Software Architecture Documentation

Supporting Development

A Comprehensive Guide to Using Architecture Documentation
for Implementation, Testing, Integration, and Deployment

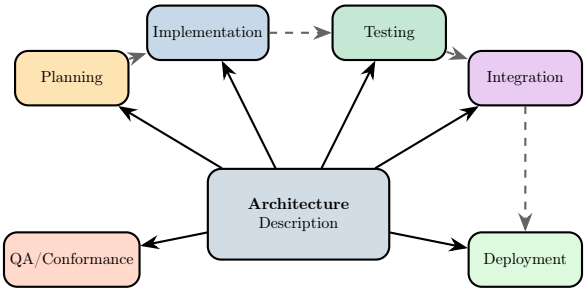
Architecture Documentation Series

Based on SEI Views and Beyond and Industry Best Practices

December 9, 2025

Abstract

Architecture documentation serves as the blueprint for system implementation. This comprehensive guide examines how architecture descriptions support the full development lifecycle—from project planning and resource estimation through implementation, testing, integration, and deployment. The document provides detailed question sets for each development stakeholder, checklists for assessing documentation readiness, and guidance for establishing conformance verification processes. Whether you are a software manager planning development resources, a developer implementing components, a tester designing test strategies, or an integrator planning system assembly, this guide ensures that architecture documentation provides the information you need to succeed.



Architecture Driving Development

Contents

1 Introduction

1.1 Purpose of This Guide

This guide examines how architecture documentation supports the software development lifecycle. Effective architecture documentation serves as a blueprint that enables development teams to:

- Understand what to build and how components fit together
- Plan and estimate development resources accurately
- Implement components that conform to architectural intent
- Test systems against architectural requirements
- Integrate components into a working whole
- Deploy systems to production environments
- Verify conformance between implementation and architecture

1.2 The Architecture-Development Relationship

Definition

Development Support: The degree to which architecture documentation provides sufficient information for development stakeholders to perform their roles effectively and know when their work is complete.

Architecture documentation bridges the gap between requirements and implementation. It provides the "what" and "why" that guide the "how" of implementation.

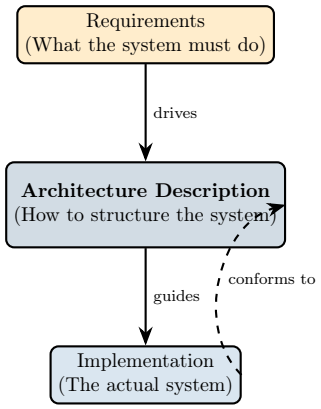


Figure 1: Architecture as Bridge Between Requirements and Implementation

1.3 Stakeholder Overview

Development support involves multiple stakeholder groups, each with distinct information needs:

Table 1: Development Stakeholders and Their Concerns

Stakeholder	Primary Role	Key Information Needs
Software Manager	Resource planning and scheduling	Implementation units; dependencies; effort estimates
Designer	Detailed design decisions	Constraints; patterns; interfaces; rationale
Implementer	Code development	Component specs; APIs; coding standards
Unit Tester	Component verification	Test criteria; dependencies; test data needs
Integrator	Component assembly	Integration order; interface contracts; runtime dependencies
System Tester	End-to-end verification	Test scenarios; quality requirements; acceptance criteria
QA/Conformance	Quality assurance	Conformance points; verification methods; traceability
Fielder/Deployer	Production deployment	Deployment topology; configuration; operational requirements

2 Implementation Unit Identification

2.1 What is an Implementation Unit?

Definition

An **implementation unit** is a portion of the system that can be assigned to a developer or team for implementation, has identifiable interfaces, can be separately compiled or built, and can be independently tested to some degree.

The architecture documentation must clearly identify all implementation units and their relationships.

