Piecing It All Together

Flexible architecture and a small-team approach are the keys to the success of large IT initiatives.

by Oleg Sadykhov

Large information technology programs are multiyear initiatives with large teams—more than 20 to 30 people, often much larger—and budgets in eight figures. Examples include an implementation of a core system such as policy administration, claims or billing, or a large data-analytics project.

The problem with large initiatives is that delivering them in a reasonable amount of time requires large teams, and large teams present large challenges such as:

- **Communication.** Smaller groups can communicate more efficiently and directly, but large teams require more communication and documentation that increase project overhead.
- **Inefficient use of resources.** Team members either wait for one another to accomplish tasks they depend on or inadvertently break each other's functionality, which introduces rework.
- **Uneven skills.** More-capable people can take on larger, more complex tasks, but it's very difficult to assemble a team of all-stars.
- **Ramp-up and knowledge transfer.** Building the initial team, addressing turnover and other team configuration changes create the need for new people to quickly get up to speed. On large projects this is even more challenging due to the size and complexity of solutions.

Because issues with large teams are well known, many modern software development methodologies (especially agile) stipulate that teams should be small. For example, in Scrum, an agile methodology, teams are typically cross-functional groups of seven or so. So why can't we break a large initiative down and create a set of small teams that works almost independently and efficiently and ultimately assembles their work products into the overall solution?

Of course, everybody tries to break large teams into sub-teams (also called work streams, tracks and the like), but very rarely is it done well, where a variety of challenges, including communication and inefficient use of resources, are truly addressed.

This is where architecture comes in. We're not talking about selection of Java versus .NET or what integration technology or what types of servers and networks to use—though these topics are also important.

Key Points

- **The Story:** Proper system architecture can help break down large, expensive multiyear tech initiatives.
- **The Background:** The idea is to implement small-team strategies that overcome communication problems and duplication of effort.
- **The Way Ahead:** Newer approaches such as CQRS can help in architecting and delivering large and complex enterprise initiatives successfully.

Instead, we're talking about architecture as decomposition: how big systems must be broken down into smaller pieces that will lend themselves well to independent work by teams. These pieces must be autonomous and loosely coupled.

The task is not easy, but it's important since without a good breakdown of systems into parts (variously called subsystems, modules, services or bounded contexts; we'll use subsystems to avoid confusion) there will be no good breakdown of large teams into efficient and independent sub-teams. It's natural for sub-teams to form around subsystems.

The stakes are even higher than just the team breakdown. If the system is not well-architected, it will result in a complex and inflexible solution. Cost overruns and major failures are inevitable. Every project

Contributor Oleg Sadykhov is a principal of X by 2, a technology consulting firm based in Farmington Hills, Mich. He may be reached at osadykhov@xby2.com

manager who has dealt with big systems has hit the "wall." Suddenly, development tasks that used to take a week to start take a month and it's not clear why. This is a direct result of increasing complexity and insufficient architecture.

If the parts of the overall solution overlap, then the teams responsible for these components will end up performing similar tasks. This leads to duplication of effort, inconsistent solutions and similar defects that show up in various parts of the system that seem elusive and hard to eliminate.

Project teams will underperform if the architecture is poor. Inefficient communication paths between teams will start dragging the initiative down. Teams will be mired in coordination and dependency nightmares and spend their valuable time adjusting to the work performed by others.

Creating Good Architecture

Let's take a simplified example to illustrate the challenges. Assume that we're working on a policy administration system where we store and maintain insurance policies. Further assume that customer information is also stored and updated by the same system.

Let's say that we want to break our system down into subsystems, and a logical choice is to separate the functionality related to policies from the functionality related to customers. So we decide to create a policy subsystem that will deal with policy information and a customer subsystem that will deal with customers.

A good architecture has subsystems that are autonomous and loosely coupled. This means we have the ability to make and release changes to subsystems independently. In our example, policy and customer subsystems should be able to grow and evolve independently of each other.

A typical system consists of user interface components, business logic components and data. When we break the system down into subsystems, we need to consider each of the layers listed above. In order to achieve the independence of subsystems we are striving for, we must deal with all of the layers (UI, business and data) and not just the business components.

