Class #7

03. Designing Software architectures

Software Architectures
Master in Informatics Engineering

Cláudio Teixeira (claudio@ua.pt)

Ensuring Portability in a Multi-Cloud World

Group assignment
20 min

https://docs.google.com/document/d/1Y8wjI8dQ1SspBuy5Y
38dln7xvO-dCuHrlOaamr7CZXE/edit?usp=sharing

Agenda

- Final practical assignment presentation

- Designing Software Architectures
    - Design principles
    - Systematic approaches:
        - Attribute driven design
        - attribute driven design (ADD)
        - architecture centric design method (ACDM)
        - architecture development method (ADM)

Final practical assignment presentation

https://docs.google.com/document/d/1XhFj-7M1LDN6G2C8wOaQ4nsKhVD6p5fYycUSnQEYeLU/edit?usp=sharing

Dates reminder

- Deliverable 1:
  - Presentation of architecture considerations and designs
  - 10 min
  - 6th and 7th May
  - Relates to “Assignment 2” (10%)
- Deliverable 2:
  - Presentation of architecture, deployment and implementation
  - Demo
  - 15 min
  - Semester’s last week
  - Relates to “Final group assignment” (50%)

Designing Software Architectures

Software architecture design

Perfect is the enemy of good.
– Voltaire

Software architecture design involves **making decisions** in order to **satisfy functional requirements, quality attributes, and constraints**. It is a **problem-solving process** that leads to the creation of an architecture design.

Software architecture design comprises defining the structures that will make up the solution and documenting them.

It's all about … decisions!

For each requirement, quality attribute or constraints there may be numerous ways to solve the problem.

You will need to consider the strengths and weaknesses of these alternatives in order to select the most appropriate choice.

A large part of software architecture design is making design decisions to resolve issues so that a solution can be implemented. As the software architect, you will be leading the decision-making process.

A completed design may not be perfect, as there will be conflicting requirements that need to be met, and trade-offs made in order to meet them.

Software architecture design

Agreeing on Terminology

Let’s define our terms for Software Development:

Element: generic term that can be used to represent any of the following terms: system, subsystem, module, or component. If we want to refer to pieces of a software application in a general way, we can refer to them as elements.

Structure: Structures are groupings of, and relations between, elements. Anything that is complex and made up of elements can be referred to as a structure.

System: represents the entire software project, including all of its subsystems. A system consists of one or more subsystems. It is the highest level of abstraction in a software architecture design.

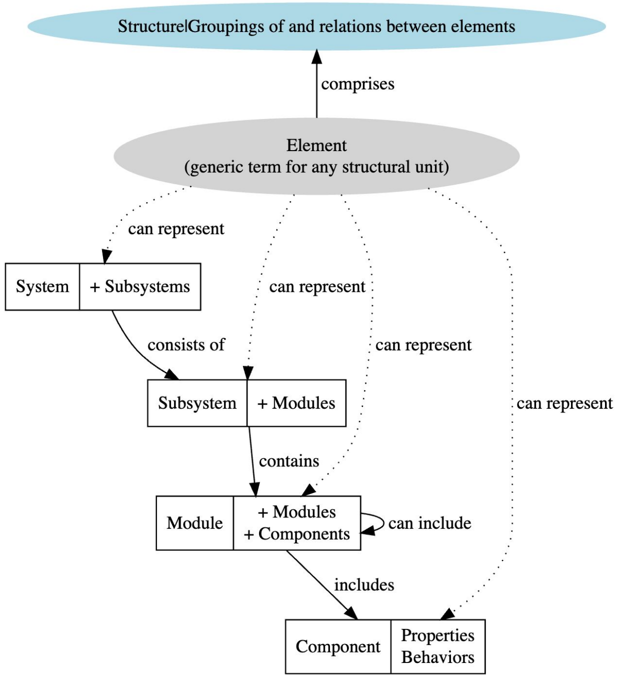
Subsystem: Subsystems are logical groupings of elements that make up a larger system. The subsystems can be created in a variety of ways, including partitioning a system by functionality.

