, dependencies, configuration, identities and automated processes, not a single authored artefact. That shift creates the engineering problem this paper addresses.

Organisations want software teams to deliver new capability quickly. Raising delivery speed without matching engineering controls introduces security weakness and technical debt, destabilises dependencies, and opens exposure through the software supply chain.

The question is accordingly not how developers can write more code. It is how an organisation can produce software efficiently while ensuring that the resulting code stays secure, maintainable, testable and governable for as long as it remains in service.

02 Research problem

Software can satisfy every functional requirement while carrying significant hidden engineering risk. The recurring forms are:

- Excessive code complexity and duplicated logic;
- Insecure coding patterns and weak input handling;
- Outdated or unmonitored dependencies;
- Credentials embedded in source repositories;
- Insufficient automated testing;
- Inconsistent coding standards and weak code review;
- Undocumented interfaces and uncontrolled third-party components;
- Deployment processes that cannot be reproduced;
- Machine-generated code accepted without verification.

None of these necessarily prevents an initial deployment. They surface later, as security incidents, production defects, rising maintenance cost, slow change cycles or dependency failure.

> **Successful execution is not evidence of sound software engineering. The problem is how programming and coding practice can be engineered so that software stays secure, maintainable, verifiable and adaptable as its functionality and its dependency ecosystem change.**

03 Research question

The principal research question is:

> **What engineering controls and programming practices enable modern enterprise software to remain secure, maintainable, testable and adaptable throughout its lifecycle?**

The supporting questions are:

1. What characteristics distinguish sustainable source code from merely functional code?

2. How should security requirements shape programming practice?
3. How do code review and automated testing reduce software risk?
4. How should third-party and open-source dependencies be governed?
5. What role should automated analysis play before software is released?
6. How should AI-generated or AI-assisted code be controlled?
7. How should operational feedback improve subsequent releases?

04 Research objectives

The study sets out to:

(i) define programming as a controlled engineering lifecycle rather than an isolated coding activity;

(ii) identify the characteristics of secure and maintainable source code;

(iii) analyse secure coding as part of development rather than as an activity appended to it;

(iv) examine automated testing and static code analysis as sources of evidence;

(v) evaluate dependency and software-supply-chain risk;

(vi) examine the governance requirements introduced by AI-assisted programming;

(vii) consolidate the analysis into an integrated engineering framework.

05 Background: from coding to software engineering

Functional code answers one question: does the software perform the intended function? Software engineering adds a set of questions that the first one cannot reach.

QUESTION	ENGINEERING PROPERTY IT TESTS
Can it be changed safely?	Modularity and change isolation
Can another engineer understand it?	Readability and structure
Can its behaviour be tested?	Testability
Can its dependencies be trusted?	Supply-chain visibility
Can weaknesses be identified before release?	Verifiability
Can a release be reproduced?	Delivery control
Can a change be traced to a requirement?	Traceability

These properties become more important as software grows, because a small coding decision repeated across a large system becomes an architectural constraint. Software quality therefore begins at the level of source code but is decided across the whole lifecycle.

06 Literature and standards review

Recognised software-engineering guidance converges on one principle: security and quality should be built into development, not inspected in afterwards. Six source families are relevant to this study.

6.1 Software product quality

The international product-quality model defines software quality as a set of characteristics with associated sub-characteristics that can be specified, measured and evaluated, instead of collapsed into a single judgement about whether the software works. Its current edition defines nine such characteristics and supersedes the 2011 text [ISO 25010 :

2023]. Functional suitability is one characteristic among nine, which is the point of the model: a product can score well on function and poorly on maintainability, security or reliability at the same time.

6.2 Secure software development

The Secure Software Development Framework organises secure-development practice into four groups — preparing the organisation, protecting the software, producing well-secured software, and responding to vulnerabilities — and is written to sit inside an existing lifecycle, not to run as a separate final activity [SSDF 1.1 : 2022]. A draft revision extends the framework as version 1.2 [SSDF 1.2 draft : 2025]; its public comment period has closed, and it is cited here as draft guidance only.

