Software Architecture Documentation

Architectural Rationale

A Comprehensive Guide to Documenting Design Decisions,
Tradeoffs, Alternatives, and Technical Debt

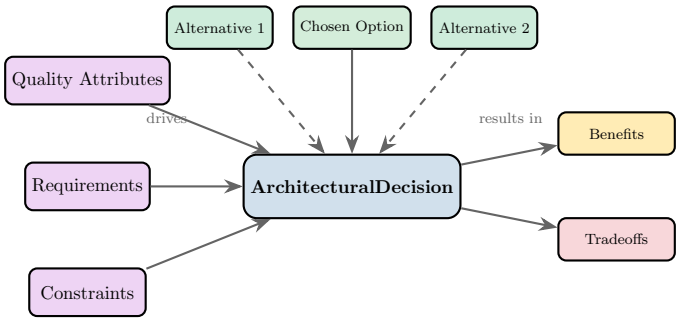
Architecture Documentation Series

Based on ADR Standards, SEI Decision Documentation, and Industry Best Practices

December 4, 2025

Abstract

Architectural rationale captures the reasoning behind design decisions—the "why" that explains the "what" shown in architectural views. Without documented rationale, architectures become inscrutable over time, leading to poor decisions, unnecessary rework, and loss of institutional knowledge. This comprehensive guide establishes principles and practices for documenting architectural rationale, including design drivers, decision records, alternatives analysis, tradeoff evaluation, pattern selection, risk assessment, and technical debt management. The document introduces the Architecture Decision Record (ADR) format, provides frameworks for systematic tradeoff analysis, and establishes governance processes for maintaining rationale documentation throughout the system lifecycle. Whether facing technology choices, structural decisions, or quality attribute tradeoffs, this guide provides the foundation for making decisions transparent, traceable, and defensible.



Contents

1 Introduction to Architectural Rationale

1.1 Definition and Purpose

Architectural rationale is the explicit documentation of the reasoning behind architectural decisions. It captures not just what was decided, but why it was decided, what alternatives were considered, what tradeoffs were made, and what consequences are expected.

Definition

Architectural Rationale is the documentation that explains the reasoning, assumptions, tradeoffs, and constraints that led to the architectural decisions reflected in a system's design. It answers the question "why is the architecture this way?" and preserves the decision-making context for future reference.

Documenting rationale serves several critical purposes. First, it provides **knowledge preservation** by capturing the reasoning that would otherwise exist only in the minds of original architects. Second, it enables **informed evolution** by allowing future architects to understand constraints and tradeoffs before making changes. Third, it ensures **decision quality** by forcing explicit consideration of alternatives and consequences. Fourth, it supports **stakeholder communication** by explaining architectural choices to diverse audiences. Fifth, it creates **accountability** through transparent, traceable decision-making. Sixth, it prevents **decision erosion** by documenting why certain approaches were rejected.

1.2 The Cost of Missing Rationale

Organizations that fail to document rationale pay significant costs over time.

Warning

Consequences of Undocumented Rationale:
Repeated Mistakes: Teams revisit rejected alternatives because they don't know why they were rejected, wasting time rediscovering the same problems.
Inappropriate Changes: Modifications that violate implicit constraints break the system in subtle ways because the constraints were never documented.
Knowledge Loss: When original architects leave, critical understanding leaves with them, creating "legacy" systems that no one fully understands.
Analysis Paralysis: Without documented precedents, teams struggle to make decisions, fearing unknown consequences.
Inconsistent Decisions: Similar problems are solved differently across the system because there's no record of previous approaches.

1.3 Rationale Within Architectural Documentation

Rationale documentation complements the structural documentation in architectural views. While primary presentations and element catalogs describe what the architecture is, rationale explains why it is that way.

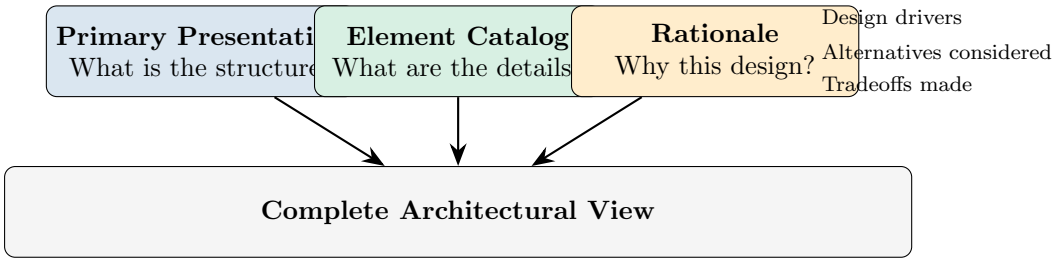


Figure 1: Rationale as Part of Complete Architectural Documentation

1.4 Standards and Frameworks

Architectural rationale documentation draws from several established approaches. The SEI (Software Engineering Institute) views and beyond approach includes rationale as part of every architectural view. Architecture Decision Records (ADRs) provide a lightweight format popularized by Michael Nygard for capturing decisions. ISO/IEC/IEEE 42010 establishes requirements for architecture rationale in architecture descriptions. Design Rationale Systems from academic research provide formal models for capturing design reasoning. TOGAF includes architecture decision documentation as part of the Architecture Development Method.

2 Design Drivers

2.1 Understanding Design Drivers

Design drivers are the forces that shape architectural decisions. They include functional requirements, quality attribute requirements, constraints, and business goals. Understanding and documenting drivers is essential because they provide the "why" behind decisions.

