Software Architecture Documentation

Primary Presentation

A Comprehensive Guide to Creating Effective
Architectural Diagrams and Visual Documentation

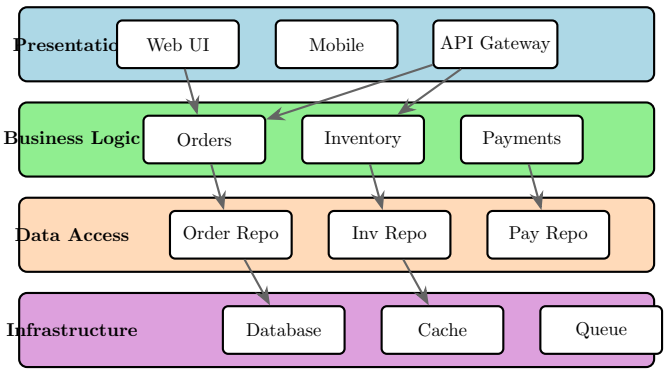
Architecture Documentation Series

Based on SEI Views and Beyond, C4 Model, UML, and Industry Best Practices

December 4, 2025

Abstract

The Primary Presentation is the visual centerpiece of an architectural view—the diagram or set of diagrams that communicates the essential structure of a system at a glance. Effective primary presentations enable stakeholders to quickly grasp architectural decisions, understand system organization, and reason about quality attributes. This comprehensive guide establishes principles and best practices for creating primary presentations that are clear, accurate, informative, and maintainable. The document covers view type selection, diagram design principles, notation standards (UML, C4, ArchiMate, and custom notations), element representation, narrative techniques, behavioral integration, and governance processes. Whether documenting module structures, component-and-connector relationships, deployment topologies, or data flows, this guide provides the foundation for professional architectural visualization.



Contents

1 Introduction to the Primary Presentation

1.1 Definition and Purpose

The Primary Presentation is the principal graphical representation of an architectural view. It is the diagram (or coordinated set of diagrams) that shows, at a glance, the elements and relationships that constitute the architecture from a particular perspective. While supporting documentation provides detail, the primary presentation provides orientation and overview.

Definition

A **Primary Presentation** is the graphical depiction of an architectural view's elements and their relationships, designed to communicate the essential structure of the system to stakeholders in a form that can be quickly comprehended and reasoned about.

The primary presentation serves several critical functions. First, it provides **orientation** by giving stakeholders a mental map of the system's structure. Second, it enables **communication** as a shared visual language for discussing architecture among diverse stakeholders. Third, it supports **analysis** by making structural patterns, dependencies, and potential issues visible. Fourth, it acts as **navigation** by providing an index to more detailed documentation. Fifth, it serves **decision support** by making architectural tradeoffs and alternatives visible.

1.2 The Role Within Architectural Documentation

Every architectural view consists of three complementary parts: the primary presentation (diagrams), the element catalog (detailed specifications), and the context documentation (rationale and background). The primary presentation is not the complete documentation—it is the visual summary that makes the detailed documentation accessible.

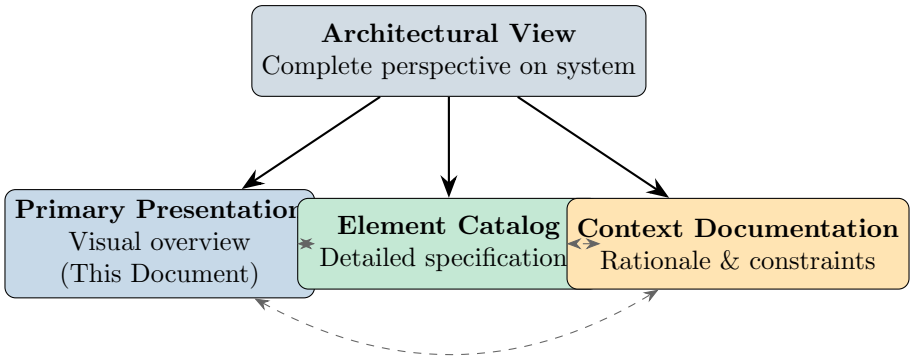


Figure 1: Primary Presentation within View Documentation Structure

1.3 Characteristics of Effective Primary Presentations

Effective primary presentations share common characteristics. They are **comprehensible**, meaning a knowledgeable stakeholder can understand the main structure within minutes, not hours. They are **accurate**, meaning the diagram faithfully represents the actual or intended architecture. They are **complete** (for their purpose), meaning all elements relevant to the view's concerns are shown. They are **consistent**, meaning notation is used uniformly throughout the diagram. They

are **minimal**, meaning unnecessary detail is omitted to maintain clarity. They are **navigable**, meaning viewers can find their way to detailed information. They are **maintainable**, meaning the diagram can be updated as the architecture evolves.

Key Point

The primary presentation should answer the question: "What is the structure of this system from this perspective?" It should not attempt to answer every question about the architecture—that is the role of the complete documentation set.

1.4 Standards and Frameworks

Primary presentation practices draw from several standards and methodologies. The SEI Views and Beyond approach explicitly defines the primary presentation as the first part of every architectural view. UML (Unified Modeling Language) provides standard notation for many diagram types. The C4 Model offers a hierarchical approach to system visualization. ArchiMate provides enterprise architecture notation. SysML extends UML for systems engineering. IEEE 42010 establishes the framework for architectural viewpoints that presentations support.