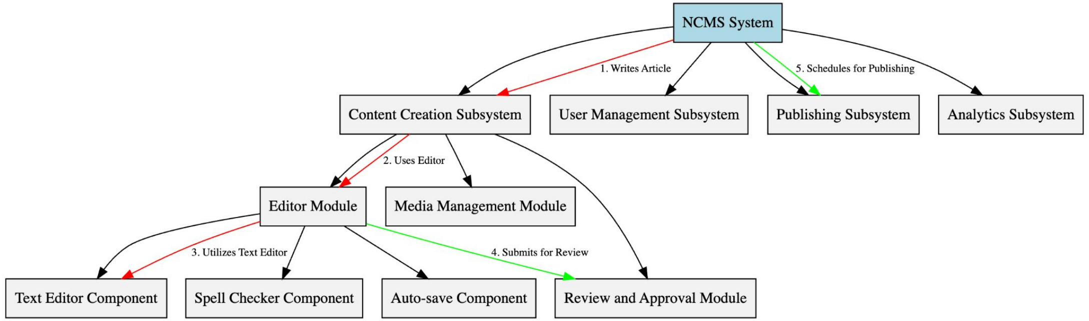
Module: like subsystems, this is a logical grouping of elements. Each module is contained within a subsystem and consists of other modules and/or components. They are typically focused on a single logical area of responsibility.

Component: Components are execution units that represent some well-defined functionality. They typically encapsulate their implementation and expose their properties and behaviors through an interface. Components are the smallest level of abstraction and typically have a relatively small scope. Components can be grouped together to form more complex elements, such as modules.

Software architecture design

Agreeing on Terminology





Possible representation of a Content Management System (CMS) for a news organization.
Scenario depicted: Creating and Publishing an Article

Architecture Design
Top-Down Approach

Begins with the highest level of system detail, specifying overall objectives and requirements before decomposing them into more detailed components. A top-down approach starts with the entire system at the highest level, and then a process of decomposition begins to work downward toward more detail.

A design using the top-down approach is typically performed iteratively, with increasing levels of decomposition. It is particularly effective if the domain is well understood (which is not always the case).

Advantages:

- Ensures alignment with business objectives from the start.
- Facilitates comprehensive understanding and planning of the system's architecture.
- Simplifies complex systems by breaking them down into manageable parts.

Use cases:

- Ideal for projects with well-defined goals and requirements, or when a clear vision of the end product exists. Particularly useful for large projects

Top-down Approach Exercise

Group assignment
45 min

https://docs.google.com/document/d/1eKvlXcnNi5b-PnASn0YL2QreDHn2keR2m3OEM7S4Iy8/edit?usp=sharing

Example

E-Commerce Platform using a top-down approach

1 => Define High-Level System Objectives and Requirements

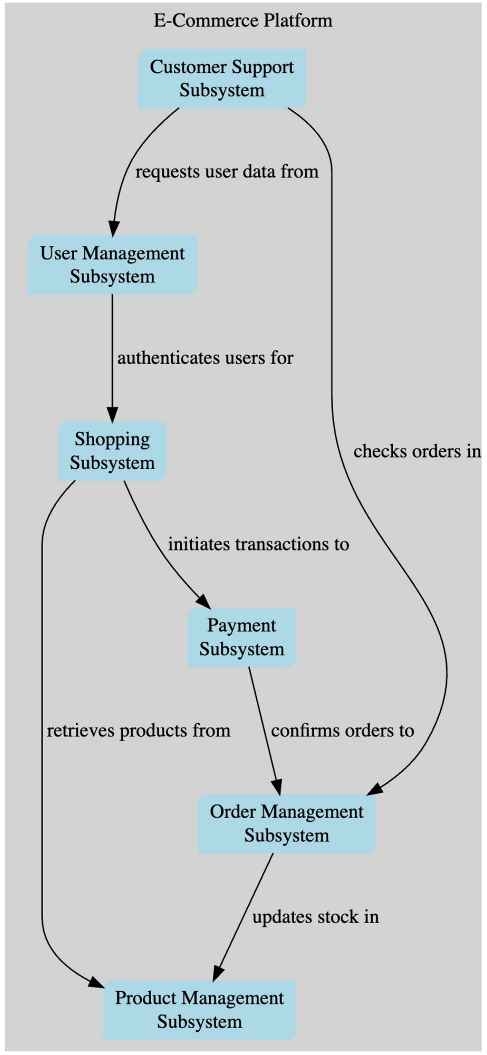
- Objective: Develop an E-Commerce Platform to enable online shopping, providing customers with a seamless shopping experience, from product browsing to checkout and payment.
- Requirements:
    1. User authentication and profile management.
    2. Product catalog with search and filter capabilities.
    3. Shopping cart and checkout process.
    4. Payment processing integration.
    5. Order management and tracking.
    6. Customer support and feedback system.

