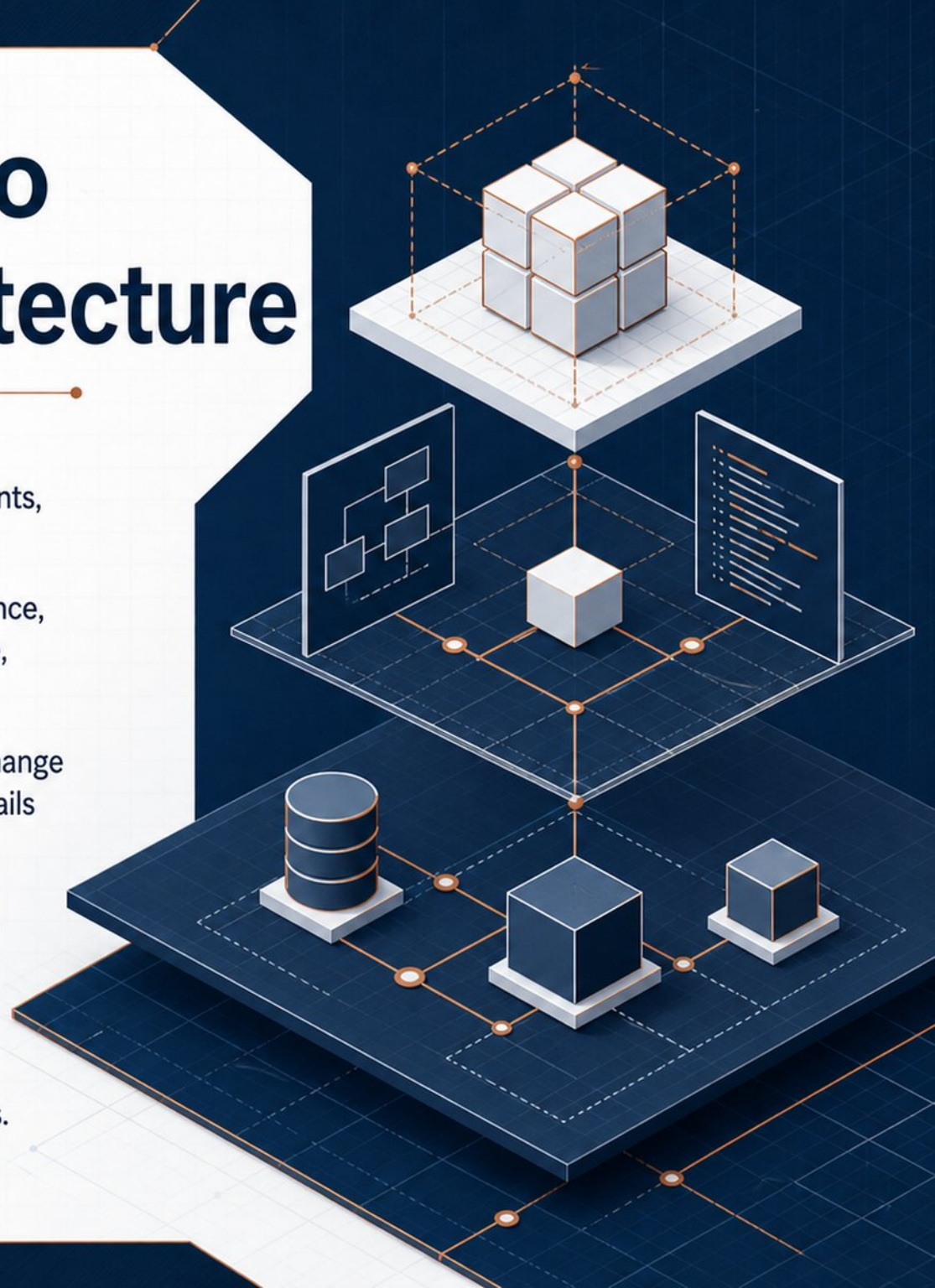


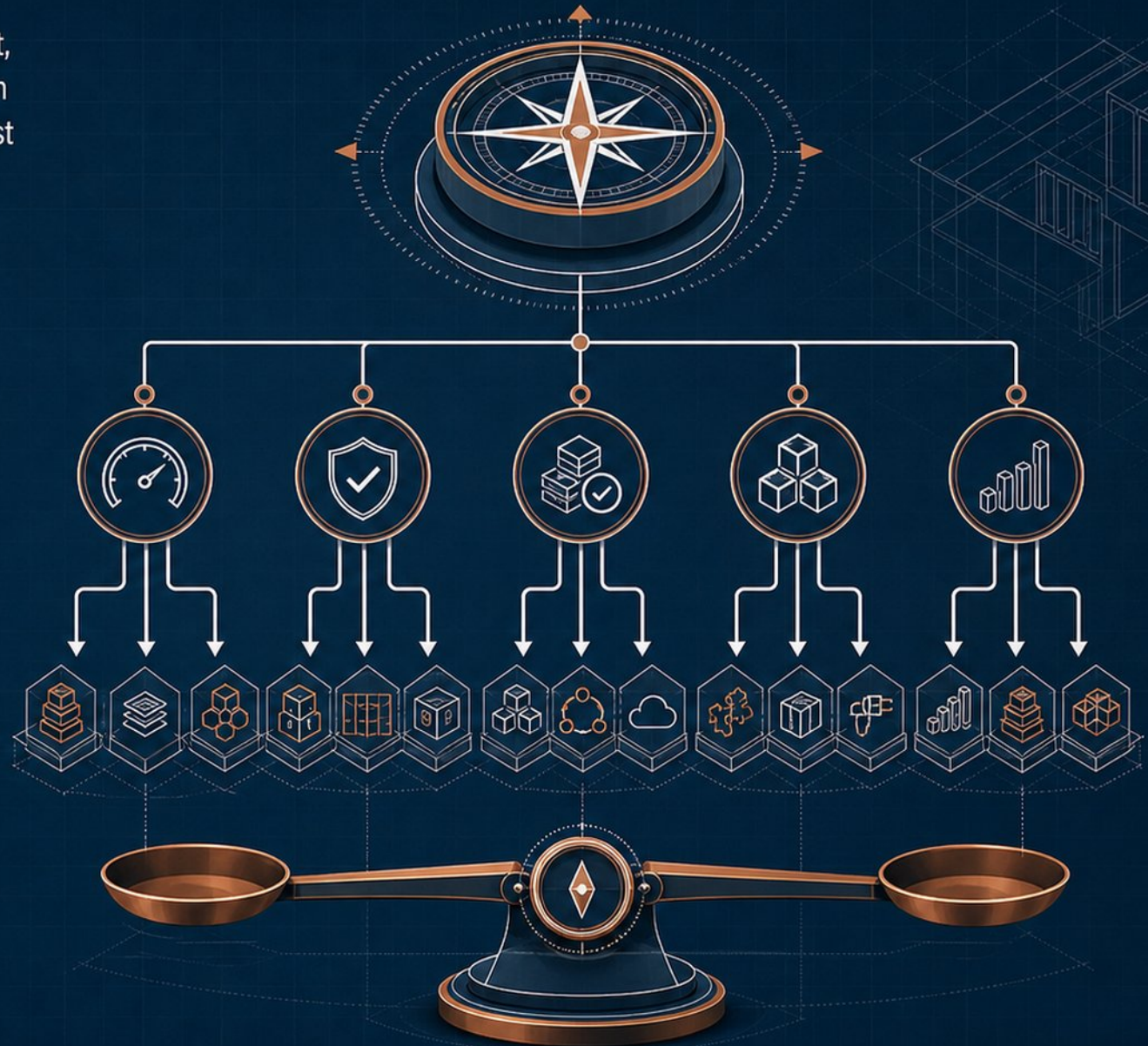
Introduction to Software Architecture

- Define architecture as the structures needed to reason about a system, encompassing components, relationships, and design principles.
- Architecture shapes quality attributes (performance, security, modifiability), influences team structure, and determines the long-term cost of change.
- Architecture addresses system-wide, hard-to-change decisions; design focuses on implementation details within a bounded context.
- Course scope: core concepts, organizing patterns, decision-making, documentation, and continuous evolution.
- Goal: making architecture a shared, practical responsibility for developers entering design roles.