2 View Types and Their Primary Presentations

2.1 Understanding View Types

Different architectural concerns require different types of views, each with appropriate primary presentation styles. The choice of view type determines what elements appear and how they should be visualized.

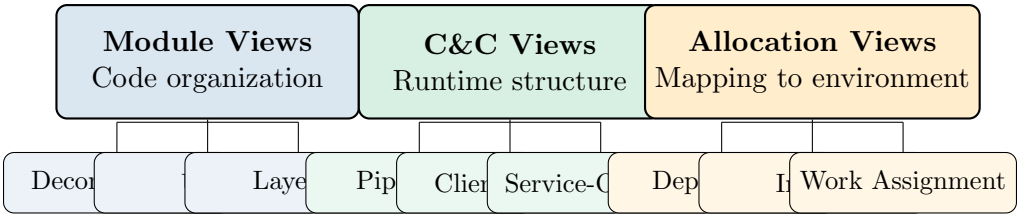


Figure 2: Architectural View Type Categories

2.2 Module Views

Module views show the code-time structure of a system—how source code is organized into modules, packages, classes, and layers. Primary presentations for module views typically use box-and-line notation showing containment and dependency relationships.

2.2.1 Module Decomposition View

The decomposition view shows how the system is divided into modules and sub-modules. The primary presentation is typically a hierarchical diagram.

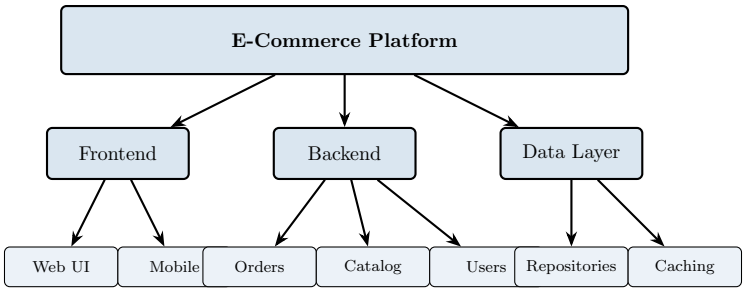


Figure 3: Module Decomposition View Example

2.2.2 Layered View

The layered view organizes modules into layers with constrained dependencies (typically, a layer may only use the layer immediately below it). The primary presentation emphasizes the layer structure and allowed relationships.

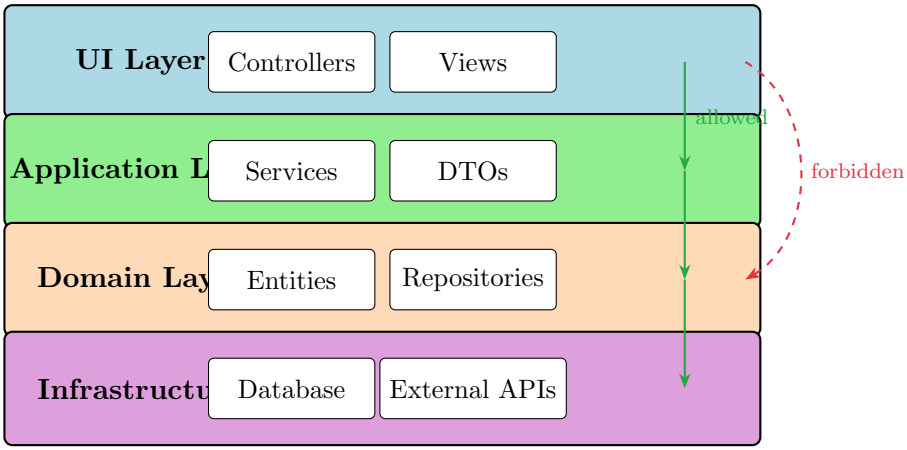


Figure 4: Layered View with Dependency Rules

2.2.3 Uses View

The uses view shows which modules use (depend on) which other modules. The primary presentation is typically a directed graph.

2.3 Component-and-Connector (C&C) Views

C&C views show the runtime structure of a system—processes, threads, services, and the connectors through which they interact. Primary presentations for C&C views emphasize components, their ports, and the connectors between them.

2.3.1 Service-Oriented View

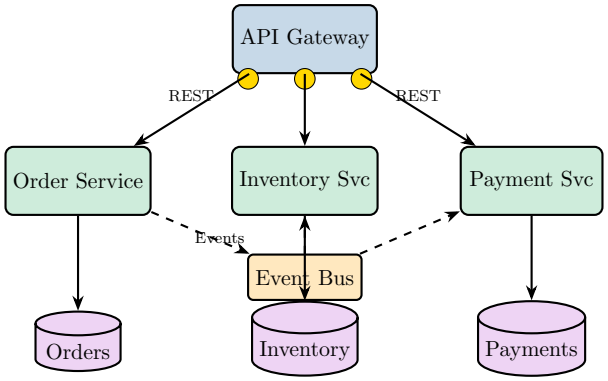


Figure 5: Service-Oriented Architecture View

2.3.2 Pipe-and-Filter View

The pipe-and-filter view shows data processing pipelines where filters transform data and pipes transport it between filters.

Figure 6: Pipe-and-Filter View

2.4 Allocation Views

Allocation views show how software elements map to non-software elements in the environment—hardware, file systems, teams, and so forth.

2.4.1 Deployment View

The deployment view shows how software artifacts are deployed to hardware or virtualized infrastructure.