> **Finding vulnerabilities and developing software so that fewer vulnerabilities are introduced are different activities. The second is the more consequential, and the first cannot substitute for it.**

6.3 Application-security verification

The Application Security Verification Standard provides structured requirements for evaluating the security of applications and related services, usable for measuring assurance, guiding developers of security controls, and specifying security requirements in procurement. Its current stable release is version 5.0.0 [ASVS 5.0 : 2025]. This carries a practical consequence for how security requirements are written: a requirement that software be secure cannot be tested, whereas a defined authentication, authorisation, validation, session-management or cryptographic requirement can be. A requirement that cannot be verified is a statement of intent, not a control.

6.4 Prevalent application-security weakness

The current edition of the widely used application-security risk list is constructed from testing data covering approximately 2.8 million applications contributed by thirteen organisations, together with a community survey, and maps 248 weakness classes into ten categories [Top 10 : 2025]. Two features of that edition matter to this paper. Software supply-chain failure appears as a top-three category in its own right, and integrity failure in software or data appears as a separate category. Both concern components an organisation incorporates, not code it writes.

6.5 Software component inventory

Guidance on the minimum elements of a software bill of materials defines three requirement categories — recorded data fields, support for automation, and the practices and processes governing how such records are requested, generated and used — and names seven data fields for each component: supplier name, component name, version, other unique identifiers, dependency relationship, author of the record and a timestamp [SBOM minimum elements : 2021]. The dependency-relationship field is the one that distinguishes an inventory from a list, because it is what records how a component came to be present.

6.6 The national control context

Software development in the Kingdom operates inside a defined control environment. Controls issued by the National Cybersecurity Authority bear on secure development, change control, logging, access management, and the classification and protection of data handled by applications [NCA DCC : 2022] [NCA ECC : 2024]. Where an application is deployed on cloud infrastructure, the Authority's cloud controls apply in addition [NCA CCC : 2020]. Applications that process personal data are further governed by the Personal Data Protection Law [PDPL : 2021], and development that introduces generative capability sits inside the national artificial-intelligence ethics principles issued by the Saudi Data and Artificial Intelligence Authority [SDAIA AI Ethics : 2023]. Addressed during design they are requirements; discovered at audit they are findings, and remediation at that point costs considerably more.

07 Research methodology

The study applies a structured qualitative software-engineering and risk analysis. It makes no claim of experimental benchmarking of languages, frameworks, tools or teams, and it analyses no confidential source code. It proceeds in five stages.

Stage 1 — Lifecycle decomposition

The lifecycle was decomposed into requirements, architecture, coding, verification, integration, release, operation and improvement, so that each control could be attached to the stage where it is effective, not the stage where it is convenient.

Stage 2 — Risk identification

Programming risk was classified by origin into code-quality risk, security risk, dependency risk, testing risk, configuration risk, release risk and maintainability risk.

Stage 3 — Engineering-control mapping

Recognised practices from the sources reviewed in clause 06 were mapped against those risk categories, and gaps recorded where a risk category had no corresponding control.

Stage 4 — Modern development analysis

The study then examined how automation, integrated security and operations practice, and AI-assisted coding change the effect of the traditional controls — separating activities suitable for automation from decisions requiring engineering judgement.

Stage 5 — Framework construction

The results were consolidated into the framework presented in clause 09.

08 Technical analysis: eight engineering dimensions

The analysis is organised as eight dimensions, lettered A to H. Each is stated as an engineering position instead of a preference, and each carries the control that gives effect to it.

8.A CODING ARCHITECTURE

Designing for change

Source code is read and modified far more often than it is originally written. Programming should therefore be optimised for future understanding as well as for present execution. The characteristics that carry that weight are clear responsibilities, meaningful naming, limited duplication, bounded complexity, defined interfaces, controlled dependencies and documentation of the assumptions a reader cannot infer.

maintainable code = understandable structure + controlled dependencies + predictable behaviour

The relation above is a conceptual expression proposed by this research and not a measured formula.

