Software Engineering

Architectural Design

Comprehensive Study Guide

1. Architectural Design

Architectural design is concerned with the structure and organization of a software system. It identifies the main structural components in a system and the relationships between them. The output of the architectural design process is an architectural model.

Key Concepts

- Structure and organization of the system
- Identification of main structural components
- Relationships between components
- Output: an architectural model

Advantages of Explicit Architecture

Having an explicit, documented architecture provides three core advantages:

- **Stakeholder communication:** A high-level view of the system that all stakeholders can discuss, even those without a technical background.
- **System analysis:** Makes it possible to analyze whether the system can meet its non-functional requirements before implementation.
- **Large-scale reuse:** The architecture can be reused across systems in the same domain.

Architecture Design Processing

The architecture design process sits within the broader software engineering lifecycle. Key steps include:

- Analyze requirements
- Define business use cases
- Choose architecture pattern/style
- Communicate architecture to stakeholders (via prototypes)
- Evaluate architecture
- Design detailed implementation architecture

Agility and Architecture Design

It is generally accepted that an early stage of agile processes is to design an overall system architecture. Architecture design activities take place throughout the project, not just at the beginning.

In agile settings, architectural work feeds continuously from Ideas and Context/Constraints into a Backlog, which then leads to Architectural Synthesis, Architectural Evaluation, and the resulting Architecture. Architecturally Significant Requirements drive the analysis phase.

Architecture Reuse

- **Same domain:** Systems in the same domain often have similar architectures that reflect domain concepts.
- **Product lines:** Application product lines are built around a core architecture with variants that satisfy particular customer requirements.
- **Patterns/styles:** The architecture of a system may be designed around one or more architectural patterns or styles.

2. Architectural Design Decisions

Architecture and Design Hierarchy

Architecture sits within a nested hierarchy of decisions, from most stable (innermost) to most changeable (outermost):

Level	Description
Requirements	What the system must do — the innermost, most stable layer.
Architecture	The structural pattern chosen to meet the requirements.
Design	The detailed component and class design.
Code	The actual implementation — the outermost, most changeable layer.

Architectural Design Process Details

- **System structuring and partitioning:** Decomposition of the system into subsystems and components.
- **Designing controlling models:** Defines control relationships between subsystems, data flow, and control flow.

Architectural Attributes (Quality Attributes)

When making architectural decisions, architects must consider numerous quality attributes:

- Performance
- Security
- Safety
- Availability
- Maintainability
- Testability
- Portability
- Scalability

Decisions Based on Architectural Attributes

Architectural decisions often involve trade-offs between competing attributes:

- Trade-off between space and time
- Trade-off between dependability and performance
- Trade-off between security and performance

The architect must choose the right architecture to reduce risks, and must assess the impact of architectural choices on quality attributes. Quality attributes are identified during the requirements process.

Other Influential Factors

Factor	Examples
Management	Schedule, budget, available resources

Market	Required features, time to market
Maintenance	Modifiability requirements
Customer	Delivery timing, target platform
End-user	Performance expectations, platform preference

Key Points on Design Decisions

- Architectural design decisions are the choices that bind the final product.
- It is very costly and sometimes impossible to fix architecture mistakes later.
- Various non-functional requirements influence the design decisions.
- An architect usually deals with trade-offs between different architectural attributes.
- Non-technical factors such as management and market may play a big role in making architectural choices.

3. Architectural Styles / Patterns

An architectural pattern is a stylized description of a reusable design solution to a commonly occurring problem, which has been tried and tested in different environments. Patterns are a means of representing, sharing, and reusing knowledge.

Architectural patterns are similar to software design patterns but have a broader scope. Patterns should include information about when they are and when they are not useful.

Combining Architectural Patterns

The architecture of a software system is almost never limited to a single architectural style. For example, you might have a SOA design composed of services developed using a layered architecture approach and a component-based architecture style. Most engines (like game engines) and frameworks use at least several architectural patterns.

List of Common Architectural Patterns

Core patterns:

- Pipes and Filters
- (N-)Layered
- (N-)Tiered
- Repository / Data Centered
- Interpreter / Virtual Machine
- Event-driven architecture
- Broker (message broker)
- Service Oriented Architecture (SOA)
- Micro-services
- Peer-to-peer
- Client/Server
- Component-based
- MVC / MVP / MVVM

Other patterns:

- Domain-driven architecture
- CQRS
- Event Sourcing

4. The Layered Architecture

Layered architecture organizes the system into layers with related functionality associated with each layer. A layer only communicates with the layer immediately below it (in a closed model) or may bypass layers (in an open model).

Typical Layers

Layer	Responsibility
Presentation layer (UI)	Handles user interaction and display.
Business layer	Implements core business logic and rules.
Data access layer	Abstracts database access.
Database layer	Stores and retrieves persistent data.