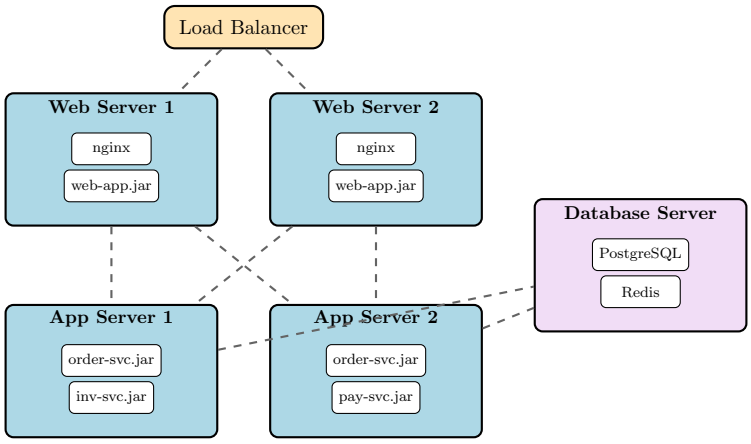


Figure 7: Deployment View

2.5 View Type Selection Guide

Table 1: View Type Selection Guide

Stakeholder Concern	Recommended View	Primary Presentation Style
Code organization	Module Decomposition	Hierarchical boxes
Build dependencies	Uses View	Directed dependency graph
Separation of concerns	Layered View	Stacked layers with rules
Runtime behavior	C&C View	Components with connectors
Service integration	Service-Oriented View	Services, APIs, messages
Data processing	Pipe-and-Filter View	Linear or branching pipeline
Hardware mapping	Deployment View	Nodes with artifacts
Team responsibilities	Work Assignment View	Modules mapped to teams
Data structure	Data Model View	Entity-relationship diagram
Security boundaries	Security View	Trust zones and controls

3 Diagram Design Principles

3.1 Visual Communication Fundamentals

Effective architectural diagrams apply principles of visual communication to maximize comprehension and minimize cognitive load.

Design Principle

Gestalt Principles for Architecture Diagrams:
Proximity: Elements that are related should be placed near each other. Group services that collaborate closely.
Similarity: Elements of the same type should look similar. Use consistent shapes, colors, and sizes for similar elements.
Enclosure: Use boundaries to show containment and grouping. Layers, subsystems, and trust boundaries benefit from enclosure.
Connection: Lines connecting elements imply relationship. Use consistent line styles for consistent relationship types.
Continuity: The eye follows smooth paths. Avoid unnecessary line crossings and jagged routes.

3.2 Clarity and Simplicity

The primary goal of a diagram is communication. Clarity must be prioritized over completeness or aesthetic concerns.

3.2.1 The Rule of Seven

Cognitive research suggests that humans can hold approximately seven items (plus or minus two) in working memory. Diagrams should respect this limitation.

Best Practice

Managing Diagram Complexity:

- Limit top-level elements to 7±2 items
- Use hierarchical decomposition for complex systems
- Create multiple focused diagrams rather than one overwhelming diagram
- Hide detail through abstraction—show it in supporting diagrams
- Use consistent chunking to group related elements

3.2.2 Progressive Disclosure

Not all stakeholders need the same level of detail. Design primary presentations to support progressive disclosure—overview first, detail on demand.

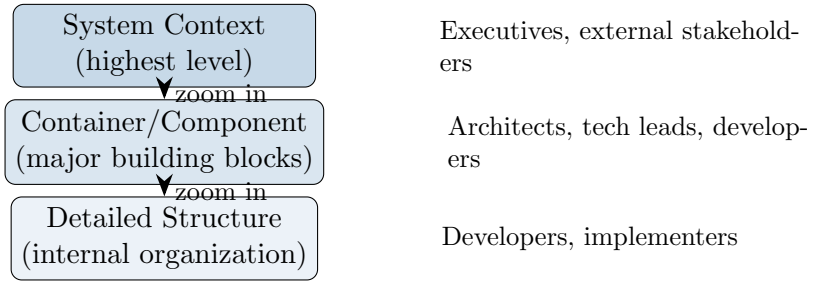


Figure 8: Progressive Disclosure Hierarchy

3.3 Consistency

Consistency reduces cognitive load by allowing viewers to learn the notation once and apply it throughout.

3.3.1 Internal Consistency

Within a single diagram, maintain consistency in shape usage (same shapes for same element types), color coding (same colors for same categories), line styles (same styles for same relationship types), sizing (similar elements have similar sizes), labeling (same format and detail level for labels), and positioning (similar elements in similar positions).

3.3.2 External Consistency

Across diagrams within a documentation set, maintain consistency in notation conventions (use the same legend throughout), abstraction levels (similar detail at similar zoom levels), terminology (same names for same concepts), and visual style (consistent look and feel).

3.4 Layout Strategies

Effective layout makes diagrams easier to read and understand.

3.4.1 Hierarchical Layout

Use hierarchical layout for decomposition, containment, and dependency relationships. Parent elements at top, children below, with dependencies flowing downward.

3.4.2 Layered Layout

Use layered layout for architectural layers, protocol stacks, and abstract-to-concrete relationships. Higher abstraction at top, lower abstraction at bottom.

3.4.3 Orthogonal Layout

Use orthogonal (grid-based) layout for clarity in complex diagrams. Elements aligned to grid, connections using horizontal and vertical segments only.

3.4.4 Circular/Radial Layout

Use circular layout to show peer relationships or to highlight a central element with its connections.