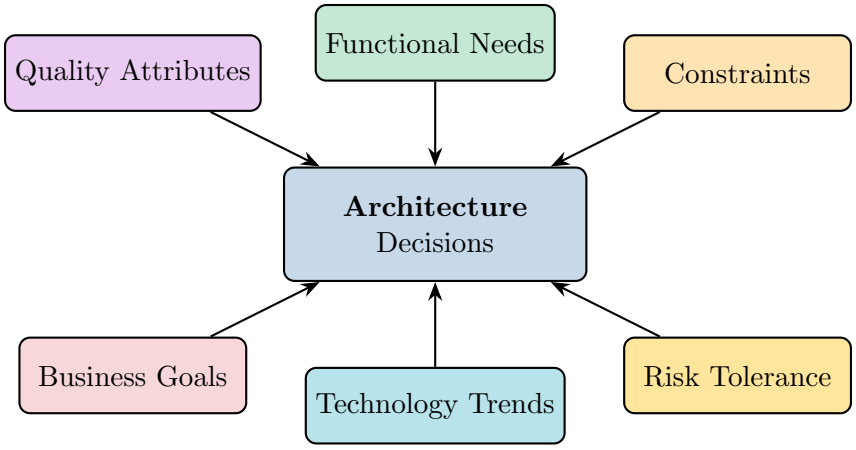


Figure 2: Categories of Design Drivers

2.2 Quality Attribute Drivers

Quality attributes (non-functional requirements) are among the most important drivers of architectural decisions. Structure exists largely to achieve quality attributes.

2.2.1 Quality Attribute Scenarios

Document quality attribute drivers as concrete, testable scenarios using the SEI quality attribute scenario format.

Template

Quality Attribute Scenario Template

Scenario ID:	Unique identifier (e.g., QA-PERF-001)
Quality Attribute:	The quality attribute addressed (e.g., Performance)
Source:	Entity that generates the stimulus
Stimulus:	The event or condition that affects the system
Environment:	Conditions under which the stimulus occurs
Artifact:	Part of system that is stimulated
Response:	How the system should respond
Response Measure:	Quantifiable measure of the response

Example

Quality Attribute Scenario: QA-PERF-001

Quality Attribute:	Performance (Latency)
Source:	External customer
Stimulus:	Submits order during peak shopping period
Environment:	Normal operation, Black Friday traffic (10x normal)
Artifact:	Order processing system
Response:	Order is validated, inventory reserved, confirmation returned
Response Measure:	95th percentile response time $\leq$ 2 seconds

Architectural Impact: This scenario drives decisions toward horizontal scaling, caching, and asynchronous processing for non-critical path operations.

2.2.2 Quality Attribute Priority Matrix

Not all quality attributes are equally important. Document relative priorities to guide tradeoff decisions.

Table 1: Quality Attribute Priority Matrix

Attribute	Priority	Key Scenarios	Architectural Impact
Availability	Critical	99.95% uptime; graceful degradation	Redundancy; health checks; failover
Performance	High	2s response at 10x load	Caching; async processing; CDN

Attribute	Priority	Key Scenarios	Architectural Impact
Security	High	PCI compliance; data protection	Encryption; auth/authz; audit logging
Scalability	High	Handle 100K concurrent users	Stateless services; horizontal scaling
Modifiability	Medium	Deploy changes within 1 day	Microservices; CI/CD; feature flags
Testability	Medium	80% code coverage; automated testing	Dependency injection; interfaces
Usability	Medium	Task completion within 3 clicks	API design; error messages
Cost	Medium	Operate within $50K/month	Cloud optimization; reserved instances

2.3 Constraints

Constraints are non-negotiable requirements that limit architectural options. Unlike quality attributes (which involve tradeoffs), constraints are absolute.

2.3.1 Constraint Categories

Table 2: Constraint Categories and Examples

Category	Examples	Typical Sources
Technical	Must use Java; must deploy to AWS; must integrate with SAP	Enterprise standards; existing investments
Organizational	Maximum team size of 8; must be maintainable by existing staff	HR policies; skill availability
Regulatory	PCI-DSS compliance; GDPR data residency; SOX audit requirements	Government; industry bodies
Contractual	Must use vendor X for payments; SLA commitments to customers	Business agreements; partnerships
Financial	Development budget of $500K; operational budget of $5K/month	Business planning; funding
Schedule	Must launch by Q4; integration complete by March	Market timing; dependencies

Category	Examples	Typical Sources
Physical	Must operate in edge locations with limited connectivity	Deployment environment

2.3.2 Constraint Documentation

Table 3: Constraint Registry

ID	Constraint	Category	Source	Impact
CON-01	Must deploy to AWS only	Technical	Enterprise IT policy	Eliminates multi-cloud options
CON-02	Java or Kotlin for backend services	Technical	Team skills; hiring	Limits technology choices
CON-03	PCI-DSS Level 1 compliance	Regulatory	Payment processing	Isolation; encryption; audit
CON-04	EU data must stay in EU	Regulatory	GDPR	Multi-region deployment
CON-05	Budget: $50K/month operations	Financial	Business case	Cloud cost optimization
CON-06	Launch by November 1	Schedule	Holiday season	Phased delivery; MVP scope

2.4 Key Requirements

Certain functional requirements significantly influence architecture. These architecturally significant requirements (ASRs) should be explicitly identified.