Layers can be marked as Closed (a request must go through this layer) or Open (a request may bypass this layer). For example: Presentation > Security > Business > Service (Open) > Data > Database.

Benefits of Layered Architecture

Benefit	Description
Separation of concerns	Each layer has a clearly defined responsibility.
Reusability	Layers can be reused across different systems.
Isolation	Changes in one layer do not affect other layers.
Abstraction	Each layer hides complexity from the layer above.
Testability	Layers can be tested independently.
Manageability	Easier to manage and maintain.

Key Points

- Used when building new facilities on top of existing systems.
- Used when development is spread across several teams.
- Used when there is a requirement for multi-level security.
- Allows replacement of entire layers so long as the interface is maintained.
- In practice, providing a clean separation between layers is often difficult.

5. The Tiered Architecture

A tier is a physical structuring mechanism for the system infrastructure. Tiered architecture shows how the system is organized into physical deployment layers, as opposed to logical layers.

Key Difference from Layered

Layers are logical divisions (separation of concerns in code). **Tiers** are physical divisions (separate machines/processes).

Typical N-Tier Example

Tier	Description
DMZ / Load Balancer	Handles incoming internet traffic and distributes it.
Web tier	Serves web pages and handles HTTP requests.
Business tier	Processes business logic.
Data tier	SQL Server (primary/secondary) stores data.

Key Points

- Used for many simple web applications.
- Allows high-performance, reliable applications due to scalable tiers.
- Provides portability.
- Monolithic design may prevent independent deployment of features.
- Can be difficult to manage in multi-tiered solutions.

6. MVC / MVP / MVVM Architectures

These three architectural patterns all separate the concerns of data, presentation, and user interaction but differ in how the three components communicate.

MVC (Model-View-Controller)

Component	Responsibility
Model	Manages the system data and associated operations on that data. Encapsulates application state. Notifies the view of state changes.
View	Defines and manages how the data is presented to the user. Renders model data. Sends user events to the controller.
Controller	Manages user interaction and passes interactions to the View and the Model. Maps user actions to model updates. Selects the view.

In a web context (e.g. PHP): the Controller handles HTTP request processing, application-specific logic, and data validation. The View generates dynamic pages and manages forms. The Model holds business logic and database access.

MVP (Model-View-Presenter)

The Presenter handles all UI logic. The View is completely passive and delegates all input to the Presenter. The Presenter updates the View directly based on changes in the Model. This makes the View easier to test.

MVVM (Model-View-ViewModel)

The ViewModel exposes data streams and commands. The View binds to the ViewModel through data binding, removing the need for the ViewModel to hold a reference to the View. Widely used in modern mobile (Android, iOS) and desktop frameworks.

Comparison Table

Aspect	MVC
Input handling	Controller
View knowledge	Controller selects view
Testability	Good
Common use	Web backends

Key Points

- Many web and mobile applications use these architectures.
- Used when there are multiple ways to view and interact with data.
- Promotes low coupling, high cohesion, testability.
- Promotes independent development and code reuse.
- Does not make sense for applications where the UI is minimal or non-existent.
- MVC is an older style; MVP and MVVM are more common in modern development.

7. Repository / Data-Centric Architecture

Sub-systems must exchange data. This may be done in two ways: shared data held in a central database accessible by all sub-systems, or each sub-system maintaining its own database and passing data explicitly to others.

When large amounts of data are to be shared, the repository model is most commonly used as it is the most efficient data sharing mechanism.

How It Works

All data in the system is managed in a central repository that is accessible to all system components. Components are independent of each other and do not need to know of the existence of other components. Adding new data to the repository can trigger an action or tool.

Example: IDE as Repository

A development environment (IDE) is a classic example. All tools work against a shared project repository:

- Editors: C# editor, HTML editor, JavaScript editor, UI designer
- Code generators: .NET generator, Machine code generator, HTML5 generator
- Code analysis software
- Report generator
- Package manager
- GIT integration tool

Unreal Engine uses several architectural patterns including Repository, Compiler, and Interpreter patterns.

Key Points

- All data is managed in a central repository accessible to all system components.
- Many tools such as IDEs and game engines use this approach.
- Useful in data-driven systems where including data in the repository triggers an action.
- Components can be independent since they do not need to know of other components.
- All data can be managed consistently.
- The repository is a single point of failure: problems affect the whole system.
- May be inefficiencies in organizing all communication through the repository.
- Distributing the repository across several computers can be difficult.

8. Interpreter / Virtual Machine Architecture

This pattern is used for designing a component that interprets programs written in a dedicated language. The interpreter acts as a sandbox where the application can run without being able to harm the host environment.