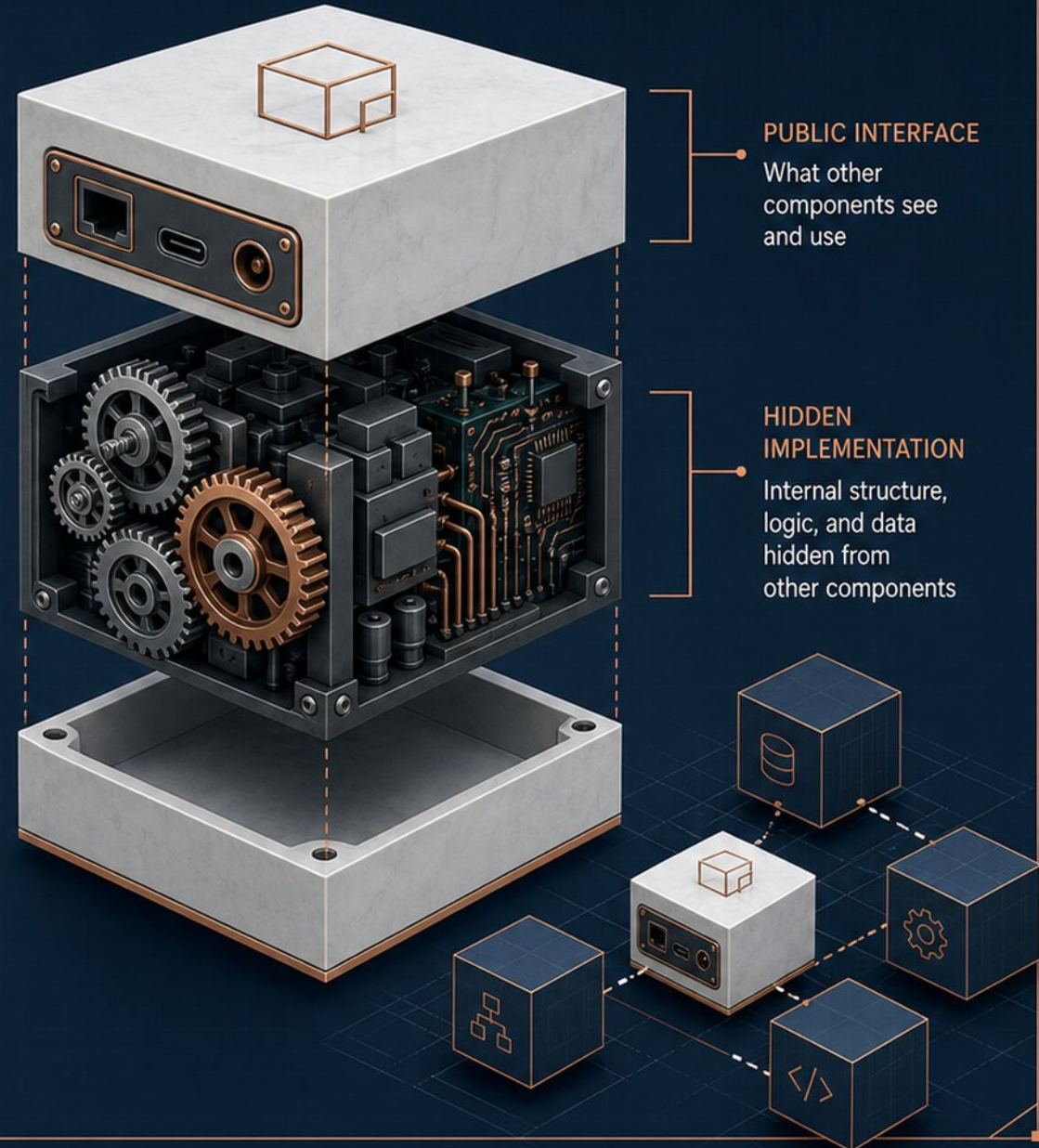
What Makes a Decision Architectural?

- Architectural decisions are those that, if changed, significantly affect system properties, coordination, or future cost and risk of change.
- Common categories: structural patterns, component boundaries, technology stacks, communication protocols, and data strategies.
- Quality attributes (the 'ilities')—performance, availability, security, modifiability, scalability—drive architectural choices.
- Runtime qualities (observed during execution) vs. development-time qualities (manifest during construction and maintenance).
- ATAM provides a structured way to evaluate decisions against competing quality attribute trade-offs.



