Software Architecture Group Assignment

Group Name: Naber

22050951010 - Yahya Çakıcı
21050911006 - Hatice Rüveyda Akça
21050911023 - İlayda Akınet

Contents

1. Architectural Patterns Comparison and Evaluation

1.1. Understanding Software Architecture Patterns

In the evolving domain of software engineering, the development of robust and scalable systems necessitates the application of well-defined design methodologies, often encapsulated within software architecture patterns. These patterns provide high-level frameworks for addressing recurrent design challenges by presenting reusable and proven solutions. Each pattern introduces a unique set of principles that govern the organization, functionality, and adaptability of software systems.

Software architecture patterns play a critical role in defining the overarching structure of a system, encompassing its components, their interactions, and its operational dynamics. By addressing fundamental concerns such as scalability, performance optimization, and maintainability, these patterns enable developers to mitigate common challenges such as system inefficiencies and operational risks. Furthermore, they enhance development efficiency, foster interdisciplinary collaboration, and ensure adaptability to evolving project requirements. For stakeholders, the implementation of architectural patterns contributes to the creation of high-quality, cost-effective software solutions delivered within reduced timeframes, solidifying their importance in contemporary software engineering practices.

1.2. Benefits of Architecture Patterns

Software architecture patterns are essential for efficient software development, enhancing processes and outcomes. They ensure scalability, agility, and platform independence while improving data integrity through secure, authorized access. User-friendly interfaces simplify operations, and their distributed structure eases maintenance and updates. These patterns also boost productivity, ensure consistent quality, and enable early issue detection.

Key Benefits:

- **Data Integrity:** Secure, authorized access promotes safe data sharing.
- **Ease of Use:** Intuitive interfaces replace complex command-line operations.
- **Independence:** Supports platform-agnostic development.
- **Maintainability:** Distributed structure simplifies updates.
- **Productivity:** Streamlines processes and sustains momentum.
- **Quality Assurance:** Ensures reusable models and consistent quality.

1.3. Key Architectural Patterns and Their Evaluation

1.3.1. Layers Pattern

The Layers architectural pattern is designed to structure applications by organizing them into a hierarchy of layers, each representing a distinct level of abstraction. In this model, each layer provides services to the layer above it while utilizing services from the layer below. This approach ensures that concerns at different abstraction levels are handled independently, creating a modular and maintainable architecture.

Key Benefits of the Layers Pattern:

- **Reusability**: Lower layers can be reused by multiple higher layers without modification. For instance, the TCP layer can serve various applications such as FTP or Telnet.
- **Standardization**: Well-defined levels of abstraction simplify standardization, making tasks and interfaces easier to develop and manage.
- **Local Dependencies**: Changes within a layer do not impact other layers as long as interfaces remain consistent, allowing independent testing and development of layers.
- **Replaceability**: Layers can be replaced or updated with different implementations without disrupting the overall system architecture.

Challenges and Solutions in the Layers Pattern:

1. **Performance Overhead**: Repeated data transformations between layers can slow performance. Optimizing data handling minimizes this impact.
2. **Dependency Management**: Poorly defined interfaces can create bottlenecks. Clear, consistent interfaces ensure smooth communication and independent development.
3. **Flexibility vs. Maintainability**: Relaxing layer boundaries improves efficiency but reduces maintainability. Adhering to strict boundaries while evaluating exceptions balances both.

1.3.2. Client-Server Pattern

The Client-Server architectural pattern structures a system where a server component provides services to multiple client components. Clients request services from the server, which remains constantly active to listen and respond. This pattern operates across process and machine boundaries, necessitating inter-process communication mechanisms. Effectively, it can be seen as an extension of the Layered pattern, with clients functioning as the higher layer and the server as the lower layer.

Key Features and Examples of Client-Server Pattern:

- **Cross-Boundary Operation**: Clients and servers may operate on separate machines, enabling distributed architectures.
- **Examples**:
    - Remote database access, where client applications interact with a central database server.
    - Remote file systems, allowing transparent access to local and remote files.
    - Web-based applications, where browsers request data from web servers.

