AL MOAMMAR INFORMATION SYSTEMS · TECHNICAL RESEARCH PAPER · APPLICATION DEVELOPMENT

Engineering Secure, Modular and Evolvable Enterprise Applications

Modern application development, API-first architecture, software quality and secure delivery

AUTHOR OF RECORD	Ehab Attar , Chief Technology officer
ORGANISATION	Al Moammar Information Systems Co. (MIS), Kingdom of Saudi Arabia
RESEARCH DOMAIN	Application Development
DOCUMENT TYPE	Technical research paper / white paper
VERSION	1.0
DATE OF PUBLICATION	05 July 2026
PERMANENT LINK	[to be assigned on the official MIS domain]

Contents

ABSTRACT

Enterprise application development has moved from the delivery of isolated software systems to a continuous engineering discipline that combines architecture, integration, cybersecurity, software quality, automation, observability and lifecycle management.

Organisations increasingly require applications that respond to changing business requirements, integrate with internal and external digital services, serve several channels, hold their security posture across frequent releases, and evolve without disruptive redesign.

This paper examines the engineering principles required to develop enterprise applications that are secure, modular, interoperable, maintainable and adaptable over time. It applies a structured qualitative methodology based on comparative analysis of recognised software-quality, application-security, API and digital-government guidance, together with the national cybersecurity and data-protection control environment of the Kingdom of Saudi Arabia.

The study examines six engineering dimensions — modular architecture, API-first design, secure development, measurable quality, automated delivery and operational observability — and expresses them as nine architectural principles.

The analysis indicates that a sustainable enterprise application cannot be judged solely by whether it satisfies its immediate functional requirements. Long-term quality depends on the architecture's capacity to absorb change, on clear component boundaries, on integration through governed interfaces, on security controls carried through the whole development lifecycle, and on measurable operational visibility.

The paper proposes an Enterprise Application Engineering Framework that brings architecture, quality, security, interoperability, automation and observability into a single model for modern application development.

The framework is then extended in two directions. A reference technology landscape maps each architectural layer to the platform categories that serve it and to the vendors that independent analyst assessment places among the market leaders for that layer, on the principle that the layer is the durable architectural object and the product within it is the replaceable one. A nine-phase implementation cycle then sets out how an application moves from discovery to continuous improvement, with each phase separated from the next by a gate expressed as a condition on evidence rather than on elapsed time.

KEYWORDS *application development; software engineering; secure software development; DevSecOps; APIs; modular architecture; software quality; application security; digital government; implementation lifecycle; enterprise architecture; data platforms.*

1 INTRODUCTION

Application development is one of the principal mechanisms by which an organisation converts business processes, policies, services and information requirements into operational digital capability.

Historically, many enterprise applications were designed as self-contained systems whose main objective was to perform defined business functions for a specific group of users. Modern digital environments require a substantially different approach: an application may now have to communicate with several internal systems, exchange data with external organisations, expose and consume APIs, use cloud services, serve web and mobile channels, integrate identity systems, process sensitive information,

operate continuously, respond quickly to changing requirements, and hold its security posture across frequent releases.

The engineering question is therefore no longer how an application should be built, but the following.

> *How should an enterprise application be engineered so that it can evolve safely while remaining secure, interoperable, maintainable and operationally reliable?*

This question carries particular weight in government and large-enterprise environments, where applications are rarely isolated from a wider digital ecosystem. Guidance issued by the Digital Government Authority on application programming interfaces emphasises consistency in API development and recommends that new applications and functions adopt API and microservices-oriented approaches rather than remain dependent on monolithic architectures (DGA, 2023).

2 RESEARCH PROBLEM

Application development frequently prioritises immediate functional delivery. A project may therefore succeed in producing the requested features while accumulating architectural limitations that only become visible later. The common consequences are:

- Tightly coupled software components;
- Direct dependencies between applications;
- Duplicated business logic;
- Inconsistent APIs;
- Difficulty introducing new functionality;
- Security controls added late in development;
- Limited automated testing;
- Slow deployment processes;
- Inadequate operational visibility;
- High maintenance effort.

These effects matter most in large digital environments, because the cost of change rises with the number of integrations, users and dependent services. The research problem is therefore how enterprise applications can be developed in a way that balances rapid delivery against security, modularity, interoperability, quality and long-term maintainability.

3 RESEARCH QUESTION

The principal research question is:

> *What engineering practices enable enterprise applications to remain secure, modular, interoperable and evolvable throughout their lifecycle?*

The supporting questions are:

- How does modular architecture influence maintainability and application evolution?
- What role should APIs play in application architecture?
- At what stages should cybersecurity requirements be integrated?
- How should software quality be evaluated beyond functional correctness?
- How can automation improve delivery and reduce deployment risk?

- What operational information should be carried into subsequent development cycles?

4 RESEARCH OBJECTIVES

The research pursues nine objectives.

i. Identify the principal engineering characteristics of sustainable enterprise applications.
ii. Analyse modularity as a mechanism for reducing software dependency.
iii. Examine API-first design as an interoperability strategy.
iv. Evaluate the integration of cybersecurity into the application lifecycle.
v. Examine recognised software-quality characteristics.
vi. Analyse the contribution of automation and DevSecOps to software delivery.
vii. Develop a practical framework for modern enterprise application engineering.
viii. Map the architectural layers of an enterprise application to the platform categories and market-leading products that implement them.
ix. Define a gated software implementation cycle that carries the framework from discovery through to operational improvement.

5 BACKGROUND

5.1 Application development as a lifecycle

Application development should not be read as programming alone. A modern software lifecycle runs from requirements through architecture, design, development, verification, deployment and operation into improvement, and each phase shapes the quality of the finished application.

A security weakness, for instance, may originate in architecture before a developer writes a line of code. Equally, an application may function correctly and still be difficult to maintain because its internal structure carries excessive dependency. Development maturity must therefore be assessed across the complete lifecycle rather than at the point of delivery.