Key Point

Identifying Architecturally Significant Requirements:
A requirement is architecturally significant if it:

- Has broad effect on the system structure
- Affects a quality attribute to a significant degree
- Involves tradeoffs between quality attributes
- Requires unusual or innovative solutions
- Has high business or technical risk
- Constrains future evolution

Table 4: Architecturally Significant Requirements

ID	Requirement	Category	Architectural Impact
ASR-01	Process 10,000 orders per minute at peak	Performance	Horizontal scaling; async processing
ASR-02	Support real-time inventory across 500 stores	Integration	Event-driven; eventual consistency
ASR-03	Enable A/B testing of all features	Modifiability	Feature flags; canary deployment
ASR-04	Maintain audit trail for 7 years	Compliance	Event sourcing; archival storage
ASR-05	Support offline operation for mobile POS	Availability	Local storage; sync protocol

3 Architecture Decision Records (ADRs)

3.1 Introduction to ADRs

Architecture Decision Records (ADRs) are a lightweight format for capturing architectural decisions. Popularized by Michael Nygard, ADRs provide a standardized, version-controlled approach to decision documentation.

Definition

An **Architecture Decision Record (ADR)** is a document that captures a single architectural decision along with its context, considered alternatives, rationale, and consequences. ADRs are typically stored alongside code in version control, making decision history part of the project repository.

3.2 ADR Format

The standard ADR format includes several key sections.

Template

Architecture Decision Record Template

Title: ADR-NNN: Short Descriptive Title

Status: [Proposed — Accepted — Deprecated — Superseded by ADR-XXX]

Date: YYYY-MM-DD

Context:
Describe the forces at play, including technological, political, social, and project constraints. This section should be value-neutral—simply describe the facts.

Decision:
State the decision in full sentences, with an active voice. "We will..."

Consequences:
Describe the resulting context after applying the decision. All consequences should be listed, both positive and negative. A decision that appears to have only positive consequences is likely missing something.

Alternatives Considered:
List alternatives that were evaluated and why they were not chosen.

3.3 ADR Lifecycle

ADRs have a lifecycle that reflects the evolution of architectural decisions.

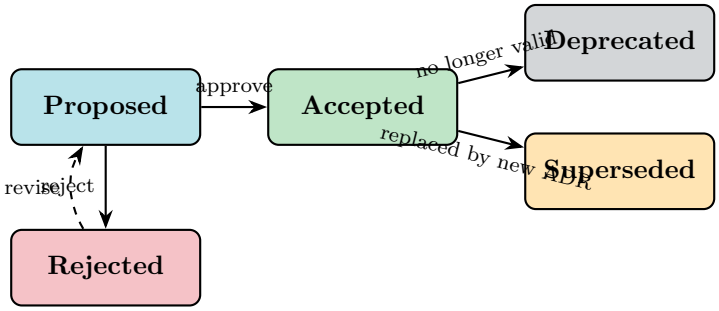


Figure 3: ADR Lifecycle States

3.3.1 ADR Status Definitions

Table 5: ADR Status Definitions

Status	Description
[PROPOSED]	Decision is proposed but not yet approved. Open for discussion.
[ACCEPTED]	Decision has been approved and is in effect. Implementation should follow.

Status	Description
[REJECTED]	Decision was considered but rejected. Rationale preserved for future reference.
[DEPRECATED]	Decision is no longer recommended but may still be in use. Plan for migration.
[SUPERSEDED]	Decision has been replaced by a newer decision. Link to replacement ADR.

3.4 Complete ADR Examples

ADR-001: Use PostgreSQL for Primary Data Storage

Status: [ACCEPTED]
Date: 2024-01-15
Deciders: Architecture Team, Database Team Lead

Context
We need to select a primary database for the e-commerce platform. The database will store orders, customers, products, and inventory data. Key requirements include:

- ACID transactions for order processing
- Complex queries for reporting and analytics
- Support for JSON documents for flexible product attributes
- Mature ecosystem with strong operational tooling
- Team familiarity and hiring availability

Current transaction volume is approximately 1,000 orders/hour with projected growth to 10,000 orders/hour within 2 years. Data volume is estimated at 500GB initially, growing to 5TB over 3 years.

Decision
We will use PostgreSQL 15 as our primary relational database.

Consequences
Positive:

- Strong ACID compliance ensures data integrity for financial transactions
- Excellent JSON/JSONB support provides flexibility without sacrificing relational model
- Mature replication and backup solutions (streaming replication, pgBackRest)
- Strong community and commercial support options
- Team has existing PostgreSQL expertise
- Available as managed service on AWS (RDS, Aurora PostgreSQL)

Negative:

- Horizontal scaling is more complex than NoSQL alternatives
- May need read replicas for high-read workloads
- Requires careful schema design and indexing for performance
- Not optimal for time-series data (will need separate solution)

Risks:

- If transaction volume exceeds projections significantly, may need sharding strategy
- Schema migrations require careful planning for zero-downtime deployments

Alternatives Considered
MySQL/MariaDB: Similar capabilities but team has less experience. PostgreSQL's JSONB support is superior.
MongoDB: Better horizontal scaling but weaker transaction support. ACID transactions added recently but less mature. Would complicate joins for reporting.
Amazon DynamoDB: Excellent scaling but requires significant application changes for query patterns. Complex for ad-hoc reporting. Higher cost at our data volumes.
CockroachDB: Better horizontal scaling than PostgreSQL but less mature, smaller talent pool, and higher operational complexity.

