# Spec-Grounded Modernization of Brownfield Software Systems: An Architecture-Aware Framework for AI-Assisted, Traceable, and Safe Evolution

*A conceptual research article extending the practitioner work on AI specification kits for brownfield software modernization*

**Bhagyeshkumar Chokhawala**

*Enterprise Architect and Applied AI Researcher, United States*

ORCID: 0009-0001-5146-832X

**Research Article · 2026**

## Abstract

Brownfield modernization is difficult because required behavior, implemented architecture, operational constraints, and business rules are distributed across code, documentation, tickets, data schemas, and tacit knowledge. AI coding agents can accelerate implementation, but without reliable system context and explicit specifications they can also amplify architectural drift, duplicate capabilities, weaken traceability, and introduce unsafe changes. This article develops a Spec-Grounded Modernization Framework (SGMF) that integrates architecture recovery, specification-driven development, AI-assisted planning, bounded implementation, and evidence-based release assurance. The framework extends a practitioner proposal that combines GitHub Spec Kit with architectural context generation and requirement parsing. It aligns the current official Spec Kit workflow—constitution, specify, clarify, plan, tasks, analyze, implement, and converge—with brownfield-specific services for context recovery, impact analysis, architecture conformance, security validation, and continuous context refresh. The article contributes an eight-stage lifecycle, an artifact and traceability model, a conceptual reference architecture, a Modernization Assurance Index, and an evaluation design for controlled industrial pilots. An illustrative billing-system scenario shows how a requirement concerning pending transactions can be connected to affected services, data changes, tests, architecture decisions, release gates, and runtime evidence. The article argues that the unit of control for AI-assisted modernization should not be the prompt or code diff alone, but a versioned evidence chain linking business intent, specification, architecture, implementation, verification, and operational outcomes. The framework provides a foundation for safer, more explainable, and more governable modernization of long-lived enterprise systems.

**Keywords:** brownfield modernization; specification-driven development; AI coding agents; software architecture recovery; requirements traceability; architecture conformance; secure software development; GitHub Spec Kit

# 1. Introduction

Long-lived software systems accumulate value and constraint at the same time. They encode critical business rules, customer history, regulatory obligations, operational knowledge, and integrations that are expensive to reproduce. They also accumulate obsolete technologies, implicit dependencies, inconsistent documentation, architectural erosion, brittle deployment procedures, and uneven test coverage. Consequently, brownfield modernization cannot be reduced to replacing a framework, moving workloads to cloud infrastructure, or asking an AI coding agent to rewrite components. Modernization is a controlled evolution problem in which new outcomes must be introduced without losing existing obligations.

The emergence of AI-assisted software development changes the economics of implementation but not the need for disciplined engineering. Large language models can draft requirements, plans, tests, and code; however, their outputs depend heavily on the quality and relevance of supplied context. Empirical studies of LLM-generated specifications and requirements documents show useful productivity potential alongside failure modes involving incompleteness, inconsistency, oversimplification, and fabricated constraints [10]-[13]. These limitations are especially consequential in brownfield environments because important constraints may be implicit in code paths, database semantics, runtime behavior, and organizational conventions rather than present in a single requirement document.

GitHub Spec Kit operationalizes specification-driven development by establishing governing principles and refining intent through a sequence of specification, planning, task generation, consistency analysis, implementation, and convergence activities [2]. The official toolkit explicitly includes iterative enhancement and legacy modernization as a development phase. Yet the tool does not, by itself, recover the architecture of an arbitrary enterprise system or guarantee that generated changes conform to undocumented boundaries. The central research problem is therefore how to ground specification-driven AI workflows in trustworthy evidence about the existing system.

This article extends the author's practitioner article, "Spec-Grounded Modernization: Leveraging AI Specification Kits for Brownfield Software Systems" [1], into a formal conceptual framework. The source article proposes architectural context generation, requirement parsing, specification generation, and continuous validation. It also presents commands such as context, parse, generate, and validate. In the current official Spec Kit, those commands are not part of the core command set; they should be understood as proposed organizational extensions or adjacent pipeline services. The framework developed here makes that distinction explicit and defines the artifacts, governance controls, and evaluation criteria needed to turn the proposal into a reproducible modernization method.

## 1.1 Research Questions

- RQ1. How can recovered architecture and operational context be represented so that AI-assisted specifications and implementation plans remain grounded in the actual brownfield system?
- RQ2. What lifecycle, artifacts, and decision gates are required to preserve traceability, architectural conformance, security, and transaction integrity during AI-assisted modernization?
- RQ3. How can organizations evaluate whether a spec-grounded modernization approach improves quality and delivery outcomes without overstating AI-generated productivity gains?

## 1.2 Contributions

- An eight-stage Spec-Grounded Modernization Framework that integrates architecture recovery with the current Spec Kit workflow.
- A layered architecture-context model that records structure, behavior, data, ownership, quality attributes, provenance, freshness, and confidence.
- A bidirectional artifact graph linking business intent to requirements, specifications, architecture decisions, tasks, code and data changes, tests, releases, and runtime evidence.
- A Modernization Assurance Index and controlled evaluation design for measuring specification quality, conformance, traceability, verification, security, and operability.
- A governance model that constrains AI coding agents through human approval, bounded change scopes, policy-as-code, and evidence-based release gates.