6 LITERATURE REVIEW

6.1 Software product quality

ISO/IEC 25010:2023 defines a product-quality model for ICT and software products. The model sets out nine quality characteristics and is intended to support requirements definition, design objectives, testing, quality assurance, acceptance criteria and product-quality evaluation across the lifecycle (ISO, 2023).

The significance is that application quality cannot be reduced to a single measurement such as functionality or performance. Development decisions must address functional requirements and quality requirements as two distinct classes of requirement.

6.2 API-oriented application architecture

DGA guidance recommends consistent API development practice and notes that new applications and functions should be designed around APIs and microservices, with a transition away from monolithic architecture where that is appropriate (DGA, 2023). The Digital Transformation Basic Standards further

address secure and effective APIs for data sharing, including endpoint documentation, access control, performance monitoring, event logging and periodic API reporting (DGA, 2024).

API design is therefore not merely an integration concern. It is part of application architecture and of lifecycle governance.

6.3 Application security as verifiable requirements

The OWASP Application Security Verification Standard provides a structured basis for defining and verifying security requirements for web applications and web services, serving both as a basis for testing security controls and as a source of requirements for secure development. Version 5.0.0, released on 30 May 2025, addresses architecture, authentication, session management, access control, validation, cryptography, logging, data protection, APIs, business logic and configuration (OWASP, 2025a; OWASP, n.d.).

> *Security should be expressed as verifiable requirements rather than treated as a final penetration-testing activity.*

6.4 Security across the development lifecycle

OWASP guidance on establishing a modern application-security programme recommends defining security requirements at the requirements stage and carrying security activity through design, build, testing and operation, and identifies the Verification Standard as the basis for application-security requirements (OWASP, 2025b).

6.5 A published framework for secure development practice

The NIST Secure Software Development Framework offers a complementary, practice-level view. It organises secure development into four groups of practices — preparing the organisation, protecting the software, producing well-secured software, and responding to vulnerabilities — and is deliberately technology-neutral so that it can be mapped onto an organisation's existing lifecycle rather than replacing it (Souppaya, Scarfone and Dodson, 2022).

Read together with the Verification Standard, the two sources cover the two halves of the same problem: what a secure application must demonstrate, and what a development organisation must practise in order to produce one.

6.6 The national regulatory context

Application engineering in the Kingdom also operates inside a defined regulatory environment. The National Cybersecurity Authority issues control sets that bear directly on application development, including controls addressing data cybersecurity and the protection of critical systems (NCA, 2022a; NCA, 2022b), alongside its essential cybersecurity controls (NCA, n.d.). Where an application processes personal data, the Personal Data Protection Law and its implementing regulations impose requirements on lawful processing, purpose limitation, data-subject rights, transfer and accountability (PDPL, n.d.).

These instruments are most economically satisfied when treated as architectural inputs at the requirements stage, and most expensively satisfied when discovered at the point of acceptance.

7 RESEARCH METHODOLOGY

This research applies a structured qualitative engineering analysis rather than an experimental or statistical method. It proceeds in five stages.

Stage 1 — Reference selection

Authoritative sources were drawn from the Digital Government Authority, ISO/IEC, OWASP, NIST, the National Cybersecurity Authority and the national data-protection framework, selected for their relevance to application development, software quality, APIs, digital-government integration, application security and secure development practice.

Stage 2 — Engineering theme identification

Recurring software-engineering themes were identified across the selected references. Six emerged: modularity, interoperability, security, quality, automation and observability.

Stage 3 — Lifecycle mapping

Each theme was mapped against the application lifecycle in order to establish at which stage it must be addressed rather than merely that it matters.

Stage 4 — Risk analysis

For each theme the study considered the engineering consequence of its omission, since a characteristic's importance is most clearly demonstrated by what fails in its absence.

Stage 5 — Framework construction

The results were consolidated into the Enterprise Application Engineering Framework presented in Section 10.

8 TECHNICAL ANALYSIS: NINE ENGINEERING PRINCIPLES

8.1 Modular architecture

A modular application separates responsibilities into clearly defined components. Figure 1 sets out a conceptual decomposition in which each layer owns a distinct concern.

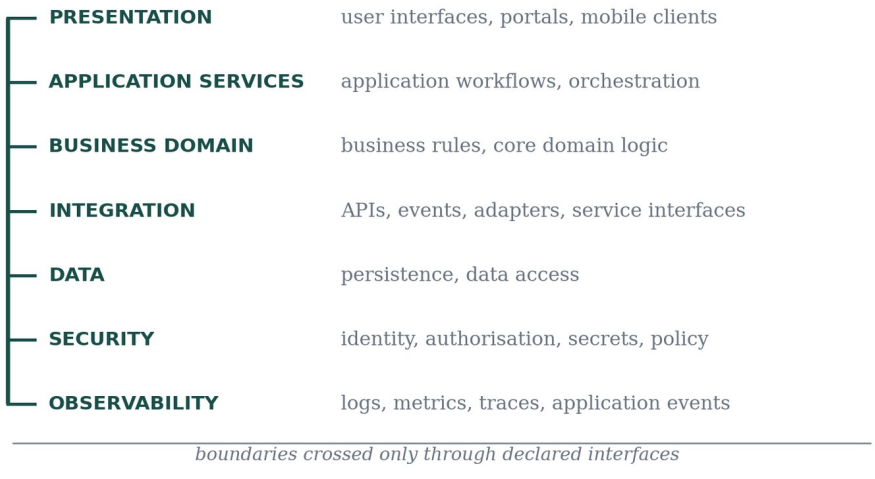

Figure 1 Conceptual layer decomposition of an enterprise application.

The purpose of modularity is not structural elegance. It limits the reach of change: when one component is altered, unrelated components should remain unaffected. Where that does not hold, the boundaries are nominal rather than real.

8.2 Controlled dependencies

Software complexity is driven less by the number of components than by the number and quality of dependencies between them. Figure 2 contrasts two designs of the same interaction.