Example

E-Commerce Platform using a top-down approach

2=> Identify Major Subsystems (from requirements)

1. User Management Subsystem: Handles user registration, authentication, and profile management.
2. Product Management Subsystem: Manages product listings, categories, and inventory.
3. Shopping Subsystem: Manages the shopping cart, wishlist, and checkout processes.
4. Payment Subsystem: Integrates with payment gateways for processing transactions.
5. Order Management Subsystem: Manages order lifecycle, from creation to delivery.
6. Customer Support Subsystem: Provides tools for customer service and feedback management.

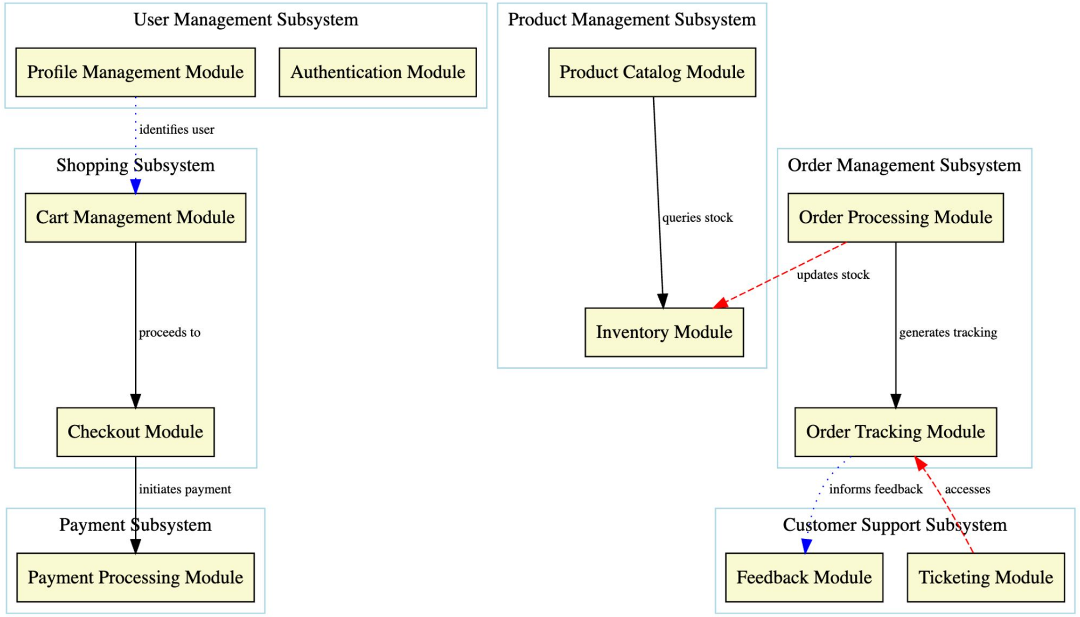


Example

E-Commerce Platform using a top-down approach

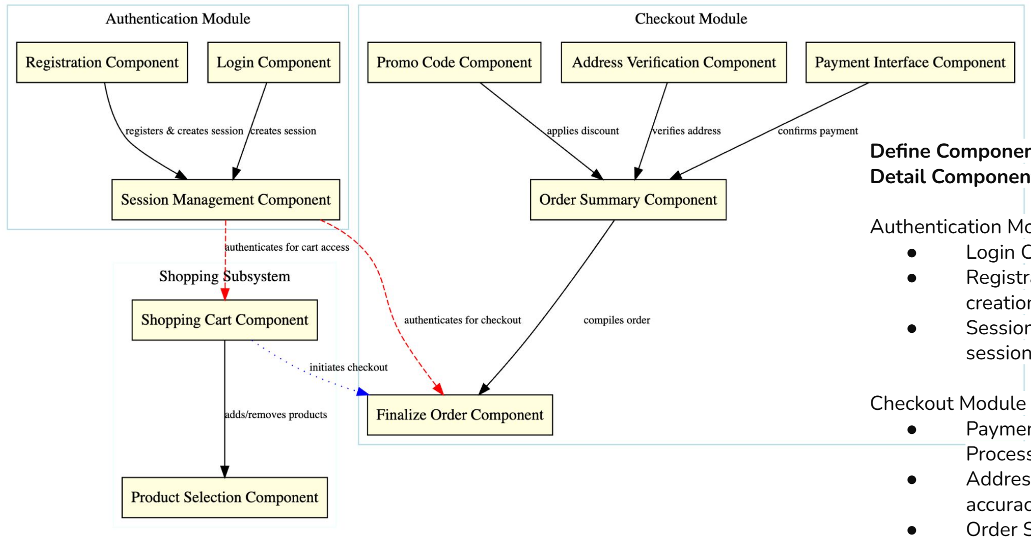
3=> Decompose Subsystems into Modules

- User Management Subsystem
  - Authentication Module, Profile Management
- Product Management Subsystem
  - Product Catalog Module, Inventory Managem
- Shopping Subsystem
  - Cart Management Module, Checkout Module
- Payment Subsystem
  - Payment Processing Module, Payment Secur
- Order Management Subsystem
  - Order Processing Module, Order Tracking Mo
- Customer Support Subsystem
  - Ticketing Module, Feedback Collection Modu



Example

E-Commerce Platform using a top-down approach



Define Components within Modules