Modularity and the reach of change

In a tightly coupled system a local change produces distant consequences. A modular system separates responsibilities so that each layer interacts through a controlled interface:

user interface → application services → domain logic → data and integration services

The engineering objective is change isolation, not the largest possible number of components. Where a modification to one module obliges modification of unrelated modules, the boundaries are nominal, and both maintainability and testability suffer for it.

Controlling complexity

Complexity is not inherently a defect; a complex business problem may require complex software. The engineering risk arises when implementation complexity exceeds problem complexity without cause — visible as deeply nested logic, excessively long functions, duplicated conditional structures, hidden side effects and dense cross-module dependency. Unnecessary complexity raises the effort required to understand the software, and code that is hard to understand is hard to secure. Complexity is therefore a security cost as much as a maintenance cost.

8.B SECURE CODING

Security begins before code

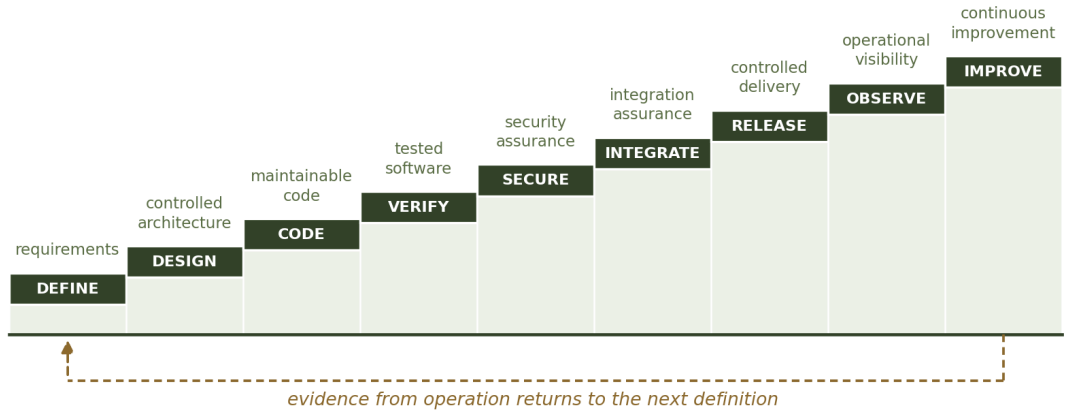
Security weakness more often originates in design decisions than in individual programming errors. Secure development therefore begins when security requirements are written, not when a scanner runs, and it follows a sequence in which each step constrains the next.

security requirements → threat analysis → secure architecture → secure coding → verification → monitoring

The framework reviewed above recommends integrating secure-development practice into the lifecycle instead of appending it. Engineers should also understand the security assumptions attaching to the code they implement, because an assumption that is not stated is an assumption that will be broken by a later change.

Figure 1 sets the nine engineering stages against the outcome each is required to produce.

Figure 1 — The nine engineering stages and the outcome each must produce



Each stage is a gate with an outcome that can be shown. A stage that produces no evidence has not been passed, only completed.

Treatment of external input

External input may arrive from users, programming interfaces, files, databases, external services or devices. It should be validated against its expected type, format, range, size and context of use. Injection and the mishandling of exceptional conditions both appear among the ten most prevalent application-security weakness categories in current data, which is the practical case for validating at the boundary and not at the point of use.

Secrets and source code

Passwords, programming-interface keys, cryptographic material and other credentials should not be embedded in source repositories. Secrets require a lifecycle of their own: secure storage, access control, rotation, revocation and auditing. A private repository is an access-control measure, not a secrets-management system, and treating it as one converts a configuration decision into a permanent exposure — because a credential committed to history remains in that history after it is deleted from the working tree.

8.C CODE QUALITY AND REVIEW

Review as quality control and knowledge transfer

Code review provides independent examination before a change becomes part of the production codebase. It can evaluate logic, maintainability and security, and it can question architectural fitness, test adequacy and dependency changes. Its purpose is not only to find error.