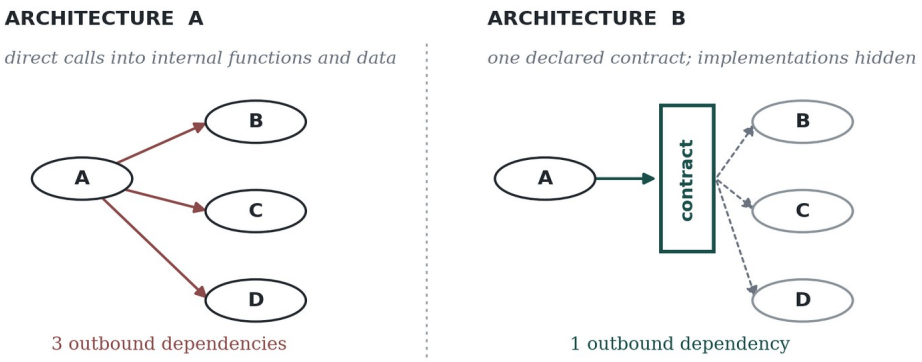

Figure 2 Direct internal calls compared with a single declared contract.

In Architecture A, module A calls internal functions in B, C and D and reaches their data structures directly, so any of the three can break it. In Architecture B, A communicates through one declared contract and the implementations behind it stay hidden, which leaves each component free to evolve independently.

8.3 API-first architecture

An API-first approach fixes the interaction before the implementation detail. The preferable sequence runs from business capability to service contract to API definition and only then to implementation, rather than building the application first and determining afterwards how others may integrate with it. The first sequence forces integration requirements into the architecture phase, which is consistent with DGA guidance on consistent API architecture and API-oriented development for new applications and functions (DGA, 2023).

A mature API design defines ownership, authentication, authorisation, request and response models, versioning, error handling, lifecycle rules, logging, documentation and performance monitoring. An API that lacks these is an integration liability presented as an integration capability.

8.4 Secure by design

Security should shape application design before implementation begins. Figure 3 places the security activity of each lifecycle phase alongside the phase itself.

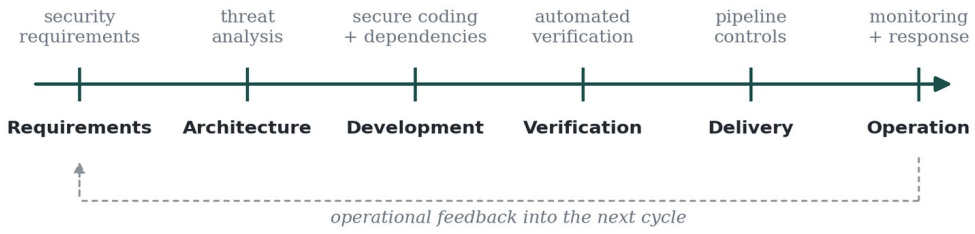

Figure 3 Security activity mapped to lifecycle phase, with operational feedback.

The Verification Standard is particularly useful here because it converts broad security objectives into controls that can be verified — authentication, authorisation, data validation, cryptography, session management, APIs, logging, error handling and data protection (OWASP, 2025a) — while the Secure Software Development Framework describes the organisational practices that produce those properties repeatably (Souppaya, Scarfone and Dodson, 2022).

8.5 Quality beyond functionality

An application may satisfy every functional requirement and still be of poor quality: hard to maintain, slow to respond, fragile under load, exposed to security weakness, awkward to integrate, or costly to operate. ISO/IEC 25010:2023 provides a structured quality model that can be applied through requirements, design, testing and evaluation (ISO, 2023).

The practical implication is that quality attributes belong in the requirements document. They do not emerge automatically from competent coding.

8.6 Automated delivery

Manual delivery introduces variability, and different environments accumulate different configurations and deployment steps. A modern pipeline runs from source code through build, static analysis, unit tests, security checks, integration tests and artefact creation to deployment and validation.

> *The engineering objective of automation is not speed for its own sake; it is repeatability, traceability and controlled change.*

8.7 DevSecOps

DevSecOps places security inside the same delivery process used to build and release the application, so that development, security and operations share one lifecycle rather than three. OWASP recommends repeatable security processes and the incorporation of security requirements and verification throughout development rather than reliance on final testing alone (OWASP, 2025b). Weaknesses identified early are, as a rule, less disruptive to correct.

8.8 Testing as an engineering feedback mechanism

Testing should verify more than functionality. A mature model spans unit, component, integration and API testing, security verification, performance testing, regression testing and acceptance testing.

Testing also reports on the architecture itself. Where a small change causes widespread regression failure, the likely cause is excessive coupling rather than inadequate tests, which makes the test suite an architectural diagnostic as well as a quality gate.

8.9 Observability and operational feedback

Engineering continues after deployment, because production environments produce information that development cannot predict. Logs record what happened; metrics describe how the application performs; traces show how a request moved through the components; events record operational change. DGA's Digital Transformation Basic Standards include requirements for monitoring API performance and analysing logs in order to identify errors and improve performance (DGA, 2024).

The result is a cycle — build, deploy, observe, learn, improve — which makes the software lifecycle continuous rather than linear.

9 Reference Technology Landscape by Layer

The principles set out in Section 8 are deliberately technology-neutral, but they are implemented with products. This section therefore extends the layer decomposition of Figure 1 into a reference technology landscape: for each architectural layer it states the engineering function the layer performs and names platforms that independent analyst assessment places among the market leaders for that function.

Three qualifications apply. First, the vendors named below are drawn from the 2025 Gartner Magic Quadrant cycle for the relevant market and are cited as evidence of market position, not as a recommendation; Gartner does not endorse the vendors it evaluates, and market position is not by itself a fitness-for-purpose assessment. Second, leadership positions are time-bound, since the quadrant for each market is reissued annually, so any shortlist derived from this section should be re-validated against the current report. Third, and most importantly for the argument of this paper, the layer is the durable object and the product is the replaceable one: an architecture that satisfies the nine principles should permit substitution within a layer without redesign of the layers above and below it.

Table 1 sets out the landscape. The layers are ordered from the user-facing surface down to the delivery and assurance toolchain that produces and sustains the whole.

