Research Report

Jordy Walraven

Table of Contents

[Introduction 1](#_Toc162276650)

[Problem 1](#_Toc162276651)

[Research questions 2](#_Toc162276652)

[What architectural styles best suit the requirements of my system? 2](#_Toc162276653)

[Which architecture styles exist and what are some pros and cons? 2](#_Toc162276654)

[Layered Pattern 3](#_Toc162276655)

[Client-Server pattern 3](#_Toc162276656)

[Event-Driver pattern 3](#_Toc162276657)

[Microkernel pattern 3](#_Toc162276658)

[Microservices 4](#_Toc162276659)

[Broker pattern 4](#_Toc162276660)

[Event-Bus pattern 4](#_Toc162276661)

[Pipe-filter pattern 5](#_Toc162276662)

[Blackboard pattern 5](#_Toc162276663)

[Bibliografie 5](#_Toc162276664)

# Introduction

# Problem

The sheer number of design and architecture patterns available is overwhelming. Each seems to offer a perfect solution, but selecting the right one for a system designed to handle a million users feels like navigating a maze. The core challenge lies in striking a delicate balance my non-functional requirements..

Through research, I aim to identify the ideal design patterns and architectural approaches that can overcome these challenges. This will help me build a system that is not only scalable but also remains understandable and manageable for future development and maintenance.

# Research questions

**Main Research:** How to design and deploy a web application that enables user-submitted system performance evaluation for game compatibility, ensuring scalability, security, reliability maintainability, and high performance?

**Sub Questions:**

* What architectural styles best suit the requirements of my system?
* How do different services or components communicate with each other?
* How can I ensure that the application is secure?
* How to deploy a scalable application?
* How do I make sure the application is reliable deployed without breaking issues?
* How do I make sure the application is reliable running in the cloud?

# What architectural styles best suit the requirements of my system?

There are a lot of different architectural styles you need to consider when creating your own web application. Especially when looking at requirements like scalability, security, reliability, maintainability and performance. First we will have a look at the different architecture styles that exist, and we will compare these with each other to see which best suit the requirements of the system.

## Which architecture styles exist and what are some pros and cons?

There exists an abundance of architectural styles, making it impractical to delve deeply into each one. Therefore, I will focus on the 15 most widely utilized patterns, as they possess sufficient support and documentation. Other styles, lacking in such resources, will not be explored in this context. The primary source I will be using is an article by (Ritvik Gupta, 2023).

The patterns I will be researching are:

* Layered pattern
* Client-server pattern
* Event-Driven pattern
* Microkernel pattern
* Microservice pattern
* Broker pattern
* Event-bus pattern
* Pipe filter patten
* Blackboard pattern
* Component-based pattern
* Service-oriented architecture
* Monolithic architecture
* Space-based architecture
* Peer-to-peer architecture
* Hybrid Architecture

### Layered Pattern

The layered pattern (Kaseb, 2022) is a pattern where each responsibility of the application is separated in layers. For example an API might consist of a presentation layer (controllers), Business layer and data access layer. This modular approach promotes segregation of concerns and facilitates easy maintenance and scalability in a complex system.

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| Promotes separation of concerns | Overhead |
| Encourages modularity |  |
| Supports parallel development |  |

### Client-Server pattern

The Client-Server pattern (Gayantha, 2020) is a fundamental architectural design that divides an application into two distinct components: the client and the server. This separation enables efficient communication and enhances scalability and maintainability. In this pattern, the client is typically a user interface or application that interacts directly with the end user, while the server is responsible for managing data processing, storage, and business logic.

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| Scalability | Data manipulation |
| Security |  |
| Reliability |  |

### Event-Driver pattern

The Event-Driven pattern enables systems to respond dynamically to events, such as user interactions or data updates. Events are emitted by producers and consumed by handlers, allowing for asynchronous, loosely-coupled communication. This pattern fosters flexibility, scalability, and real-time responsiveness in software architectures.