# 2. Background and Related Work

## 2.1 Brownfield Modernization as Architecture-Constrained Evolution

Brownfield modernization differs from greenfield development because the current system is simultaneously the implementation, a partial specification, and a source of historical constraints. Architecture documentation may be incomplete or stale, while the executable system contains dependencies not represented in design artifacts. Architecture recovery uses source code, repository history, build information, runtime traces, and other evidence to reconstruct views of the implemented system. Recent mapping work on mining software repositories for software architecture identifies recovery, conformance, architectural-smell detection, and decision support as major uses of repository evidence [6].

Modernization therefore requires two complementary forms of knowledge. Prescriptive knowledge expresses intended behavior and architectural rules. Descriptive knowledge represents what the system currently does and how its elements interact. A safe modernization method must compare and reconcile both. If an AI agent receives only prescriptive requirements, it may invent structures that conflict with the implementation. If it receives only code context, it may preserve accidental complexity and reproduce obsolete behavior. Spec grounding combines both forms and records disagreements as explicit decisions rather than silently resolving them in generated code.

## 2.2 Specification-Driven Development

Spec Kit describes specification-driven development as a process in which specifications become primary, executable artifacts rather than disposable documentation [2]. The core workflow begins with a project constitution, then develops a feature specification, clarifies ambiguity, creates an implementation plan, decomposes the plan into tasks, analyzes consistency and coverage, implements the tasks, and converges remaining work. The emphasis on intent before implementation and multi-step refinement provides a stronger control surface than one-shot prompt-to-code generation.

For brownfield use, this workflow must be enriched with an evidence package describing existing boundaries, interfaces, data semantics, quality-attribute constraints, ownership, and runtime dependencies. Spec Kit supports extensions and presets, which provide a practical mechanism for adding context-recovery commands, regulated templates, architecture checks, and domain-specific traceability requirements. Thus, spec-grounded modernization is not a competing tool; it is an architectural pattern for extending a specification-driven process with brownfield evidence and assurance services.

## 2.3 Requirements Traceability and Change Impact

Requirements traceability connects the origin and rationale of a requirement to design, implementation, verification, and release artifacts. Controlled studies have found that developers with traceability support can perform maintenance tasks faster and produce more correct solutions [7], while later empirical work links traceability to more effective evolution and maintenance [8]. A systematic mapping study of traceability in maintenance and evolution reports benefits for change impact analysis, comprehension, compliance, verification, and reuse, while also identifying cost and upkeep as persistent barriers [9].

AI-assisted modernization creates both a need and an opportunity for more automated traceability. Generated plans, tasks, and code changes already pass through machine-readable workflows. If each generated artifact carries stable identifiers, provenance, and relationship metadata, the development pipeline can maintain a live evidence graph rather than a manually curated spreadsheet. However, automatically generated links must expose confidence and permit human correction; otherwise traceability can become precise-looking but incorrect.

## 2.4 Secure and Governed AI-Assisted Development

The NIST Secure Software Development Framework recommends integrating secure practices into any software development life cycle rather than treating security as a final inspection [14]. Its generative-AI community profile extends these concerns to AI model and system development [15]. For AI-assisted code changes, secure development requires protected code and prompt context, controlled model access, review of generated dependencies, source and artifact provenance, automated analysis, test evidence, vulnerability handling, and auditable approvals. CodeQL

demonstrates how semantic code analysis can be integrated into developer workflows to identify security and correctness issues as code changes [16].