ADR-002: Adopt Event-Driven Architecture for Service Communication

Status: [ACCEPTED]
Date: 2024-01-22
Deciders: Architecture Team, Platform Team Lead

Context
Our microservices architecture requires inter-service communication. Currently considering two primary patterns:

- Synchronous: Direct HTTP/gRPC calls between services
- Asynchronous: Event-driven communication via message broker

Key considerations include:

- Resilience to service failures
- Ability to handle traffic spikes
- Maintaining data consistency across services
- Supporting audit requirements (7-year retention)
- Team familiarity with patterns

The Order Service must coordinate with Inventory, Payment, and Notification services. Order completion rate must remain above 99.9% even during partial outages.

Decision
We will adopt an event-driven architecture using Apache Kafka as the primary message broker for inter-service communication. Synchronous calls will be limited to:

- User-facing APIs requiring immediate response
- Queries that cannot tolerate eventual consistency

Consequences
Positive:

- Services are decoupled; failures don't cascade
- Natural buffering handles traffic spikes
- Event log provides audit trail and enables replay
- Supports multiple consumers without sender changes
- Easier to add new services (subscribe to existing events)

Negative:

- Eventual consistency requires careful design
- Debugging distributed workflows is more complex
- Additional infrastructure (Kafka cluster) to operate
- Team needs training on event-driven patterns
- Message schema evolution requires governance

Risks:

- Consumer lag could cause stale data if not monitored
- Poison messages could block processing without dead-letter handling

Alternatives Considered
Pure Synchronous (REST/gRPC): Simpler mental model but creates tight coupling. Single service failure can cascade. Difficult to handle spikes without circuit breakers everywhere.
RabbitMQ: Simpler than Kafka but lacks event replay capability. Our audit requirements benefit from Kafka's log retention.
AWS SQS/SNS: Easier to operate but less control over partitioning. Kafka's consumer groups provide better scaling model for our use case.

3.5 ADR Organization and Naming

Best Practice

ADR Organization Guidelines:
Location: Store ADRs in version control alongside code, typically in `docs/adr/` or `architecture/decisions/`.
Naming: Use sequential numbering with descriptive titles:

- `0001-use-postgresql-for-primary-storage.md`
- `0002-adopt-event-driven-architecture.md`
- `0003-select-kubernetes-for-orchestration.md`

Format: Markdown is most common for compatibility with GitHub/GitLab rendering.
Index: Maintain an index file (README.md) linking to all ADRs with status.
Immutability: Once accepted, ADRs should not be modified except to update status. Create new ADRs to supersede old ones.

3.6 Decision Log Summary

Maintain a summary table for quick reference to all decisions.

Table 6: Architecture Decision Log

ID	Decision	Status	Date	Key Drivers
ADR-001	PostgreSQL for primary DB	[ACCEPTED]	2024-01-15	ACID; team skills
ADR-002	Event-driven architecture	[ACCEPTED]	2024-01-22	Resilience; audit
ADR-003	Kubernetes for orchestration	[ACCEPTED]	2024-02-01	Scaling; portability
ADR-004	JWT for authentication	[ACCEPTED]	2024-02-08	Stateless; standards
ADR-005	GraphQL for mobile API	[PROPOSED]	2024-02-15	Bandwidth; flexibility
ADR-006	Session-based auth	[SUPERSEDED]	2024-01-10	Superseded by ADR-004

4 Alternatives Analysis

4.1 Systematic Alternatives Evaluation

Thorough evaluation of alternatives is essential for sound decision-making. A systematic approach ensures consistency and completeness.

4.1.1 Alternatives Identification

Before evaluating alternatives, ensure you have identified a sufficient range of options.

Best Practice

Finding Alternatives:

- Research industry practices and case studies
- Consult technology radar and analyst reports
- Review competitor and peer architectures
- Brainstorm with diverse team members
- Consider "do nothing" as a baseline
- Include build vs. buy vs. open-source options
- Explore hybrid approaches

4.2 Evaluation Criteria

Define explicit criteria before evaluating alternatives to ensure objective comparison.

Table 7: Common Evaluation Criteria

Category	Criteria	Typical Measures
Fitness	Meets functional requirements	Feature coverage percentage
Fitness	Meets quality attributes	Scenario satisfaction
Cost	Initial development cost	Person-months; $
Cost	Ongoing operational cost	Monthly $; person-hours
Cost	Licensing cost	Annual $; per-user $
Risk	Technical risk	Probability × impact
Risk	Vendor/community viability	Market share; funding
Capability	Team skills match	Training required
Capability	Tool and ecosystem maturity	Age; adoption; documentation
Strategic	Alignment with roadmap	Fit with future plans
Strategic	Vendor relationship	Existing contracts; support

4.3 Decision Matrix Method

A decision matrix provides structured comparison of alternatives against weighted criteria.

Example

Database Selection Decision Matrix

Criteria Weights:	ACID Compliance	25%
	Query Flexibility	20%
	Scalability	20%
	Operational Simplicity	15%
	Team Experience	10%
	Cost	10%

Scoring (1-5 scale):

Criterion	Weight	Postgres	MySQL	MongoDB	DynamoDB
ACID Compliance	25%	5	4	3	4
Query Flexibility	20%	5	4	4	2
Scalability	20%	3	3	5	5
Operational Simplicity	15%	4	4	3	5
Team Experience	10%	5	3	2	2
Cost	10%	4	4	3	3
Weighted Score		**4.25**	**3.65**	**3.45**	**3.65**

Result: PostgreSQL scores highest, primarily due to strong ACID compliance, query flexibility, and team experience.

4.4 Pros and Cons Analysis

For each alternative, document advantages and disadvantages systematically.