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| Asynchronous | Complexity |
| Scalable | Debugging |
| Modular | Overhead |
| Scalable |  |

### Microkernel pattern

The microkernel architecture pattern (Heusser, 2020) is a software design approach that sits between monolithic and microservices architectures. It is characterized by a central core system that contains only the essential components necessary to run the application, and additional functionalities are implemented as plugins. This architecture allows for a high degree of modularity and flexibility, as new features or modifications can be added or changed without altering the core system. The core system and plugins communicate through interprocess communication mechanisms provided by the microkernel, ensuring that they remain isolated from each other.

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| Modularity | Performance overhead |
| Flexibility | Complexity |
| Stability | Dependent on kernel |

### Microservices

Microservices architecture (IBM, n.d.) is a cloud-native approach where a single application is composed of many loosely coupled and independently deployable smaller components or services. These services are organized by business capability, often referred to as a bounded context. This architecture allows for easier code updates, as new features or functionality can be added without affecting the entire application. Teams can use different stacks and programming languages for different components, and components can be scaled independently, reducing waste and cost associated with scaling entire applications.

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| Fault isolation | Complexity |
| Independent scaling | Resource intensive |
| Smaller and faster deployment | Data inconsistency |
| Scalability |  |

### Broker pattern

The Broker Pattern (Wikipedia, 2023) is an architectural pattern used to structure distributed software systems with decoupled components that interact by remote procedure calls. It involves an intermediary software entity, known as a broker, which is responsible for coordinating communication between clients and servers. The broker acts as a middleman, receiving messages from one component and forwarding them to the appropriate recipient. This pattern allows components to remain decoupled and focused on their own responsibilities, while still being able to communicate and collaborate with other components in the system.

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| Decoupling | Single point of failure |
| Scalability | Complexity |
| Changeability | Performance overhead |

### Event-Bus pattern

The Event Bus pattern is a design pattern that facilitates communication between components in a distributed system by using an event bus as a central hub. This pattern is particularly useful in large-scale applications where components need to interact without being tightly coupled, adhering to principles of loose coupling and separation of concerns. The event bus acts as a pipeline, where components (referred to as subscribers) can register to receive specific types of events. When an event occurs, it is dispatched to the event bus, which then forwards it to all registered subscribers that are interested in that type of event. This mechanism allows components to communicate asynchronously, enabling them to operate independently and react to events as they occur.

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| Loose coupling | Complexity |
| Asynchronous | Debugging |
| Scalability | Performance overhead |

### Pipe-filter pattern

The Pipe and Filter pattern is an architectural pattern that structures a system as a sequence of processing elements, where each element is a filter that performs a specific operation on the data. Data flows through the system in a pipeline, where each filter takes input from the previous filter and passes its output to the next filter in the sequence. This pattern is particularly useful for data processing and transformation tasks, where operations can be performed in a series of steps.

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| Modularity | Complexity |
| Flexibility | Latency |

### Blackboard pattern

The Blackboard pattern is a way to solve complex problems by breaking them down into smaller, manageable parts. It involves a central "blackboard" where different parts of a program, called "agents," can share information and work together to find a solution.

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| Modularity | Complexity |
| Flexibility | Coordination |
| Collaboration | Security |

# Bibliografie

Gayantha, K. (2020, Januari 2020). *Client-Server architectural pattern*. Opgehaald van Medium: https://medium.com/@96kavindugayantha/software-architectural-patterns-17b05bfa3aef

Heusser, M. (2020, March 10). *What is a microkernel architecture, and is it right for you?* Opgehaald van TechTarget: https://www.techtarget.com/searchapparchitecture/tip/What-is-a-microkernel-architecture-and-is-it-right-for-you

Kaseb, K. (2022, March 27). *The Layered Architecture Pattern in Software Architecture*. Opgehaald van Medium: https://medium.com/kayvan-kaseb/the-layered-architecture-pattern-in-software-architecture-324922d381ad

Ritvik Gupta, A. S. (2023, October 15). *software-architecture-patterns-types*. Opgehaald van Turing: https://www.turing.com/blog/software-architecture-patterns-types/