Challenges and Solutions in the Client-Server Pattern:

1. **Overhead in Communication**: Inter-process communication and data transformation add latency. Efficient protocols and optimized data marshaling can mitigate this.
2. **Transparency**: Distributed systems require location and platform transparency, ensuring clients do not need to know specific server details. Intermediate layers like caching, load balancing, or security mechanisms enhance this transparency.
3. **Thread Management**: Servers often handle requests in separate threads, requiring robust concurrency management for scalability.

1.3.3. Master-Slave Pattern

The Master-Slave architectural pattern is designed to support fault tolerance and parallel computation by dividing tasks between a central master component and multiple slave components. The master delegates work to identical slaves, gathers their results, and computes the final outcome. This approach is highly suitable for systems requiring process control, embedded systems, large-scale parallel computations, and fault tolerance.

Key Features and Examples of the Master-Slave Pattern:

- **Fault Tolerance**: The master applies strategies to ensure system reliability, such as selecting the first result or the majority of results from slaves. While slave failures can be managed, master failure leads to system-wide failure.
- **Parallel Computing**: Tasks are divided into independent subtasks processed concurrently by slaves, as seen in matrix computations.
- **Computational Accuracy**: Delegating tasks to multiple implementations ensures accuracy by applying strategies like averaging or majority voting.

Challenges and Solutions in the Master-Slave Pattern:

1. **Latency**: Communication delays between master and slaves can hinder performance, especially in real-time systems. Optimizing communication processes mitigates this.
2. **Applicability**: The pattern is limited to problems that can be decomposed into parallel tasks. Thorough problem analysis ensures appropriate use.
3. **Fault Handling**: Slave failures can be managed with timeouts and redundancy, but master failures require additional mechanisms like backups or failover systems.

1.3.4. Pipe-Filter Pattern

The Pipe-Filter architectural pattern structures systems that process data streams through a series of sequential steps. Each step is encapsulated in a filter component, which processes incoming data and passes the output to the next step via pipes. This design enables data buffering and synchronization between components, supporting efficient and modular data processing.

Key Features and Examples of the Pipe-Filter Pattern:

- **Data Flow**: Filters consume and produce data incrementally, allowing for continuous processing without waiting for all input to be consumed.
- **Examples**:
  - Unix shell commands (e.g., **cat file | grep xyz | sort | uniq > out**), where each command acts as a filter in a data processing pipeline.
  - Compilers, where data flows through sequential filters like lexical analysis, parsing, semantic analysis, and code generation.

Challenges and Solutions in the Pipe-Filter Pattern:

1. **Performance Overhead**: Standardized interfaces may cause inefficiencies due to data transformation. Optimizing input/output formats minimizes this issue.
2. **Deadlocks**: Filters requiring all data before producing output can cause deadlocks if buffer sizes are insufficient. Proper buffer management prevents this.
3. **Interactive Limitations**: The pattern is not well-suited for interactive applications due to its linear data flow structure. Alternative designs may be required for interactivity.

1.3.5. Broker Pattern

The Broker architectural pattern organizes distributed systems with decoupled components that communicate through remote service invocations. A broker component coordinates communication by forwarding client requests to the appropriate server and transmitting results or exceptions. This decoupling eliminates the need for components to know each other's location or implementation details, simplifying system design.

Key Features and Examples of the Broker Pattern:

- **Dynamic Communication**: Clients request services from the broker, which redirects the request to an appropriate server based on a registry of published services.
- **Examples**:
  - **Web Services**: Brokers like UDDI registries manage service discovery and invocation, using protocols like WSDL and SOAP.
  - **CORBA**: Facilitates communication between heterogeneous object-oriented systems.

Challenges and Solutions in the Broker Pattern:

1. **Standardization**: Service descriptions require a uniform standard (e.g., IDL or binary formats) to enable seamless interaction. Clear standards ensure compatibility.
2. **Complexity in Cooperation**: When multiple brokers interact, bridges are needed to translate protocols. Well-designed bridges ensure smooth cross-broker communication.
3. **Dynamic Changes**: While the pattern supports adding, deleting, and relocating objects dynamically, this flexibility demands careful management of the registry and dependencies.