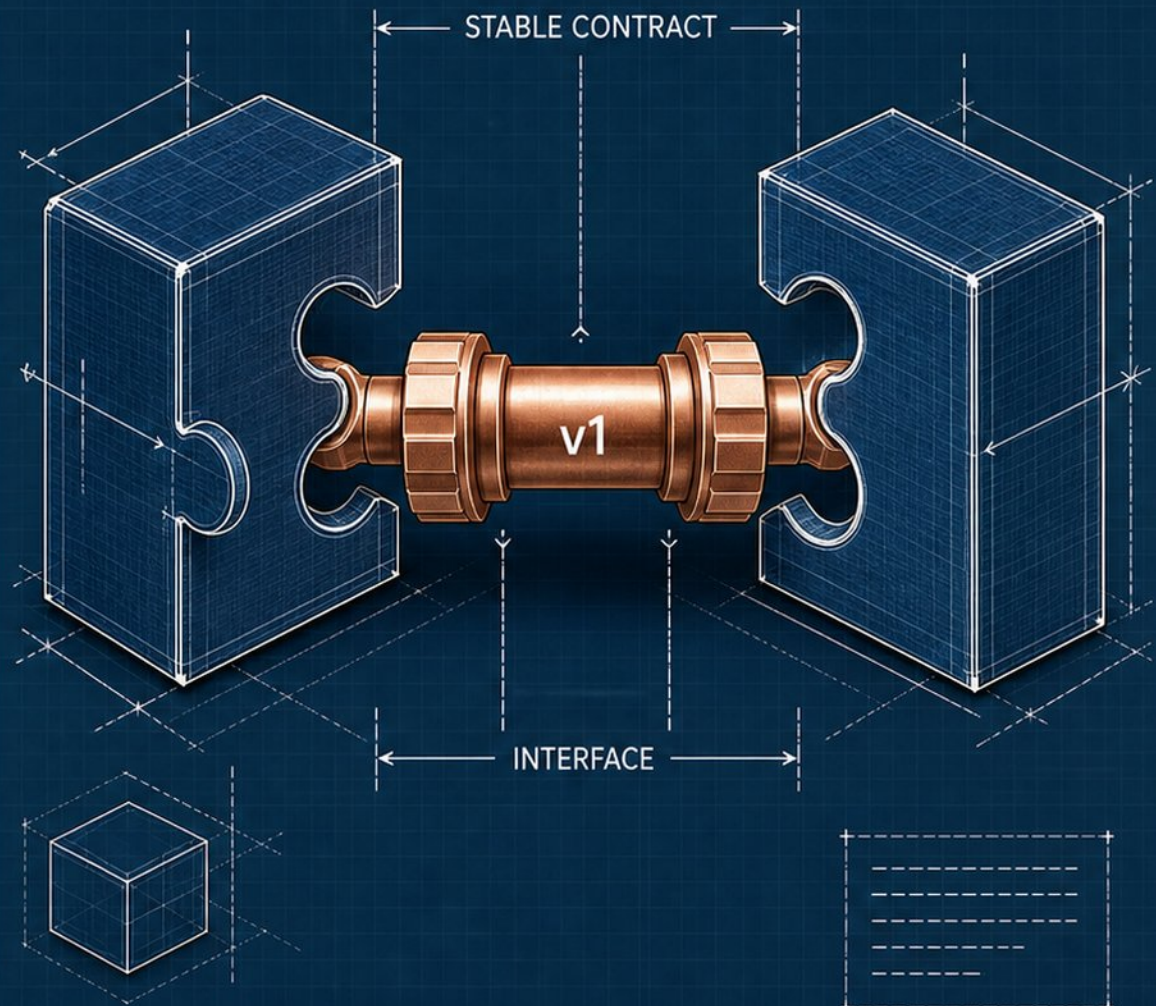
Architectural Components and Their Responsibilities

- ◆ A component is a modular, encapsulated unit with a specific responsibility
- ◆ Components expose behavior only through well-defined interfaces and hide implementation
- ◆ High cohesion: responsibilities inside a component belong together
- ◆ Low coupling: minimal dependencies on other components
- ◆ Components are independently replaceable and deployable units