Table 8: Alternative: MongoDB for Primary Storage

Category	Pros	Cons
Functionality	Flexible schema for varying product types; native JSON	Weaker multi-document transactions; joins require application logic
Performance	Excellent write throughput; built-in sharding	Index management complexity; query optimization less mature
Operations	Easy horizontal scaling; managed service available	Memory-hungry; backup complexity for large datasets

Category	Pros	Cons
Team	Modern developer experience	Limited team experience; training needed
Ecosystem	Good driver support; active community	Fewer BI tool integrations
Cost	Open source core; pay for Enterprise features	Higher storage costs; Atlas pricing at scale

4.5 Sensitivity Analysis

Test how robust the decision is to changes in criteria weights or scores.

Key Point

Perform sensitivity analysis when:

- The top alternatives are close in score
- Criteria weights are uncertain or contested
- Future conditions may change criteria importance
- Stakeholders have different priorities

Ask: "What would have to change for a different alternative to win?" If the answer is "very little," the decision may need more deliberation.

5 Tradeoff Analysis

5.1 Understanding Tradeoffs

Architecture is fundamentally about tradeoffs. Improving one quality attribute often comes at the expense of another. Documenting tradeoffs explicitly helps stakeholders understand the implications of decisions.

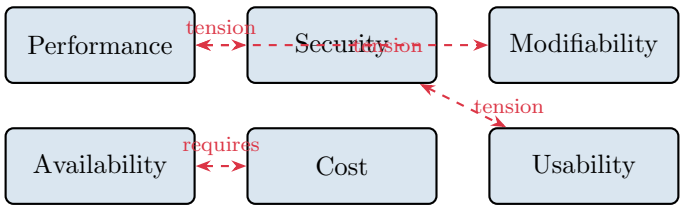


Figure 4: Common Quality Attribute Tradeoffs

5.2 Common Tradeoff Patterns

Table 9: Common Architectural Tradeoffs

Tradeoff	Description	Mitigation Strategies
Performance vs. Security	Encryption and validation add latency	Hardware acceleration; caching; async validation
Availability vs. Consistency	CAP theorem; strong consistency limits availability	Eventual consistency where acceptable; careful partition design
Modifiability vs. Performance	Abstraction layers add overhead	Profile-driven optimization; bypass for hot paths
Simplicity vs. Flexibility	Generic solutions are more complex	Build for current needs; design for extension
Time-to-Market vs. Quality	Fast delivery may incur technical debt	MVP scope; planned remediation; feature flags
Cost vs. Scalability	Over-provisioning wastes money; under-provisioning limits growth	Auto-scaling; reserved capacity for baseline
Usability vs. Security	Security measures can frustrate users	Risk-based authentication; SSO; biometrics

5.3 ATAM-Style Tradeoff Documentation

The Architecture Tradeoff Analysis Method (ATAM) provides a structured approach to analyzing tradeoffs.

Template

Tradeoff Point Documentation

Tradeoff ID: Unique identifier
Decision: The architectural decision creating the tradeoff
Quality Attributes Affected: List of QAs in tension
Description: Explanation of the tradeoff
Sensitivity: How sensitive is the outcome to this tradeoff?
Stakeholder Impact: Which stakeholders are affected?
Mitigation: How the negative impacts are addressed

Example

Tradeoff Point: TP-001

Decision: ADR-002 (Event-driven architecture)
Quality Attributes: Resilience (+) vs. Consistency (-)
Description: By adopting event-driven architecture, we gain resilience through service decoupling but accept eventual consistency. The Order Service publishes events that Inventory and Notification services consume asynchronously. This means a customer may briefly see stale inventory data after a purchase.
Sensitivity: Medium. Consistency windows are typically under 1 second, but during high load or failures, could extend to minutes.
Stakeholder Impact:

- Customers: May see "in stock" for items just purchased by others
- Operations: Must monitor consumer lag and handle compensation
- Finance: Rare edge cases may require manual reconciliation

Mitigation:

- Optimistic UI updates with confirmation
- Real-time inventory checks at checkout (synchronous for critical path)
- Consumer lag alerting with automatic scaling
- Compensation events for conflict resolution

5.4 Tradeoff Visualization

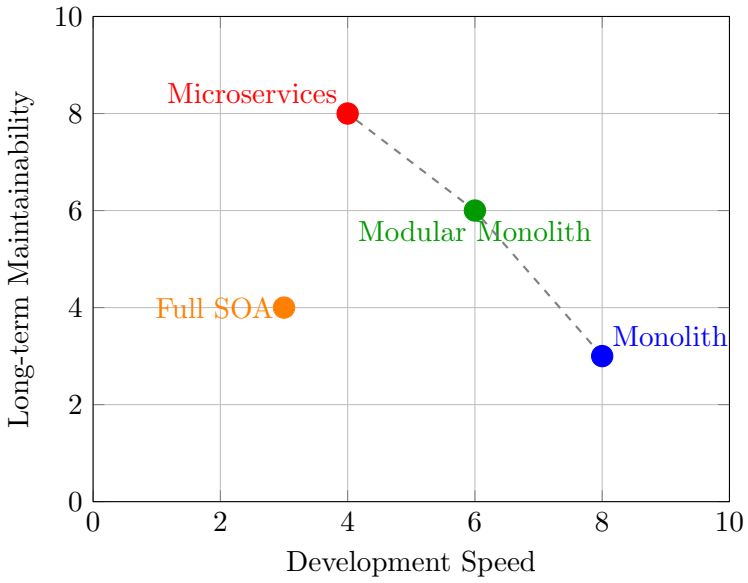


Figure 5: Architecture Style Tradeoff Visualization

6 Architectural Patterns and Styles

6.1 Pattern Selection Rationale

Architectural patterns encode proven solutions to recurring problems. Documenting why specific patterns were chosen connects the architecture to its design drivers.

