## The top 5 software architecture patterns: How to make the right choice

How many plots are there in Hollywood movies? Some critics say there are only five. How many ways can you structure a program? Right now, the majority of programs use one of five architectures.

Mark Richards is a Boston-based software architect who’s been thinking for more than 30 years about how data should flow through software. His new (free) book, *[Software Architecture Patterns](http://www.oreilly.com/programming/free/software-architecture-patterns.csp" \t "_blank)*, focuses on five architectures that are commonly used to organize software systems. The best way to plan new programs is to study them and understand their strengths and weaknesses.

In this article, I’ve distilled the five architectures into a quick reference of the strengths and weaknesses, as well as optimal use cases. Remember that you can use multiple patterns in a single system to optimize each section of code with the best architecture. Even though they call it computer science, it’s often an art.

**Testing in the Agile Era: Top Tools and Processes**

[**GET REPORT**](https://software.microfocus.com/en-us/assets/application-delivery-management/it-central-station-testing-agile-era?utm_campaign=00134846&source=techbeacon&utm_medium=article&utm_content=middle)

## Layered (n-tier) architecture

This approach is probably the most common because it is usually built around the database, and many applications in business naturally lend themselves to storing information in tables.

This is something of a self-fulfilling prophecy. Many of the biggest and best software frameworks—like Java EE, Drupal, and Express—were built with this structure in mind, so many of the applications built with them naturally come out in a layered architecture.

The code is arranged so the data enters the top layer and works its way down each layer until it reaches the bottom, which is usually a database. Along the way, each layer has a specific task, like checking the data for consistency or reformatting the values to keep them consistent. It’s common for different programmers to work independently on different layers.

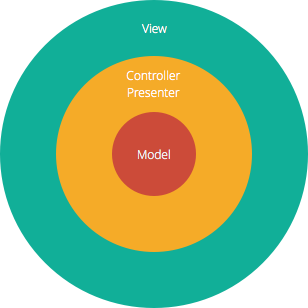


Image credit: [Izhaki](http://stackoverflow.com/a/30745329" \t "_blank)

The Model-View-Controller (MVC) structure, which is the standard software development approach offered by most of the popular web frameworks, is clearly a layered architecture. Just above the database is the model layer, which often contains business logic and information about the types of data in the database. At the top is the view layer, which is often CSS, JavaScript, and HTML with dynamic embedded code. In the middle, you have the controller, which has various rules and methods for transforming the data moving between the view and the model.

The advantage of a layered architecture is the separation of concerns, which means that each layer can focus solely on its role. This makes it:

* Maintainable
* Testable
* Easy to assign separate "roles"
* Easy to update and enhance layers separately

Proper layered architectures will have isolated layers that aren’t affected by certain changes in other layers, allowing for easier refactoring. This architecture can also contain additional open layers, like a service layer, that can be used to access shared services only in the business layer but also get bypassed for speed.

Slicing up the tasks and defining separate layers is the biggest challenge for the architect. When the requirements fit the pattern well, the layers will be easy to separate and assign to different programmers.

**Caveats:**

* Source code can turn into a “big ball of mud” if it is unorganized and the modules don’t have clear roles or relationships.
* Code can end up slow thanks to what some developers call the “sinkhole anti-pattern.” Much of the code can be devoted to passing data through layers without using any logic.
* Layer isolation, which is an important goal for the architecture, can also make it hard to understand the architecture without understanding every module.
* Coders can skip past layers to create tight coupling and produce a logical mess full of complex interdependencies.
* Monolithic deployment is often unavoidable, which means small changes can require a complete redeployment of the application.

**Best for:**

* New applications that need to be built quickly
* Enterprise or business applications that need to mirror traditional IT departments and processes
* Teams with inexperienced developers who don’t understand other architectures yet
* Applications requiring strict maintainability and testability standards

## Event-driven architecture

Many programs spend most of their time waiting for something to happen. This is especially true for computers that work directly with humans, but it’s also common in areas like networks. Sometimes there’s data that needs processing, and other times there isn’t.

The event-driven architecture helps manage this by building a central unit that accepts all data and then delegates it to the separate modules that handle the particular type. This handoff is said to generate an “event,” and it is delegated to the code assigned to that type.   
  
Programming a web page with JavaScript involves writing the small modules that react to events like mouse clicks or keystrokes. The browser itself orchestrates all of the input and makes sure that only the right code sees the right events. Many different types of events are common in the browser, but the modules interact only with the events that concern them. This is very different from the layered architecture where all data will typically pass through all layers. Overall, event-driven architectures:

* Are easily adaptable to complex, often chaotic environments
* Scale easily
* Are easily extendable when new event types appear

**Caveats:**

* Testing can be complex if the modules can affect each other. While individual modules can be tested independently, the interactions between them can only be tested in a fully functioning system.
* Error handling can be difficult to structure, especially when several modules must handle the same events.
* When modules fail, the central unit must have a backup plan.
* Messaging overhead can slow down processing speed, especially when the central unit must buffer messages that arrive in bursts.
* Developing a systemwide data structure for events can be complex when the events have very different needs.
* Maintaining a transaction-based mechanism for consistency is difficult because the modules are so decoupled and independent.

