24

Independence

A good architecture must support

- The use cases of the system
- The operation of the system
- The development of the system
- The deployment of the system

Use Cases

The first bullet—use cases—means that the architecture of the system must support the intent of the system. If the system is a shopping cart application, then the architecture must support shopping cart use cases. Indeed, this is the first priority of the architecture. The architecture must support the use cases.

However, as we discussed previously, architecture does not wield much influence over the behavior of the system. There are very few behavioral options that the architecture can leave open. But influence isn't everything. The most important thing a good architecture can do to support behavior is to clarify and expose that behavior so that the intent of the system is visible at the architectural level.

A shopping cart application with a good architecture will *look* like a shopping cart application. The use cases of that system will be plainly visible within the structure of that system. Developers will not have to hunt for behaviors, because those behaviors will be first-class elements visible at the top level of the system. Those elements will be classes or functions or modules that have prominent positions within the architecture and will have names that clearly describe their function.

Operation

Architecture plays a more substantial, and less cosmetic, role in supporting the operation of the system. If the system must handle 100,000 transactions per second, the architecture must support that kind of throughput and response time for each use case that demands it. If the system must query big data cubes in milliseconds, then the architecture must be structured to allow this kind of operation.

For some systems, this will mean arranging the processing elements of the system into an array of little services that can be run in parallel on many different servers. For other systems, it will mean a plethora of little lightweight threads sharing the address space of a single process within a single processor. Still other systems will need just a few processes running in isolated address spaces. And some systems can even survive as simple monolithic programs running in a single process.

As strange as it may seem, this decision is one of the options that a good architecture leaves *open*. A system that is written as a monolith and that depends upon that monolithic structure cannot easily be upgraded to multiple processes, multiple threads, or microservices should the need arise. On the other hand, an architecture that maintains the proper isolation of its components and does not assume the means of communication between those components will be much easier to transition through the spectrum of threads, processes, and services as the operational needs of the system change over time.

Development

Architecture plays a significant role in supporting the development environment. This is where Conway’s law and the Single Responsibility Principle (SRP) come into play. A system that must be developed by an organization with many teams and many concerns must have an architecture that facilitates independent actions by those teams so that the teams do not interfere with each other during development.

This is accomplished by properly partitioning the system into well-isolated, independently developable components. Those components can then be allocated to teams that can work independently of each other.

Deployment

Finally, the architecture plays a huge role in the ease with which the system is deployed. The goal is “immediate deployment.” A good architecture does not rely on dozens of little configuration scripts and property file tweaks. It does not require manual creation of directories or files that must be arranged just so. A good architecture helps the system be immediately deployable after each build.

Again, this is achieved through the proper partitioning and isolation of the components of the system, including those master components that tie the whole system together and ensure that each component is properly started, integrated, and supervised.

Leaving Options Open

A good architecture balances all the above concerns with a component structure that mutually satisfies them all. Sounds easy, right? Well, it’s easy for me to write that.

The reality is that this is pretty hard. The problem is that most of the time we don’t know what all the use cases are, nor do we know the operational constraints, the team structure, or the deployment requirements. Worse, even if we *did* know them, they will all change as the system moves through its lifecycle. In short, the goals we must meet are indistinct and inconstant. Welcome to the real world.

But all is not lost. There are principles of architecture that are relatively inexpensive to implement and that can help balance those concerns, even when you don’t have a clear picture of the goals you have to hit. Those principles help us partition our systems into well-isolated components that allow us to leave as many options open as possible, for as long as possible.

A good architecture makes the system easy to change, in all the ways that it must change, by leaving options open.