1.3.6. Peer-to-Peer Pattern

The Peer-to-Peer (P2P) architectural pattern is a symmetric variant of the Client-Server model, where nodes (peers) can act as both clients and servers. Peers dynamically switch roles, providing or requesting services as needed. Communication between peers can be implicit (e.g., through a stream) or explicit, and multi-threading is typically used to handle simultaneous interactions.

Key Features and Examples of the Peer-to-Peer Pattern:

- **Dynamic Roles**: Peers can adapt to act as either clients or servers based on system needs.

- **Examples**:
  - **DNS**: Distributed domain name resolution.
  - **File Sharing**: Platforms like BitTorrent and Gnutella.
  - **Distributed Search**: Search engines like Sciencenet.
  - **Collaborative Tools**: Multi-user applications, such as shared drawing boards.

Challenges and Solutions in the Peer-to-Peer Pattern:

1. **Quality of Service**: Voluntary cooperation among peers can lead to inconsistent service quality. Incentivizing reliable participation helps mitigate this.
2. **Security Risks**: Decentralization makes security guarantees challenging. Strong encryption and authentication mechanisms can address vulnerabilities.
3. **Performance Variability**: Performance depends on the number of active peers, improving with more participants but potentially lagging with fewer. Efficient peer discovery and resource sharing strategies alleviate this.

1.3.7. Event-Bus Pattern

The Event-Bus architectural pattern is designed to handle events in a decoupled manner. Event sources publish messages to specific channels on an event bus, while listeners subscribe to these channels to receive notifications. Message generation and delivery are asynchronous, allowing event sources to continue processing without waiting for listeners to react.

Key Features and Examples of the Event-Bus Pattern:

- **Asynchronous Communication**: Event sources generate messages without blocking for listener acknowledgments.
- **Flexible Channels**: Channels can be explicit, with direct subscriptions to publishers, or implicit, with subscriptions to event types.
- **Examples**:
  - **Software Development Tools**: Components like build tools, unit test runners, and debugging tools communicate via an event bus.
  - **Real-Time Systems**: Middleware like OpenSplice for data distribution.
  - **Trading Systems**: For monitoring and handling financial transactions.

Challenges and Solutions in the Event-Bus Pattern:

1. **Scalability**: As all messages pass through the event bus, increased traffic can create bottlenecks. Optimizing message routing and increasing bus capacity can mitigate this.
2. **Delivery Assumptions**: Ensuring message ordering, distribution, and timeliness can be complex. Developers must design listeners to handle these uncertainties.
3. **Overhead**: Adding transformations or scripting services to the bus may introduce performance overhead. These should be implemented judiciously.

1.3.8. Model-View-Controller Pattern

The Model-View-Controller (MVC) architectural pattern organizes interactive applications into three components:

- **Model**: Manages the core functionality and data.
- **View**: Displays information to the user and supports multiple visual representations.
- **Controller**: Handles user input and updates the model accordingly.

This decoupled structure makes MVC particularly suitable for applications with multiple graphical user interfaces (GUIs), as the model remains independent of the number and type of GUIs. Consistency between the model and view is maintained through notifications, often leveraging the Observer pattern.

Key Features and Examples of the MVC Pattern:

- **Dynamic Interface Management**: Allows easy modifications to the application's look and feel.
- **Examples**:
  - **Web Presentation**: Used in web application architectures.
  - **Document View Architecture**: Seen in systems like Windows applications, enabling multiple views (e.g., print layout, web layout).

Challenges and Solutions in the MVC Pattern:

1. **Complexity**: Separation of model, view, and controller adds design complexity. Combining simpler elements outside the pattern can reduce overhead.
2. **Unnecessary Updates**: User actions may trigger redundant updates across components. Optimizing notifications can address this.
3. **Tight Coupling**: Views and controllers are often closely related to the model, making changes ripple across components. Clear interface design mitigates this issue.