6.2 Pattern Documentation

For each major pattern employed, document its selection rationale.

Table 10: Architectural Pattern Selection

Pattern	**Problem Addressed**	**Quality Attributes**	**ADR Reference**
Microservices	Independent deployment; team autonomy	Modifiability; Scalability	ADR-007
Event Sourcing	Audit trail; temporal queries	Auditability; Recoverability	ADR-009
CQRS	Separate read/write optimization	Performance; Scalability	ADR-010
API Gateway	Unified entry point; cross-cutting concerns	Security; Manageability	ADR-011
Circuit Breaker	Cascading failure prevention	Availability; Resilience	ADR-012
Saga	Distributed transaction coordination	Consistency; Resilience	ADR-013

6.3 Pattern Interaction Map

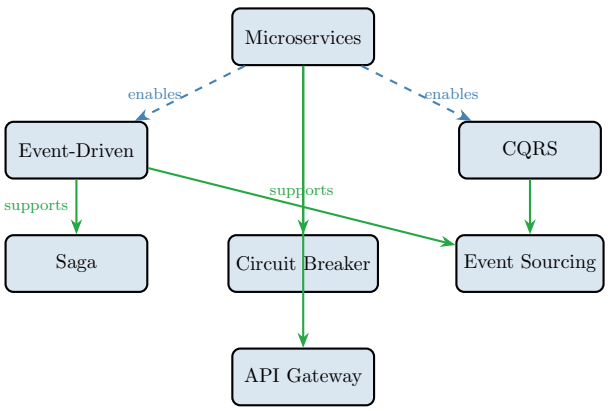


Figure 6: Pattern Relationships in the Architecture

6.4 Pattern Application Guidelines

Document how patterns should be applied consistently across the system.

Pattern: Circuit Breaker Application Guidelines

When to Apply:
- All synchronous calls to external services
- All synchronous inter-service calls
- Database connections (with appropriate thresholds)

Configuration Standards:
- Failure threshold: 5 failures in 10 seconds
- Open duration: 30 seconds
- Half-open test requests: 3
- Timeout: Match SLA of downstream service

Fallback Strategies:
- Return cached data where acceptable
- Return degraded response with explanation
- Queue request for later retry
- Fail fast with clear error message

Monitoring:
- Alert when circuit opens
- Track circuit state transitions
- Dashboard for all circuit breaker states

7 Risks and Technical Debt

7.1 Architectural Risk Management

Architectural decisions involve uncertainty. Documenting risks ensures they are managed proactively rather than discovered in production.

7.1.1 Risk Identification

Table 11: Architectural Risk Register

ID	Risk Description	Probability	Impact	Mitigation Strategy
R-01	Kafka cluster failure causes event loss	Low	Critical	Multi-AZ deployment; replication factor 3; disaster recovery runbook
R-02	Database scaling limits reached before migration	Medium	High	Monitor growth; prepared sharding strategy; Aurora Serverless evaluation

ID	Risk Description	Probability	Impact	Mitigation Strategy
R-03	Team unable to maintain microservices complexity	Medium	High	Platform team support; standardization; training program
R-04	Third-party payment provider outage	Medium	Critical	Secondary provider integration; graceful degradation
R-05	Event schema evolution breaks consumers	High	Medium	Schema registry; compatibility rules; consumer contract testing

7.1.2 Risk Heat Map

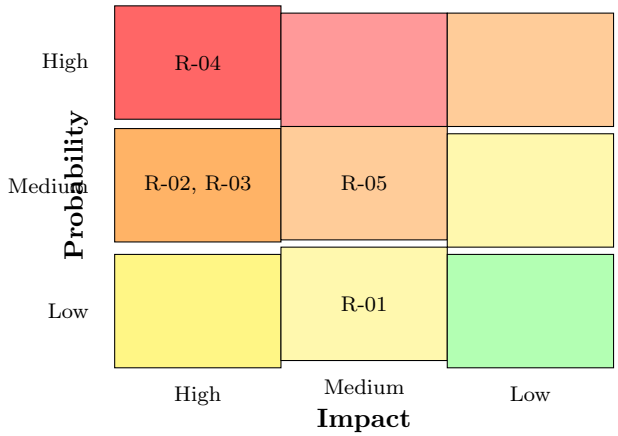


Figure 7: Risk Heat Map

7.2 Technical Debt Documentation

Technical debt is intentional deviation from ideal architecture, accepted for short-term benefit with planned future remediation.

Definition

Technical Debt is the implied cost of additional rework caused by choosing an easier solution now instead of using a better approach that would take longer. Like financial debt, it incurs "interest" in the form of increased maintenance cost and reduced agility.

7.2.1 Technical Debt Categories

Table 12: Technical Debt Categories

Category	Description	Examples
Deliberate/Prudent	Conscious choice with clear payoff	"Ship now, refactor before scale"
Deliberate/Reckless	Conscious choice ignoring consequences	"No time for design"
Inadvertent/Prudent	Learned better approach after building	"Now we know how to do it properly"
Inadvertent/Reckless	Due to lack of skill or knowledge	"What's layering?"