2.2 Unit Identification from Architecture Views

Different architectural views contribute to implementation unit identification:

Table 2: Views and Implementation Unit Information

View Type	Unit Information	Development Use
Module Decomposition	Code modules; packages; classes	Work breakdown; team assignment
Uses View	Module dependencies	Build order; impact analysis

Layered View	Layer membership; allowed dependencies	Dependency rules; portability
Component-Connector	Runtime components; connectors	Runtime behavior; communication
Deployment View	Nodes; allocation of components	Deployment planning; resource needs
Data Model View	Data entities; relationships	Database design; data access

2.3 Implementation Unit Registry

Implementation Unit Registry Template

Unit ID: [Unique identifier, e.g., IU-ORDER-001]
Unit Name: [Descriptive name]
Description: [Brief description of responsibility]
Architecture Element: [Reference to architectural element(s)]
Type: [Service / Library / Component / Module / Database / Configuration]
Owner: [Team or individual responsible]
Dependencies:
- Build-time: [Units required for compilation]
- Runtime: [Units required for execution]
- Test: [Units/resources required for testing]

Interfaces:
- Provided: [APIs/services this unit exposes]
- Required: [APIs/services this unit consumes]

Quality Requirements:
- Performance: [Response time, throughput targets]
- Reliability: [Availability, error handling requirements]
- Security: [Authentication, authorization requirements]

Constraints: [Technology, standards, patterns that must be followed]
Estimated Effort: [Story points / person-days / lines of code]
Status: [Not Started / In Progress / Code Complete / Tested / Integrated]

2.4 Example Implementation Unit Catalog

Table 3: Example Implementation Unit Catalog

Unit ID	Name	Type	Owner	Build Deps	Effort
IU-001	Order Service	Service	Order Team	IU-010, IU-011	40 pts
IU-002	Payment Service	Service	Payment Team	IU-010, IU-012	35 pts
IU-003	Inventory Service	Service	Inventory Team	IU-010	30 pts

IU-004	User Service	Service	Identity Team	IU-010, IU-012	25 pts
IU-005	API Gateway	Component	Platform Team	IU-010	20 pts
IU-010	Common Library	Library	Platform Team	–	15 pts
IU-011	Messaging Library	Library	Platform Team	IU-010	10 pts
IU-012	Security Library	Library	Security Team	IU-010	15 pts
IU-020	Order Database	Database	DBA Team	–	10 pts
IU-021	User Database	Database	DBA Team	–	8 pts

3 Development Planning Support

3.1 Resource Estimation

Architecture documentation should enable accurate resource estimation by providing:

- Clear scope definition for each implementation unit
- Complexity indicators (interfaces, dependencies, quality requirements)
- Technology constraints affecting effort
- Reuse opportunities (existing components, libraries, patterns)
- Risk factors (new technology, complex integration, unclear requirements)

Best Practice

Estimation Guidance from Architecture:

1. **Count implementation units:** Establish baseline scope
2. **Assess complexity:** Consider interfaces, dependencies, quality requirements
3. **Identify reuse:** Reduce estimates for reused/adapted components
4. **Add integration effort:** Account for component integration work
5. **Include architecture-specific tasks:** Frameworks, infrastructure, cross-cutting concerns
6. **Apply risk factors:** Adjust for uncertainty and technical risk

3.2 Dependency Analysis

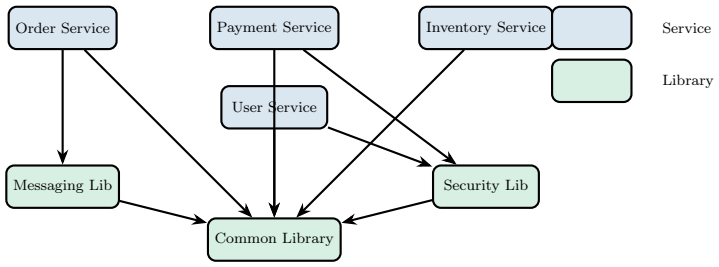


Figure 2: Build Dependency Graph

3.3 Development Schedule Planning

Architecture dependencies determine feasible development sequences:

Table 4: Development Phasing Based on Dependencies

Phase	Units	Prerequisites	Parallel Opportunities
Phase 1	Common Library	None	Single unit
Phase 2	Messaging Lib, Security Lib	Common Library	Both can be parallel
Phase 3	Order, Payment, Inventory, User Services	Phase 2 complete	All services can be parallel
Phase 4	API Gateway	Services available	Depends on service APIs
Phase 5	Integration Testing	All units complete	–

3.4 Parallel Development Identification

Key Point

Enabling Parallel Development:
Architecture documentation enables parallel development when it clearly defines:

- **Interface contracts:** Teams can work against defined interfaces before implementations exist
- **Dependency direction:** Lower-level components developed first or mocked
- **Integration points:** Clear handoff points between teams
- **Shared resources:** Coordination requirements for shared components