1.3.9. Blackboard Pattern

The Blackboard architectural pattern addresses problems without deterministic solution strategies. It involves multiple specialized subsystems collaborating through a shared data store, known as the blackboard. Subsystems contribute knowledge by adding data to the blackboard and retrieve relevant data through pattern matching. This approach supports iterative and incremental problem-solving, yielding partial or approximate solutions.

Key Features and Examples of the Blackboard Pattern:

- **Shared Data Store**: All components access and contribute to a common repository.
- **Examples**:
    - **Speech Recognition**: Analyzing audio signals to identify spoken words.
    - **Submarine Detection**: Combining sensor data for threat detection.
    - **3D Molecular Structure Inference**: Interpreting experimental data to deduce molecular structures.
    - **Tuple Space Systems**: Such as JavaSpaces, enabling distributed data sharing.

Challenges and Solutions in the Blackboard Pattern:

1. **Data Structure Modification**: Changes to the blackboard's structure impact all connected components. Adopting flexible and well-defined structures mitigates disruption.
2. **Process Coordination**: Components must agree on the shared data's structure. Establishing synchronization and access control mechanisms ensures smooth collaboration.
3. **Complexity in Scaling**: Adding new components is straightforward, but managing dependencies in large systems requires robust design.

1.3.10. Interpreter Pattern

The Interpreter architectural pattern is designed for systems that need to process and execute programs written in a dedicated language. It facilitates easy replacement and modification of the interpreted program, making it suitable for applications requiring frequent updates or customization.

Key Features and Examples of the Interpreter Pattern:

- **Dynamic Interpretation**: Executes programs dynamically without the need for prior compilation.
- **Examples**:
  - **Rule-Based Systems**: Expert systems for decision-making.
  - **Web Scripting Languages**: Client-side JavaScript and server-side PHP.
  - **Document Languages**: Postscript for graphical and text content.

Challenges and Solutions in the Interpreter Pattern:

1. **Performance Issues**: Interpreted languages are typically slower than compiled ones. Optimizing the interpreter or compiling performance-critical sections can address this.
2. **Testing and Stability**: The ease of replacing programs can lead to insufficient testing, compromising stability and security. Enforcing rigorous testing and validation mitigates risks.

2. Cloud Native Patterns

Cloud-native patterns are critical in designing modern software systems optimized for scalability, resilience, and maintainability in cloud environments. These patterns leverage cloud-native capabilities such as elasticity, distributed systems, and container orchestration to create robust architectures. Below is a detailed comparison of ten cloud-native architectural patterns, including their features, examples, benefits, and challenges.

2.1. Microservices Pattern

The Microservices pattern breaks applications into a collection of small, independently deployable services that perform specific business functions.

- **Key Features**:
  - Services communicate via lightweight protocols (e.g., REST, gRPC).
  - Independent development, deployment, and scaling of services.
- **Examples**:
  - E-commerce platforms with separate services for inventory, payments, and user management.
- **Benefits**:
  - Scalability: Each service can scale independently.
  - Fault Isolation: Failure in one service does not affect the entire system.
- **Challenges**:
  - Increased complexity in communication and data consistency.
  - Requires robust monitoring and orchestration.

2.2. Service Mesh Pattern

Service Mesh abstracts communication between microservices, handling discovery, routing, and security.

- **Key Features**:
  - Layered control over service-to-service communication.
  - Enhanced observability with built-in monitoring and tracing.
- **Examples**:
  - Istio or Linkerd used in Kubernetes environments.
- **Benefits**:
  - Simplifies service communication and management.
  - Improves security through mutual TLS and fine-grained policies.
- **Challenges**:
  - Overhead in managing the mesh infrastructure.
  - Increased latency due to additional communication layers.

2.3. API Gateway Pattern

The API Gateway pattern centralizes external client communication with backend microservices through a single entry point.

- **Key Features**:
  - Handles requests, authentication, caching, and load balancing.
  - Aggregates responses from multiple services.
- **Examples**:
  - Netflix’s Zuul, Amazon API Gateway.
- **Benefits**:
  - Simplifies client interaction with microservices.
  - Improves performance through caching and rate limiting.
- **Challenges**:
  - Single point of failure if not redundantly designed.
  - Complex configuration for dynamic microservices.

