Scrum.org WHITEPAPERS

YOU CAN'T SCALE ENTERPRISE AGILITY WITHOUT ARCHITECTURE

KURT BITTNER AND PIERRE PUREUR | AUGUST 2022

Key Take-Aways

- In order to scale enterprise agility, you need to enable teams to work as independently as possible by employing a platform approach. Cross-team dependencies kill agility.
- Products need to be small, which means that they should be viewed as evolving MVPs focused on specific customer groups with common sets of unmet needs.
- The platform also has an MVA that evolves - applications running on the platform will inform what the platform needs to do, and apps will continually be refactored to add to the platform
- Internal open source practices are important to building and growing platforms - you can't have a platform team/organization without amplifying dependencies
- Automated testing is essential to creating a stable platform
- Versioning of services is crucial to prevent breaking existing applications

Scaling agility means different things to different organizations. For some, it's simply getting a lot of teams using empiricism to make better decisions by using fast feedback loops. While this can be challenging, the difficulties mostly relate to growing enough high-performing, self-managing teams in a consistently repeatable way. All organizations have this challenge.

But some organizations have a further challenge: finding a way to get these high-performing, self-managing teams to work together on large and complex products, usually ones that existed long before the organization had its sights on using empiricism to solve complex problems. These organizations often assume that in order to deliver large complex products they will have to, to a large degree, dilute team autonomy and resort to a more top-down, planned way of working. Unfortunately, these top-down approaches to scaling agility often dilute team autonomy and lengthen (or even largely eliminate) fast feedback loops so much that they become mere agile façades overlaying a traditional program management approach.

Scaling agility and gaining the benefits of team autonomy and fast empirical feedback loops often means that organizations need to adopt a different approach to the way their products are defined and built, including the way they are architected. The reason is that cross-team dependencies kill autonomy, introduce delays, and waste effort as **teams invest their effort in things that they might, if they obtained faster feedback, realize have no value**.

To achieve fast empirical feedback loops, organizations need to break down large products into smaller, more independent products, but in doing so they must keep those products connected by uniting them with a common underlying platform architecture. In a sense, the large complex product is broken down into a large number of related but independent products united by this common platform. Figure 1 illustrates visually how this works and contrasts with the traditional approach.

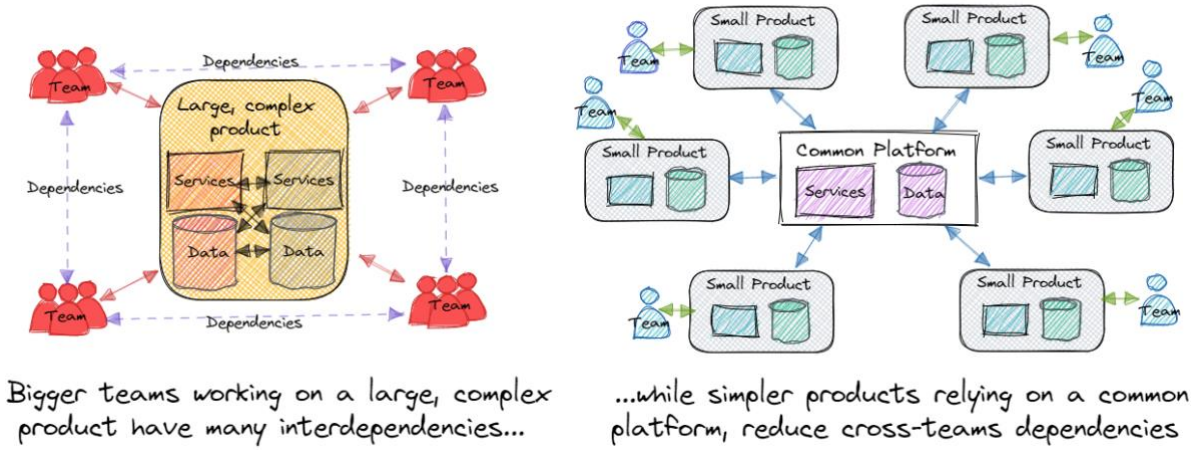

Figure 1: Simpler products based on a common platform reduce cross-team dependencies

Development challenges for large applications

Traditional organizations are typically structured as a set of silos, each of which has a particular specialty. Teams in these organizations can be cross-functional, but often the people are assigned to more than one team at a time. When part-time team membership is the norm, products can take a very long time to develop because of scheduling challenges.