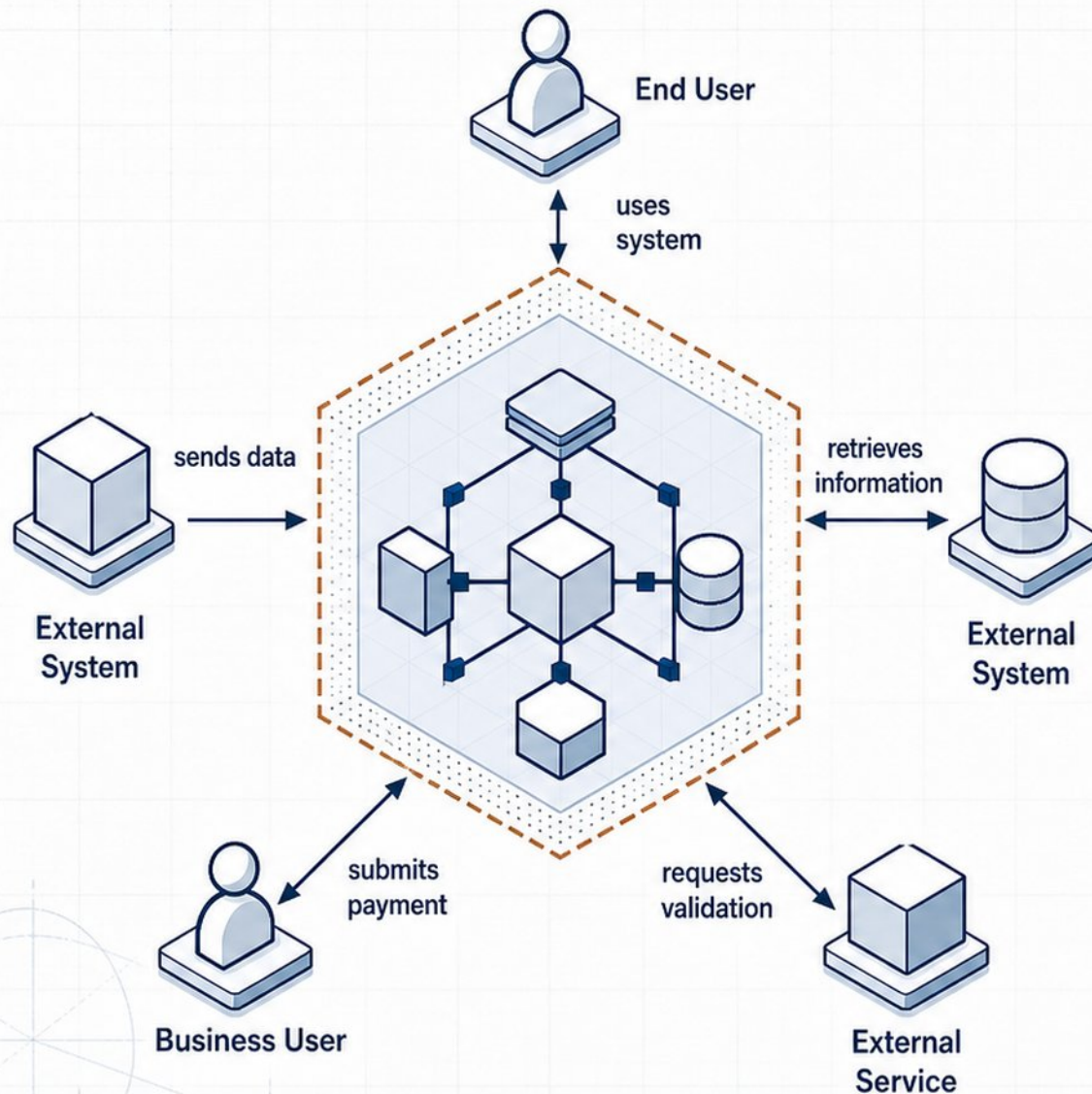


Interfaces and Contracts: The Architectural Glue

- An interface is a stable contract specifying operations, inputs, outputs, pre/postconditions, and error semantics.
- Synchronous (REST, gRPC) suits low-latency needs; asynchronous (messaging, events) decouples for resilience and scale.
- URI path versioning (/v1/) is the pragmatic default; date-based (Stripe-Version) excels at scale.
- Additive-only change: new optional fields and endpoints are safe; renaming or removing requires a new version.
- Deprecation lifecycle: Sunset/Deprecation headers, 12-month minimum runway, and transparent communication.



System Boundaries and Context



System boundaries define what is inside vs. outside the system



Boundary decisions align with team ownership and org structure



The C4 model provides hierarchical architecture visualization



Context Diagram (Level 1): center the system, surround with users and external systems



Label all relationships with intent (e.g., "submits payment")

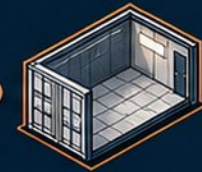


Keep technology-agnostic for shared technical/non-technical understanding

Zooming In: Containers and Components in the C4 Model



- **Container (Level 2):** A deployable unit that executes code or stores data (e.g., web apps, databases, file systems)



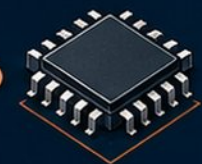
- **Container Diagram:** Zooms in to show technology choices, communication protocols (HTTPS, gRPC), and runtime responsibility distribution



- **Component (Level 3):** A grouping of related functionality inside a container, exposed through a well-defined interface (e.g., controllers, services)



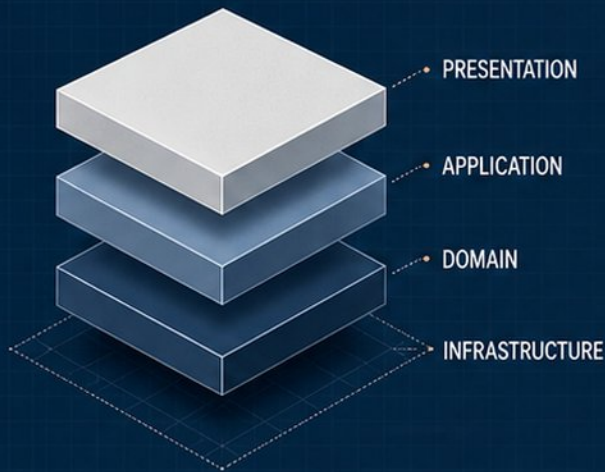
- **Component Diagram:** Developer-focused view of internal structure and dependencies within a single container