Review also distributes understanding. Where software is reviewed by several engineers, knowledge of the system is not concentrated in one person, and the organisation's ability to change that software does not depend on that person's availability. For long-lived enterprise software this second function is frequently the more valuable of the two.

Enforcement of coding standards

A documented coding standard that is not applied at review or in the pipeline is a statement of intent. The engineering value of a standard comes from the point at which it is checked, which is why the standards worth writing are the ones that can be checked.

8.D AUTOMATED VERIFICATION

Automation repeats; it does not judge

Modern development depends on automated controls in the delivery pipeline. A mature pipeline runs a defined sequence in which each step can fail the change:

commit → build → static analysis → dependency analysis → unit tests → security testing → integration tests → artefact

Automation removes the variability of manual execution and makes the same checks run on every change. But automated tools apply defined rules and detect defined patterns; they do not independently judge architectural fitness or business-logic correctness. Automation should therefore augment engineering judgement, never stand in for it, and a pipeline that passes is evidence that the defined checks passed, which is a narrower claim than it looks.

8.E TESTING AS EVIDENCE

Test categories and the questions they answer

Testing converts assumptions about behaviour into repeatable evidence. The categories are not interchangeable, and a high count in one does not compensate for the absence of another.

TEST CATEGORY	QUESTION IT ANSWERS
Unit	Does this component behave correctly in isolation?
Integration	Do components interact correctly across their interfaces?
Regression	Did this change alter behaviour that previously worked?
Security	Does the implementation resist the threats stated in the requirements?
Performance	Does the software behave acceptably under expected load?
Acceptance	Does the software satisfy the stated requirement, or only the assumed one?

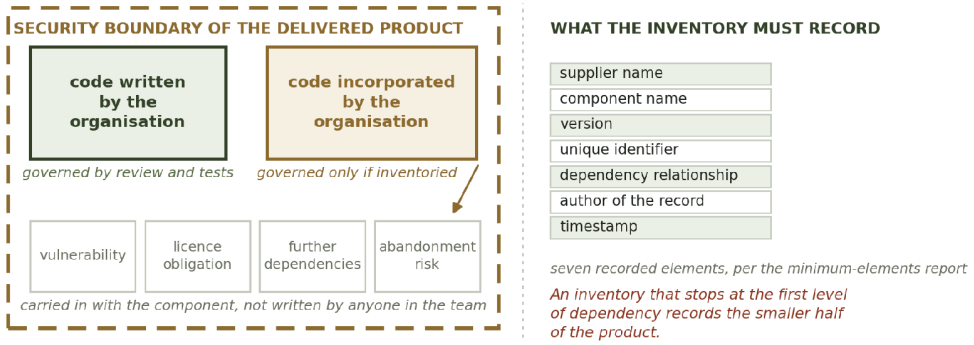
Testing is accordingly part of the engineering feedback loop, not a final quality-control activity. What the suite covers matters more than how much of it there is.

8.F DEPENDENCIES AND THE SOFTWARE SUPPLY CHAIN

Dependencies as part of the product

A modern application rarely consists entirely of internally written code; it may depend on hundreds of external packages, each of which may depend on others. This produces a distinction that governs the rest of this dimension: code written by the organisation and code incorporated by the organisation both determine the security of the delivered product, but only the first is visible by default. Figure 2 sets out that distinction and the inventory elements that make the incorporated half visible.

Figure 2 — The security boundary of a delivered product is wider than the code written inside it



An organisation is answerable for the security of code it incorporated as much as for code it wrote. The difference is that only one of the two is visible by default.

Using a library transfers functionality into the product and imports risk with it: vulnerability, licensing obligation, compatibility constraint, abandonment, and further dependencies of its own. Dependency governance should therefore define approved sources, version control, vulnerability monitoring, update policy, licence review and replacement planning.