4 Design Guidance and Constraints

4.1 Architectural Constraints

The architecture documentation should clearly communicate constraints that implementations must follow:

Table 5: Types of Architectural Constraints

Constraint Type	**Description**	**Examples**
Technology	Required or prohibited technologies	"Must use Java 17+"; "No vendor-specific SQL"
Dependency	Allowed/prohibited module dependencies	"UI layer cannot access database directly"
Communication	Required communication patterns	"Services communicate via REST or events only"
Data	Data management rules	"Each service owns its data"; "No shared databases"
Security	Security requirements	"All external APIs require authentication"
Performance	Performance boundaries	"Response time <500ms at p95"
Pattern	Required design patterns	"Use repository pattern for data access"
Standard	Standards compliance	"RESTful APIs follow OpenAPI 3.0"

4.2 Design Rules and Principles

Architectural Design Rules

Dependency Rules:
1. Higher layers may depend on lower layers, not vice versa
2. Domain layer has no external dependencies
3. Infrastructure adapters depend on domain interfaces
4. No circular dependencies between modules

Communication Rules:
1. Synchronous calls only for queries; commands use async messaging
2. All inter-service communication through defined APIs
3. No direct database access across service boundaries
4. Events published for all state changes

Data Rules:
1. Each service owns its database schema
2. No foreign keys across service boundaries
3. Data replication via events, not shared tables
4. Sensitive data encrypted at rest and in transit

Error Handling Rules:
1. All exceptions logged with correlation ID
2. Circuit breakers on all external calls
3. Retry with exponential backoff for transient failures
4. Graceful degradation when dependencies unavailable

4.3 Pattern and Style Guidance

Table 6: Architectural Patterns and Their Application

Pattern	When to Apply	Implementation Guidance
Repository	Data access layer	Abstract data store; return domain objects
Factory	Complex object creation	Encapsulate construction logic
Strategy	Varying algorithms	Interface with multiple implementations
Observer/Event	State change notification	Event bus for loose coupling
Circuit Breaker	External service calls	Fail fast after threshold; auto-recover
Retry	Transient failures	Exponential backoff; max attempts
Saga	Distributed transactions	Compensating actions for rollback
CQRS	Read/write separation	Separate models for queries and commands

4.4 Variability and Change Guidance

Architecture documentation should identify what is likely to change and how to accommodate it:

Table 7: Change Accommodation Guidance

Change Type	**Architecture Provision**	**Implementation Approach**
New payment provider	Payment gateway abstraction	Implement new adapter; configure
Database migration	Repository pattern	Change repository implementation
UI framework change	Layered architecture	Replace UI layer only
New business rules	Rule engine / Strategy	Add new rule implementation
Scaling requirements	Stateless services	Horizontal scaling; no code change
New integration	API gateway / Adapters	Add new adapter; register route

5 Testing Support

5.1 Architecture-Based Testing

Architecture documentation supports testing by defining what to test, how to test it, and when testing is complete.

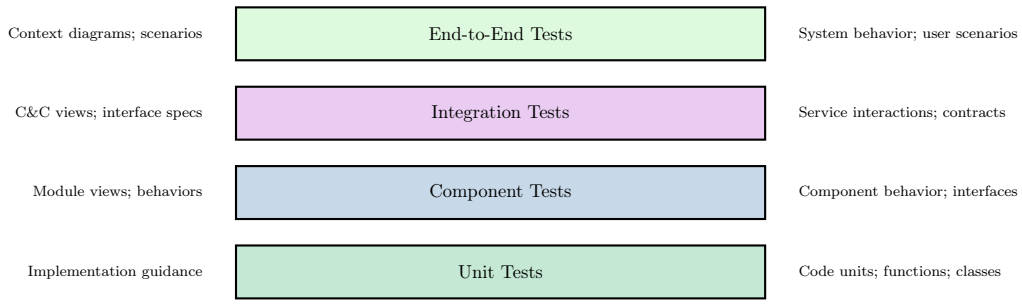


Figure 3: Test Pyramid and Architecture Documentation

5.2 Test Information from Architecture

Architecture-Derived Test Information

From Module Views:
- Unit boundaries for isolation testing
- Dependencies requiring mocks/stubs
- Internal interfaces for component testing