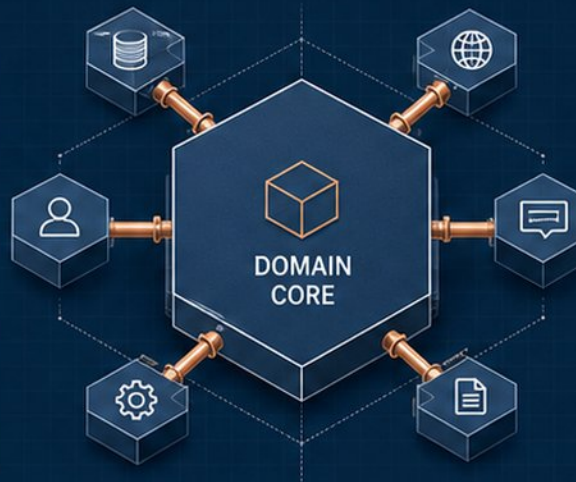
- **Code Diagram (Level 4):** Automatable, on-demand implementation details—typically generated from source code, not drawn manually

Organizing Components: Key Architectural Patterns

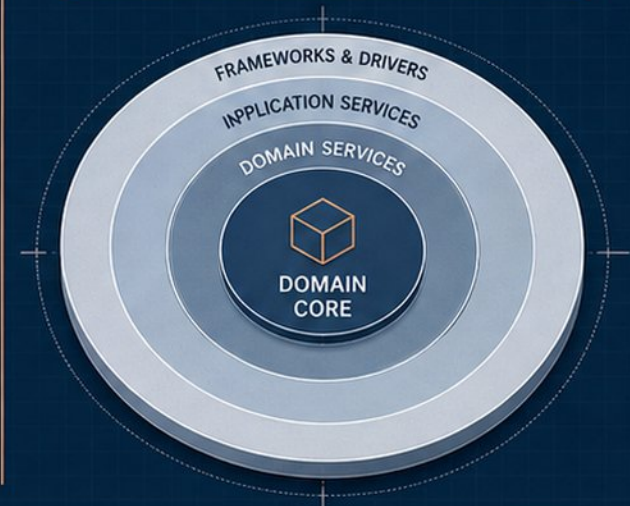
LAYERED ARCHITECTURE



HEXAGONAL (PORTS & ADAPTERS)



CLEAN/ONION ARCHITECTURE



- Layered Architecture: simple and familiar, but risks business-logic leakage and poor scalability
- Hexagonal (Ports & Adapters): isolates domain core behind interfaces for swappable infrastructure and deep testability
- Clean/'Onion Architecture: concentric layers with strict inward-dependency rules for long-lived complex domains
- Start simple with Layered or Modular Monolith; evolve to Hexagonal or Clean only when complexity justifies the cost
- These patterns are "domain-centric, dependency-direction control" — essentially the same spirit with different ceremony

Modern Architectural Styles: Monoliths, Microservices, and Event-Driven Systems



- "Modular Monolith: single deployable unit with enforced boundaries, microservices benefits without ops overhead"



- "Microservices: independently deployable services owning data, scaling teams at cost of complexity"



- "Beware the Distributed Monolith: services sharing databases or requiring synchronized releases"



- "Event-Driven: async events and messaging for decoupling, trades consistency for resilience"