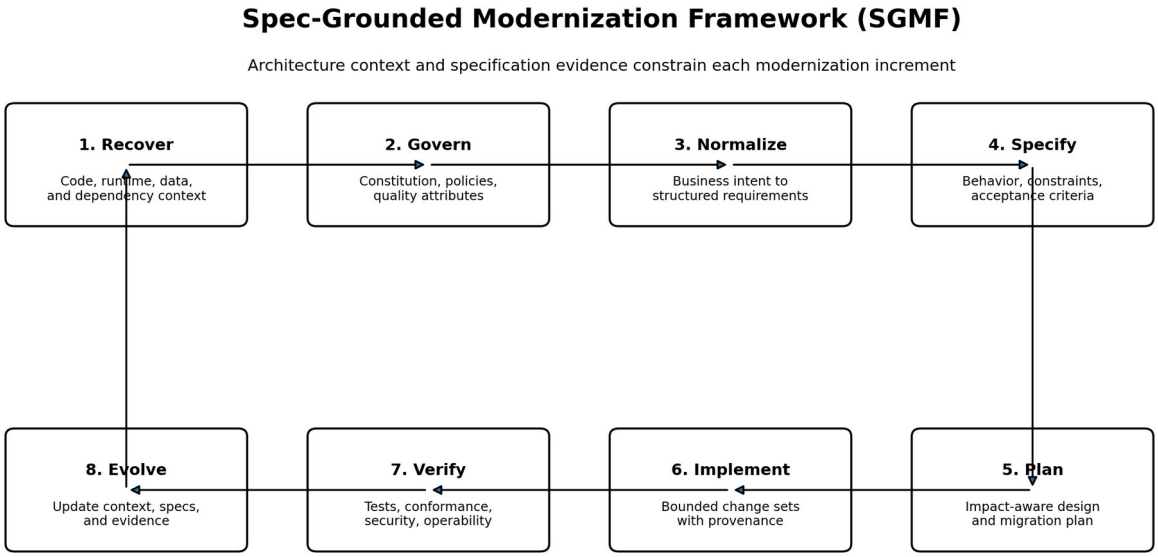


*Figure 1. Eight-stage Spec-Grounded Modernization Framework.*

## 3. Research Design

This work uses a conceptual design-science approach. The problem and initial solution elements were derived from the practitioner article [1]. The design was then refined through synthesis of primary and authoritative sources on specification-driven development, software architecture, architecture recovery, requirements traceability, AI-assisted requirements and code generation, and secure software development. The resulting framework is an artifact intended for subsequent empirical evaluation rather than a claim of proven effectiveness.

The source article reports preliminary improvements in redundant module creation, specification accuracy, and traceability. Because it does not provide a repository description, sample size, baseline protocol, measurement instrument, or statistical analysis, this manuscript does not treat those percentages as validated results. Instead, they are reformulated as hypotheses and measurable outcomes for a controlled pilot. This distinction is important because evidence about AI development productivity can be highly sensitive to task selection, evaluator expertise, system familiarity, and the quality of the underlying test oracle.

### 3.1 Design Principles

**Evidence before generation.** AI agents should not generate implementation plans until system context has been collected, versioned, and assessed for freshness and confidence.

**Intent before technology.** The feature specification should define outcomes, users, rules, exceptions, and acceptance criteria before selecting implementation mechanisms.

**Bounded autonomy.** Agents may propose or execute only within an approved scope, repository, branch, architecture zone, and data-access boundary.

**Bidirectional traceability.** Every implementation artifact must link backward to intent and forward to verification and operational evidence.

**Conformance as code.** Architecture rules, security controls, interface contracts, and quality thresholds should be automated where practical.

**Reversibility by design.** Modernization increments require rollback, feature isolation, data migration rehearsal, and operational observability.

**Context is a maintained product.** The architecture context must evolve with each accepted change and retain provenance and confidence.

## 4. Spec-Grounded Modernization Framework

SGMF organizes modernization as a closed-loop process in which specifications and context co-evolve. The framework's eight stages are described below. The stages may be iterated within a feature branch or modernization increment, but the control sequence should not be bypassed when a change affects high-risk data, transactions, interfaces, or regulated behavior.

### 4.1 Recover the Implemented Context

The first stage creates an evidence-based architecture context from source code, build manifests, dependency files, API descriptions, database schemas, infrastructure definitions, architecture decision records, tickets, repository history, and runtime telemetry. Static tools can reveal call, import, data-flow, and ownership relationships; dynamic traces reveal runtime paths that static analysis may over- or under-approximate. The output is a versioned context graph, not merely a diagram.

### 4.2 Establish Governing Principles

The project constitution records non-negotiable principles, such as transaction integrity, backwards compatibility, privacy, observability, testing, interface versioning, dependency policy, and permitted technology choices. These principles become reusable constraints for all later specifications and plans.

### 4.3 Normalize Business Requirements

Requirement documents, tickets, policies, and stakeholder narratives are transformed into structured statements covering actors, triggers, preconditions, normal flows, exceptions, business rules, data obligations, quality attributes, and acceptance criteria. LLMs may draft this structure, but ambiguity and inferred requirements must be marked explicitly rather than presented as facts.

### 4.4 Generate and Clarify the Feature Specification

The feature specification states what must change and why, while referencing the recovered context. It identifies affected capabilities and likely architecture zones without prematurely fixing a detailed solution. Clarification questions resolve missing rules, conflict between current behavior and desired behavior, and uncertainty about legacy obligations.

### 4.5 Produce an Impact-Aware Plan

The plan joins the specification to the architecture graph. It identifies impacted modules, interfaces, schemas, events, deployment units, tests, security boundaries, owners, and operational dependencies. Architecture decisions record alternatives, trade-offs, and assumptions. The plan also defines a migration and rollback strategy.

### 4.6 Execute Bounded Implementation

Tasks are decomposed into small, reviewable increments. Agent permissions are limited to approved repositories and paths. Generated code includes provenance linking it to task and specification identifiers. Data migrations, interface changes, and dependency upgrades are separated where possible to improve reversibility and fault isolation.

### 4.7 Verify Multi-Layer Conformance

Verification combines specification coverage, unit and integration tests, contract tests, data migration tests, architecture conformance checks, static security analysis, dependency review, performance tests, observability checks, and rollback rehearsal. The release decision is based on an evidence packet, not the agent's explanation of its own output.

### 4.8 Update Context and Learn

After acceptance, the recovered context, specification set, traceability graph, decisions, and operational baselines are updated. Runtime evidence is compared with expected outcomes. Deviations create new tasks or specification changes, allowing the system context and modernization plan to converge over time.