Detail Components and their interactions

Authentication Module

- Login Component: Manages user login processes, authenticating credentials.
- Registration Component: Handles new user registration, including data validation and account creation..
- Session Management Component: Manages user sessions post-login or registration, providing session tokens for authenticated access to various subsystems.

Checkout Module Components

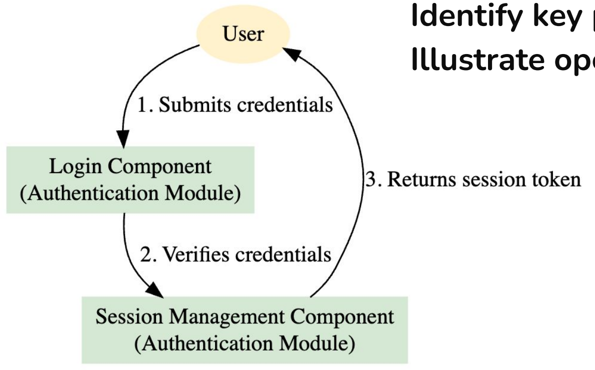
- Payment Interface Component: Initiates payment transactions and interacts with the Payment Processing Module to confirm payments.
- Address Verification Component: Verifies the shipping address provided by the customer to ensure accuracy and deliverability.
- Order Summary Component: Compiles order details, including items, quantities, prices, discounts applied through the Promo Code Component, and shipping information verified by the Address Verification Component. It acts as a central component that orchestrates the final steps before order finalization.
- Promo Code Component: Applies discount codes to the order, adjusting the total price accordingly.
- Finalize Order Component: Finalizes the order by compiling all order details into a final summary, triggering stock updates and order record creation. It represents the last step in the checkout process within the module.