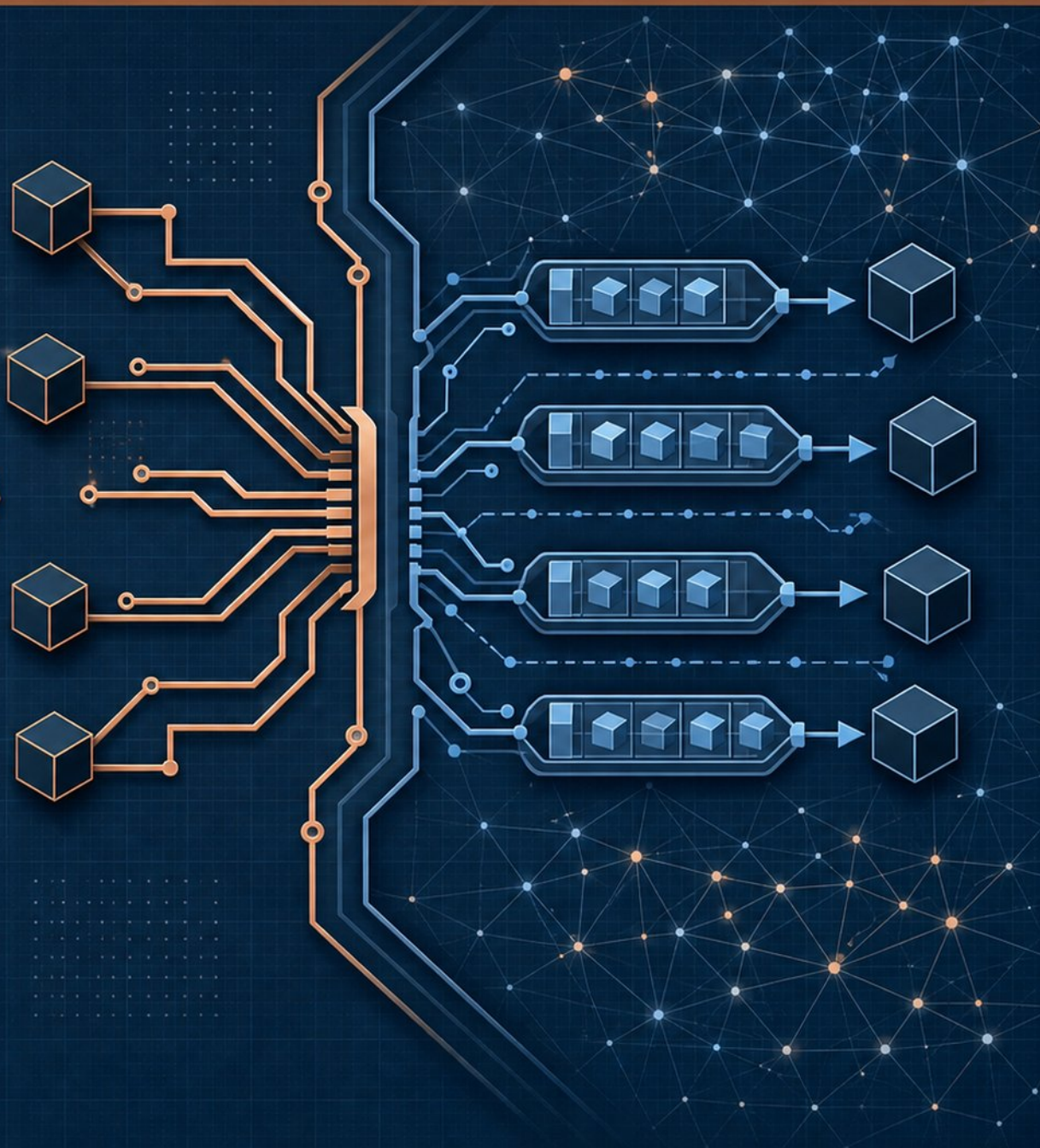


- "Decision guide: start with Modular Monolith below ~30 engineers; adopt Microservices only when org demands it"



Connecting Components: Communication and Integration

- » Local calls are fast and reliable; network calls introduce latency, partial failure, and the fallacies of distributed computing
- » Synchronous patterns (REST, gRPC) provide immediate responses; asynchronous patterns (pub/sub, event streaming) maximize decoupling and resilience
- » Every remote call is a contract: the interface defines the integration risk, and the message schema is a shared agreement
- » Communication style shapes performance, scalability, modifiability, and fault tolerance
- » JSON is human-readable; Protobuf and Avro optimize speed and schema evolution; backward/forward compatibility is essential



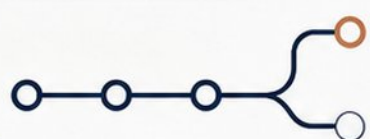
Design Decisions and Trade-Offs

- - No universally best pattern: every option fits a specific context and time
- - Use weighted matrix analysis: score options across 5–8 quality attributes
- - Core trade-off pairs: security vs. performance, consistency vs. availability
- - Trade-off points benefit one attribute but degrade another
- - The goal: make trade-offs visible, explicit, and stakeholder-aligned



Documenting Architecture Decisions with ADRs

- Architecture Decision Record (ADR): a one-page "why" document
- Standard structure: Title, Status, Context, Decision, Consequences, Alternatives
- Stored in-repo as versioned markdown; accepted records are immutable—only supersede
- Preserves institutional memory: records what was chosen and why
- "ADR-0047: Use Redis for Session Storage"—a concrete, honest template