## 5. Architecture Context Model

A brownfield context package should be explicit enough for machine reasoning but compact enough to be reviewed and refreshed. It can be represented as a graph or linked set of JSON/YAML documents. Each element should include a stable identifier, type, owner, source, version, timestamp, confidence, and relationships. Table 1 defines the recommended context dimensions.

| Dimension | Representative content | Primary evidence |
|---|---|---|
| Structural | Modules, services, packages, deployment units, libraries, repositories | Source and build analysis |
| Interface | APIs, events, files, queues, commands, contracts, versions | OpenAPI/AsyncAPI, code, gateways |
| Data | Entities, schemas, ownership, retention, consistency, migrations | DDL, ORM, catalogs, lineage |
| Behavioral | Business flows, state transitions, error paths, transaction boundaries | Code paths, tests, traces, process models |
| Quality attributes | Availability, latency, security, privacy, scalability, maintainability | SLOs, policies, tests, incidents |
| Operational | Deployments, configuration, dependencies, observability, rollback | IaC, CI/CD, telemetry, runbooks |
| Organizational | Owners, teams, approval rights, support model, domain boundaries | CODEOWNERS, catalogs, RACI |
| Historical | Decisions, deprecations, incidents, workarounds, migration attempts | ADRs, tickets, postmortems, commits |

*Table 1. Recommended architecture-context dimensions.*

Context quality should be scored along four axes: completeness, semantic accuracy, freshness, and provenance. Confidence must be attached to both nodes and relationships. For example, a direct call visible in code may receive high structural confidence, while an inferred business ownership relationship extracted from tickets may receive moderate confidence pending validation. A plan that depends on low-confidence relationships should trigger targeted investigation before implementation.

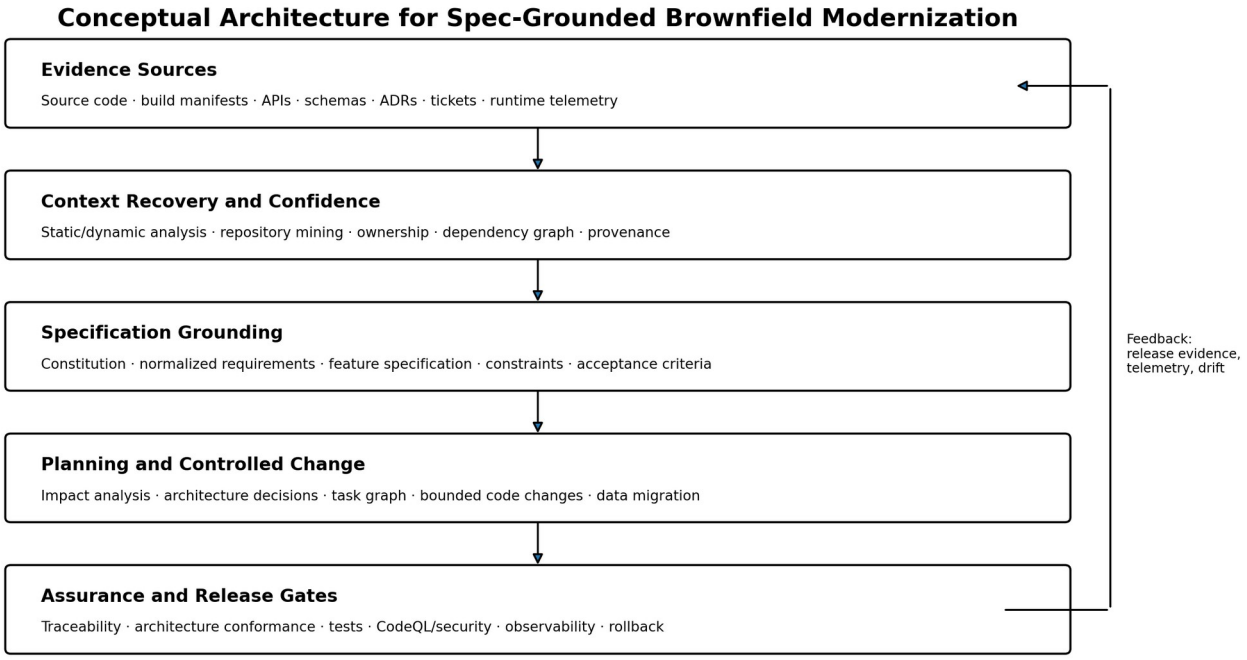


*Figure 2. Conceptual architecture for spec-grounded brownfield modernization.*

## 6. Artifact and Traceability Model

The framework treats modernization artifacts as a connected evidence graph. Stable identifiers permit automated coverage analysis and change-impact queries. The minimum artifact set is shown in Table 2. Organizations may implement the graph in repository files, an application lifecycle management platform, a knowledge graph, or a combination, provided that links remain versioned and queryable.