Layer	Engineering function	Representative market-leading platforms
Experience and presentation	Web, mobile and portal delivery; multi-channel rendering; accessibility	Angular, React and Vue for web; Flutter and React Native for cross-platform mobile; native iOS and Android toolchains
API management and gateway	Contract publication, authentication, rate limiting, versioning, developer portal, usage telemetry	Salesforce MuleSoft Anypoint, Google Apigee, Microsoft Azure API Management, IBM API Connect, Kong, Amazon API Gateway
Integration and iPaaS	Application-to-application and B2B integration, event routing, orchestration, data mapping	Leaders in the 2025 iPaaS quadrant include Boomi, SAP Integration Suite, Informatica, Microsoft Azure Integration Services, Salesforce MuleSoft and Workato (Gartner, 2025a)
Business applications — ERP	Finance, procurement, supply chain, human capital and core transactional process	SAP Cloud ERP, Oracle Fusion Cloud ERP and NetSuite, Microsoft Dynamics 365, Workday, Infor, IFS and Epicor appear as Leaders across the 2025 service-centric and product-centric ERP quadrants (Gartner, 2025c)
Business applications — CRM and service management	Sales, marketing, customer service and enterprise service workflow	Salesforce, Microsoft Dynamics 365, Oracle, SAP and ServiceNow
Identity and access management	Authentication, federation, single sign-on, multi-factor authentication, authorisation, privileged access	Microsoft Entra ID, Okta, Ping Identity, CyberArk for privileged access, and Keycloak in open-source deployments
Application runtime and orchestration	Container hosting, elastic scaling, service networking, workload isolation	Kubernetes distributions including Red Hat OpenShift, Amazon EKS, Azure Kubernetes Service and Google GKE; serverless runtimes such as AWS Lambda and Azure Functions
Operational data — transactional DBMS	Persistence of application state under transactional guarantees	Oracle Database, Microsoft SQL Server and Azure SQL, PostgreSQL, MySQL, MongoDB, Amazon Aurora and Google Cloud Spanner
Data lake and lakehouse	Raw and curated storage of large multi-format datasets over open table formats	Leaders in the 2025 cloud DBMS quadrant include Databricks, Snowflake, Microsoft, Google, Amazon Web Services, Oracle, IBM and MongoDB (Gartner, 2025d); Apache Iceberg and Delta Lake are the prevailing open table formats
Data warehouse and semantic layer	Modelled, governed analytical storage and the shared definition of business measures	Snowflake, Google BigQuery, Amazon Redshift, Microsoft Fabric and Azure Synapse, Databricks SQL, Teradata and Oracle Autonomous Data Warehouse
Data integration, ELT and streaming	Ingestion, transformation, change data capture and event streaming	Informatica, Microsoft Fabric Data Factory, Qlik Talend, Fivetran, dbt, Apache Kafka with Confluent, Apache NiFi and Apache Spark
Data governance and cataloguing	Lineage, data quality, master data, classification and policy enforcement	Informatica, Collibra, Alation, Microsoft Purview, Databricks Unity Catalog and SAP Master Data Governance
Analytics, business intelligence and visualisation	Self-service analysis, governed reporting, dashboards and natural-language query	Leaders in the 2025 analytics and business intelligence quadrant are Microsoft Power BI, Salesforce Tableau, Google Looker, Qlik, Oracle Analytics Cloud and

Layer	Engineering function	Representative market-leading platforms
		ThoughtSpot (Gartner, 2025b)
Data science, machine learning and AI	Feature engineering, model training, deployment and model monitoring	Databricks, Microsoft Azure Machine Learning, Google Vertex AI, Amazon SageMaker, Dataiku, SAS and IBM watsonx
Observability and monitoring	Logs, metrics, distributed traces, alerting and service-level measurement	Datadog, Dynatrace, Splunk, Elastic, Grafana with Prometheus and New Relic, with OpenTelemetry as the vendor-neutral instrumentation standard
Delivery toolchain — CI/CD and DevSecOps	Source control, build, pipeline orchestration, artefact management and infrastructure as code	GitHub Actions, GitLab, Azure DevOps, Jenkins, Argo CD, Terraform, Ansible and HashiCorp Vault
Application security testing	Static, dynamic, dependency, secrets and container scanning inside the pipeline	Black Duck, Veracode, Checkmarx, Snyk, SonarQube, GitHub Advanced Security and OWASP ZAP

Table 1 Reference technology landscape by architectural layer.

Read against the nine principles, the landscape yields a small number of selection tests that matter more than feature comparison. A platform in the integration layer should be judged on whether it externalises its contracts — whether the interfaces it publishes are portable descriptions rather than proprietary configuration — because that determines whether the layer can later be replaced. A platform in the data layer should be judged on whether it stores in open formats, since open table formats decouple the storage of data from the engine that queries it. A platform in the delivery layer should be judged on whether its pipeline definitions live in the application repository as code, because a pipeline that exists only inside a product's console is not reproducible.

Two further considerations are specific to the environment described in Section 6.6. Where an application processes personal or otherwise regulated data, the deployment region and the data-residency options of every layer in the stack become architectural constraints rather than commercial preferences, and the availability of a platform in a Kingdom-hosted region may narrow the shortlist before any functional evaluation begins. Equally, the logging, retention and access-control capabilities of the identity, data and observability layers are the practical means by which the national control sets are satisfied, so those capabilities belong in the selection criteria rather than in a subsequent compliance exercise.

The landscape is therefore presented as a map rather than a shortlist. Its purpose is to make explicit which architectural decision each layer represents, so that a product choice can be traced back to an engineering requirement — and, where the requirement later changes, so that the affected layer can be identified and substituted without disturbing the rest of the architecture.

10 THE ENTERPRISE APPLICATION ENGINEERING FRAMEWORK

The nine principles above resolve into six engineering dimensions, each with a diagnostic question and a desired characteristic, as set out in Table 2.