An inventory is only as good as its depth

A component inventory should record, for each component, the seven elements set out in clause 6.5, and should reach transitive dependencies, not stop at those declared directly. The dependency-relationship element is what makes the record an inventory and not a list. Components that go unnamed are the ones whose risk never gets examined.

Integrity beyond the repository

The source repository is one stage in software production. The full chain runs from developer to repository, build system, dependencies, artefact repository, deployment pipeline and production, and compromise at any stage reaches the delivered software. Software security accordingly covers not only source code but the systems that transform source into a deployable artefact.

8.G CONTROLLED DELIVERY AND TRACEABILITY

Integrated development, security and operations

Traditional delivery can separate development, security and operations into sequential teams, which places security and operational requirements after the decisions they should have shaped. Integrating the three moves those requirements ahead of release instead of behind it. The engineering result to look for is not a named practice but a specific capability: security and operational requirements affecting software while it can still be changed cheaply.

Traceability and investigation

A controlled lifecycle can answer, for any line in production: who changed it, why, against which requirement, who reviewed it, which tests ran, which version shipped, and which dependencies were included. The chain that answers those questions is:

requirement → change → review → test → build → release

Traceability creates accountability in ordinary operation and is the difference between investigating an incident and reconstructing it from memory. In regulated and government environments it is also the evidence that a control was applied and not merely adopted.

Reproducibility of releases

Reproducible builds and recorded component inventories are what make a delivered artefact accountable. Where a deployment cannot be reproduced from a recorded state, the organisation cannot establish what it actually shipped, and every assurance statement about that release rests on recollection.

8.H AI-ASSISTED DEVELOPMENT

Speed changes; accountability does not

Machine assistance accelerates code generation, refactoring, documentation, test creation, debugging and code explanation. It can raise engineering productivity materially. Generated code can nevertheless contain logical error, insecure pattern, inefficiency, reference to an interface that does not exist, an inappropriate dependency, an unexamined licence position, or an assumption its author never stated because it had no author.

Generated code should therefore pass the same controls as written code, and in security-sensitive contexts stronger ones. Figure 3 states the resulting engineering position.

Figure 3 — Five controls between a machine suggestion and production code

Machine assistance shortens the distance from requirement to candidate code. It does not shorten the distance from candidate code to trusted code.

> **Machine assistance may accelerate code creation. It does not transfer engineering accountability away from the organisation that deploys the result.**

Guidance extending secure-development practice to generative systems treats their introduction as a change to the security problem across the lifecycle rather than as a new tool inside it [SSDF AI : 2024]. Its relevance is not confined to organisations that build models: it establishes that introducing generative capability into a development process is itself a change in the security problem. In the Kingdom, that engineering position sits alongside the national artificial-intelligence ethics principles, which require documented human oversight and accountability for automated decision-making [SDAIA AI Ethics : 2023].

Human accountability

A model can generate code, a pipeline can test it, and automation can deploy it. Accountability for the result does not transfer to any of them. A responsible operating model names a code owner, a reviewer, a security owner, a release authority and an operational owner; and for high-impact systems human approval should remain mandatory for architectural decisions, security-sensitive code, destructive operations, privileged access and critical releases. Automation can execute a responsibility; it cannot hold one.

09 The Secure and Sustainable Software Engineering Framework

The eight dimensions are expressed as a nine-stage lifecycle, each stage carrying an engineering question and the outcome that answers it.

STAGE	ENGINEERING QUESTION	REQUIRED OUTCOME	DIMENSIONS
Define	What must the software do, and what must it protect?	Stated requirements, including security requirements	B
Design	How are responsibilities separated?	Controlled architecture	A, B
Code	Is the implementation understandable and secure?	Maintainable code	A, B, C
Verify	Does evidence support the expected behaviour?	Tested software	D, E

STAGE	ENGINEERING QUESTION	REQUIRED OUTCOME	DIMENSIONS
Secure	Have weaknesses and dependencies been assessed?	Security assurance	B, F
Integrate	Can changes interact safely?	Integration assurance	E, G
Release	Is deployment repeatable and traceable?	Controlled delivery	F, G
Observe	How does the software behave in production?	Operational visibility	G
Improve	What should change on the evidence?	Continuous improvement	C, G, H

Expressed as a single statement, the framework holds that:

sustainable software = maintainable code + secure development + automated verification + dependency governance + traceable delivery + operational feedback + governed machine assistance

This is a conceptual outcome of the present research. It is not presented as an official standard or a regulatory formula, and it carries no authority beyond the analysis set out above.