7.2.2 Technical Debt Register

Table 13: Technical Debt Register

ID	Debt Description	Interest Cost	Payoff Effort	Remediation Plan
TD-01	Monolithic user service (auth + profile + preferences)	Medium: harder to scale auth independently	3 sprints	Split in Q3; extract auth service first
TD-02	Hardcoded feature flags in code	High: deployments needed for flag changes	1 sprint	Implement LaunchDarkly in Q2
TD-03	Synchronous inventory checks at checkout	Low: works at current scale	2 sprints	Convert to async reservation when >5K orders/hour
TD-04	No automated performance tests	Medium: manual testing delays releases	2 sprints	Implement Gatling suite Q2
TD-05	Direct database access from API layer	High: tight coupling; hard to optimize	4 sprints	Introduce repository pattern; scheduled for H2

7.3 Debt Repayment Strategy

Best Practice

Managing Technical Debt:
Track It: Maintain a technical debt register as part of the architecture documentation.
Quantify It: Estimate "interest" (ongoing cost) and "principal" (remediation cost) for prioritization.
Budget for It: Allocate capacity (e.g., 20% of sprints) for debt repayment.
Prevent It: Use architecture reviews and coding standards to prevent inadvertent debt.
Communicate It: Ensure stakeholders understand the cost of carrying debt.
Prioritize It: Pay high-interest debt first; defer low-interest debt that may never need payment.

8 Traceability

8.1 Importance of Traceability

Traceability connects decisions to their drivers and consequences, enabling impact analysis when requirements change, validation that decisions satisfy requirements, compliance demonstration for audits, and knowledge navigation across documentation.

8.2 Traceability Matrix

Table 14: Decision Traceability Matrix

Decision	Requirements	Quality Scenarios	Elements	Validation
ADR-001	ASR-01, ASR-04	QA-PERF-001, QA-REL-002	ORDER-DB, INV-DB	Load test; backup test
ADR-002	ASR-02, ASR-04	QA-AVAIL-001, QA-AUDIT-001	EVENT-BUS, all services	Chaos testing; replay test
ADR-003	ASR-01, ASR-03	QA-SCALE-001	K8s cluster	Scale test; failover test
ADR-004	CON-03	QA-SEC-001	AUTH-SVC	Penetration test; audit

8.3 Bidirectional Tracing

8.3.1 Forward Tracing: Requirements to Decisions

Table 15: Requirements to Decisions Tracing

Requirement	Description	Satisfied By Decisions
ASR-01	Process 10,000 orders/minute at peak	ADR-001, ADR-002, ADR-003, ADR-010
ASR-02	Real-time inventory across 500 stores	ADR-002, ADR-009
ASR-03	Enable A/B testing of all features	ADR-003, ADR-014
ASR-04	Maintain audit trail for 7 years	ADR-001, ADR-002, ADR-009
CON-03	PCI-DSS Level 1 compliance	ADR-004, ADR-015, ADR-016

8.3.2 Backward Tracing: Decisions to Requirements

Table 16: Decisions to Requirements Tracing

Decision	Decision Summary	Driven By Requirements
ADR-001	PostgreSQL for primary DB	ASR-01 (ACID for orders), ASR-04 (audit), CON-02 (team skills)
ADR-002	Event-driven architecture	ASR-02 (real-time), ASR-04 (audit), QA-AVAIL-001 (resilience)
ADR-009	Event sourcing for orders	ASR-04 (audit), ASR-02 (temporal queries), QA-AUDIT-001

8.4 Validation Coverage

Track how decisions are validated to ensure they achieve intended outcomes.

Table 17: Decision Validation Coverage

Decision	Validation Method	Validation Artifact	Frequency	Owner
ADR-001	Load testing	Gatling scripts; reports	Pre-release	QA Team
ADR-002	Chaos engineering	Gremlin scenarios	Monthly	SRE Team
ADR-003	Scale testing	K6 scripts; metrics	Quarterly	Platform Team

Decision	Validation Method	Validation Artifact	Frequency	Owner
ADR-004	Security audit	Pen test report	Annually	Security Team
ADR-009	Event replay testing	Replay scripts; comparison	Pre-release	Dev Team

9 Decision Governance

9.1 Decision-Making Process

Establish a clear process for making and documenting architectural decisions.

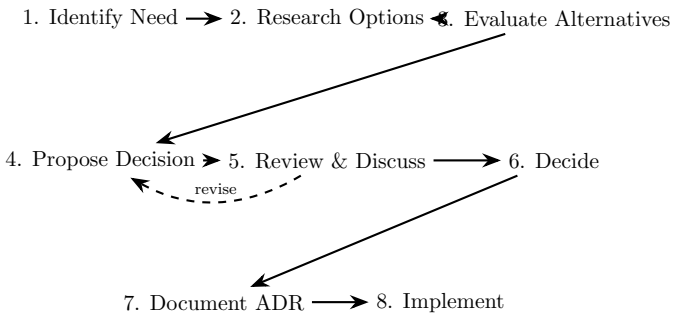


Figure 8: Decision-Making Process

9.2 Decision Authority Matrix

Define who has authority to make different types of decisions.