Example

E-Commerce Platform using a top-down approach

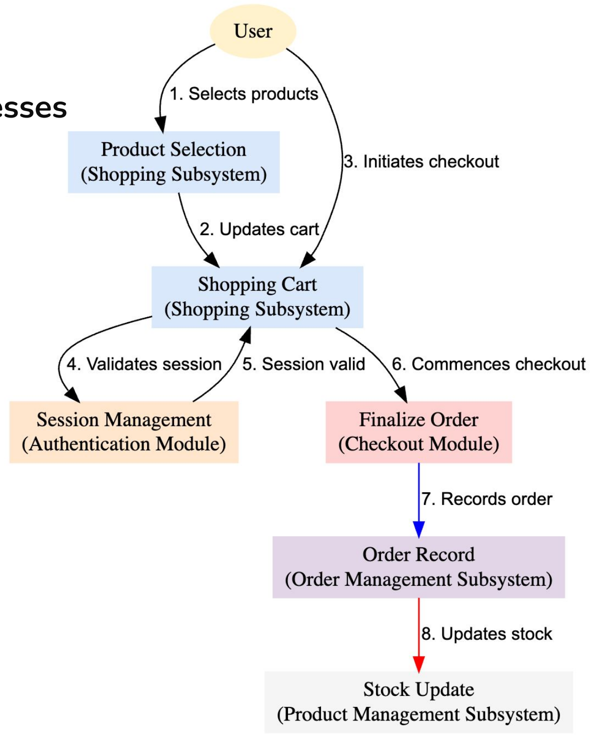
Identify key processes

Illustrate operations flow across components for key processes



User
1. Submits credentials
Login Component (Authentication Module)
2. Verifies credentials
Session Management Component (Authentication Module)
3. Returns session token

Process 1: User Login and Session Management



User
1. Selects products
Product Selection (Shopping Subsystem)
3. Initiates checkout
2. Updates cart
Shopping Cart (Shopping Subsystem)
4. Validates session
5. Session valid
6. Commences checkout
Session Management (Authentication Module)
Finalize Order (Checkout Module)
7. Records order
Order Record (Order Management Subsystem)
8. Updates stock
Stock Update (Product Management Subsystem)

Process 2: Complete Checkout to Stock Update

Architecture Design
Bottom-Up Approach

Starts with the development of individual components or subsystems based on available technologies or existing solutions, which are then integrated to form the whole system. It begins with the components that are needed for the solution, and then the design works upward into higher levels of abstraction.

Unlike the top-down approach, which begins with the high-level structure, there is no up-front architecture design with the bottom-up approach. The architecture emerges as more work is completed. The bottom-up approach does not require that the domain be well-understood, as the team only focuses on a small piece at a time.

Advantages:

- Greater level of simplicity. The team only has to focus on individual pieces and builds only what it needs for a particular iteration.
- Allows for early testing and validation of components, reducing risks.
- Encourages reuse of existing components, potentially speeding up development.
- Flexible in accommodating changes and additions to the system.

Use Cases:

- Suitable for projects where requirements are expected to evolve or are not fully defined, or when leveraging existing technologies and components is preferred.

Architecture Design

Bottom-Up Approach - Where to start?

- Identify Existing Components and Functionalities
- Create new Components and Functionalities
- Group Components into Logical Modules
- Integrate Modules into Subsystems
- Build the System from Subsystems

Top-Down or Bottom-Up

- Top-Down if at least 2 are true:
  - The project is large in size
  - The project is complex
  - Enterprise software is being designed for an organization
  - The team is large, or there are multiple teams that will be working on the project
  - The domain is well-understood
- Bottom-up if at least 2 are true:
  - The project is small in size
  - The project is not very complex
  - Enterprise software for an organization is not being designed
  - The team is small, or there is only a single team
  - The domain is not well-understood

Designing Software Architectures:

Design principles

Design principles and solutions: design concepts

- Do not reinvent the wheel!
- (Re)use proven design principles and solutions, referred to as **design concepts**
    - Allowing you to build architectures on a solid foundation of established principles.
    - This approach not only streamlines the design process but also ensures a more resilient and well-structured architecture.
- Design concepts:
    - **Software architecture patterns:** provide solutions for recurring architecture design problems. Patterns are discovered while observing what people were doing successfully to solve a particular problem, and then documenting those patterns so that they can be reused.
    - **Reference architectures:** are templates for an architecture that is best suited to a particular domain. Reference architectures are proven, in both technical as well as business contexts, as viable solutions for certain problems.
    - **Tactics:** proven techniques to influence quality attribute scenarios
    - **Externally developed software:** buy or build - pros & cons? OSS (Open source software) vs not OSS?

Design rationale

Design Rationale explains the reasoning and justification behind design decisions in software architecture.