10 Findings

The study produces eight findings.

finding 01 **Functional correctness is necessary and insufficient.** Software can execute correctly while remaining insecure, opaque or unmaintainable.

finding 02 **Maintainability is decided in structure, not in documentation.** Readability, bounded complexity and real module boundaries determine whether software can be changed safely; documentation describes a structure, it does not create one.

finding 03 **Security cannot be added after implementation.** Secure coding cannot fully compensate for insecure architecture, and late testing finds defects without undoing the decision that produced them.

finding 04 **Verification converts assumption into evidence.** Testing, static analysis, dependency analysis and security verification produce repeatable evidence; a passing pipeline evidences the defined checks and nothing wider.

finding 05 **Dependencies are inside the security boundary.** An organisation inherits the risk of every component it incorporates, and most of a modern product is incorporated, not written.

finding 06 **An inventory that stops at declared dependencies records the smaller part of the product.** Transitive components are where unexamined risk accumulates, and the dependency-relationship record is what makes them visible.

finding 07 **Traceability is the precondition of accountability.** An organisation that cannot reconstruct why a change was made, who reviewed it and what was tested will conduct incident analysis by inference.

finding 08 **Machine assistance shifts effort, not responsibility.** Generation changes where engineering time is spent and leaves accountability where it was; the control that matters is the gate, applied identically regardless of origin.

11 Discussion

Programming maturity is often measured by lines of code, number of engineers or delivery speed. Those measures describe activity. They do not describe quality, and an organisation that manages them will optimise them.

A more useful model asks whether the software is understandable, secure, testable, traceable, maintainable and observable — six properties that can each be evidenced. On that model the development objective changes from producing more code to producing trusted software capable of sustainable change.

Three shifts account for most of the difference between software that ages well and software that does not. The first is from authorship to assembly: when the majority of a product is imported, the discipline that determines its quality moves from how well the team writes to how well it selects, inventories, updates and replaces what it imports. The second is from detection to prevention: testing remains necessary, but an organisation whose entire security posture rests on finding defects has accepted that defects will be introduced at the rate its process allows. The third is from tool trust to verified trust: as generation, analysis and deployment become automated, the temptation is to treat the output of a trusted tool as trusted output, when the engineering position is the reverse — the tool is trusted to perform a step, and the output is trusted only once it has passed the control that applies to it.

12 Practical applications

The framework applies to new development, modernisation of existing systems, enterprise and government applications, programming interfaces, digital platforms, cloud-native systems, mobile applications, integration software, internally developed tools and AI-enabled applications. Used diagnostically, it reads as follows.

OBSERVED CONDITION	INDICATED GAP
No independent code review	Quality and security control gap (dimension C)
High complexity and duplication	Maintainability risk (dimension A)
Credentials committed to repositories	Secrets-management gap (dimension B)
Security requirements written for the penetration test	Secure-development gap (dimension B)
Tests present but concentrated in one category	Verification-coverage gap (dimension E)
No inventory of transitive dependencies	Supply-chain visibility gap (dimension F)
Builds not reproducible between environments	Delivery-control gap (dimension G)
Cannot reconstruct why a released change was made	Traceability gap (dimension G)
Generated code merged without the standard review	AI-governance gap (dimension H)

13 Applied context

Al Moammar Information Systems develops and maintains software for enterprise and government organisations, in environments where applications coexist with platforms, data systems, integration services and infrastructure. That practice is the applied context of this paper: it is where the distinctions set out above are met under delivery obligation instead of in principle.