From C&C Views:
- Service contracts for contract testing
- Communication paths for integration testing
- Runtime configurations for scenario testing

From Behavior Descriptions:
- Scenarios for end-to-end testing
- State transitions for state-based testing
- Error conditions for negative testing

From Quality Requirements:
- Performance targets for load testing
- Reliability requirements for chaos testing
- Security requirements for penetration testing

From Deployment Views:
- Environment configurations for deployment testing
- Resource requirements for capacity testing
- Network topology for failover testing

5.3 Test Success Criteria

Table 8: Architecture-Based Test Success Criteria

Test Level	Success Criteria Source	Example Criteria
Unit Test	Interface specifications	All public methods tested; edge cases covered
Component Test	Component behavior specs	All interfaces exercised; error paths tested
Integration Test	Interface contracts	Contracts verified; communication paths tested
System Test	Quality attribute scenarios	Performance targets met; security verified
Acceptance Test	Stakeholder scenarios	Use cases pass; business rules verified

5.4 Test Environment Requirements

Architecture documentation should specify test environment needs:

- **Unit testing:** Minimal; mocks for dependencies

- **Component testing:** Component under test; stubbed dependencies
- **Integration testing:** Multiple components; test databases; message brokers
- **System testing:** Full environment; production-like data; external system simulators
- **Performance testing:** Scaled environment; load generators; monitoring

6 Integration and Deployment Support

6.1 Integration Planning

Architecture documentation supports integration by defining:

Table 9: Integration Information from Architecture

Information	Source	Integration Use
Integration units	Module decomposition	What must be integrated
Integration order	Dependency analysis	Sequence of integration
Interface contracts	Interface specifications	Verification criteria
Runtime dependencies	C&C view	Resources needed for integration
Integration test cases	Behavior descriptions	How to verify integration
Configuration	Deployment view	Environment setup

6.2 Integration Strategies

Architecture-Driven Integration Strategies

Bottom-Up Integration:
- Start with lowest-level components (libraries, utilities)
- Progress upward through dependency layers
- Each level tested before moving up
- Requires drivers to test components

Top-Down Integration:
- Start with highest-level components
- Progressively integrate lower levels
- Uses stubs for unimplemented components
- Early validation of system structure

Feature-Based Integration:
- Integrate components needed for specific features
- Vertical slices through architecture
- Early delivery of working functionality
- May require both stubs and drivers

Continuous Integration:
- Integrate frequently (multiple times per day)
- Automated build and test
- Early detection of integration issues
- Requires robust test automation

6.3 Deployment Information

Table 10: Deployment Information from Architecture

Information	Architecture Source	Deployment Use
Deployment topology	Deployment view	Infrastructure provisioning
Component allocation	Deployment mapping	Where to deploy each component
Resource requirements	Quality attribute analysis	Sizing and capacity
Configuration	Variability guide	Environment-specific settings
Dependencies	Runtime dependencies	Deployment order; health checks
Network requirements	C&C communication	Firewall rules; load balancers

6.4 Deployment Checklist

Architecture-Based Deployment Checklist

Pre-Deployment:

- ☐ All implementation units identified in architecture are built
- ☐ All integration tests pass
- ☐ Deployment topology matches architecture deployment view
- ☐ Configuration matches architecture specifications
- ☐ Resource allocation meets architecture requirements

Deployment Verification:

- ☐ All components deployed to correct nodes
- ☐ All communication paths operational
- ☐ All external integrations connected
- ☐ Health checks passing
- ☐ Monitoring and alerting configured

Post-Deployment:

- ☐ Smoke tests pass
- ☐ Performance within architecture targets
- ☐ Security controls verified
- ☐ Deployment documented

7 Conformance Verification

7.1 What is Architectural Conformance?

Definition

Architectural Conformance: The degree to which the implemented system adheres to the prescribed architecture. Conformance verification ensures that implementation decisions align with architectural intent.