| Artifact | Purpose | Accountable owner |
|---|---|---|
| Project constitution | Engineering and governance principles | Architect / engineering leadership |
| Architecture context snapshot | Recovered system evidence and confidence | Architecture / platform team |
| Normalized requirement set | Business rules, quality attributes, exceptions | Product owner / business analyst |
| Feature specification | Outcome, scenarios, constraints, acceptance criteria | Product and engineering |
| Impact and implementation plan | Affected elements, decisions, migration, rollback | Technical lead / architect |
| Task graph | Reviewable implementation and assurance tasks | Engineering team |
| Change manifest | Files, schemas, dependencies, interfaces, provenance | Developer / AI agent |
| Verification evidence | Tests, scans, conformance, performance, approvals | Quality and security teams |
| Release evidence | Deployment, flags, rollback, known risks | Release owner |
| Operational outcome record | SLOs, incidents, adoption, business outcome | Service owner |

*Table 2. Core artifacts and accountable ownership.*

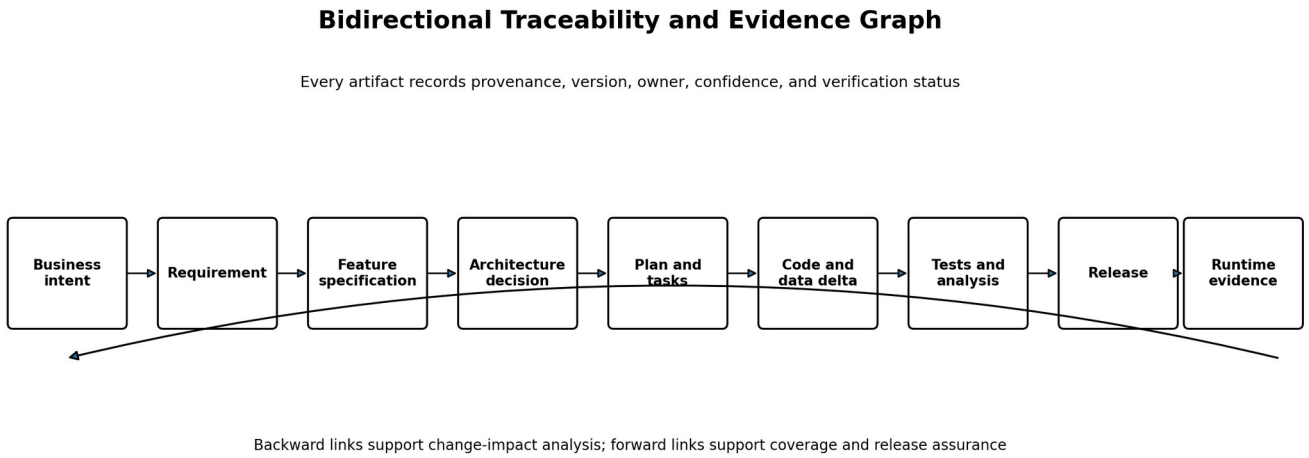


*Figure 3. Bidirectional traceability and evidence chain.*

The traceability graph enables three critical queries. Coverage asks whether every approved requirement has corresponding design, implementation, and verification evidence. Impact asks which downstream artifacts may be affected when a requirement, interface, or architecture element changes. Justification asks why a code or configuration element exists and which outcome or constraint it serves. These queries provide stronger governance than a narrative AI explanation because they can be independently recomputed from versioned artifacts.

## 7. Governance, Security, and Human Control

Spec grounding reduces but does not eliminate risk. AI agents can misinterpret context, follow malicious repository instructions, expose confidential source code, introduce vulnerable dependencies, or optimize for stated acceptance criteria while violating unstated operational obligations. Governance should therefore combine preventive constraints, detective controls, and recoverability.

| Risk | Failure mode | Control |
|---|---|---|
| Stale or incomplete context | Plan omits hidden coupling or operational dependency | Freshness checks; confidence thresholds; targeted dynamic tracing |
| Hallucinated requirement | Generated specification invents a rule or removes an exception | Source citation; explicit inference labels; stakeholder approval |
| Over-broad code change | Agent modifies unrelated modules or refactors beyond scope | Path allowlists; diff-size thresholds; task-level branches |
| Architecture drift | Change introduces duplicate capability or forbidden dependency | Architecture tests; dependency rules; ADR review |
| Weak test oracle | Tests confirm generated behavior but not intended behavior | Independent acceptance criteria; mutation tests; human scenario review |
| Security regression | New vulnerability, secret exposure, or unsafe dependency | CodeQL; dependency scanning; secret scanning; threat review |
| Data migration failure | Loss, inconsistency, or irreversible schema change | Dual-read/write strategy; rehearsal; backup; rollback checkpoints |
| Traceability decay | Links become stale after manual edits or later changes | CI link validation; converge step; release-blocking coverage thresholds |

*Table 3. Principal risks and assurance controls.*

### 7.1 Human Approval Gates

- Gate A - specification approval: confirms business rules, exceptions, quality attributes, and non-goals.
- Gate B - architecture and impact approval: confirms affected boundaries, trade-offs, data and interface implications, migration, and rollback.
- Gate C - change approval: reviews code and data deltas, provenance, test adequacy, dependency changes, and security findings.
- Gate D - release approval: confirms operational readiness, observability, support ownership, feature isolation, and rollback evidence.
- Gate E - outcome review: compares operational and business evidence with the specification and updates the context and backlog.