Best Practice

Layout Guidelines:

- Minimize line crossings—each crossing increases cognitive load
- Maintain consistent spacing between elements
- Align elements to an implicit grid
- Leave adequate white space—crowded diagrams are hard to read
- Place the most important elements prominently (center or top)
- Consider reading direction (left-to-right, top-to-bottom in Western cultures)

3.5 Color Usage

Color is a powerful visual channel but must be used carefully.

Table 2: Effective Color Usage

Purpose	Approach	Guidelines
Categorization	Distinct hues	Use 5–7 colors maximum; ensure adequate contrast
Layering	Value gradients	Darker for lower layers, lighter for higher

Purpose	Approach	Guidelines
Status/State	Semantic colors	Red=error, yellow=warning, green=healthy
Emphasis	Saturation	Saturated for focus, desaturated for context
Grouping	Background fills	Light backgrounds for regions; consistent within groups

Warning

Color Accessibility:

- Approximately 8% of men have color vision deficiency
- Never use color as the only distinguishing characteristic
- Combine color with shape, pattern, or label differences
- Test diagrams with color blindness simulators
- Ensure sufficient contrast for readability

4 Notation Standards and Conventions

4.1 Choosing a Notation

The choice of notation affects how effectively the diagram communicates to its intended audience. Consider standardization (well-known notations reduce learning curve), expressiveness (the notation should represent needed concepts), tool support (modeling tools should support the notation), and audience familiarity (match notation to stakeholder expectations).

4.2 UML for Architecture

The Unified Modeling Language provides several diagram types useful for architectural documentation.

4.2.1 UML Component Diagram

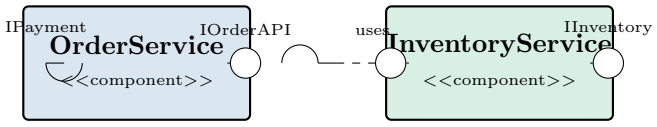


Figure 9: UML Component Diagram Notation

4.2.2 UML Deployment Diagram

Table 3: UML Deployment Diagram Elements

Symbol	Name	Description
	Node	Physical or virtual computing resource
	Artifact	Deployable unit (JAR, WAR, executable)
	Communication Path	Network connection between nodes
	Deployment	Artifact deployed to node

4.3 C4 Model Notation

The C4 model provides a hierarchical approach with four levels: Context, Container, Component, and Code.

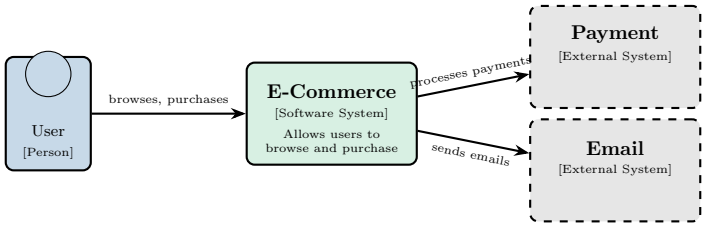


Figure 10: C4 Context Diagram Notation

4.3.1 C4 Notation Elements

Table 4: C4 Model Element Types

Element	Visual Style	Description
Person	Stick figure or labeled box	A human user of the system
Software System	Solid box with label	The system being described or an external system
Container	Solid box (different color)	An application or data store within the system
Component	Solid box (lighter)	A grouping of related functionality within a container
External System	Dashed box (gray)	A system outside your control
Relationship	Arrow with label	Describes the interaction between elements

4.4 ArchiMate Notation

ArchiMate provides enterprise architecture notation spanning business, application, and technology layers.

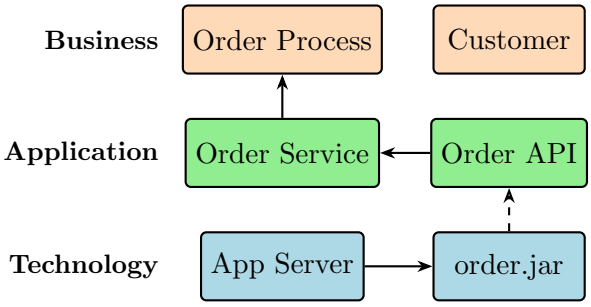


Figure 11: ArchiMate Cross-Layer Notation

4.5 Custom Notation

When standard notations are insufficient or inappropriate, custom notation may be used. However, custom notation requires careful documentation.

Template

Custom Notation Documentation Requirements:
For each custom element type:

- Visual representation (shape, color, border style)
- Semantic meaning (what it represents)
- Usage guidelines (when to use this element type)
- Examples (concrete instances)

For each custom relationship type:

- Visual representation (line style, arrowheads, decorations)
- Semantic meaning (what relationship it represents)
- Directionality (what source and target mean)
- Labels (what information goes on the line)

5 Element Key and Legend Design

5.1 Purpose of the Legend

Every primary presentation must include a legend (element key) that explains the notation used. The legend serves as a reference for interpreting the diagram, defines the visual vocabulary, ensures consistent interpretation across readers, and documents any non-standard notation.

5.2 Legend Structure

A comprehensive legend includes element types, relationship types, and any special annotations or conventions.

