**Layered Architecture Pattern**

Is the most common Architecture pattern, also known as the N-Tier Architecture Pattern.

**Pattern Description**

Components within the layered architecture pattern are organized into horizontal layers, each layer performing a specific role within the application (e.g., presentation logic or business logic). Although the layered architecture pattern does not specify the number and types of layers that must exist in the pattern, most layered architectures consist of four standard layers: presentation, business, persistence, and database.



Each layer of the layered architecture pattern has a specific role and responsibility within the application. For example, a presentation layer would be responsible for handling all user interface and browser communication logic, whereas a business layer would be responsible for executing specific business rules associated with the request. Each layer in the architecture forms an abstraction around the work that needs to be done to satisfy a particular business request. For example, the presentation layer doesn't need to know or worry about how to get customer data; it only needs to display that information on a screen in particular format. Similarly, the business layer doesn't need to be concerned about how to format customer data for display on a screen or even where the customer data is coming from; it only needs to get the data from the persistence layer, perform business logic against the data (e.g., calculate values or aggregate data), and pass that information up to the presentation layer.

## Key Concept

A Request must go through every layer, its not possible to skip one. This concept is known as “layers of isolation”. The layers of isolation concept means that changes made in one layer of the architecture generally don't impact or affect components in other layers: the change is isolated to the components within that layer. (les coupling => easier to change and maintain). The layers of isolation concept also means that each layer is independent of the other layers, thereby having little or no knowledge of the inner workings of other layers in the architecture.

**Open**

ToDO

**Pros:**

Solid General Oattern, good starting point for most applications.

Easy to test

Easy to implement

**Cons:**

* Performance (see sinkhole anti-pattern)
* Scaleability
* Can be hard to deploy

**Architecture sinkhole anti-pattern**: A Request flows through multiple layer as a simple pass-trough, the pass trough layers doesn’t (or very little) processing. It’s common to have a few of these “sinkholes”, but it shouldn’t overpasse 20% of the requests. In the case there are a lot of simple-pass troughs, you should implement a “open Layer”

For example, assume the presentation layer responds to a request from the user to retrieve customer data. The presentation layer passes the request to the business layer, which simply passes the request to the persistence layer, which then makes a simple SQL call to the database layer to retrieve the customer data. The data is then passed all the way back up the stack with no additional processing or logic to aggregate, calculate, or transform the data.

## Rating:

**Overall agility: Low**

Overall agility is the ability to respond quickly to a constantly changing environment. While change can be isolated through the layers of isolation feature of this pattern, it is still cumbersome and time-consuming to make changes in this architecture pattern because of the monolithic nature of most implementations as well as the tight coupling of components usually found with this pattern.

**Ease of deployment: Low**

Depending on how you implement this pattern, deployment can become an issue, particularly for larger applications. One small change to a component can require a redeployment of the entire application (or a large portion of the application), resulting in deployments that need to be planned, scheduled, and executed during off-hours or on weekends. As such, this pattern does not easily lend itself toward a continuous delivery pipeline, further reducing the overall rating for deployment.

**Testability: High**

Because components belong to specific layers in the architecture, other layers can be mocked or stubbed, making this pattern is relatively easy to test. A developer can mock a presentation component or screen to isolate testing within a business component, as well as mock the business layer to test certain screen functionality.

**Performance: low**

While it is true some layered architectures can perform well, the pattern does not lend itself to high-performance applications due to the inefficiencies of having to go through multiple layers of the architecture to fulfill a business request.

**Scalability: Low**

Because of the trend toward tightly coupled and monolithic implementations of this pattern, applications build using this architecture pattern are generally difficult to scale. You can scale a layered architecture by splitting the layers into separate physical deployments or replicating the entire application into multiple nodes, but overall the granularity is too broad, making it expensive to scale.

**Ease of development: High**

Ease of development gets a relatively high score, mostly because this pattern is so well known and is not overly complex to implement. Because most companies develop applications by separating skill sets by layers (presentation, business, database), this pattern becomes a natural choice for most business-application development. The connection between a company's communication and organization structure and the way it develops software is outlined is what is called Conway's law. You can Google "Conway's law" to get more information about this fascinating correlation.

Source: <https://www.oreilly.com/ideas/software-architecture-patterns/page/2/layered-architecture#sapr_0104_img>

**Intent**: «Standart» architectural pattern. Matching most of the traditional IT-Communication structure, making it a natural choice for most business applications. For example: OSI modell

**Applicability:** In most applciations cause the way of layering is very natural and matching most of our Communication and business structure. Bad Idea in a often changed environment because of bad Scaleability and deployment. Can have a bad performance (sinkhole anti-pattern)

**Consequences:**

**Pros:**

+ Solid General Oattern, good starting point for most applications.

+ Easy to test

+ Easy to implement

**Cons:**

* Performance (see sinkhole anti-pattern)
* Scaleability
* Can be hard to deploy

**Structure**

For Example the OSI Layer Diagramm

**Variants:** Open and close Layers.