### 7.2 Mapping to the Official Spec Kit Workflow

Table 4 maps the framework to the current core and optional Spec Kit commands. Context recovery and architecture validation are modeled as extensions or external services because they are not standard core commands. This mapping preserves fidelity to the official toolkit while retaining the source article's modernization intent.

| SGMF activity | Spec Kit mechanism | Brownfield extension |
|---|---|---|
| Govern | /speckit.constitution | Add brownfield principles, security, compatibility, and transaction-integrity policies |
| Normalize / Specify | /speckit.specify and /speckit.clarify | Ground requirements in context identifiers and explicit evidence |
| Plan | /speckit.plan | Perform architecture impact analysis; create ADRs, migration, and rollback plan |
| Decompose | /speckit.tasks | Generate bounded development and assurance tasks |
| Analyze | /speckit.analyze and /speckit.checklist | Check cross-artifact consistency, coverage, and quality criteria |
| Implement | /speckit.implement | Execute approved tasks with limited permissions and provenance |
| Converge | /speckit.converge | Compare codebase with spec, plan, and tasks; add remaining work |
| Recover / Validate | Extension or adjacent pipeline service | Generate context snapshots and enforce architecture/security/operability gates |

*Table 4. Alignment with the current official Spec Kit workflow.*

## 8. Evaluation Model

The framework should be evaluated through controlled comparisons on real or representative repositories. A pilot can assign comparable modernization features to a conventional AI-assisted workflow and to SGMF, using the same developers, models, repositories, and review standards where feasible. Measurements should include both speed and downstream quality; a faster implementation with lower correctness, higher review effort, or more architectural drift is not a successful outcome.

### 8.1 Modernization Assurance Index

The Modernization Assurance Index (MAI) summarizes release readiness across six dimensions. Each dimension is scored from 0 to 100 using predefined evidence. Weights are illustrative and should be calibrated by risk and domain.

$$MAI = 0.18SC + 0.18AC + 0.16TC + 0.18VA + 0.15SR + 0.15OR$$

*SC = specification completeness; AC = architecture conformance; TC = traceability completeness; VA = verification adequacy; SR = security and risk control; OR = operability and reversibility.*

| Dimension | Weight | Evidence examples |
|---|---|---|
| Specification completeness (SC) | 18% | Rules, scenarios, exceptions, quality attributes, acceptance criteria |
| Architecture conformance (AC) | 18% | Boundary rules, dependency constraints, interface and data consistency |
| Traceability completeness (TC) | 16% | Bidirectional links from intent through runtime evidence |
| Verification adequacy (VA) | 18% | Unit, integration, contract, migration, performance, and acceptance evidence |
| Security and risk control (SR) | 15% | Static analysis, dependency and secret checks, threat mitigations |
| Operability and reversibility (OR) | 15% | Telemetry, deployment readiness, feature isolation, rollback rehearsal |

*Table 5. Illustrative Modernization Assurance Index dimensions.*

Illustrative decision thresholds are: MAI below 60, experiment only; 60-74, controlled pilot; 75-84, limited production with heightened monitoring; and 85 or above, eligible for standard enterprise release. A high aggregate score must not override a critical failure. For example, unresolved high-severity security findings, missing rollback for destructive data migration, or unapproved business-rule changes remain release blockers regardless of the total.

## 8.2 Core Outcome Metrics

| Category | Example measures | Evaluation question |
|---|---|---|
| Delivery | Lead time, implementation time, review time, rework effort | Does the method improve flow without shifting work downstream? |
| Correctness | Acceptance pass rate, escaped defects, regression defects | Does the change implement the intended behavior? |
| Architecture | New violations, duplicate capability, coupling change, smell count | Does the system remain structurally coherent? |
| Traceability | Requirement-to-test coverage, link precision/recall, stale links | Can impact and justification be reconstructed? |
| Security | Introduced vulnerabilities, dependency risk, secrets, remediation time | Does AI-assisted change preserve secure development outcomes? |
| Operability | SLO change, incident rate, rollback success, observability coverage | Can the change be operated and reversed safely? |
| Human factors | Comprehension, trust calibration, onboarding time, cognitive load | Does the evidence help reviewers understand and control the change? |

*Table 6. Evaluation metrics for an industrial pilot.*

## 8.3 Research Propositions

- P1. Architecture-context grounding will reduce changes that create duplicate modules or violate established dependency boundaries compared with prompt-only AI-assisted development.
- P2. Structured specifications and bidirectional traceability will improve maintenance-task correctness and reduce reviewer effort, particularly when requirements affect multiple artifacts.
- P3. Context confidence and freshness will moderate the effectiveness of spec-grounded modernization; stale context may create false assurance.
- P4. Multi-layer evidence gates will reduce escaped defects and security regressions but may increase early-stage planning effort.
- P5. The greatest net benefit will occur for medium-to-high complexity changes with distributed rules and dependencies, rather than trivial localized edits.

## 9. Illustrative Brownfield Billing Scenario

Consider a Java-based billing service with modules for bill retrieval, payment posting, adjustments, balance calculation, and persistence. A new rule requires that a payment not be finalized while earlier transactions remain pending, and that pending payments be represented as provisional credits in balance calculations. Although the requirement appears local, it changes transaction state, balance semantics, data schema, API responses, reporting, reconciliation, and operational monitoring.