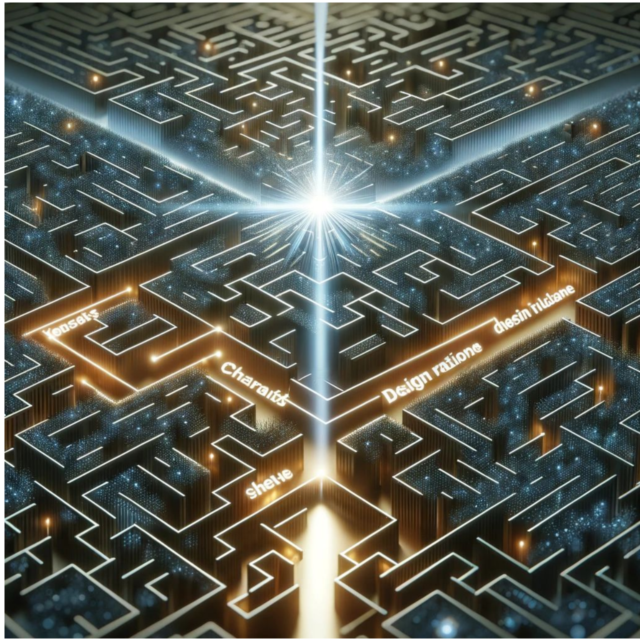
It highlights the value of documenting ***the whys*** and ***the why nots*** behind architectural choices, facilitating better understanding, maintenance, and future decision-making.

Key Components of Design Rationale:

- Decisions: What architectural choices were made.
- Reasons: Why these choices were preferred over alternatives.
- Implications: The impact of these decisions on the system's quality attributes.

Why Design Rationale Matters:

- Enhanced Communication: Clarifies architectural decisions to all stakeholders.
- Improved Decision Making: Provides a foundation for making informed future architectural decisions.
- Facilitates Evolution: Helps in understanding the impact of changing requirements on the existing architecture.



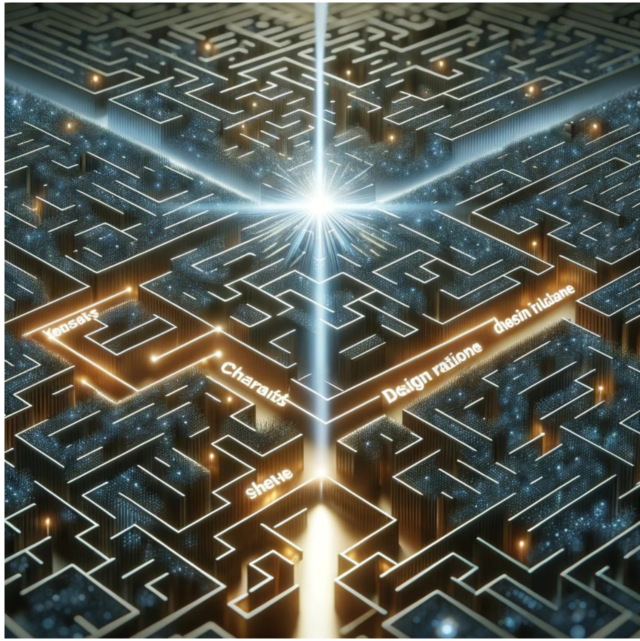
Design rationale

How to Document Design Rationale:

- Architectural Decision Records (ADRs): A concise textual format for capturing a single architectural decision.
- Design Diagrams with Annotations: Visual representations of architecture supplemented with rationale annotations.
- Decision Matrix: Comparing alternatives based on criteria like cost, feasibility, and impact on quality attributes.