7.2 Conformance Points

Architecture documentation should identify specific conformance points—aspects of the implementation that must match the architecture:

Table 11: Conformance Point Categories

Category	Conformance Points	Verification Method
Structure	Module decomposition matches architecture	Static analysis; code review
Dependencies	Only allowed dependencies exist	Dependency analysis tools

Category	Conformance Points	Verification Method
Interfaces	APIs match specifications	Contract testing; API comparison
Behavior	Runtime behavior matches scenarios	Integration testing; monitoring
Quality	Quality targets met	Performance testing; security scanning
Deployment	Deployment matches topology	Infrastructure inspection
Standards	Coding standards followed	Linting; static analysis
Patterns	Required patterns implemented	Code review; architecture fitness functions

7.3 Conformance Verification Methods

Conformance Verification Approaches

Static Analysis:
- Dependency checking (forbidden dependencies)
- Layer violation detection
- Coding standard enforcement
- Architecture rule checking

Dynamic Analysis:
- Runtime behavior verification
- Performance measurement
- Communication pattern verification
- Resource usage monitoring

Manual Review:
- Architecture review boards
- Code reviews with architecture focus
- Design document reviews
- Deployment verification

Architecture Fitness Functions:
- Automated architecture tests
- Continuous conformance checking
- Metrics and thresholds
- Build pipeline integration

7.4 Conformance Verification Process

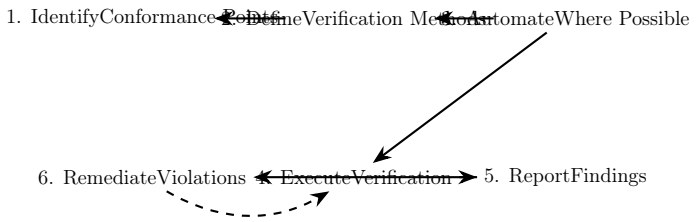


Figure 4: Conformance Verification Process

8 Quality Assurance Support

8.1 QA Information Needs

QA stakeholders require specific information from architecture documentation:

Table 12: QA Information Requirements

QA Activity	Required Information	Architecture Source
Baseline management	Version; status; change history	Document control section
Decision tracking	Key decisions; rationale	Rationale documentation
Traceability	Requirements to architecture mapping	Traceability matrices
Conformance planning	Conformance points; methods	Conformance specification
Issue management	Open issues; deferred decisions	Open issues section
Test alignment	Test approach consistency	Behavior descriptions

8.2 Architecture Baselining

Best Practice

Architecture Baseline Requirements:

1. **Version identification:** Clear version number and date
2. **Status indication:** Draft/Review/Approved/Deprecated
3. **Change history:** Record of all changes with rationale
4. **Approval record:** Who approved and when
5. **Distribution list:** Who has received the baseline
6. **Configuration control:** How changes are managed

8.3 Open Decisions and Deferred Items

Architecture documentation should clearly identify:

- **Open decisions:** Architectural questions not yet resolved
- **Deferred decisions:** Decisions intentionally left to implementation
- **Known issues:** Problems or limitations in the architecture
- **Technical debt:** Shortcuts that need future attention
- **Assumptions:** Assumptions that may need validation

Table 13: Open Items Tracking

ID	Description	Type	Impact	Resolution
OI-001	Cache eviction strategy	Deferred	Medium	Implementer choice
OI-002	Multi-region deployment	Open	High	Q3 decision
OI-003	Legacy API deprecation	Known Issue	Medium	Migration plan needed
OI-004	Database sharding approach	Open	High	Performance testing

9 Review Question Sets

9.1 Questions for Software Managers

Software Manager Questions

Implementation Unit Identification:

1. Can you identify the complete set of implementation units from the architecture documentation?
2. Is each unit clearly defined with scope and boundaries?
3. Are unit ownership and responsibilities clearly assigned?

Resource Planning:

4. Can you estimate development effort for each unit?
5. Are complexity factors (interfaces, quality requirements) documented?
6. Can you identify reuse opportunities that reduce effort?
7. Are there risk factors that affect estimates?

Dependency Management:

8. Can you determine build dependencies between units?
9. Can you identify runtime dependencies?
10. Are dependency directions clear (what depends on what)?

Schedule Planning:

11. Can you lay out a development schedule based on dependencies?
12. Can you identify units that can be developed in parallel?
13. Can you plan for architecture prototype/proof of concept?
14. Are milestone criteria defined?

Constraint Assessment:

15. Does the architecture overconstrain developers unnecessarily?
16. Are constraints clearly justified with rationale?
17. Is there appropriate flexibility for implementation decisions?

9.2 Questions for Designers and Implementers

Designer/Implementer Questions

Dependency Understanding:

1. Can you identify allowed dependencies for your unit?
2. Can you identify prohibited dependencies?
3. Are dependency rules clearly documented and justified?