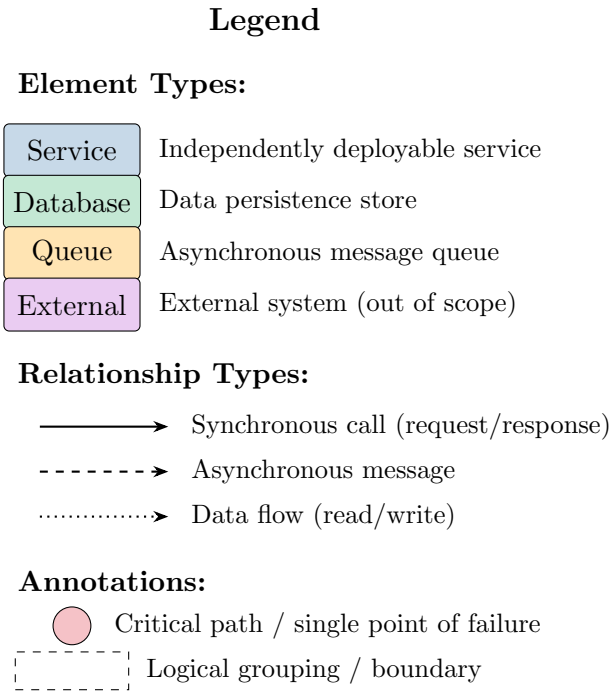


Figure 12: Comprehensive Legend Example

5.3 Element Type Documentation

For each element type in the legend, document the visual representation, semantic meaning, typical properties, and examples from this diagram.

Table 5: Element Type Documentation Template

Type Name	Visual	Meaning	Examples
Service	Service	Independently deployable unit providing business capability	Order Service, Payment Service
Database		Persistent data storage with query capability	OrderDB, UserDB
Queue	Queue	Asynchronous message buffer	Event Bus, Task Queue
External	External	System outside our control	Payment Gateway, Email Provider

5.4 Relationship Type Documentation

Table 6: Relationship Type Documentation Template

Type	Visual	Meaning	Examples
Sync Call	——→	Synchronous request-response invocation	API calls, RPC
Async Message	- - - - - - →	Fire-and-forget or pub/sub message	Events, commands
Data Flow	·····→	Data transfer (read or write)	DB queries, file writes
Dependency	——→	Compile-time or structural dependency	Uses, imports
Contains	◆——	Composition / containment	Module contains class

6 Narrative Description Techniques

6.1 The Role of Narrative

The narrative description is a textual walkthrough that guides readers through the primary presentation. It complements the visual diagram by explaining intent, significance, and context that cannot be conveyed visually.

Key Point

The narrative should answer questions the diagram raises but cannot answer: Why is the system structured this way? What are the significant design decisions? How does this structure support quality attributes?

6.2 Narrative Structure

An effective narrative follows a logical structure that builds understanding progressively.

1. **Overview:** Start with a one-paragraph summary of what the diagram shows
2. **Major Elements:** Describe the primary elements and their responsibilities
3. **Key Relationships:** Explain the most important interactions and dependencies
4. **Data and Control Flow:** Describe how data and control move through the structure
5. **Quality Attribute Support:** Explain how the structure supports key quality attributes
6. **Notable Decisions:** Highlight significant design decisions and their rationale

6.3 Writing Effective Narratives

Best Practice

Narrative Writing Guidelines:
Be purposeful: Every sentence should add value. Avoid restating what is obvious from the diagram.
Use consistent terminology: Use element names exactly as they appear in the diagram.
Explain significance: Don't just describe what is shown; explain why it matters.
Connect to concerns: Relate structural choices to stakeholder concerns and quality attributes.
Be concrete: Use specific examples of scenarios and interactions.
Maintain appropriate detail: Match detail level to audience; don't overwhelm with implementation specifics.

6.4 Narrative Example

Example

Narrative for Service-Oriented Architecture Diagram

Overview: This diagram presents the runtime structure of the e-commerce platform as a collection of independently deployable services communicating through synchronous APIs and asynchronous events.

Major Elements: The system consists of four primary services. The *API Gateway* serves as the single entry point for all client requests, handling authentication, rate limiting, and request routing. The *Order Service* manages the complete order lifecycle from creation through fulfillment. The *Inventory Service* maintains real-time inventory levels and handles stock reservations. The *Payment Service* processes payments through integration with external payment providers.

Key Relationships: The API Gateway routes all external requests to appropriate backend services, providing a unified interface while hiding internal complexity. The Order Service orchestrates order creation by coordinating with Inventory (to reserve stock) and Payment (to process charges) services. All services publish domain events to the Event Bus, enabling loose coupling and eventual consistency.

Quality Attribute Support: This structure supports several key quality attributes. *Availability* is enhanced through service independence—failure of the Payment Service does not affect product browsing. *Scalability* is achieved by allowing individual services to scale independently based on their specific load. *Modifiability* benefits from clear service boundaries—changes to payment processing are isolated to one service.

Notable Decisions: The choice of asynchronous events for inter-service communication (rather than synchronous calls for everything) was driven by resilience requirements. Services can continue operating during temporary unavailability of other services, with events queued for later processing.

7 Behavioral Overview Integration

7.1 Static vs. Dynamic Views

The primary presentation typically shows static structure—elements and their relationships at rest. However, architecture is also about behavior—how elements interact over time. The behavioral overview bridges this gap.

7.2 Behavioral Summary Approaches

7.2.1 Scenario Walkthroughs

Describe key scenarios as sequences of interactions through the structure.

Example

Scenario: Customer Places Order

1. Customer submits order through Web UI to API Gateway
2. API Gateway authenticates request and routes to Order Service
3. Order Service validates order and requests inventory reservation from Inventory Service
4. Inventory Service reserves stock and returns confirmation
5. Order Service requests payment processing from Payment Service
6. Payment Service processes payment through external gateway
7. Order Service publishes OrderCreated event to Event Bus
8. Notification Service (subscribed to events) sends confirmation email

7.2.2 Interaction Diagrams

Reference supporting sequence or collaboration diagrams that detail important interactions.