Practical Application of Design Rationale:

- Case Study Overview: Brief introduction to a project or system.
- Decision Analysis: Example of a critical architectural decision made in the project, including the rationale behind it.
- Lessons Learned: Insights gained from applying design rationale in the project.
- Future maintenance and project evolution



https://learning.oreilly.com/library/view/software-architecture-the/9781492086888/ch01.html
https://learning.oreilly.com/library/view/software-architecture-the/9781492086888/app02.html#sec-adr-list

ADR Examples

ADR: Using a Sidecar for Operational Coupling
Context
Each service in our microservices architecture requires common and consistent operational behavior; leaving that responsibility to each team introduces inconsistencies and coordination issues.
Decision
We will use a sidecar component in conjunction with a service mesh to consolidate shared operational coupling.
The shared infrastructure team will own and maintain the sidecar for service teams; service teams act as their customers. The following services will be provided by the sidecar:

- *Monitoring*
- *Logging*
- *Service discovery*
- *Authentication*
- *Authorization*

Consequences
Teams should not add domain classes to the sidecar, which encourages inappropriate coupling.
Teams work with the shared infrastructure team to place shared, operational *libraries in the sidecar if enough teams require it.*

ADR: Migrate Sysops Squad Application to a Distributed Architecture
Context
The Sysops Squad is currently a monolithic problem ticket application that supports many different business functions related to problem tickets, including customer registration, problem ticket entry and processing, operations and analytical reporting, billing and payment processing, and various administrative maintenance functions. The current application has numerous issues involving scalability, availability, and maintainability.
Decision
We will migrate the existing monolithic Sysops Squad application to a distributed architecture. Moving to a distributed architecture will accomplish the following:

- *Make the core ticketing functionality more available for our external customers, therefore providing better fault tolerance*
- *Provide better scalability for customer growth and ticket creation, resolving the frequent application freeze-ups we've been experiencing*
- *Separate the reporting functionality and reporting load on the database, resolving the frequent application freeze-ups we've been experiencing*
- *Allow teams to implement new features and fix bugs much faster than with the current monolithic application, therefore providing for better overall agility*
- *Reduce the amount of bugs introduced into the system when changes occur, therefore providing better testability*
- *Allow us to deploy new features and bug fixes at a much faster rate (weekly or even daily), therefore providing better deployability*

Consequences
The migration effort will cause delays for new features being introduced since most of the developers will be needed for the architecture migration.
The migration effort will incur additional cost (cost estimates to be determined).
Until the existing deployment pipeline is modified, release engineers will have to manage the release and monitoring of multiple deployment units.
The migration effort will require us to break apart the monolithic database.

https://learning.oreilly.com/library/view/software-architecture-the/9781492086888/app02.html#sec-adr-list

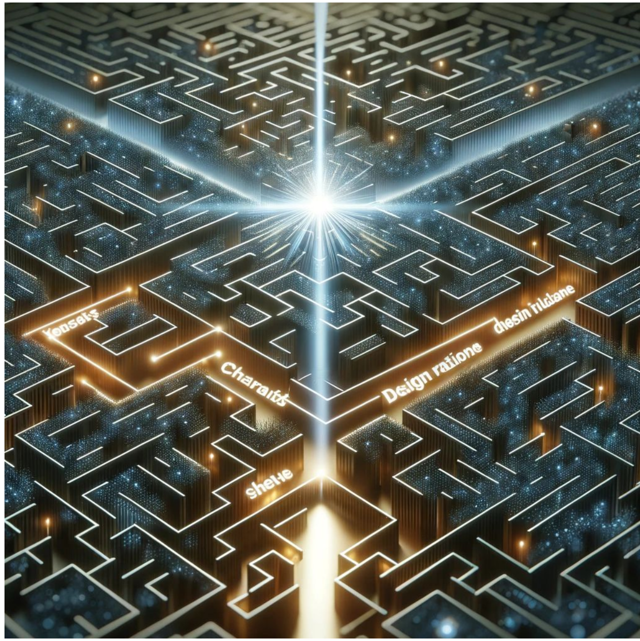
Design rationale

Challenges and Solutions

- Time and Effort: Balancing the depth of documentation with project timelines.
- Evolving Decisions: Keeping the rationale documentation updated as decisions evolve.
- Engagement: Ensuring all stakeholders contribute to and understand the design rationale.

Best Practices for Effective Design Rationale

- Start Early: Incorporate rationale documentation from the beginning of the design process.
- Keep it Accessible: Ensure the documentation is easily accessible and understandable to all stakeholders.
- Regular Reviews: Periodically review and update the rationale to reflect any changes in the project scope or objectives.



ADR Discussion

Sysops Squad Saga: Customer Registration Granularity