2.4. Serverless Pattern

The Serverless pattern offloads infrastructure management, allowing developers to focus on writing functions executed on-demand.

- **Key Features**:
  - Event-driven execution.
  - Pay-per-use billing model.
- **Examples**:
  - AWS Lambda, Google Cloud Functions.
- **Benefits**:
  - Cost-efficient: Resources are allocated only when needed.
  - Simplifies scalability and resource management.
- **Challenges**:
  - Limited execution duration and resource constraints.
  - Vendor lock-in due to proprietary implementations.

2.5. Sidecar Pattern

The Sidecar pattern runs helper components alongside the main application container to handle auxiliary tasks.

- **Key Features**:
    - Containers share the same pod but operate independently.
    - Commonly used for logging, monitoring, or networking.
- **Examples**:
    - Envoy proxy as a sidecar for service communication.

- **Benefits**:
    - Simplifies application design by offloading non-core functions.
    - Promotes reusability of sidecar components.
- **Challenges**:
    - Additional resource consumption per node.
    - Complexity in managing multiple sidecars.

2.6. Strangler Pattern

The Strangler pattern incrementally replaces legacy systems by building new features in a modern architecture while phasing out the old.

- **Key Features**:
    - Dual-running systems during migration.
    - Gradual replacement of legacy components.
- **Examples**:
    - Migrating a monolithic application to microservices.
- **Benefits**:
    - Reduces migration risk by allowing incremental changes.
    - Minimizes downtime and disruption.
- **Challenges**:
    - Requires careful integration of new and old systems.
    - Extended transition period increases maintenance complexity.

2.7. Circuit Breaker Pattern

The Circuit Breaker pattern prevents cascading failures by monitoring and controlling requests to failing services.

- **Key Features**:
    - Tracks service health and triggers fallbacks or timeouts.
    - Supports retries after cooldown periods.
- **Examples**:
    - Netflix Hystrix for fault tolerance.

- **Benefits**:
  - Improves system resilience and stability.
  - Protects against service overload.
- **Challenges**:
  - Requires fine-tuning to avoid unnecessary trip events.
  - Adds latency during health-check evaluations.

2.8. Blue-Green Deployment Pattern

The Blue-Green Deployment pattern minimizes downtime during releases by maintaining two environments: one active (blue) and one staged (green).

- **Key Features**:
  - Seamless switching between environments.
  - Allows rollback to the previous version if needed.
- **Examples**:
  - Deployment pipelines in Kubernetes clusters.
- **Benefits**:
  - Ensures zero downtime during deployments.
  - Provides a safe rollback mechanism.
- **Challenges**:
  - Resource-intensive due to duplicate environments.
  - Requires automated deployment orchestration.

2.9. Event Sourcing Pattern

The Event Sourcing pattern stores system state changes as immutable events, reconstructing current states by replaying these events.

- **Key Features**:
  - Log-based state reconstruction.
  - Maintains an audit trail for all changes.
- **Examples**:
  - Financial systems tracking transactions.
- **Benefits**:
  - Ensures consistency and traceability.
  - Supports debugging and replaying events for analysis.
- **Challenges**:
  - High storage requirements for event logs.
  - Complexity in event versioning and schema evolution.

2.10. Observability Pattern

The Observability pattern integrates monitoring, logging, and tracing to provide real-time insights into system behavior.

- **Key Features**:
    - Centralized telemetry data collection.
    - End-to-end tracing of requests.
- **Examples**:
    - Prometheus for metrics, Grafana for visualization.
- **Benefits**:
    - Enhances system reliability and debugging.
    - Provides actionable insights for optimization.
- **Challenges**:
    - High overhead in data collection and processing.
    - Requires skilled resources for configuration and maintenance.

3. Enterprise Patterns

Enterprise patterns are foundational for designing scalable, robust, and maintainable enterprise applications. They address recurring problems in business environments, focusing on system architecture, data management, and application integration. Below is a detailed comparison of ten enterprise architectural patterns, highlighting their features, examples, benefits, and challenges.