The paper presents no delivered system as a research experiment, claims no research contribution from any particular engagement, and names no client, no vendor and no product. Those choices keep the paper within the boundary of research and keep commercial and client information out of a public document.

Placeholder for MIS input — delete this box before publication. One verified engineering case may be inserted here: the application or component; the client sector, anonymised unless written consent to name the client has been obtained; the engineering problem; the architecture and interface decisions; the development methodology; the source-control and code-review approach; the verification regime and its coverage; the security verification applied; the dependency-governance approach; the delivery pipeline; the use of machine assistance, if any; and the measurable outcome. Only facts the responsible MIS team can substantiate should be published, and no vendor or product should be named without that vendor's agreement.

14 Limitations

Five limitations should be stated. The study is a qualitative engineering analysis and includes no controlled experiment, benchmark or measurement of languages, tools or teams. No proprietary source code and no confidential client system were examined. Practice varies with system criticality, regulatory exposure, technology stack and organisational maturity,

so the dimensions describe engineering intent rather than a prescribed configuration; the paper deliberately prescribes no programming language, framework, cloud platform or development tool.

Further, one of the sources relied upon is a draft whose comment period has closed and which had not been issued in final form at the date of publication. And the treatment of AI-assisted programming addresses the engineering control model, not the behaviour of any specific generation system, which changes faster than a published paper can track.

15 Future research

Useful extensions include quantitative measurement of defect density against review coverage; the relationship between dependency depth and time to remediate a disclosed vulnerability; reproducible-build adoption and its effect on incident investigation; measurement of technical-debt interest in delivery terms; automated architecture-conformance checking; component-inventory maturity, including how far inventories reach in practice; the security properties of generated code compared with written code under identical review; and the effect of machine assistance on the distribution of engineering knowledge within a team.

The natural continuation of this study is an assessment instrument rather than a further paper:

AI-Assisted Secure Software Engineering Maturity Model

Such a model would score an organisation across machine-assistance usage, human oversight, code verification, security assurance, traceability and production governance, taking verification evidence, not process documentation, as the primary indicator.

16 Research conclusions

The study reaches eight conclusions, followed by an explicit answer to the research question.

16.1 Good software is engineered for change

Sustainability depends on how safely software can be understood and modified, not on whether its present functionality works. Quality characteristics that are not specified are not built.

16.2 Security begins before source code

Security requirements and architecture precede implementation, and a requirement that cannot be verified is not a control.

16.3 Verification converts assumption into evidence

Testing, static analysis, dependency analysis and security verification provide repeatable evidence about a specific set of defined properties.

16.4 External dependencies are inside the security boundary

An organisation inherits risk from every third-party and open-source component incorporated into its product, and that inheritance is not reduced by the component being widely used.

16.5 Inventory depth determines dependency governance

Recording declared dependencies without their transitive components records the smaller part of the product; the dependency relationship is the element that makes governance possible.

16.6 Traceability is fundamental to controlled delivery

An organisation should be able to connect requirement, change, review, test, build and release, and a release that cannot be reproduced cannot be assured.

16.7 Production operation is part of software engineering

Operational behaviour is evidence, and evidence that does not return to the next definition is discarded information.

16.8 Machine assistance does not remove engineering accountability

Generative capability can accelerate software creation; responsibility for correctness, security and suitability remains with the organisation deploying the code.

16.9 Answer to the research question

The research question asked what engineering controls and programming practices enable modern enterprise software to remain secure, maintainable, testable and adaptable throughout its lifecycle. On this analysis, the answer is as follows.

> **Sustainable software requires modular and understandable code; security integrated from requirements onward and expressed as verifiable requirements; independent human review together with automated verification; testing distributed across categories rather than concentrated in one; a component inventory that reaches transitive dependencies, with policy for update and replacement; traceable and reproducible delivery; operational feedback returned to the next definition; and machine assistance governed by the same gate as written code — with named human accountability that no automation assumes.**

No programming language, development framework or machine-assistance tool independently provides these characteristics. Engineering maturity should therefore be assessed by an organisation's ability to hold all eight dimensions at once, and to produce the evidence that each was held.