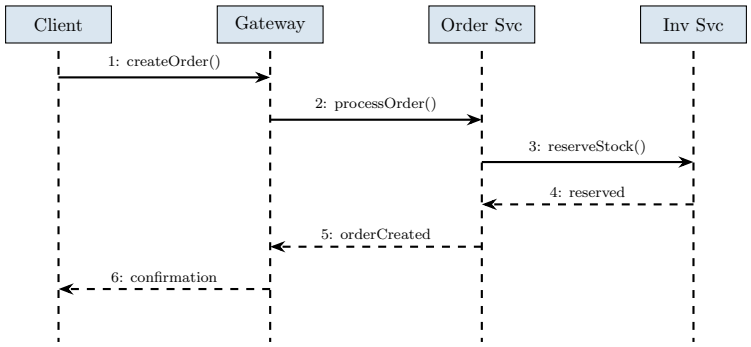


Figure 13: Order Creation Sequence (Simplified)

7.2.3 State and Lifecycle

For elements with significant state, summarize their lifecycle.

Table 7: Element Lifecycle Summary

Element	Key States	Lifecycle Notes
Order Service	Starting, Ready, Degraded, Stopping	Starts with database connection; enters Degraded if DB unavailable
Order Entity	Pending, Confirmed, Shipped, Delivered, Cancelled	Transitions driven by events; terminal states are Delivered, Cancelled
Inventory Reservation	Pending, Committed, Released	15-minute timeout on Pending; auto-release if not committed

7.3 Behavioral References

Link to detailed behavioral models maintained separately.

Table 8: Behavioral Model References

Scenario/Behavior	Model Type	Reference
Order Creation Flow	Sequence Diagram	[DOC:behavior/seq-order-create.md]
Order Cancellation	Sequence Diagram	[DOC:behavior/seq-order-cancel.md]
Payment Processing	Activity Diagram	[DOC:behavior/act-payment.md]
Order State Machine	State Diagram	[DOC:behavior/state-order.md]
Inventory Saga	Saga Diagram	[DOC:behavior/saga-inventory.md]

8 Constraints and Design Rules

8.1 Purpose of Documented Constraints

Constraints and design rules are architectural decisions that restrict how elements may be used, connected, or changed. Documenting these rules in the primary presentation ensures that implementers understand and follow the intended architecture.

8.2 Types of Constraints

8.2.1 Structural Constraints

Structural constraints govern how elements may be organized and related.

Table 9: Structural Constraints

Constraint Type	Description	Example
Layering Rules	Allowed dependencies between layers	UI layer may only call Application layer
Containment Rules	What may contain what	Services contain components; components contain classes
Cardinality	Required number of elements	Each service must have exactly one database
Exclusion	Elements that may not coexist	Synchronous and async variants of same interface
Required Elements	Elements that must exist	Every service must have health check endpoint

8.2.2 Communication Constraints

Communication constraints govern how elements may interact.

Table 10: Communication Constraints

Constraint Type	Description	Example
Protocol Requirements	Required communication protocols	All service-to-service calls use gRPC
Direction Rules	Allowed call directions	Services may not call the API Gateway
Intermediary Requirements	Required intermediate elements	All external calls must go through Gateway
Async Requirements	When async is required	Cross-domain communication must be async

8.2.3 Technology Constraints

Technology constraints mandate or prohibit specific technologies.

Table 11: Technology Constraints

Constraint Type	Description	Example
Language/Framework	Mandated or prohibited technologies	Services must use Java or Go
Data Stores	Allowed database technologies	Relational data uses PostgreSQL only
Infrastructure	Required platforms	All services deploy to Kubernetes
Libraries	Mandated libraries for cross-cutting concerns	All services use OpenTelemetry for tracing

8.3 Constraint Documentation Format

Template

Constraint Specification Template

Constraint ID: Unique identifier
Name: Short descriptive name
Category: Structural / Communication / Technology / Quality
Statement: Precise statement of the constraint
Rationale: Why this constraint exists
Enforcement: How the constraint is enforced (review, tooling, tests)
Exceptions: Any allowed exceptions and approval process
Related Decisions: Link to architectural decision record

8.4 Example Constraints

Constraint: LAYER-001 – Layer Dependency Direction

Statement: Dependencies between layers must flow downward only. A higher layer may depend on a lower layer, but a lower layer may not depend on a higher layer.

Rationale: Enforcing unidirectional dependencies ensures that lower layers remain stable and reusable. Changes to the UI layer should not require changes to business logic or data access layers.

Enforcement:

- Static analysis tools (ArchUnit, NDepend) run in CI pipeline
- Architecture review checklist includes dependency verification
- Violations fail the build

Exceptions: Dependency inversion through interfaces is allowed (lower layer defines interface, higher layer implements). Must be documented in element catalog.

Constraint: COMM-003 – Cross-Domain Async Requirement

Statement: Communication between services in different business domains must use asynchronous messaging via the Event Bus. Synchronous calls are prohibited.

Rationale: Asynchronous communication between domains reduces coupling, improves resilience (services can operate during partner unavailability), and supports independent scaling and deployment.

Enforcement:

- Service mesh policies reject cross-domain synchronous calls
- Architecture review for new integrations
- Domain boundary documentation maintained

Exceptions: Time-critical operations (payment confirmation) may use synchronous calls with architecture board approval. Must implement circuit breaker pattern.