Real-World Examples

- Web browser executing JavaScript
- JVM (Java Virtual Machine)
- Lua scripting in game engines
- Roblox Studio uses a Lua virtual machine
- Krpano virtual tour scripting

When to Use

- When users or specialists need to modify the behaviour of the application.
- When the rules of the application need to be extended long after the application is finished.
- When the application may not directly access the environment (sandboxing requirement).

Key Points

- Enhances flexibility because replacing an interpreted program is easy.
- Performance may be an issue compared to compiled code.
- May require a custom debugger.
- May require the development and implementation of a feature-rich programming language.

9. Pipes and Filters Architecture

The Pipes and Filters architecture is used to divide a larger processing task into a sequence of smaller, independent processing steps (Filters) that are connected by channels (Pipes). The central concept is processing of a data stream.

Core Concepts

Component	Description
Filter	An independent processing step that transforms or processes data.
Pipe	A channel that connects filters and transfers data between them.
Source	Produces the initial data stream.
Sink	Consumes the final processed data stream.

A classic example from Unix/Linux shells:

```
ls -l | grep key | less
```

Another example: command 1 | command 2 | command 3 | ...

When to Use

- When transformations are sequential (batch sequential model).
- Data processing systems.
- When each processing step is independent and can be developed separately.

Advantages

- Easy to understand.
- Supports transformation reuse.
- Opportunities for running parallel instances.
- Supports resiliency.
- Fantastic scalability opportunities.

Challenges

- Overhead from passing data between filters.
- Added complexity.
- Reliability concerns if a filter fails.
- Idempotency: filters may need extra design effort to be safely re-runnable.

10. Event-Driven Architectures

An event-driven architecture consists of event producers that generate a stream of events, and event consumers that listen for and react to those events. There is an event ingestion mechanism between producers and consumers.

Publisher / Subscriber Model

Publishers emit events without knowing who will receive them. Subscribers register interest in specific event types. A pubsub mechanism routes events from publishers to interested subscribers.

Event Stream Model

Producers write events to a persistent event stream (stored as a file, database, or in memory). Multiple clients can read from the stream independently, at their own pace.

Key Points

- Promotes loose coupling between components.
- Promotes separation of concerns.
- Reliability issues may emerge (event loss, ordering issues).
- Complex control flow can be difficult to debug.
- Easy to understand paradigm, but implementation differs across programming languages.

11. Event Bus Architecture

The Event Bus pattern has four major components that work together to provide a decoupled messaging infrastructure.

The Four Components

Component	Description
Event source	Produces events when something of interest happens.
Channel	A named conduit for a particular type of event.
Event Bus	The central routing mechanism that directs events to appropriate channels.
Event listener	Subscribes to one or more channels and reacts to events.

Example: GreenRobot EventBus (Android)

EventBus is an open-source library for Android and Java using the publisher/subscriber pattern for loose coupling. It enables central communication between decoupled classes, simplifying code and removing dependencies.

Key Points

- Used when it is needed to decouple components to achieve effective distribution.
- New publishers, subscribers, and connections can be added easily.
- Effective for highly distributed applications.
- Scalability may be a problem if all messages travel through the same bus.

12. Message Broker Architecture

In the message broker pattern, servers publish their messages to topics inside a broker (middleware). Client subscribers receive messages that were published to topics they are subscribed to.

How It Works

Servers produce messages and send them to named channels (topics) within the broker. Clients subscribe to channels of interest. The broker stores and forwards messages, decoupling producers from consumers in both time and space.

Popular Implementations

- Apache ActiveMQ
- Apache Kafka
- RabbitMQ
- JBoss Messaging

Key Points

- Used to structure distributed systems with decoupled components.
- Standardization is still evolving across implementations.
- Highly reliable and scalable in enterprise settings.

13. Client-Server Architecture

In a client-server architecture, the functionality of the system is organized into services, where each service may be delivered from a separate server. Clients access those services over a network (typically the Internet).

How It Works

Clients send requests to specific servers (e.g. a video server, a photo server, or a web server). The server processes the request and returns a response. Communication occurs through the Internet or an internal network.

Key Points (Advantages)

- Distributed systems accessed over the Internet are normally organized as client-server systems.
- Easier to manage system configuration compared to fully distributed (serverless) systems.
- Better security can be achieved in centralized server-based systems.

Key Points (Disadvantages)

- Each service is a single point of failure, making it susceptible to denial-of-service attacks or server failure.
- Performance may be unpredictable because it depends on the network as well as the system.
- May be management problems if servers are owned by different organizations.

14. Peer-to-Peer Architecture

Peer-to-peer (P2P) systems are decentralised systems where computations may be carried out by any node in the network. The overall system is designed to take advantage of the computational power and storage of a large number of networked computers.