When products become too large for a single team to develop, traditional organizations break the work down using components, whose teams are often determined by the technical expertise the team needs to work on the component. This leads to skill specialization across teams, which further complicates cross-team communication. This pattern was described a long time ago in a **paper by Melvin Conway**, and is sometimes referred to as "Conway's Law", in which the structure of an application tends to mirror the structure of the organization that produced it.

Team specialization also introduces an application integration problem since some team now needs to assemble the components into a running application. It also decreases focus on customers, since most teams only work on their components, not the application as a whole. The integration problems and further delays caused by having separate component teams often results in problems that aren't addressed, or even discovered, until late in the application delivery cycle, and the lack of customer focus of most teams means that components may work fine on their own but can't be integrated into a working customer product. When faced with critical deadlines, late-cycle integration problems often result in acrimonious finger pointing. Sounds familiar?

Breaking large and complex products into simpler, independent products

The first step toward breaking these dependencies is to create more, yet smaller products, each of which does less for its customers or users. For many in large organizations, this is going to sound like a step backward toward anarchy; here's why it's essential:

Let's consider an application developed by an insurance company that handles homeowners insurance. This application is very large, because the number of different customer risk situations that it needs to deal with are immense. Every possible variation that a customer's situation could introduce has to be considered, and any time a new variation is discovered, the application has to be modified. Of course it does!

Or does it? There is nothing that says that a single application needs to handle all possible customer variations. What if there was a much smaller application that dealt with a particular group of customers and their specific issues? We see this all the time on our mobile devices, where we have different apps for finding a person's house on a map versus finding a place to charge our electric vehicle. Would it be better to have one mapping app that did everything? Maybe, but you may have to wait a very long time for it to become available, and it still might not do everything you need.

So let's say that we break down a large monolithic homeowners application into separate applications for people who rent, people who own a condominium in a multi-unit building, and people who own a single-family home. Each of these types of customers has different risk profiles and different needs for coverages. Breaking a big application into smaller ones gives us a better chance for a single small team to be able to develop an application, and it also results in bringing teams closer to their customers. We could further break the applications down by states since insurance law is governed by state insurance commissions and laws vary greatly from one state to another. Each team might own one or more of these smaller applications, but they would be freer to meet the needs of their customers.

A product is really just a vehicle for delivering outcomes, and in so doing, testing assumptions about what customers find valuable. Large and complex products with many different kinds of customers/users make testing assumptions about value more difficult because they lack focus. A large product may really be simply an agglomeration of lots of different customer needs, usually one that has lost focus on its customers. **Breaking down large, complex products into smaller, simpler products** makes things easier for the teams that develop the products by reducing the scope of their concerns, giving them a better chance at being able to form a self-managing team that can benefit from fast feedback loops.

From the perspective of the organization, however, having lots of small, independent products usually triggers fears of certain anarchy and imminent chaos. Ultimately all these applications need to work together and integrate with each other to support a broader business. How can that happen with many small teams working independently of each other?

The other problem that each team faces is that, because they are working on similar problems, they are likely to have to develop a lot of the same base functionality. The result, most people will say, is going to be a lot of duplicated effort, or worse: different solutions to the same problems.

These two issues usually lead organizations to retreat back to a more centrally planned approach that, regardless of whether they are called value streams or release trains, have most of the characteristics of a traditional program management approach: long delivery cycles and delayed, and sometimes non-existent, feedback loops. These organizations end up achieving few of the benefits they were looking for from agility.