3.1. Layered Architecture Pattern

The Layered Architecture pattern organizes applications into horizontal layers, each with a specific responsibility, such as presentation, business logic, and data access.

- **Key Features**:
  - Separation of concerns.
  - Clear interaction rules between layers.
- **Examples**:
  - Enterprise applications with user interfaces, service layers, and databases.
- **Benefits**:
  - Easy to maintain and scale.
  - Promotes reusability and modularity.
- **Challenges**:
  - May introduce latency due to inter-layer communication.
  - Can lead to tightly coupled layers if not designed properly.

3.2. Microservices Architecture Pattern

Microservices divide applications into small, independent services that communicate over APIs.

- **Key Features**:
  - Independent deployment and scaling of services.
  - Decentralized data management.
- **Examples**:
  - Large-scale e-commerce systems with separate services for inventory, payment, and user accounts.
- **Benefits**:
  - Fault isolation and scalability.
  - Facilitates agile development.
- **Challenges**:
  - Complex communication and data consistency.
  - Requires robust monitoring and orchestration.

3.3. Event-Driven Architecture Pattern

Event-driven systems rely on the generation, detection, and processing of events to drive business logic.

- **Key Features**:
  - Asynchronous event handling.
  - Real-time responsiveness.
- **Examples**:
  - Banking systems for transaction alerts.
  - Inventory systems triggering stock replenishment events.
- **Benefits**:
  - High responsiveness and scalability.
  - Decouples event producers and consumers.
- **Challenges**:
  - Complex debugging and monitoring.
  - Potential message loss without proper handling.

3.4. Service-Oriented Architecture (SOA) Pattern

SOA structures systems around loosely coupled services, each offering a specific business capability.

- **Key Features**:
  - Service reuse and composability.
  - Standards-based communication protocols.
- **Examples**:
  - ERP systems with modular services for HR, finance, and supply chain.
- **Benefits**:
  - Encourages interoperability across platforms.
  - Simplifies integration in heterogeneous environments.
- **Challenges**:
  - High initial setup complexity.
  - Governance and standardization issues.

3.5. Repository Pattern

The Repository pattern centralizes data access logic, abstracting the underlying data sources.

- **Key Features**:
  - Provides a consistent data access API.
  - Simplifies unit testing by abstracting the data layer.
- **Examples**:
  - Data repositories in enterprise applications for managing customers, orders, or inventory.
- **Benefits**:
  - Reduces code duplication.
  - Improves maintainability and testability.
- **Challenges**:
  - May introduce performance bottlenecks if not optimized.
  - Complexity in managing large data repositories.

3.6. Saga Pattern

The Saga pattern handles distributed transactions by breaking them into a series of coordinated, smaller transactions.

- **Key Features**:
  - Compensating actions for rollback.
  - Decentralized transaction management.
- **Examples**:
  - E-commerce order processing across payment, inventory, and shipping services.
- **Benefits**:
  - Ensures consistency in distributed systems.
  - Avoids locking resources for long periods.
- **Challenges**:
  - Requires careful design of compensating actions.
  - Debugging is complex due to the distributed nature.

3.7. CQRS (Command Query Responsibility Segregation) Pattern

CQRS separates read and write operations into different models, optimizing each for its specific use case.

- **Key Features**:
  - Distinct data models for queries and commands.
  - Supports event sourcing for state reconstruction.
- **Examples**:
  - Financial systems where reads and writes have vastly different performance needs.
- **Benefits**:
  - Improves scalability and performance.
  - Simplifies complex domain logic.
- **Challenges**:
  - Increases system complexity with multiple models.
  - Eventual consistency can introduce challenges.

3.8. Strangler Fig Pattern

This pattern replaces legacy systems incrementally by building new functionalities alongside the old system and phasing out the legacy components.

- **Key Features**:
  - Gradual migration.
  - Dual-running systems during transition.
- **Examples**:
  - Modernizing monolithic systems into microservices.
- **Benefits**:
  - Reduces migration risk.
  - Minimizes downtime and disruption.
- **Challenges**:
  - Extended maintenance of both systems.
  - Integration between new and old components.