Separation of the business components is probably the easiest task (though still not easy) and as such it's the only thing that is typically attempted. Take for example many implementations of SOA where there is an incorrect belief that by exposing business components as Web services, we achieve good architecture, which implies loose coupling and autonomy.

There is no hard-and-fast rule to say when data should be centralized and when replication makes sense.

But if both the customer and the policy subsystems continue to share the same underlying database, then when we work on customer functionality we can inadvertently break policy. Therefore, when we release one, we must test the other as well. And when the database is down, then both subsystems are down. So where is the autonomy?

The same can be said of the UI. It's natural for the same screen to present information about both policies and customers, but the way such screens are often built further entangles the two subsystems.

To summarize: The challenge of architecture is to keep the criteria of autonomy and loose coupling of subsystems very clearly in mind and tackle the difficult issues such as data separation and disentanglement of user interface to accomplish it.

Data Replication

So what's the problem with breaking the data down and letting the subsystems own their data and avoiding one large database that supports it all?

The problem is that the policy subsystem legitimately needs customer information, and the customer subsystem may also need to know information about the policies the customer owns.

In other words, the data needs to be shared, and we have a couple of choices for accomplishing that:

- Store data in one place and provide access to it. In this scenario customer data is truly owned and only accessible by the customer subsystem. To share data, the customer subsystem can either expose Web services or provide UI screens or widgets for others to use.
- Allow partial replication of data. In this scenario we allow replication of some of the customer data elements so the policy subsystem can store it in its own database.

The idea of storing data in one place initially looks like a winner, but after a deeper look, it presents several challenges. In many data-intensive situations, centralized data access presents a performance challenge. Many off-the-shelf packages expect data they need to be stored locally in the application database (this is called a replication scenario). Some of the popular enterprise subsystems will be difficult to scale, since everyone will need the same data. They also will be hard to change without affecting everyone. Downtime of one subsystem will lead to downtime of others that depend on it.

The bottom line is that data replication is an important tool to achieve our goals of autonomy and loose coupling of subsystems.

There is no hard-and-fast rule to say when data should be centralized and when replication makes sense. To decide, think about the amount of data in play, how much data processing is done by the subsystem that owns the data, how impactful the downtime of one subsystem will be on another, and so on.

Of course, the moment we start down the path of data replication is the moment we need a good framework to think about data ownership and rules we impose on our subsystems. One such framework is master data management, which is a tried-and-true way to track owners of data and the rules that should be followed.

CQRS Aids Architecture

Another relatively recent framework that helps one think about separation between subsystems, data ownership and much more is called Command and Query Responsibility Separation. The idea is fairly simple: It advocates separation of data that supports modifications of the system from data that supports inquiries.

According to CQRS, systems should have two parts—one for data modifications and the other for data inquiries—each with their own databases.

Even without fully following CQRS, these concepts introduced by CQRS are helpful when working on and discussing architecture like:

- **Commands.** Requests to modify data (that is, to create a new customer). Their names should be in the present tense and sound like commands. Their processing involves data validation, execution of business rules and changing of data.
- **Events.** Notifications about data changes (such as, new customer created). Names of events should use past tense to indicate that they're just a notification of a change that's already occurred.

Events notify interested parties that a data change has been approved and recorded. They're typically used to trigger other processing or to update decentralized data replicas.

- **Queries.** Read-only requests to access data can be a search or access to specific data record details.

Here's how the CQRS approach can be applied to our example of policy and customer subsystems. Policy and customer subsystems will "own" their data and be responsible for processing commands to change their data. Upon successful changes of data, they will publish events notifying others.

We will also create a new subsystem—operational data store—that will combine data from policy and customer components in a way that is optimized for inquiries. ODS will change its data in response to events published by policy and customer subsystems.

Now, to build a solution that involves interactions with policy and customer subsystems we can use ODS to fulfill inquiries, but when we need to make changes we will send commands to the respective subsystems (policy and customer) to process them.

To summarize: Software architects should consider newer approaches such as CQRS when working through challenges of decomposition of their systems. The goal is to achieve autonomy and loose coupling of subsystems.

If they succeed, they can pave the way for small, independent and efficient teams that have a much better chance to deliver large and complex enterprise initiatives successfully. BR