DIMENSION	CORE QUESTION	DESIRED CHARACTERISTIC	PRINCIPLES
Architecture	Can components evolve independently?	Modularity	8.1, 8.2
Integration	Can the application interact through governed interfaces?	Interoperability	8.3
Security	Are controls integrated throughout development?	Secure by design	8.4, 8.7
Quality	Are non-functional characteristics defined and verified?	Measurable quality	8.5, 8.8

DIMENSION	CORE QUESTION	DESIRED CHARACTERISTIC	PRINCIPLES
Delivery	Can changes be built and released consistently?	Automation	8.6
Operations	Can behaviour and failure be understood after deployment?	Observability	8.9

Table 2 The six engineering dimensions of the framework.

Expressed as a single statement, a sustainable enterprise application is the combination of modular architecture, governed APIs, secure development, measurable quality, automated delivery and operational feedback. This formulation is developed for the present paper and should not be read as an official DGA, ISO, NIST or OWASP model.

11 APPLYING THE FRAMEWORK ACROSS THE LIFECYCLE

Mapping the framework onto the lifecycle prevents architecture, quality and security from being deferred to the end of development, which is the point at which each is most expensive to correct. Table 3 states the principal engineering concern at each stage.

LIFECYCLE STAGE	PRINCIPAL ENGINEERING CONCERN
Requirements	Functional, quality and security requirements stated together
Architecture	Modularity, interfaces, data design, security architecture
Development	Coding quality and dependency control
Verification	Functional, quality and security testing
Delivery	Repeatability and automation
Operation	Monitoring and observability
Improvement	Feedback-driven development

Table 3 Principal engineering concern by lifecycle stage.

11.1 The implementation cycle as a governed sequence

The stage mapping above states what each phase of the lifecycle must address. It does not state how an implementation moves from one phase to the next, and in practice that transition is where most delivery risk accumulates. A project that treats phases as calendar periods will pass from design into build because the design window has closed rather than because the design is complete. The remedy is to express the implementation cycle as a sequence of phases separated by gates, where each gate is a condition on evidence rather than a date.

The nine phases described below are not a waterfall. Phases 4 to 6 iterate — build, verify and integrate repeat once per increment — and the cycle as a whole closes back on itself through Phase 9. What is sequential is the dependency: an increment cannot be verified against security requirements that were never written, and it cannot be deployed repeatably through a pipeline that does not yet exist. The value of naming the phases is that it makes those dependencies explicit and therefore auditable.

Each phase below is described in the same four terms: the objective it serves, the principal activities it comprises, the artefacts it produces, and the failure that results when it is compressed or skipped. The fourth term is deliberate: as Section 7 noted, the importance of an engineering activity is most clearly demonstrated by what breaks in its absence.

11.2 Phase 1 — Discovery and requirements

The objective of discovery is to establish what the application must do, how well it must do it, and under what constraints — before any architectural commitment is made. The activities comprise stakeholder and process analysis, definition of functional scope, elicitation of the quality characteristics that matter for this application, identification of the data the application will hold and its classification, enumeration of the systems it must integrate with, and derivation of the applicable security and regulatory requirements.

Two of these deserve emphasis because they are the ones most often deferred. Quality requirements — availability, response time under stated load, recoverability, maintainability — must be stated as measurable targets at this point, since ISO/IEC 25010:2023 is only useful as a design input if its characteristics are converted into specific acceptance thresholds (ISO, 2023). Security requirements must likewise be written as verifiable statements at this point, which is the purpose the Application Security Verification Standard is designed to serve (OWASP, 2025a; OWASP, 2025b). Where personal data is in scope, the lawful basis, retention period and data-subject rights obligations are requirements in exactly the same sense (PDPL, n.d.).

The artefacts are a scoped requirements baseline covering functional, quality, security, integration and data requirements; a data classification register; and a first statement of the regulatory obligations that apply. The characteristic failure of a compressed discovery phase is not missing features — those surface quickly and are corrected — but missing non-functional requirements, which surface at acceptance testing or in production and are then expensive to satisfy because the architecture was chosen without them.

11.3 Phase 2 — Architecture and solution design

The objective of the design phase is to decide the structure within which every later decision will be made. The activities comprise decomposition of the application into components with stated responsibilities, definition of the interfaces between them, selection of the technology for each architectural layer, data modelling and storage design, security architecture, and the definition of the deployment topology and environment strategy.

This is the phase in which the principles of Section 8 are actually exercised. Component boundaries are drawn here (8.1); the dependency structure between components is decided here (8.2); the API contracts are specified before implementation here (8.3); and the security design — trust boundaries, authentication and authorisation model, cryptographic decisions, data protection at rest and in transit — is settled here (8.4). The technology selections drawn from the landscape in Section 9 are made at this point and, critically, are recorded with the requirement that justified each one, so that a later substitution can be assessed against the original rationale rather than re-argued from first principles.

The artefacts are an architecture description covering components, interfaces and data; a set of API contracts expressed in a portable specification format; a security architecture and threat model; and a documented record of the significant technology decisions and their justification. The characteristic failure of a compressed design phase is a structure that is never explicitly decided and therefore emerges from implementation order — the first component written becomes the de facto centre of the application, and the dependency structure that results is an accident rather than a design.

11.4 Phase 3 — Environment, pipeline and toolchain establishment

The objective of this phase is to make the delivery mechanism exist before it is needed. The activities comprise provisioning of the development, test, staging and production environments; expression of that provisioning as infrastructure code so that environments are reproducible rather than hand-built; construction of the build and deployment pipeline; integration of the static analysis, dependency scanning, secrets detection and container scanning stages into that pipeline; establishment of the artefact repository and versioning scheme; and configuration of the observability stack so that telemetry exists from the first deployment rather than being retrofitted after the first incident.

Placing this phase before the build is the practical content of the automation and DevSecOps principles (8.6, 8.7). A pipeline established after development has begun will be shaped by the code that already exists; a pipeline established first shapes the code instead, because a build that must pass static analysis and dependency scanning on every commit produces different code from one that is scanned at the end. The NIST Secure Software Development Framework treats this preparation of the organisation and its toolchain as a distinct practice group for the same reason (Souppaya, Scarfone and Dodson, 2022).