3.9. Circuit Breaker Pattern

The Circuit Breaker pattern prevents cascading failures in distributed systems by monitoring requests to potentially failing services.

- **Key Features**:
  - Tracks service health and triggers fallback mechanisms.
  - Protects against system overload.
- **Examples**:
  - APIs with intermittent connectivity.
- **Benefits**:
  - Increases system stability and resilience.
  - Prevents overload on failing components.
- **Challenges**:
  - Proper configuration is essential to avoid false positives.
  - Adds latency for health checks and retries.

3.10. Proxy Pattern

The Proxy pattern provides a surrogate or placeholder for another object, controlling access to it.

- **Key Features**:
  - Mediates access to resources.
  - Supports lazy initialization and caching.
- **Examples**:
  - API Gateways in microservices.
  - Database proxies for query optimization.
- **Benefits**:
  - Improves security by abstracting resource access.
  - Enhances performance with caching.
- **Challenges**:
  - May introduce latency due to intermediate processing.
  - Increased complexity in proxy management.

4. Cloud Native Patterns vs. Enterprise Patterns: Key Differences

- **Purpose and Focus**:
  - **Cloud Native Patterns**: Designed specifically for cloud environments, focusing on scalability, resilience, and efficient resource usage. These patterns leverage cloud infrastructure to create applications optimized for distributed systems and dynamic workloads.
  - **Enterprise Patterns**: Focus on solving common problems in enterprise application development, such as integration, transaction management, and data consistency, typically within on-premises or hybrid environments.

- **Infrastructure Dependency**:
  - **Cloud Native Patterns**: Depend heavily on cloud-native technologies, such as containers (Docker), orchestration (Kubernetes), and managed services (e.g., AWS Lambda or Azure Functions).
  - **Enterprise Patterns**: Often built with a more traditional technology stack, focusing on compatibility with existing enterprise systems and legacy applications.

- **Resilience and Scalability**:
  - **Cloud Native Patterns**: Prioritize auto-scaling, fault tolerance, and high availability through patterns like Circuit Breaker, Retry, or Bulkhead. They assume failure is common and design for recovery.
  - **Enterprise Patterns**: Focus more on consistency, transactional integrity, and reliability within a stable, often controlled environment.

- **Deployment and Updates**:
  - **Cloud Native Patterns**: Emphasize continuous integration/continuous deployment (CI/CD), immutable infrastructure, and blue-green or canary deployments for seamless updates.
  - **Enterprise Patterns**: May rely on traditional release cycles and manual deployments, though modern enterprises increasingly adopt DevOps practices.

- **State Management**:
  - **Cloud Native Patterns**: Prefer stateless architectures and externalized state management using services like distributed caches or databases.
  - **Enterprise Patterns**: Often designed with stateful components, using local databases or in-memory sessions.

- **Examples of Patterns**:
    - **Cloud Native Patterns**: Microservices, Service Mesh, Sidecar, Circuit Breaker, API Gateway, Event Sourcing.
    - **Enterprise Patterns**: Repository Pattern, Unit of Work, Transaction Script, Data Mapper, Business Logic Layer.

- **Usage Scenarios**:
    - **Cloud Native Patterns**: Ideal for cloud-first applications requiring rapid scaling, global distribution, and resilience.
    - **Enterprise Patterns**: Suitable for traditional enterprise systems needing integration with existing infrastructure or adherence to complex business rules.