Table 18: Decision Authority Matrix (RACI)

Decision Type	Responsible	Accountable	Consulted
Strategic (patterns, platforms)	Architecture Team	Chief Architect	Tech Leads; CTO
Tactical (service design)	Tech Lead	Chief Architect	Architecture Team
Implementation (libraries, tools)	Developer	Tech Lead	Team
Cross-cutting (security, compliance)	Specialist	Chief Architect	Legal; Security
Emergency (production fixes)	On-call Engineer	Tech Lead	Architecture Team

9.3 Review Cadence

Table 19: Decision Review Schedule

Review Type	Frequency	Scope	Participants
ADR Review	Per ADR (async + meeting if needed)	Single decision	Author + reviewers
Architecture Sync	Weekly	Current decisions; blockers	Architecture Team
Architecture Review Board	Monthly	Strategic decisions	Architects + leads
Technical Debt Review	Quarterly	Debt register; priorities	Tech leads + PMs
Architecture Retrospective	Semi-annually	All decisions; lessons learned	Full team

10 Open Issues and Future Decisions

10.1 Pending Decisions

Document decisions that are needed but not yet made.

Table 20: Pending Decision Queue

ID	Decision Needed	Urgency	Target Date	Blocker/Dependencies
PD-01	GraphQL vs REST for mobile API	High	2024-03-01	Mobile team input needed
PD-02	Multi-region deployment strategy	Medium	2024-04-15	Cost analysis pending
PD-03	Search engine selection	Medium	2024-03-15	Requirements clarification
PD-04	Observability platform selection	Low	2024-05-01	Budget approval

10.2 Questions and Unknowns

Table 21: Open Questions

ID	Question	Impact If Unresolved	Owner / Due Date
Q-01	What is the expected international expansion timeline?	Multi-region decision depends on this	Product / Feb 15
Q-02	Will we need to support on-premise deployment?	Affects containerization strategy	Sales / March 1
Q-03	What are the actual peak traffic projections for Year 2?	Influences scaling architecture	Analytics / Feb 28
Q-04	Are there planned acquisitions that would require integration?	May affect integration architecture	Exec / Ongoing

11 Appendix A: ADR Template

Architecture Decision Record Template

ADR-NNN: [Short Title]

Status: [Proposed — Accepted — Deprecated — Superseded by ADR-XXX]
Date: YYYY-MM-DD
Deciders: [List of people involved in decision]

Context
[Describe the issue motivating this decision. What is the problem? What are the constraints? What forces are at play?]

Decision
[State the decision. Use active voice: "We will..."]

Consequences
[What are the results of this decision? Include both positive and negative consequences.]
Positive:
Benefit 1
Benefit 2
Negative:
Drawback 1
Drawback 2
Risks:
Risk 1

Alternatives Considered
[What other options were evaluated?]
Alternative 1: [Description and why rejected]
Alternative 2: [Description and why rejected]

Related Decisions
[Links to related ADRs]

References
[Links to relevant documentation, research, or resources]

12 Appendix B: Decision Checklist

12.1 Before Making a Decision

- ☐ Problem is clearly understood and documented
- ☐ Relevant stakeholders are identified
- ☐ Design drivers (QAs, constraints, requirements) are documented
- ☐ Multiple alternatives have been identified

- ☐ Evaluation criteria are defined and weighted
- ☐ Alternatives have been systematically evaluated
- ☐ Tradeoffs are understood and documented
- ☐ Risks are identified

12.2 When Documenting a Decision

- ☐ Context explains the situation objectively
- ☐ Decision is stated clearly and actionably
- ☐ Both positive and negative consequences are listed
- ☐ Rejected alternatives and reasons are documented
- ☐ Traceability to requirements is established
- ☐ Validation approach is identified
- ☐ ADR is reviewed by appropriate stakeholders
- ☐ ADR is stored in version control

12.3 After Making a Decision

- ☐ Decision is communicated to affected teams
- ☐ Implementation guidance is provided if needed
- ☐ Validation tests are implemented
- ☐ Metrics/monitoring for consequences are established
- ☐ Technical debt register is updated if applicable
- ☐ Decision log summary is updated

13 Appendix C: Glossary

ADR Architecture Decision Record; a document capturing a single architectural decision

ASR Architecturally Significant Requirement; a requirement with significant impact on architecture

ATAM Architecture Tradeoff Analysis Method; a structured approach to evaluating architecture quality

Design Driver A force (requirement, constraint, goal) that shapes architectural decisions

Quality Attribute A measurable property of a system such as performance, security, or availability

Quality Attribute Scenario
A concrete, testable specification of a quality attribute requirement

Rationale The reasoning behind a decision, including alternatives considered and tradeoffs made

Technical Debt
Implied cost of future rework caused by choosing easier solutions now

Traceability The ability to relate decisions to requirements and other artifacts

Tradeoff A situation where improving one quality attribute negatively impacts another

14 Appendix D: References

1. Nygard, M. (2011). "Documenting Architecture Decisions." Cognitect Blog. Retrieved from https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions

2. Clements, P., et al. (2010). *Documenting Software Architectures: Views and Beyond* (2nd ed.). Addison-Wesley.

3. Bass, L., Clements, P., & Kazman, R. (2021). *Software Architecture in Practice* (4th ed.). Addison-Wesley.

4. Keeling, M. (2017). *Design It! From Programmer to Software Architect.* Pragmatic Bookshelf.

5. Kazman, R., Klein, M., & Clements, P. (2000). "ATAM: Method for Architecture Evaluation." SEI Technical Report CMU/SEI-2000-TR-004.

6. Tyree, J., & Akerman, A. (2005). "Architecture Decisions: Demystifying Architecture." IEEE Software, 22(2), 19-27.

7. Cunningham, W. (1992). "The WyCash Portfolio Management System." OOPSLA '92 Experience Report. (Origin of "technical debt" metaphor)

8. IEEE. (2011). *ISO/IEC/IEEE 42010:2011 Systems and Software Engineering—Architecture Description.*