



Home > [Articles](#) > [Software Development & Management](#) > [Architecture and Design](#)

The Changing Field of Software Architecture

By [Rick Kazman](#), [Len Bass](#), [Paul Clements](#)

Dec 12, 2012

 [Print](#)  [Share This](#)

Page 1 of 1

The authors of *Software Architecture in Practice*, 3rd Edition discuss how technologies like cloud and edge-dominant systems have changed (and not changed) the field of software architecture in the ten years since the last edition of their book was published.

From the author of



[Software Architecture in Practice, 3rd Edition](#)

[Learn More](#)  [Buy](#)

The discipline of “software architecture” was not formally studied until about 1990. Since then the term has become fashionable, and many software professionals now sport some form of “architect” title on their business cards. Enterprise architects, solution architects, application architects, and others are as common as beans. But, more significantly, the practice of software architecture has become indispensable technically, and a critical enabler of competitive advantage organizationally.

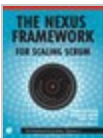
What is software architecture? Definitions abound. Not surprisingly, we like ours best:

The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.

This definition stands in contrast to many other definitions that talk about the “early” or “major” design decisions. Many architectural decisions are made early, but not all are. Many decisions are made early that are not architectural. And it’s hard to make a decision and tell whether or not it’s “major.” So we prefer to focus attention on structures, relationship, and properties.

Related Resources

[Store](#) [Articles](#) [Blogs](#)



[Nexus Framework for Scaling Scrum, The: Continuously Delivering an Integrated Product with Multiple Scrum Teams](#)

By [Kurt Bittner](#), [Patricia Kong](#), [Dave West](#)

Book \$27.99



[Adaptive Code: Agile coding with design patterns and SOLID principles, 2nd Edition](#)

By [Gary McLean Hall](#)

Book \$39.99



[SOA Design Patterns \(paperback\)](#)

By [Thomas Erl](#)


Book \$59.99

[See All Related Store Items](#)

From the author of



[Software Architecture in Practice, 3rd Edition](#)

[Learn More](#)  [Buy](#)

This focus implies that architecture is an abstraction. An architecture comprises software elements and how the elements relate to each other. Our definition, with its emphasis on abstractions—structures and properties—makes it clear that an architecture selects certain details and suppresses others:

- It omits information about elements that is not useful for reasoning about the system.
- It omits information that has no ramifications outside of the design and implementation of a single element.
- It omits private details of elements—details having to do solely with internal implementation.

We humans have finite cognitive capabilities; we can only keep so many details in our heads at once. But systems know no physical limits—they may become arbitrarily complex, quickly outstripping our abilities to understand them (if “understand” means mastery of every detail of the system). Thus abstraction is essential to taming the complexity of an architecture. We simply cannot, and do not want to, deal with all of the complexity all of the time! Furthermore, by focusing on the abstractions, we can do a substantial amount of planning, design, and analysis before committing enormous amounts of resources to code. This is no different from other mature engineering disciplines: a civil engineer, for example, creates and analyzes blueprints before digging foundations, pouring concrete, and bolting beams together. This is just plain good common sense.

With the increasing complexity of the systems and systems of systems that envelope our lives these days, architecture has taken an increasingly central role. It has been almost a decade since the publication of the second edition of *Software Architecture in Practice*. During that time, the field of software has grown enormously in scope and in its impact on society. Concurrent with this explosion in software, software architecture has broadened its focus from being primarily internally and technically oriented—addressing problems of how to design, evaluate, and document complex software systems—to including external impacts as well. It affects your business and its goals; it affects your people and what they do and how they are trained; in some cases it affects the broader technical environment; and in just a few cases it affects society at large (think, for example, of how society has been affected by mobile applications, peer-to-peer networks, social networks, and service computing). Architectures can have a profound effect on your organizational goals, your people, your software development life cycle, and on the way you manage a project.

The past ten years have also seen dramatic changes in the types of systems being constructed and thus in the types of concerns facing architects. Large data, social media, and the cloud are all areas that, at most, were embryonic ten years ago and now are not only mature but also extremely influential. Software has become ubiquitous: in our computers, obviously, but also in our smart phones, our toys, our tools, our refrigerators and rice cookers and cars and thermostats and razors. In fact, software has not just become ubiquitous, but it has become *essential* in all of these domains (well, maybe not razors) and it has become essential to business success. Look at the ads for a modern automobile and you will see an enormous emphasis on features enabled by software: anti-lock brakes, adaptive cruise-control, GPS, Bluetooth, anti-theft devices, parking sensors...the list goes on and on. Without such features, a car simply is not competitive in today's market. Getting a head start on your competitors in providing such features can be a huge advantage. At the heart of it all lies the software architecture; the artifact that mediates between business goals and the achievement of those goals. Software is at the heart of our lives—the social networks and email and phone calls and instant messages that we exchange, the business-to-business electronic data interchange that keeps our supply chains running, the signaling data that keeps our roadways and airways and trains and power grids operating and cooperating. And, once again, software architecture lies at the heart of it all. These systems are simply too complex, too business-critical and life-critical to be designed in an ad hoc fashion.