Simpler products need a common platform

The solution to these problems, lack of common solutions to shared problems, and lack of cooperation between disparate applications, is to use a common platform as a foundation for the applications. A

platform is an aggregation of services that solves problems in a particular technical and/or business domain.

In a previous article, we discussed the benefits and pitfalls of frameworks. Platforms tend to encompass many different frameworks, and while frameworks tend to be domain-independent, platforms tend to be domain specific. In the context we are using here, a framework would be used by many organizations, while a platform is organization-specific for reasons we will discuss later. Think of a platform as a domain-specific set of architecture patterns, practices, and tactics, supported by shared components and **frameworks**. They are "a common way of doing things across a problem domain".

As with frameworks, platforms make decisions for an application development team. This provides benefits as they don't need to revisit decisions that have already been made by the organization, but the team does need to understand the choices a platform will make for them if they use it.

A platform that supports a homeowner's insurance domain would include all the basic code for issuing new or updated policies and coverages, but would also contain common solutions for security, transaction consistency, regulatory compliance, and so forth, because every team does not need to be burdened with having to develop the functionality common to all applications.

As we explained in a **previous article**, a conscious architectural focus is necessary to meet the quality attribute requirements associated with the delivery of larger products. Being able to develop new applications and rapidly deliver smaller products using a common platform that provides a **Minimum Viable Architecture (MVA)** for those applications, provides significant benefits to achieving fast feedback loops.

The common platform also has an MVA that evolves based on the feedback loop provided by the applications that use it. Applications running on the platform inform what the platform needs to do, and apps are continually refactored to add to the common functionality of the platform, therefore benefitting other apps that use the same platform foundation.

Getting started with the common platform

How do the teams get started with building a common platform? Ideally, they would start simultaneously from a blank slate, and would not need to reuse any existing functionality already implemented in legacy systems. Each team would design an MVA for their product, and they would agree upon a common platform architecture definition by gathering the common elements between the MVAs.

In reality, this won't often happen in most cases. One team is likely to be ahead of the others, and that team will need to identify which elements of their MVA belong to the common platform architecture, without enough knowledge (if any) about the other products using the platform. In this situation, it is best not to over-design the common platform MVA, and to keep in mind Continuous Architecture Principle 3: Delay design decisions until absolutely necessary.

It is also likely that product teams will be directed to reuse some existing functionality from a legacy system, in an attempt to reduce costs and deliver a product faster. Going back to the homeowners policy administration system that we mentioned earlier in this article, let's say that we want to deliver a product to cater for the insurance needs of people who rent luxury condominium units in expensive beachfront locations. It is likely that some of the "common" functionality needed by the new product already exists in a legacy homeowner policy administration system, and that functionality will either

need to be extracted from the legacy system and ported to the common platform, or be temporarily invoked as a service from the new platform, assuming that this functionality can be accessed via an API - see "**Chipping away at the monolith: applying MVPs and MVAs to legacy applications**" article. Either way, products relying on the common platform must be isolated from the way this functionality is implemented. This is especially important if the team decides to initially access the functionality via an API, and ports it to the common platform at a later date.