Constraints and Patterns:

4. Can you identify applicable architectural constraints?
5. Are required patterns and styles documented?
6. Is guidance provided for pattern application?
7. Are technology constraints clear?

Requirements Traceability:

8. Can you navigate from your unit to its requirements?
9. Are quality requirements for your unit clear?
10. Are derived requirements documented?

Cross-Cutting Concerns:

11. Is error handling approach defined?
12. Is resource management approach defined?
13. Is data persistence approach defined?
14. Is security implementation approach defined?
15. Is logging and monitoring approach defined?

Change and Evolution:

16. Can you identify what is likely to change?
17. Is impact of changes on your design clear?
18. Are variation points documented?
19. Is decision stability indicated (firm vs. tentative)?

Conformance:

20. Do you understand how conformance will be verified?
21. Are conformance points for your unit identified?
22. Do you know when your implementation is complete?

9.3 Questions for Integrators and Fielders

Integrator/Fielder Questions

Integration Planning:

1. Can you identify all units requiring integration?
2. Is integration order determinable from dependencies?
3. Are interface contracts defined for integration verification?

Resource Requirements:

4. Can you determine resources needed for each unit?
5. Are runtime dependencies clearly documented?
6. Is infrastructure requirements documented?

Integration Testing:

7. Are integration test obligations defined?
8. Is integration success criteria clear?
9. Are integration test environments specified?

Deployment Planning:

10. Is deployment topology clearly defined?
11. Are configuration requirements documented?
12. Is deployment order specified?
13. Are rollback procedures defined?

Conformance:

14. Do you understand deployment conformance criteria?
15. Are operational conformance points identified?

9.4 Questions for Testers

Tester Questions

Test Isolation:

1. Can you identify units testable in isolation?
2. Are dependencies for isolated testing documented?
3. Is mock/stub guidance provided?

Test Requirements:

4. For each unit, what data is needed for testing?
5. What special hardware or infrastructure is needed?
6. What other units must be available?

Test Success Criteria:

7. Are test success criteria defined for each unit?
8. Are quality attribute targets testable?
9. Are acceptance criteria defined?

System Testing:

10. Can the system be tested as a whole?
11. Are end-to-end scenarios documented?
12. Are system test environments specified?

9.5 Questions for QA Stakeholders

QA Stakeholder Questions

Baseline and History:

1. Is the architecture documentation baselined?
2. Is change history maintained?
3. Are versions clearly identified?

Decisions and Rationale:

4. Are key decisions identified?
5. Is rationale captured for decisions?
6. Are open decisions documented?
7. Are deferred decisions clearly marked?

Traceability:

8. Is there traceability to requirements?
9. Is there traceability to test artifacts?
10. Is there traceability to implementation artifacts?

Consistency:

11. Are known inconsistencies documented?
12. Are view-to-view correspondences defined?
13. Are model-to-artifact mappings documented?

Conformance Process:

14. Are conformance points identified?
15. Is there a documented conformance verification method?
16. Does the architecture content support conformance checking?
17. Is there a formal conformance process?

Issue Management:

18. How are developer issues captured?
19. How are issues resolved and reflected in the architecture?
20. Is there a feedback loop from implementation to architecture?

9.6 Questions for All Stakeholders

Universal Questions

1. Can you identify open, partially resolved, or unresolved issues in the architecture documentation?
2. Are there areas of the architecture that are unclear or ambiguous?
3. Can you identify where automated tools will be used in development?
4. Does the architecture documentation have content in formats processable by tools?
5. Is the architecture documentation accessible and navigable?
6. Can you find the information you need without excessive searching?
7. Is terminology used consistently throughout the documentation?
8. Are cross-references accurate and helpful?