A good example of architecture's modern role is how architecture is central to the creation of “edge-dominant” systems. An edge-dominant system is one that crucially on the inputs of users for its success. Think Wikipedia, Facebook, YouTube, Craigslist, Twitter, Flickr, and Pinterest. YouTube serves up approximately 1 billion videos a day. Twitter boasts that its users tweet 50 million times per day. Facebook recently announced that it has over 1 billion users and serves up about 30 billion pieces of content each day. Flickr recently announced that users had uploaded more than 6 billion photos. And edge-

From the author of



[Software Architecture in Practice, 3rd Edition](#)

[Learn More](#)

 [Buy](#)

dominant systems are not limited to open-content systems. Open-source software has exactly the same model of free contributions by “the crowd,” and it has resulted in some of the most important software in the world, standing at the heart of our business and information technology infrastructure: Linux and Apache run about two thirds of the world’s web servers. The Android operating system runs more smart phones than all other operating systems combined. MySQL, Eclipse, JBoss, Joomla, PHP, Firefox, FileZilla, OpenOffice, Hadoop, Wordpress, Twiki...the list of important open source applications is long and impressive and runs much of our personal and corporate lives.

Why do we care about this class of systems? Because the architectures of such systems have some important differences from the architectures that you would build for traditional systems. In the field of software architecture, we are fond of analogies with traditional architecture; designing a complex piece of software has much in common with designing a building. By way of contrast, we call this new class of systems “Metropolis” systems, in that they resemble a city more than a single building, with millions of stakeholders, continuous change, conflicting requirements, and no single planning authority.

The key architectural choice for a Metropolis edge-dominant system is the distinction between *core* and *edge*. That is, the architecture of these systems is, without fail, bifurcated into

- a core (or kernel) infrastructure that is designed by a small, coherent team and that defines the system’s basic structure, quality attributes and tradeoffs. This code is typically highly modular, slow to change, and robust.
- a set of edge functions or services that are built on the core. These deliver the majority of the function and end-user value and change relatively rapidly. And these edge functions are typically independent of each other.

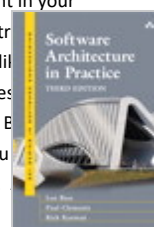
The core (often called a platform) is usually implemented as a set of services; complex platforms have hundreds of these. The functions and services at the edge may number in the thousands or even more (think of the number of apps for Android, the number of plug-ins for Firefox, or the number of applications and drivers for Linux). Without this key architectural structure, these systems would not be able to grow (seemingly) without bounds.

All our familiar software development life cycles—waterfall, Agile, iterative, or prototyping-based—are broken in an edge-dominant, crowdsourced world. These models all assume that requirements can be known; software is developed, tested, and released in planned increments; projects have dedicated finite resources; and management can “manage” these resources. None of these conditions is true in the Metropolis. Requirements emerge from the crowds; the software is perpetually changing, with no notion of a single stable state; project resources come and go with the whims of the crowds; and management can only influence but not control the contributors.

Another set of architectural technologies that have become important in recent years, and which look to be dominant for many years to come, is cloud-based systems. The cloud provides an elastic set of resources through the use of virtual machines, virtual networks, and virtual file systems. Cloud-based architectures provide enormous economies of scale for many businesses. In one sense the cloud is nothing new; we have had time-sharing computers that provided virtualized resources since the 1960s. But cloud-based systems deliver their services ubiquitously (and cheaply!), over the internet. And the various service models— Software as a Service, Platform as a Service, and Infrastructure as a Service—and deployment options—private cloud, public cloud, hybrid cloud, community cloud—have added a new set of design options (and tradeoffs) for today’s software architect.

The point of these examples is twofold: 1) architecture keeps changing, and you need to stay on top of the latest technologies and their underpinnings to stay relevant in your organization; 2) architecture is architecture (apologies to Gertrude Stein). Did we just contradict ourselves? What we mean is that while technologies like edge-dominant systems change and bring with them enormous consequences, the fundamentals behind these technologies do *not* change. This is good news! By understanding the design primitives of architecture and the way in which you document, plan, and analyze these artifacts, you can master this critical skill. on!

From the author of



[Software Architecture in Practice, 3rd Edition](#)

[Learn More](#)

[Buy](#)

From the author of



Software Architecture in Practice, 3rd Edition

[Learn More](#)

[Buy](#)