9 Quality Attribute Visualization

9.1 Making Quality Attributes Visible

While structure is the primary focus of architectural diagrams, quality attributes often drive structural decisions. Making quality considerations visible helps stakeholders understand why the architecture is designed as it is.

9.2 Visual Annotation Techniques

9.2.1 Color Coding for Criticality

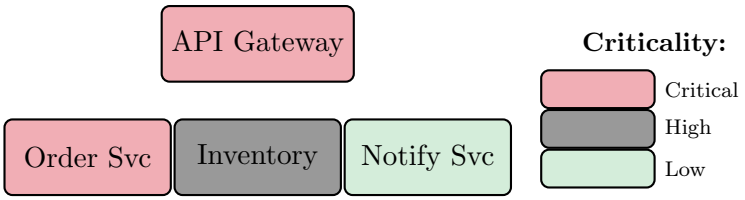


Figure 14: Criticality Visualization

9.2.2 Annotations for Performance

Add performance annotations directly to elements or relationships.

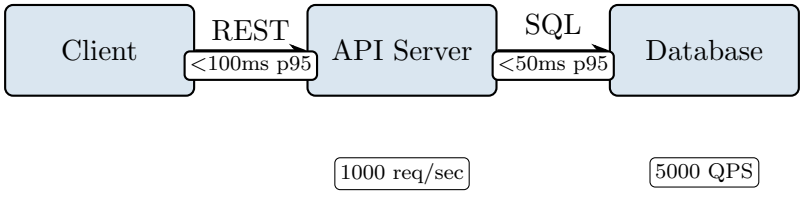


Figure 15: Performance Annotations

9.2.3 Security Zone Boundaries

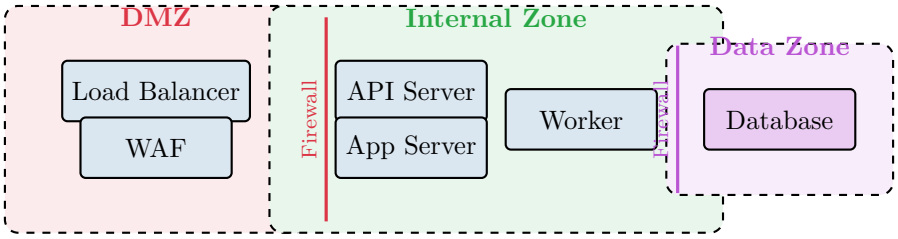


Figure 16: Security Zone Visualization

9.2.4 Redundancy and Failover

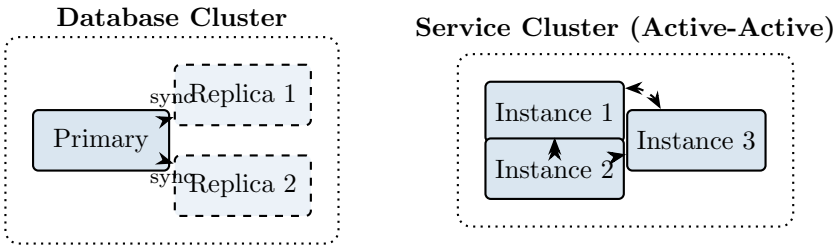


Figure 17: Redundancy and Failover Patterns

9.3 Quality Attribute Summary Table

Include a summary table linking structural elements to quality attributes.

Table 12: Quality Attribute Summary by Element

Element	Avail.	Perf.	Security	Notes
API Gateway	99.99%	<50ms	TLS, Auth	Redundant; rate limiting
Order Service	99.95%	<200ms	Internal	3 replicas; circuit breaker
Database	99.99%	<10ms	Encrypted	Primary + 2 replicas
Event Bus	99.9%	best effort	Internal	3-node Kafka cluster

10 Known Issues and Future Work

10.1 Purpose of Issue Documentation

Documenting known issues and incomplete areas is essential for honest communication with stakeholders. It sets appropriate expectations, identifies areas needing attention, and provides context for future work.

10.2 Issue Categories

10.2.1 Accuracy Issues

The diagram does not fully represent the current or intended state.

Table 13: Accuracy Issues

ID	Issue	Impact	Resolution Plan
A1	Legacy Order Service not shown	Incomplete picture of dependencies	Add to diagram Q2
A2	Cache layer omitted for simplicity	May confuse performance analysis	Add cache to v2.0

10.2.2 Completeness Issues

Areas that are not yet fully documented or designed.

Table 14: Completeness Issues

ID	Issue	Impact	Resolution Plan
C1	Mobile backend not designed	Mobile integration unclear	Design sprint scheduled
C2	Disaster recovery not shown	Operational gaps	DR architecture in progress

10.2.3 Pending Decisions

Architectural decisions that affect the diagram but are not yet made.

Table 15: Pending Decisions

ID	Decision Needed	Diagram Impact	Target Date
D1	Event bus technology selection	May change connector notation	March 15
D2	Multi-region strategy	Major structural changes	April 1

11 Diagram Maintenance and Governance

11.1 Keeping Diagrams Current

Diagrams that don't match reality are worse than no diagrams—they actively mislead. Establishing governance processes ensures diagrams remain accurate and useful.

11.2 Update Triggers

Diagrams should be updated when new elements are added to the system, elements are removed or deprecated, relationships between elements change, quality attribute requirements change significantly, technology choices change, and organizational boundaries change (for work assignment views).