The artefacts are environment definitions held as code, a working pipeline that builds, tests, scans and deploys an application skeleton end to end, an artefact repository with a defined versioning and promotion scheme, and a baseline observability configuration. The characteristic failure of a compressed set-up phase is environment divergence: configurations accumulate independently in each environment, defects become environment-specific, and the phrase that signals the failure — that the application works in test but not in production — becomes a recurring feature of the project rather than an anomaly.

11.5 Phase 4 — Iterative build

The objective of the build phase is to produce working, verified increments of the application rather than a single deliverable at the end. The activities comprise implementation against the agreed interfaces, unit and component testing written alongside the code, code review, dependency management, continuous integration on every change, and incremental demonstration of completed functionality to the stakeholders who defined it.

Two disciplines distinguish a build phase that preserves the architecture from one that erodes it. The first is dependency control: a component that reaches directly into another component's internals rather than through its declared interface has silently changed the architecture, and only review of the dependency structure — not review of the code line by line — will detect it (8.2). The second is that the definition of a completed increment includes its tests, its documentation and its telemetry, not merely its functional behaviour; an increment that works but cannot be observed in production is not finished, it is deferred.

The artefacts are versioned, reviewed increments in the artefact repository, each with its automated test suite, updated interface documentation and instrumentation in place. The characteristic failure of an unmanaged build phase is architectural drift: the delivered system passes its functional tests while no longer matching its architecture description, at which point the description ceases to be a reliable basis for any subsequent change.

11.6 Phase 5 — Verification and security assurance

The objective of verification is to demonstrate that the application satisfies its requirements — all of them, not only the functional set. The activities comprise functional and regression testing, integration and contract testing across component and system boundaries, performance and load testing against the thresholds stated in Phase 1, security verification against the requirements derived in Phase 1, resilience and failover testing, accessibility testing, and user acceptance testing.

Security verification at this phase is a confirmation exercise, not a discovery exercise. Where the Verification Standard was used to write requirements in Phase 1, verification consists of testing against those stated controls — authentication, session management, access control, input validation, cryptography, logging, error handling, data protection, API behaviour and configuration (OWASP, 2025a). Where it was not, this phase becomes the first point at which security is considered, and the findings are then architectural rather than incidental, which is precisely the outcome the lifecycle integration guidance is intended to prevent (OWASP, 2025b).

The artefacts are test results traceable to specific requirements, a performance report against the stated thresholds, a security verification report with findings and their disposition, and a signed acceptance record. The characteristic failure of a compressed verification phase is a narrow interpretation of testing: functional coverage is demonstrated, non-functional characteristics are asserted rather than measured, and the first genuine load the application experiences is the one that arrives in production.

11.7 Phase 6 — Data migration and integration readiness

The objective of this phase is to establish that the application will behave correctly against real data and real counterparties rather than test fixtures. It is treated separately here because it is routinely absorbed into other phases and is routinely the cause of delayed go-live. The activities comprise profiling of the source data, definition and implementation of the migration and transformation rules, reconciliation of migrated data against source records, cleansing where source quality is inadequate, trial migrations at production scale, and end-to-end testing of every external integration under production-equivalent conditions including authentication, error handling and throughput.

Two conditions are worth stating explicitly. A migration is not validated until it has been rehearsed at full volume, because time-to-complete is itself a cutover constraint and is not derivable from a partial run. An integration is not validated until it has been exercised against the counterparty's actual endpoint, because the divergence between a documented interface and its implementation is discovered only by use — which is the operational argument for the interface governance and monitoring requirements set out in the digital transformation standards (DGA, 2024).

The artefacts are documented migration and transformation rules, reconciliation evidence at record level, a full-scale trial migration result with timings, and integration test evidence for each interface. The characteristic failure here is a cutover that overruns its window, or a go-live in which the application is functionally correct while the data it operates on is incomplete, unreconciled or of unknown provenance.

11.8 Phase 7 — Release, cutover and go-live

The objective of release is to place the application into production in a controlled and reversible manner. The activities comprise release planning and scheduling, execution of the deployment through the established pipeline rather than by hand, database and schema migration, configuration and secrets management for the production environment, smoke verification immediately after deployment, and a documented rollback path that has itself been tested.

The engineering requirement at this phase is reversibility. A deployment that cannot be reversed converts every release into an irreversible commitment and therefore raises the cost of releasing, which in turn reduces release frequency and increases the size of each release — the precise dynamic that automation is intended to break (8.6). Progressive release techniques, in which a new version is exposed first to a limited population and promoted on the evidence of its telemetry, are the practical expression of the same requirement.

The artefacts are a release record identifying exactly which artefact version was deployed, the deployment and rollback procedures as executed, post-deployment verification results, and the updated configuration baseline. The characteristic failure here is the manual release: undocumented, unrepeatable, dependent on the individual who performed it, and impossible to reverse under time pressure.

11.9 Phase 8 — Hypercare and operational handover

The objective of this phase is to transfer the application from the team that built it to the team that will run it, without losing the knowledge that made it work. The activities comprise a defined period of elevated support immediately after go-live, monitoring of the telemetry established in Phase 3 against expected behaviour, triage and correction of early defects, tuning of alert thresholds against observed rather than assumed behaviour, completion of operational documentation and runbooks, and training of the support and operations functions.

Handover is an engineering deliverable rather than an administrative one. An application handed over without runbooks, without documented failure modes and without alerting calibrated to real behaviour has been transferred in name only, and the operations team will escalate to the development team indefinitely — which is the practical meaning of a system that is in production but not yet operable.

The artefacts are operational runbooks, a documented incident and escalation model, alert definitions with tuned thresholds, a service-level definition and its measurement method, and a signed handover record. The characteristic failure of an absent hypercare phase is a support model that never stabilises, in which the development team remains the operations team by default and its capacity for new work is consumed by the application it has already delivered.

11.10 Phase 9 — Continuous improvement and evolution

The objective of the final phase is to convert operational experience into engineering change, which closes the cycle rather than ending it. The activities comprise analysis of the logs, metrics and traces produced in operation, review of incidents for their architectural rather than immediate causes, measurement of the quality characteristics that were specified in Phase 1 against their behaviour in production, management of dependency currency and technical debt, periodic re-verification of security controls, and the feeding of all of this into the requirements baseline for the next increment.