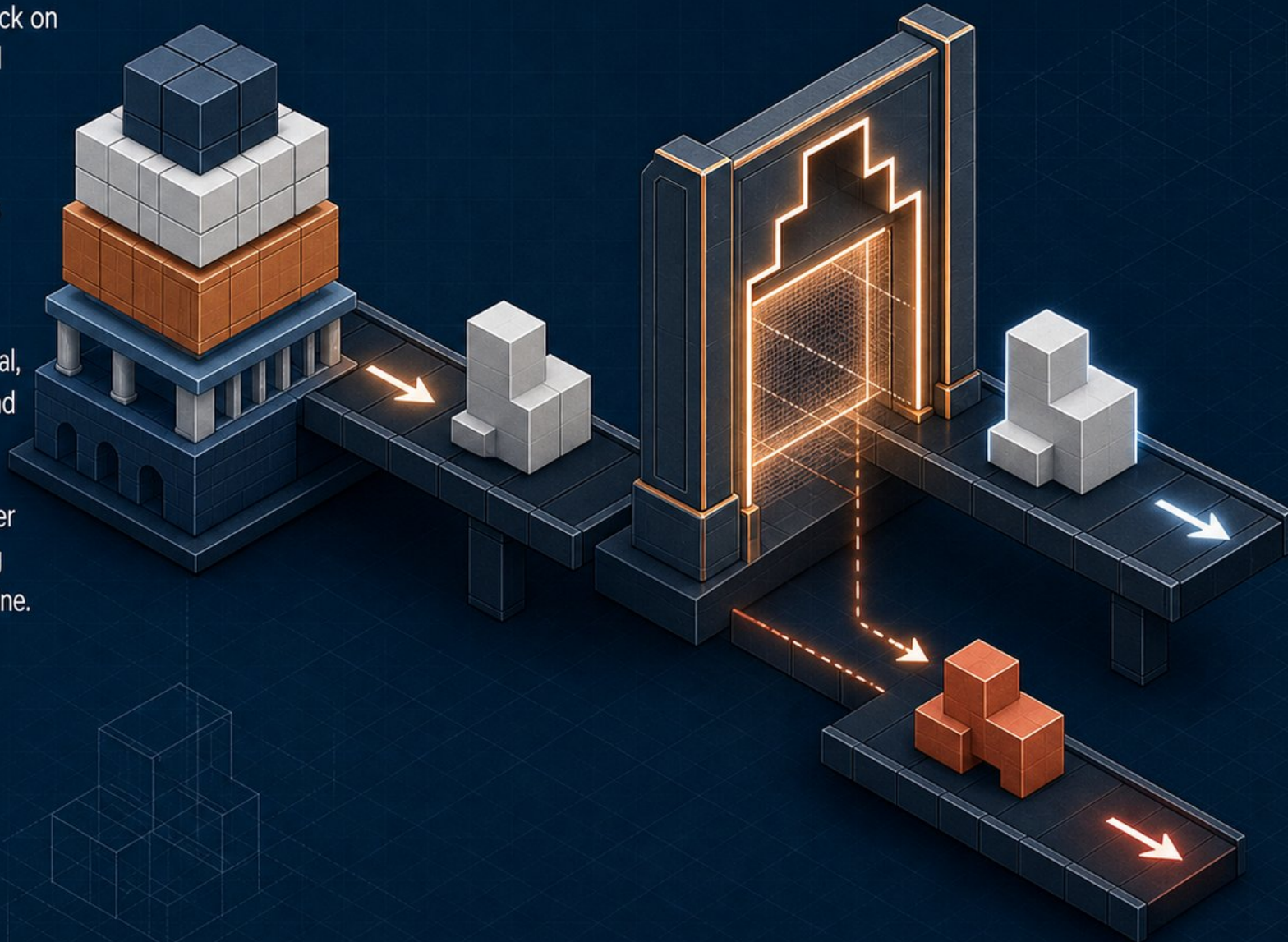


A diagram showing a sequence of four circles connected by a line. The line starts from the left, passes through three circles, and then branches into two lines, each ending in a circle. The top-right circle is orange, while the others are white.

TITLE	
STATUS	
CONTEXT	
DECISION	
CONSEQUENCES	
ALTERNATIVES	

Evolving Architecture Over Time with Fitness Functions

- A fitness function is an objective, repeatable check on one or more architectural characteristics.
- It acts as a guardrail, preventing code changes from degrading critical quality attributes.
- Categories include structural, behavioral, operational, and semantic fitness checks.
- ArchUnit rules enforce layer dependencies and naming conventions in the CI pipeline.
- Fitness functions shift governance left, turning periodic reviews into continuous, automated feedback.



Putting It All Together: A Practical Blueprint



1. Step 1: Define system context with a C4 System Context Diagram — users, external systems, and scope.



2. Step 2: Decompose into containers and components; assign responsibilities and define interfaces.



3. Step 3: Capture structural decisions (modular monolith vs. microservices) and communication patterns in ADRs.



4. Step 4: Write 3–5 measurable quality attribute scenarios with fitness functions to guard evolution.



5. Pitfalls to avoid: over-engineering, ignoring trade-offs, skipping documentation — good architecture is iterative and evidence-guided.

PersonWise.

PersonWise • AI digital-human course platform



Scan the code or click anywhere to watch this course live, with voice Q&A

personwise.ai/t/01339f9f/public