### 9.1 Context Recovery

The context snapshot identifies the PaymentService, BalanceService, repositories, transaction-status schema, REST interfaces, clearing events, reconciliation job, and dashboards. Static analysis shows method and data dependencies; runtime traces reveal that the reconciliation job updates balances asynchronously after clearing. The architecture record states that the balance service is the sole owner of calculated balance state. These facts constrain the solution and prevent the agent from duplicating balance logic in the payment service.

### 9.2 Specification

- When an account has one or more prior PENDING transactions, a new payment is accepted but remains PENDING and must not be included as cleared funds.
- Pending payments are displayed as provisional credits and must not reduce the settled amount due.
- When all predecessor transactions clear, the payment is eligible for transition to CLEARED through the existing clearing workflow.
- The API must preserve compatibility for existing clients; a new status field is optional during the transition window.
- Reconciliation totals, audit events, and observability metrics must distinguish pending, cleared, and failed transactions.
- No direct balance mutation is permitted outside BalanceService.

### 9.3 Impact-Aware Plan and Tasks

The plan maps each rule to affected methods, schema changes, event handlers, API contract changes, tests, dashboards, and migration actions. It selects an additive transaction-status column, a backward-compatible API field, and a feature flag. Tasks separate schema migration, domain-state changes, balance calculations, contract updates, reconciliation behavior, tests, and telemetry. Each task carries requirement and architecture identifiers.

### 9.4 Verification and Release

Verification includes state-transition tests, ordering and concurrency tests, API contract tests, balance invariant tests, migration tests on production-like data, CodeQL and dependency scans, architecture checks enforcing the BalanceService ownership rule, and a rollback rehearsal. The release begins with shadow calculation and comparison metrics, then a limited cohort, before full activation. Runtime discrepancies create new convergence tasks and update the context snapshot.

| Release criterion | Illustrative acceptance condition |
|---|---|
| Requirement coverage | 100% of approved rules linked to tests and code changes |
| Architecture conformance | No new ownership or dependency violations |
| Transaction integrity | No double posting; invariant violations = 0 in test and shadow mode |
| Compatibility | Existing client contract tests pass |
| Security | No new high or critical findings |
| Operability | Metrics, logs, alert thresholds, feature flag, and rollback verified |

*Table 7. Example release evidence for the billing scenario.*

# 10. Discussion

## 10.1 From Prompt Grounding to System Grounding

Retrieval-augmented prompting is often described as grounding, but brownfield modernization requires more than retrieving nearby source files. System grounding includes architectural boundaries, runtime behavior, data ownership, quality attributes, operational constraints, and historical decisions. It also requires an explicit mechanism for representing uncertainty and conflict. SGMF therefore treats context as a governed engineering artifact with provenance and confidence, not as an opaque bundle of text inserted into a model prompt.

## 10.2 Specifications as Coordination and Control Artifacts

Specifications serve multiple roles: they communicate intent, constrain generation, drive task decomposition, provide test oracles, and support audit. Their value depends on precision without premature design lock-in. The framework separates feature behavior from implementation plans so that teams can review business meaning before debating technology. This also makes it possible to regenerate plans when the architecture context changes while preserving approved intent.

## 10.3 Architecture Work Changes Rather Than Disappears

AI agents do not eliminate architecture work. They shift architecture toward explicit principles, machine-readable context, automated conformance, evidence review, and continuous evolution. Architects must define boundaries and decision rights, curate context, resolve conflicts between intended and implemented architecture, and design controls proportionate to risk. The framework therefore supports an architecture-as-code operating model while preserving human accountability for high-impact decisions.

## 10.4 Cost and Adoption Considerations

Spec grounding introduces upfront work: recovering architecture, cleaning requirements, defining policies, instrumenting traceability, and automating gates. The investment is unlikely to be justified for every small change. Adoption should prioritize systems where change impact is difficult to understand, business rules are distributed, defects are costly, compliance evidence is required, or multiple teams modify shared capabilities. Reusable organizational presets, extensions, context schemas, and assurance pipelines can reduce marginal cost over time.

# 11. Limitations and Future Research

This article presents a conceptual artifact and an illustrative scenario, not a completed industrial experiment. The framework requires validation across languages, architectural styles, repository sizes, regulatory contexts, and AI coding agents. The MAI weights and thresholds are design hypotheses rather than universal standards. Context recovery tools may produce incomplete or noisy models, and organizations may lack the documentation or runtime telemetry needed to validate inferred relationships.

Future research should develop benchmark repositories with intentionally incomplete documentation, hidden dependencies, architecture rules, and representative change requests. Experiments should compare prompt-only, repository-context, specification-driven, and full spec-grounded workflows. Important research questions include how to measure context confidence, how much traceability automation remains accurate over time, how agent autonomy affects reviewer behavior, whether evidence packets improve trust calibration, and how to prevent specification artifacts from becoming a new source of stale documentation.

Further work should also examine multi-service and event-driven systems, database decomposition, mainframe integration, regulated workflows, and modernization across organizational boundaries. Tooling research is needed for incremental context updates, graph-based impact analysis, architecture-policy generation, and evaluation of semantic alignment between requirements, code, and runtime behavior.