P2P Models

Model	Description
Decentralized	No central authority. Each node connects directly to other nodes. Data and routing are distributed.
Semi-centralized	A discovery service (super-peer) helps nodes find each other, but data transfer is direct between nodes.

Popular P2P Systems

- File sharing: BitTorrent protocol
- Messaging: Jabber
- Payments: Bitcoin
- Databases: Freenet (decentralized database)
- Phone systems: Viber
- Computation: SETI@home

Key Points

- Effective when a system is computationally or network intensive.
- Takes advantage of the computational power and storage of many networked computers.
- Security and privacy concerns are the principal reason why P2P architectures are not widely used in enterprise settings.

15. Service Oriented Architecture (SOA) and Microservices

Service Oriented Architecture (SOA)

SOA is an architecture that structures the application as a set of loosely coupled, collaborating services. Services communicate using synchronous protocols (REST, SOAP) or asynchronous messaging. The three-party model involves a Service Registry, Service Provider, and Service Consumer:

- Service Provider publishes the service description to the Service Registry.
- Service Consumer discovers the service via the Service Registry.
- Service Consumer invokes the service directly on the Service Provider.

Key Principles of SOA

- Services are autonomous
- Services are distributable
- Services are loosely coupled
- Services share schema and contract, not class
- Compatibility is based on policy

Main Benefits of SOA

- Domain alignment
- Abstraction
- Discoverability
- Interoperability
- Rationalization

Microservices

Microservices are a special case of SOA where the services are almost completely decoupled and autonomous. Each microservice has its own database, is deployed independently, and communicates through APIs.

A typical microservices system: a Mobile App and Browser connect through an API Gateway, which routes requests to Account Service, Inventory Service, and Shipping Service, each with their own database.

Main Benefits of Microservices

- All benefits of SOA
- Improved continuous delivery (independent deployment)
- Improved fault isolation (one service failure does not crash others)
- Independent development by separate teams

Summary: Evolution of Web Architectures

Style	Coupling
Monolithic	Tightly coupled

Service-oriented (SOA)	Loosely coupled
Microservices	Decoupled

Key Points

- Structures the application as a set of loosely coupled, collaborating services.
- Services communicate using synchronous protocols.
- Microservices are a special case of SOA with nearly complete decoupling and autonomy.
- Not appropriate for real-time applications.
- SOA architecture implementations can be complex.

16. Component-Based Architecture (CBA)

Component-based architecture focuses on the decomposition of the design into individual functional or logical components. The components expose well-defined communication interfaces containing methods, events, and properties.

Key Principles of CBA

Principle	Description
Reusable	Components can be used in multiple systems.
Replaceable	Components can be substituted with other similar components.
Not context specific	Components work in different environments.
Extensible	Components can add behaviour to other components.
Encapsulated	Internals are hidden; only the interface is exposed.
Independent	Components have minimal dependencies on other components.

Main Benefits of CBA

- Ease of deployment
- Reduced cost
- Ease of development
- Reusability
- Mitigation of technical complexity

Shortcomings

- Effort required for development of components.
- Component maintenance costs.
- Reliability concerns if components change.
- Sensitivity to interface changes.

Examples in Practice

- Visual Basic / .NET UI component libraries
- Unreal Engine and Unity game component systems
- React and Vue.js frontend component frameworks

17. Summary Comparison Table

The following table provides a quick reference for all major architectural patterns covered in this guide.

Pattern	Best Used When	Main Drawback
Layered	Building on top of existing systems; multi-team development; multi-level security	Clean separation between layers is often difficult in practice
Tiered	Web applications needing physical deployment separation; scalable infrastructure	Monolithic design prevents independent feature deployment
MVC/MVP/MVVM	Multiple views/interactions with same data; web and mobile apps	Overkill for minimal UI; adds complexity for simple data models
Repository	Many tools sharing a central data store; IDE-like systems	Single point of failure; distributing the repository is hard
Interpreter/VM	Users need to modify app behaviour; sandbox security required	Performance overhead; requires custom debugger
Pipes & Filters	Sequential data processing; transformations that can be parallelized	Communication overhead; idempotency design effort
Event-Driven	Loosely coupled components; asynchronous processing	Complex control flow; reliability concerns
Event Bus	Decoupling components in a distributed system	Bus itself can become a scalability bottleneck
Message Broker	Distributed systems needing reliable async messaging	Standardization not yet settled; operational complexity
Client/Server	Internet-accessed systems; centralized management/security	Each service is a single point of failure
Peer-to-peer	Computationally or network intensive distributed tasks	Security and privacy concerns limit adoption
SOA	Integrating heterogeneous enterprise systems	Complex implementation; not suitable for real-time
Microservices	Large teams; need for independent deployment and fault isolation	Distributed system complexity; network overhead
CBA	Systems built from reusable functional units; frameworks	Component maintenance costs; sensitivity to interface changes