17 Conclusion

Programming remains at the centre of software development, but the definition of high-quality software has moved beyond the act of writing code. A modern application is assembled more than it is authored, verified more than it is inspected, and increasingly generated in part rather than typed in full — and each of those three shifts moves the determinants of quality away from the keyboard.

This paper examined that lifecycle through its risks and controls and consolidated the result into eight engineering dimensions, from coding architecture to machine assistance. The central conclusion is that trustworthiness is a property the lifecycle produces and the evidence demonstrates, not a property that successful execution implies.

The governing question at the end of a release is therefore not whether the software works, but whether the organisation can show why it should be trusted to keep working — and can show it for the code it wrote, the code it incorporated and the code it generated, on the same terms.

REFERENCES

Sources are cited in the text by the bracketed sigla in the first column.

SIGLUM	REFERENCE
ISO 25010 : 2023	International Organization for Standardization, ISO/IEC 25010:2023, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model, 2nd edition. Geneva: ISO/IEC, November 2023. Defines nine quality characteristics; supersedes ISO/IEC 25010:2011.
SSDF 1.1 : 2022	M. Souppaya, K. Scarfone and D. Dodson, Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities, NIST Special Publication 800-218. Gaithersburg, MD: NIST, February 2022.
SSDF 1.2 draft : 2025	Secure Software Development Framework (SSDF) Version 1.2: Recommendations for Mitigating the Risk of Software Vulnerabilities, NIST Special Publication 800-218 Revision 1, Initial Public Draft. Gaithersburg, MD: NIST, 17 December 2025; public comment period closed 30 January 2026. Cited as a draft; not issued in final form at the date of publication.
SSDF AI : 2024	H. Booth, M. Souppaya, A. Vassilev, M. Ogata, M. Stanley and K. Scarfone, Secure Software Development Practices for Generative AI and Dual-Use Foundation Models: An SSDF Community Profile, NIST Special Publication 800-218A. Gaithersburg, MD: NIST, July 2024.

SIGLUM	REFERENCE
ASVS 5.0 : 2025	OWASP Foundation, Application Security Verification Standard (ASVS), Version 5.0.0, released 30 May 2025. Wakefield, MA: OWASP Foundation.
Top 10 : 2025	OWASP Foundation, OWASP Top 10:2025 — the ten most critical application-security risk categories. Constructed from testing data covering approximately 2.8 million applications contributed by thirteen organisations together with a community survey; 248 weakness classes mapped across ten categories. Released November 2025 at the OWASP Global AppSec Conference, Washington, DC.
SBOM minimum elements : 2021	United States Department of Commerce, The Minimum Elements for a Software Bill of Materials (SBOM) Pursuant to Executive Order 14028 on Improving the Nation's Cybersecurity. Washington, DC: Department of Commerce, 12 July 2021.
NCA DCC : 2022	National Cybersecurity Authority, Data Cybersecurity Controls. Riyadh: NCA, Kingdom of Saudi Arabia, 2022.
NCA ECC : 2024	National Cybersecurity Authority, Essential Cybersecurity Controls, ECC-2:2024. Riyadh: NCA, Kingdom of Saudi Arabia, 2024. Supersedes ECC-1:2018.
NCA CCC : 2020	National Cybersecurity Authority, Cloud Cybersecurity Controls, CCC-1:2020. Riyadh: NCA, Kingdom of Saudi Arabia, 2020.
PDPL : 2021	Personal Data Protection Law, promulgated by Royal Decree No. M/19 of 09/02/1443H (16 September 2021), as amended. Riyadh: Saudi Data and Artificial Intelligence Authority (SDAIA), Kingdom of Saudi Arabia.
SDAIA AI Ethics : 2023	Saudi Data and Artificial Intelligence Authority, AI Ethics Principles, version 2. Riyadh: SDAIA, Kingdom of Saudi Arabia, 2023.