This is the phase in which observability earns its cost (8.9). Telemetry that is collected but never analysed is an expense; telemetry that changes the next increment is an engineering input. The same applies to incidents: an incident closed by restoring service has been handled, whereas an incident closed by removing the condition that permitted it has been engineered. Digital transformation guidance requires the monitoring of interface performance and the analysis of logs precisely because the analysis, not the collection, is where the improvement originates (DGA, 2024).

The artefacts are a periodic operational review, a maintained technical-debt and dependency register, updated quality measurements against the original targets, and an amended requirements baseline for the next cycle. The characteristic failure at this phase is the application that is considered finished at go-live: dependencies age, security posture decays relative to a changing threat environment, and the accumulated cost of deferred change eventually presents itself as a replacement programme rather than a maintenance one.

11.11 Phase gates and the evidence each produces

A phase gate is useful only if it tests evidence. Table 4 states, for each phase, the condition that should be satisfied before the next phase begins and the artefact that demonstrates it. The intent is not procedural

weight but traceability: at any point in an implementation it should be possible to say which gate the work has passed and on what evidence.

Phase	Gate condition before the next phase begins	Evidence produced
1 Discovery and requirements	Functional, quality, security, integration and data requirements are stated and measurable	Requirements baseline; data classification register; regulatory obligations statement
2 Architecture and design	Components, interfaces, data model and security architecture are decided and recorded	Architecture description; API contracts; threat model; technology decision record
3 Environment and pipeline	An application skeleton builds, scans and deploys end to end without manual steps	Environment definitions as code; working pipeline; artefact repository; observability baseline
4 Iterative build	Each increment is reviewed, tested, instrumented and matches the architecture description	Versioned increments; automated test suites; updated interface documentation
5 Verification and security assurance	Functional, performance and security requirements are demonstrated, not asserted	Traceable test results; performance report against thresholds; security verification report
6 Data migration and integration readiness	Migration is rehearsed at full volume and every interface is tested against its real endpoint	Migration rules; record-level reconciliation; trial migration timings; integration evidence
7 Release and cutover	Deployment is pipeline-executed, verified after deployment, and reversible by a tested path	Release record with artefact version; deployment and rollback procedures; smoke results
8 Hypercare and handover	Operations can run the application without recourse to the build team	Runbooks; incident and escalation model; tuned alerts; service-level definition; handover record
9 Continuous improvement	Operational evidence has been analysed and has amended the requirements baseline	Operational review; technical-debt and dependency register; quality measurements; revised baseline

Table 4 Implementation phase gates and the evidence each produces.

12 FINDINGS

The research produces nine findings.

Finding 1. **Functional completion is not evidence of application quality.** A working application may still carry architectural, security, maintainability or operational weakness.

Finding 2. **Modularity is a precondition of sustainable change.** Applications with real component boundaries are easier to evolve than tightly coupled systems.

Finding 3. **APIs belong to application architecture, not to integration afterthought.** API-first design determines whether an application can participate in a wider digital ecosystem.

Finding 4. **Security begins with requirements.** Final-stage testing cannot compensate for insecure architecture or absent security requirements.

Finding 5. **Quality must be explicit and measurable.** Quality characteristics belong in requirements, testing and acceptance criteria.

Finding 6. **Automation buys repeatability before it buys speed.** Automated build, verification and deployment reduce inconsistency and produce clearer traceability.

Finding 7. **Production information should shape development.** Observability closes the loop by returning real operational behaviour into subsequent releases.

Finding 8. **The architectural layer is durable; the product inside it is not.** Technology selection is defensible only where the layer boundary permits substitution without redesign of the layers above and

below it, which makes portable contracts and open storage formats selection criteria rather than preferences.

Finding 9. Implementation phases must be separated by evidence, not by dates. A gate that tests for a named artefact — a measurable requirement, a tested rollback, a reconciled migration — is auditable; a gate that tests whether a scheduled window has closed is not.

13 DISCUSSION

The analysis suggests that modern application development is better understood as continuous product engineering than as a sequence that terminates at deployment. Three transitions carry most of the weight: from monolithic functionality to modular capability; from security verified after development to security integrated throughout it; and from project delivery to a continuous application lifecycle.

Each transition becomes more consequential as applications grow interconnected with digital platforms, shared APIs, common data, external services and cloud environments — that is, precisely as the environment in which most enterprise and government applications now operate.

14 PRACTICAL APPLICATIONS

The framework can be applied when developing new enterprise applications, modernising or decomposing legacy applications, introducing APIs, designing mobile applications, building government digital services, introducing DevSecOps, improving application security, defining application-quality requirements, or migrating applications to cloud environments.

It also serves as an assessment checklist. High component dependency indicates maintainability risk; undefined API ownership indicates integration-governance risk; security requirements written only in preparation for penetration testing indicate a secure-development gap; manual deployment indicates a repeatability risk; and absent application telemetry indicates an operational-visibility gap.

15 APPLIED CONTEXT

Al Moammar Information Systems develops applications in environments where they coexist with digital platforms, enterprise systems, data environments, integration services and infrastructure. That setting is the applied context of this paper: it is where the engineering tensions described above are encountered rather than theorised.

The paper does not claim that any particular application was developed according to the framework proposed here, and does not present delivered work as a research experiment. The separation is deliberate, and it is what allows the paper to stand as research rather than as a corporate profile.

> **Placeholder for MIS input — delete this box before publication.** One validated application-development case may be inserted here: application or solution name; the business problem addressed; the development scope performed by MIS; the application architecture; APIs and integrations; the security approach; the development and testing methodology; the deployment model; the operational monitoring model; and the technical outcome. Only facts the responsible MIS team can substantiate should be published, and any client name requires that client's documented consent.