10 Development Readiness Assessment

10.1 Readiness Checklist

Architecture Documentation Development Readiness

Implementation Unit Identification:

- ☐ All implementation units identified
- ☐ Unit boundaries and responsibilities defined
- ☐ Unit ownership assigned
- ☐ Effort estimates possible

Dependency Documentation:

- ☐ Build dependencies documented
- ☐ Runtime dependencies documented
- ☐ Dependency rules and constraints defined
- ☐ Development sequence determinable

Design Guidance:

- ☐ Architectural constraints documented
- ☐ Required patterns identified
- ☐ Technology constraints specified
- ☐ Cross-cutting concern approaches defined

Interface Specifications:

- ☐ External interfaces specified
- ☐ Internal interfaces defined
- ☐ Interface contracts documented
- ☐ Data formats specified

Quality Requirements:

- ☐ Performance requirements allocated
- ☐ Security requirements allocated
- ☐ Other quality requirements allocated
- ☐ Test criteria defined

Conformance Definition:

- ☐ Conformance points identified
- ☐ Verification methods defined
- ☐ Completion criteria established

Documentation Quality:

- ☐ Documentation baselined
- ☐ Terminology consistent
- ☐ Cross-references accurate
- ☐ Open issues documented

10.2 Readiness Assessment Matrix

Table 14: Development Readiness Assessment

Criterion	**Ready**	**Partial**	**Not Ready**	**Notes**
Unit identification	☐	☐	☐	
Dependencies	☐	☐	☐	
Constraints	☐	☐	☐	
Interfaces	☐	☐	☐	
Quality requirements	☐	☐	☐	
Test criteria	☐	☐	☐	
Conformance points	☐	☐	☐	
Rationale	☐	☐	☐	
Documentation baseline	☐	☐	☐	
Overall Assessment	☐	☐	☐	

11 Appendix A: Implementation Unit Template

Detailed Implementation Unit Specification

1. Identification
- Unit ID: [IU-XXX-NNN]
- Unit Name: [Descriptive name]
- Version: [X.Y.Z]
- Status: [Planned / In Development / Complete]

2. Architecture Mapping
- Module View Element: [Reference]
- C&C View Element: [Reference]
- Deployment View Element: [Reference]

3. Responsibility
- Primary Responsibility: [Main purpose]
- Secondary Responsibilities: [Additional duties]
- Out of Scope: [What this unit does NOT do]

4. Dependencies
- Build Dependencies: [Required for compilation]
- Runtime Dependencies: [Required for execution]
- Test Dependencies: [Required for testing]
- Optional Dependencies: [Enhanced functionality]

5. Interfaces
- Provided Interfaces: [APIs exposed]
- Required Interfaces: [APIs consumed]
- Events Published: [Events emitted]
- Events Subscribed: [Events handled]

6. Quality Requirements
- Performance: [Specific targets]
- Availability: [Uptime requirements]
- Security: [Security requirements]
- Other: [Additional quality requirements]

7. Constraints
- Technology: [Required technologies]
- Standards: [Standards to follow]
- Patterns: [Required patterns]

8. Test Criteria
- Unit Test Coverage: [Target percentage]
- Integration Test Scenarios: [Key scenarios]
- Performance Test Criteria: [Targets]

9. Conformance Points
- Structure: [Conformance checks]
- Behavior: [Conformance checks]
- Quality: [Conformance checks]

12 Appendix B: Glossary

Architecture Conformance
: Degree to which implementation adheres to prescribed architecture

Build Dependency
: Dependency required at compile/build time

Conformance Point
: Specific aspect of implementation that must match architecture

Fitness Function
: Automated test verifying architectural characteristic

Implementation Unit
: Portion of system assignable to developer for implementation

Integration Process of combining components into working assemblies

Runtime Dependency
: Dependency required during system execution

Test Criteria Conditions that determine test success or failure

Traceability Ability to trace relationships between artifacts

13 Appendix C: References

1. Clements, P., et al. (2010). *Documenting Software Architectures: Views and Beyond* (2nd ed.). Addison-Wesley.
2. Bass, L., Clements, P., & Kazman, R. (2021). *Software Architecture in Practice* (4th ed.). Addison-Wesley.
3. Ford, N., Parsons, R., & Kua, P. (2017). *Building Evolutionary Architectures*. O'Reilly Media.
4. Rozanski, N., & Woods, E. (2011). *Software Systems Architecture* (2nd ed.). Addison-Wesley.
5. ISO/IEC/IEEE 42010:2011. *Systems and software engineering—Architecture description*.
6. Cohn, M. (2005). *Agile Estimating and Planning*. Prentice Hall.
7. Humble, J., & Farley, D. (2010). *Continuous Delivery*. Addison-Wesley.
8. Richards, M., & Ford, N. (2020). *Fundamentals of Software Architecture*. O'Reilly Media.