11.3 Review Process

1. **Scheduled Reviews:** Quarterly reviews of all primary presentations
2. **Change-Triggered Reviews:** Review after significant system changes
3. **Stakeholder Feedback:** Process for stakeholders to report inaccuracies
4. **Implementation Comparison:** Periodic comparison with actual implementation

11.4 Version Control

Table 16: Diagram Version History

Version	Date	Author	Changes
1.0	2024-01-15	J. Smith	Initial diagram
1.1	2024-02-20	A. Jones	Added Event Bus; updated service names
1.2	2024-03-15	J. Smith	Added caching layer; updated legend
2.0	2024-06-01	Architecture Team	Major revision for microservices migration

11.5 Tooling Recommendations

Table 17: Diagramming Tool Comparison

Tool	Strengths	Best For
Structurizr	C4 model; diagram-as-code; versioning	Teams adopting C4; code-first approach
Lucidchart	Collaborative; polished output; templates	Distributed teams; executive presentations
draw.io	Free; version control friendly; flexible	Budget-conscious; Git integration

Tool	Strengths	Best For
PlantUML	Text-based; automated generation; CI-friendly	Developer documentation; automation
Miro	Collaborative whiteboarding; flexible	Workshops; brainstorming; informal
Enterprise Architect	Full UML; model repository; traceability	Large enterprises; formal modeling

12 Appendix A: Notation Quick Reference

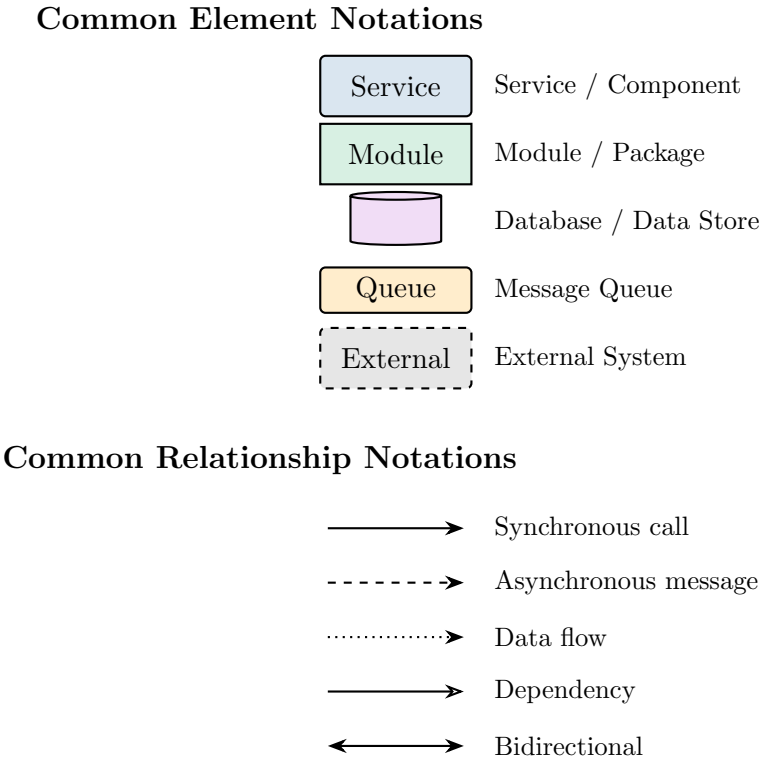


Figure 18: Notation Quick Reference

13 Appendix B: Primary Presentation Checklist

13.1 Completeness Checklist

All significant elements from the view are shown

All significant relationships are shown

Element names match catalog entries

Legend/key explains all notation used

Diagram has descriptive title

Version and date are indicated

Narrative description is provided

Constraints and rules are documented

Known issues are documented

13.2 Quality Checklist

Diagram is comprehensible at a glance

Complexity is appropriate ($\leq$15 elements typically)

Layout is clean and organized

Line crossings are minimized

Notation is used consistently

Colors are accessible (not color-only differentiation)

Text is readable at intended display size

White space is adequate

13.3 Accuracy Checklist

Diagram matches current/intended implementation

Relationships accurately represent actual interactions

Technology choices are correctly represented

Boundaries and groupings are accurate

Quality attribute annotations are current

Reviewed by knowledgeable stakeholder

14 Appendix C: Glossary

C4 Model A hierarchical approach to software architecture diagramming with four levels: Context, Container, Component, and Code

Component A modular part of a system with defined interfaces that encapsulates implementation

Connector An architectural element that mediates interaction between components

Element A fundamental building block represented in an architectural view

Legend A key explaining the notation used in a diagram

Module A code unit that implements a coherent set of responsibilities

Narrative The textual description accompanying a diagram

Notation The visual vocabulary used to represent architectural concepts

Primary Presentation
The main diagram(s) showing an architectural view

Quality Attribute
A measurable property of a system (performance, security, etc.)

Relationship A connection between architectural elements

View A representation of a system from a particular perspective

Viewpoint A specification of conventions for constructing and using a view

15 Appendix D: References

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

2. Brown, S. (2018). *The C4 Model for Visualising Software Architecture*. Retrieved from https://c4model.com

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

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

5. Object Management Group. (2017). *OMG Unified Modeling Language (UML) Version 2.5.1*.

6. The Open Group. (2019). *ArchiMate 3.1 Specification*. Van Haren Publishing.

7. Tufte, E. R. (2001). *The Visual Display of Quantitative Information* (2nd ed.). Graphics Press.

8. Rozanski, N., & Woods, E. (2012). *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives* (2nd ed.). Addison-Wesley.