# 12. Conclusion

AI-assisted implementation can accelerate brownfield modernization only when generation is constrained by reliable evidence about the system and explicit agreement about intended behavior. The Spec-Grounded

Modernization Framework integrates architecture recovery, specification-driven development, impact-aware planning, bounded implementation, multi-layer verification, and continuous context evolution. It aligns the official Spec Kit lifecycle with brownfield-specific extensions for context and assurance, while avoiding the inaccurate assumption that AI tooling natively understands an enterprise system's architecture.

The framework's central principle is that modernization should be governed through a versioned evidence chain connecting business intent, specification, architecture, tasks, implementation, verification, release, and operational outcomes. This evidence chain makes AI-assisted changes more explainable, reviewable, reversible, and measurable. It also provides a practical path for organizations to use AI coding agents without surrendering architecture discipline, security controls, or accountability for long-lived software systems.

## References

[1] B. Chokhawala, "Spec-Grounded Modernization: Leveraging AI Specification Kits for Brownfield Software Systems," KAIRI / Medium, Oct. 31, 2025.

[2] GitHub, "Spec Kit: Define what to build before building it—with any AI coding agent," github/spec-kit, 2026. Accessed July 21, 2026.

[3] L. Bass, P. Clements, and R. Kazman, Software Architecture in Practice, 4th ed. Addison-Wesley, 2021.

[4] P. Clements et al., Documenting Software Architectures: Views and Beyond, 2nd ed. Addison-Wesley, 2010.

[5] ISO/IEC/IEEE 42010:2022, Systems and Software Engineering—Architecture Description, International Organization for Standardization, 2022.

[6] M. Soliman, M. Albonico, I. Malavolta, and A. Wortmann, "Mining software repositories for software architecture—A systematic mapping study," Information and Software Technology, vol. 181, art. 107677, 2025. doi: 10.1016/j.infsof.2025.107677.

[7] P. Mäder and A. Egyed, "Assessing the effect of requirements traceability for software maintenance," in Proc. 28th IEEE International Conference on Software Maintenance, 2012, pp. 171-180. doi: 10.1109/ICSM.2012.6405269.

[8] P. Mäder and A. Egyed, "Do developers benefit from requirements traceability when evolving and maintaining a software system?" Empirical Software Engineering, vol. 20, pp. 413-441, 2015. doi: 10.1007/s10664-014-9314-z.

[9] F. Tian, P. Liang, and M. A. Babar, "The impact of traceability on software maintenance and evolution: A mapping study," Journal of Software: Evolution and Process, vol. 33, no. 10, e2374, 2021. doi: 10.1002/smr.2374.

[10] D. Xie et al., "How effective are large language models in generating software specifications?" arXiv:2306.03324, 2023.

[11] M. Krishna, B. Gaur, A. Verma, and P. Jalote, "Using LLMs in software requirements specifications: An empirical evaluation," arXiv:2404.17842, 2024.

[12] B. Wei et al., "From requirements to code with LLMs," arXiv:2406.10101, 2024.

[13] R. Al-Msie'deen, "Requirements traceability: Recovering and visualizing traceability links between requirements and source code of object-oriented software systems," arXiv:2307.05188, 2023.

[14] M. Souppaya, K. Scarfone, and D. Dodson, Secure Software Development Framework (SSDF) Version 1.1, NIST SP 800-218, 2022. doi: 10.6028/NIST.SP.800-218.

[15] H. Booth et al., Secure Software Development Practices for Generative AI and Dual-Use Foundation Models: An SSDF Community Profile, NIST SP 800-218A, 2024. doi: 10.6028/NIST.SP.800-218A.

[16] GitHub, "CodeQL documentation: Query-based static analysis," 2026. Accessed July 21, 2026.

[17] R. Seacord, D. Plakosh, and G. Lewis, Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices. Addison-Wesley, 2003.

[18] P. Kruchten, "The 4+1 view model of architecture," IEEE Software, vol. 12, no. 6, pp. 42-50, 1995. doi: 10.1109/52.469759.

[19] R. Kazman, M. Klein, and P. Clements, ATAM: Method for Architecture Evaluation, CMU/SEI-2000-TR-004, Software Engineering Institute, 2000.

[20] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, 2003.

[21] O. Gotel and A. Finkelstein, "An analysis of the requirements traceability problem," in Proc. First International Conference on Requirements Engineering, 1994, pp. 94-101. doi: 10.1109/ICRE.1994.292398.

[22] P. Rempel and P. Mäder, “Preventing defects: The impact of requirements traceability completeness on software quality,” IEEE Transactions on Software Engineering, vol. 43, no. 8, pp. 777-797, 2017. doi: 10.1109/TSE.2016.2622264.

[23] M. Fowler, “Strangler Fig Application,” martinfowler.com, 2019.

## Author Note

This manuscript expands and formalizes the author’s practitioner article published on Medium. It is intended as a conceptual research contribution and a foundation for empirical industrial validation. The illustrative scenario and assurance thresholds are design examples and should not be interpreted as independently validated performance results.