References

1. Open Universiteit, Software Architecture Learning Unit 3: Architectural Patterns. Open Universiteit, pp. 42–43. [Layers Pattern]
2. Open Universiteit, Software Architecture Learning Unit 3: Architectural Patterns. Open Universiteit, pp. 44–45. [Client-Server Pattern]
3. Open Universiteit, Software Architecture Learning Unit 3: Architectural Patterns. Open Universiteit, pp. 46–47. [Master-Slave Pattern]
4. Open Universiteit, Software Architecture Learning Unit 3: Architectural Patterns. Open Universiteit, pp. 47–49. [Pipe-Filter Pattern]
5. Open Universiteit, Software Architecture Learning Unit 3: Architectural Patterns. Open Universiteit, pp. 49–51. [Broker Pattern]
6. Open Universiteit, Software Architecture Learning Unit 3: Architectural Patterns. Open Universiteit, pp. 51–52. [Peer-to-Peer Pattern]
7. Open Universiteit, Software Architecture Learning Unit 3: Architectural Patterns. Open Universiteit, pp. 52–53. [Event-Bus Pattern]
8. Open Universiteit, Software Architecture Learning Unit 3: Architectural Patterns. Open Universiteit, pp. 53–54. [Model-View-Controller (MVC) Pattern]
9. Open Universiteit, Software Architecture Learning Unit 3: Architectural Patterns. Open Universiteit, pp. 55–56. [Blackboard Pattern]
10. Open Universiteit, Software Architecture Learning Unit 3: Architectural Patterns. Open Universiteit, pp. 56–57. [Interpreter Pattern]
11. C. Davis, Cloud Native Patterns: Designing Change-Tolerant Software. Manning Publications, 2019, pp. 81-105. [Chapter 4: Event-driven microservices]
12. C. Davis, Cloud Native Patterns: Designing Change-Tolerant Software. Manning Publications, 2019, pp. 108-138. [Chapter 5: App redundancy]
13. C. Davis, Cloud Native Patterns: Designing Change-Tolerant Software. Manning Publications, 2019, pp. 139-169. [Chapter 6: Application configuration]
14. C. Davis, Cloud Native Patterns: Designing Change-Tolerant Software. Manning Publications, 2019, pp. 170-206. [Chapter 7: Application lifecycle]
15. C. Davis, Cloud Native Patterns: Designing Change-Tolerant Software. Manning Publications, 2019, pp. 207-230. [Chapter 8: Accessing apps]
16. C. Davis, Cloud Native Patterns: Designing Change-Tolerant Software. Manning Publications, 2019, pp. 231-266. [Chapter 9: Interaction redundancy]
17. C. Davis, Cloud Native Patterns: Designing Change-Tolerant Software. Manning Publications, 2019, pp. 267-294. [Chapter 10: Fronting services]
18. C. Davis, Cloud Native Patterns: Designing Change-Tolerant Software. Manning Publications, 2019, pp. 295-319. [Chapter 11: Troubleshooting]
19. C. Davis, Cloud Native Patterns: Designing Change-Tolerant Software. Manning Publications, 2019, pp. 320-356. [Chapter 12: Cloud-native data]
20. G. Hohpe and B. Woolf, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley, 2004, pp. 81–105. [Layered Architecture Pattern]

21. G. Hohpe and B. Woolf, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley, 2004, pp. 120–145. [Microservices Architecture Pattern] .
22. G. Hoolf, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley, 2004, pp. 146–175. [Event-Driven Architecture Pattern] .
23. G. Hohpe and B. prise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2004, pp. 176–200. [Service-Oriented Architecture (SOA) Pattern] .
24. G. Hohpe and B. Woolf, Enteation Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley, 2004, pp. 201–220. [Repository Pattern] .
25. G. Hohpe and B. Woolf, Enterprise Integns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley, 2004, pp. 221–245. [Saga Pattern] .
26. G. Hohpe and B. Woolf, Enterprise Integration Patteg, Building, and Deploying Messaging Solutions. Addison-Wesley, 2004, pp. 246–270. [CQRS Pattern] .
27. G. Hohpe and B. Woolf, Enterprise Integration Patterns: Designi and Deploying Messaging Solutions. Addison-Wesley, 2004, pp. 271–290. [Strangler Fig Pattern] .
28. G. Hohpe and B. Woolf, Enterprise Integration Patterns: Designing, Buildingng Messaging Solutions. Addison-Wesley, 2004, pp. 291–315. [Circuit Breaker Pattern] .
29. G. Hohpe and B. Woolf, Enterprise Integration Patterns: Designing, Building, and Deplog Solutions. Addison-Wesley, 2004, pp. 316–350. [Proxy Pattern] .