16 LIMITATIONS

Five limitations should be stated plainly. The study rests on qualitative comparative analysis rather than controlled software experiments. No proprietary source code and no confidential customer architecture were examined. The proposed framework has not been tested against a statistically significant portfolio of applications. Development practice varies with application criticality, regulatory exposure, technology architecture and organisational maturity. And the reference technology landscape in Section 9 reflects the analyst assessments of a single evaluation cycle: market positions change, products are acquired and renamed, and the named platforms should therefore be treated as an illustration of the layer model rather than as a durable shortlist.

The framework should therefore be treated as a structured engineering model rather than a universal implementation prescription.

17 FUTURE RESEARCH

Useful extensions include quantitative software maintainability, application coupling metrics, API maturity assessment, artificial-intelligence-assisted software engineering, secure software supply chains, automated architecture conformance, DevSecOps maturity, application observability, automated vulnerability detection and technical-debt measurement.

The natural continuation of this study is an Enterprise Application Engineering Maturity Model that evaluates organisations across the six dimensions proposed here and derives measurable maturity indicators for each.

18 CONCLUSION

Application development has moved beyond programming and functional delivery. Modern enterprise applications operate as parts of wider digital ecosystems, and must therefore address architecture, interoperability, cybersecurity, software quality, delivery automation and operations together rather than in sequence.

The research identifies six foundational characteristics — modularity, interoperability, secure development, measurable quality, automated delivery and observability — and treats them as lifecycle concerns rather than as independent technology features.

The governing question is accordingly no longer whether the application works, but whether it can continue to work, integrate, remain secure and evolve as its requirements and its surrounding ecosystem change. Sustainable application development begins at the point where that second question enters architectural and engineering decisions.

Abbreviations

ABI, analytics and business intelligence. API, application programming interface. ASVS, Application Security Verification Standard. CDC, change data capture. CI/CD, continuous integration and continuous delivery. CRM, customer relationship management. DAST, dynamic application security testing. DBMS, database management system. DevSecOps, development, security and operations. DGA, Digital Government Authority. ELT, extract, load and transform. ERP, enterprise resource planning. iPaaS, integration platform as a service. ISO/IEC, International Organization for Standardization and International Electrotechnical Commission. MDM, master data management. NCA, National Cybersecurity Authority. NIST, National

Institute of Standards and Technology. OWASP, Open Worldwide Application Security Project. PDPL, Personal Data Protection Law. SAST, static application security testing. SCA, software composition analysis. SDLC, software development lifecycle. SSDF, Secure Software Development Framework.

Notice on Vendor References and Trademarks

The product and platform names appearing in Section 9 are included to illustrate the architectural layer model and to indicate the current shape of the market. Their inclusion is not a recommendation, an endorsement, a statement of suitability for any particular requirement, or an indication of a commercial relationship between Al Moammar Information Systems and the vendor concerned. Omission of a product carries no adverse implication.

Where market position is attributed to analyst assessment, the attribution is to the cited Gartner Magic Quadrant report for the market in question. Gartner does not endorse any vendor, product or service depicted in its research publications, and its research consists of the opinions of its research organisation and should not be construed as statements of fact. Positions are those of the stated publication date and are subject to change in subsequent editions.

All product names, logos and brands are the property of their respective owners, and all company, product and service names used in this paper are for identification purposes only.

REFERENCES

Digital Government Authority (2023) Guideline: Application Programming Interfaces (APIs), Version 1.0. Riyadh: Digital Government Authority, Kingdom of Saudi Arabia.

Digital Government Authority (2024) Digital Transformation Basic Standards, Version 3.0. Riyadh: Digital Government Authority, Kingdom of Saudi Arabia.

Gartner (2025a) Magic Quadrant for Integration Platform as a Service. Humphreys, A., Comes, A., Guttridge, K. and Wilkins, A., 19 May 2025. Stamford, CT: Gartner, Inc.

Gartner (2025b) Magic Quadrant for Analytics and Business Intelligence Platforms. Ganeshan, A., Macari, E., O'Brien, J., Schlegel, K. and Long, C., 16 June 2025. Stamford, CT: Gartner, Inc.

Gartner (2025c) Magic Quadrant for Cloud ERP for Service-Centric Enterprises. Anderson, R., Jartelius, J., Kienast, T., Grinter, S., Torii, D. and Paradarami, C., 13 October 2025. Stamford, CT: Gartner, Inc.

Gartner (2025d) Magic Quadrant for Cloud Database Management Systems. Cook, H., Gu, X., Ramakrishnan, R., Rosenbaum, A. and Miraz, M., 18 November 2025. Stamford, CT: Gartner, Inc.

International Organization for Standardization (2023) ISO/IEC 25010:2023 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model. Geneva: ISO.

National Cybersecurity Authority (2022a) Data Cybersecurity Controls. Riyadh: National Cybersecurity Authority, Kingdom of Saudi Arabia.

National Cybersecurity Authority (2022b) Critical Systems Cybersecurity Controls. Riyadh: National Cybersecurity Authority, Kingdom of Saudi Arabia.

National Cybersecurity Authority (n.d.) Essential Cybersecurity Controls (ECC). Riyadh: National Cybersecurity Authority, Kingdom of Saudi Arabia. [edition and year of issue to be confirmed before publication]

OWASP Foundation (2025a) Application Security Verification Standard (ASVS), Version 5.0.0, released 30 May 2025. Wakefield, MA: OWASP Foundation.

OWASP Foundation (2025b) Establishing a Modern Application Security Program, OWASP Top 10:2025. Wakefield, MA: OWASP Foundation.

OWASP Foundation (n.d.) OWASP Developer Guide: Application Security Verification Standard. Wakefield, MA: OWASP Foundation.

Personal Data Protection Law (n.d.) Personal Data Protection Law of the Kingdom of Saudi Arabia and its Implementing Regulations. Riyadh: Saudi Data and Artificial Intelligence Authority. [Royal Decree number and dates of issue and amendment to be confirmed before publication]

Souppaya, M., Scarfone, K. and Dodson, D. (2022) Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities, NIST Special Publication 800-218. Gaithersburg, MD: National Institute of Standards and Technology.