**Best for:**

* Asynchronous systems with asynchronous data flow
* Applications where the individual data blocks interact with only a few of the many modules
* User interfaces

## Microkernel architecture

Many applications have a core set of operations that are used again and again in different patterns that depend upon the data and the task at hand. The popular development tool Eclipse, for instance, will open files, annotate them, edit them, and start up background processors. The tool is famous for doing all of these jobs with Java code and then, when a button is pushed, compiling the code and running it.

In this case, the basic routines for displaying a file and editing it are part of the microkernel. The Java compiler is just an extra part that’s bolted on to support the basic features in the microkernel. Other programmers have extended Eclipse to develop code for other languages with other compilers. Many don’t even use the Java compiler, but they all use the same basic routines for editing and annotating files.

The extra features that are layered on top are often called plug-ins. Many call this extensible approach a plug-in architecture instead.

Richards likes to explain this with an example from the insurance business: “Claims processing is necessarily complex, but the actual steps are not. What makes it complex are all of the rules.”

The solution is to push some basic tasks—like asking for a name or checking on payment—into the microkernel. The different business units can then write plug-ins for the different types of claims by knitting together the rules with calls to the basic functions in the kernel.

**Caveats:**

* Deciding what belongs in the microkernel is often an art. It ought to hold the code that’s used frequently.
* The plug-ins must include a fair amount of handshaking code so the microkernel is aware that the plug-in is installed and ready to work.
* Modifying the microkernel can be very difficult or even impossible once a number of plug-ins depend upon it. The only solution is to modify the plug-ins too.
* Choosing the right granularity for the kernel functions is difficult to do in advance but almost impossible to change later in the game.

**Best for:**

* Tools used by a wide variety of people
* Applications with a clear division between basic routines and higher order rules
* Applications with a fixed set of core routines and a dynamic set of rules that must be updated frequently

## Microservices architecture

Software can be like a baby elephant: It is cute and fun when it’s little, but once it gets big, it is difficult to steer and resistant to change. The microservice architecture is designed to help developers avoid letting their babies grow up to be unwieldy, monolithic, and inflexible. Instead of building one big program, the goal is to create a number of different tiny programs and then create a new little program every time someone wants to add a new feature. Think of a herd of guinea pigs.

[via GIPHY](http://giphy.com/gifs/cute-flood-ehWeTxjeCy2li)

“If you go onto your iPad and look at Netflix’s UI, every single thing on that interface comes from a separate service,” points out Richards. The list of your favorites, the ratings you give to individual films, and the accounting information are all delivered in separate batches by separate services. It’s as if Netflix is a constellation of dozens of smaller websites that just happens to present itself as one service.

This approach is similar to the event-driven and microkernel approaches, but it’s used mainly when the different tasks are easily separated. In many cases, different tasks can require different amounts of processing and may vary in use. The servers delivering Netflix’s content get pushed much harder on Friday and Saturday nights, so they must be ready to scale up. The servers that track DVD returns, on the other hand, do the bulk of their work during the week, just after the post office delivers the day’s mail. By implementing these as separate services, the Netflix cloud can scale them up and down independently as demand changes.

**Caveats:**

* The services must be [largely independent](http://techbeacon.com/5-fundamentals-successful-microservice-design) or else interaction can cause the cloud to become imbalanced.
* [Not all applications](http://techbeacon.com/why-your-project-may-not-be-ready-microservices) have tasks that can’t be easily split into independent units.
* Performance can suffer when tasks are spread out between different microservices. The communication costs can be significant.
* Too many microservices can confuse users as parts of the web page appear much later than others.

**Best for:**

* Websites with small components
* Corporate data centers with well-defined boundaries
* Rapidly developing new businesses and web applications
* Development teams that are spread out, often across the globe

## Space-based architecture

Many websites are built around a database, and they function well as long as the database is able to keep up with the load. But when usage peaks, and the database can’t keep up with the constant challenge of writing a log of the transactions, the entire website fails.

The space-based architecture is designed to avoid functional collapse under high load by splitting up both the processing and the storage between multiple servers. The data is spread out across the nodes just like the responsibility for servicing calls. Some architects use the more amorphous term “cloud architecture.” The name “space-based” refers to the “tuple space” of the users, which is cut up to partition the work between the nodes. “It’s all in-memory objects,” says Richards. “The space-based architecture supports things that have unpredictable spikes by eliminating the database.”

Storing the information in RAM makes many jobs much faster, and spreading out the storage with the processing can simplify many basic tasks. But the distributed architecture can make some types of analysis more complex. Computations that must be spread out across the entire data set—like finding an average or doing a statistical analysis—must be split up into subjobs, spread out across all of the nodes, and then aggregated when it’s done.

**Caveats:**

* Transactional support is more difficult with RAM databases.
* Generating enough load to test the system can be challenging, but the individual nodes can be tested independently.
* Developing the expertise to cache the data for speed without corrupting multiple copies is difficult.

**Best for:**

* High-volume data like click streams and user logs
* Low-value data that can be lost occasionally without big consequences—in other words, not bank transactions
* Social networks

These are Richards' big five. They may be just what you need. And if they’re not, the right solution could be a mixture of two. Or maybe even three. Be sure to [download his book for free](http://www.oreilly.com/programming/free/software-architecture-patterns.csp); it contains a lot more detail. ***If you can think of more, please let us know in the comments.***