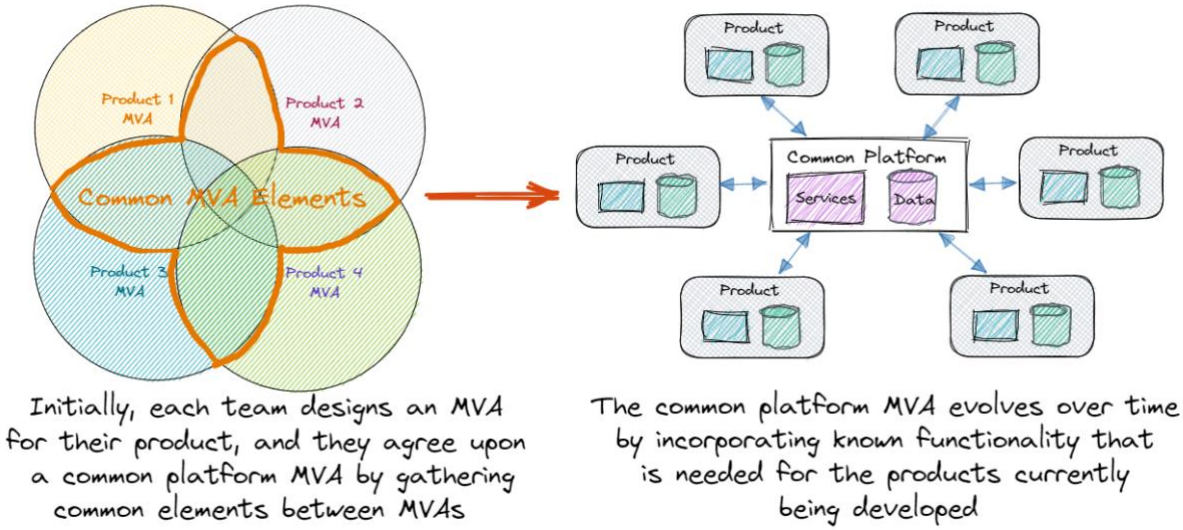

Figure 2: A common platform MVA evolves from common elements between individual product MVAs

Once the initial MVA for the common platform has been agreed upon, the team has a decision to make in each Sprint: do they add new functionality to the product, or to the platform? The challenge here is that functionality that might be needed for other products should become part of the platform, but you don't know what other products might need yet. As we mentioned in a previous **article**, "The MVA [...] consists of a minimal set of technical decisions that are tested and evolved using empiricism over time. These decisions are complemented by a minimal set of architectural practices that help the team to keep the product architecturally viable while they evolve it". Following this approach, it is best not to over-design the common platform and only incorporate known functionality that is needed for the products currently being developed.

Having a Common Platform does not mean having a Common Platform Team

Traditional organizations are used to having things like a shared common platform owned by a dedicated team. This isn't necessary and it actually worsens the problem of cross-team dependencies.

Introducing a common platform team centralizes the dependencies that applications have on the platform and focuses them on the team that would own the platform. This creates a single-team bottleneck on any shared functionality that teams need from the platform. When this happens, several things can occur:

- The common platform team becomes a bottleneck, so that everyone is slowed down to their speed

- To manage priorities, the common platform team creates a complicated requirements gathering process that become a source of new bottlenecks
- Application teams, frustrated by the bottlenecks they experience, decide to develop their own services that really should be common but the teams can't wait
- The resulting redundancies create loads of technical debt that usually is never resolved and, as it must be maintained and improved over time, acts as a permanent drag on application team effectiveness.

The solution to this problem is a concept called **innersourcing**, which is similar to open source development but open to teams only within the same company, potentially expanded to partners and contractors. In short, any of the application development teams should be able to add or modify platform code, so long as they do not break another application. This means that every code change must be verified to not break a service interface or cause an application to fail to meet a QAR that it has previously satisfied. The only practical way to achieve this is to have a robust set of platform-related tests built into an automated continuous delivery pipeline.

The most important things these tests verify is that service interfaces are not broken by changes to the platform. When a platform service needs to change, the team making the change should create a new version of the service. Applications using older versions of the service can migrate to newer versions when it fits within their priorities. Ensuring that service interfaces don't change protects applications from platform changes that might otherwise break the application.

The second kind of automated test looks at QARs and ensures that changes to the platform don't break QAR commitments that the platform has implicitly made to applications that use the platform. Changes made to the platform should improve QAR performance, not degrade it.

Teams developing applications are then faced with a choice for every service they develop: "is this service unique to our application, or could it be useful (if generalized) to other teams". The best way to answer this question is usually to ask the other teams if they need something like the service. If the answer is 'yes', it belongs in the platform. Other teams might have already developed something similar that they have not yet contributed to the platform.

An important consideration in answering this question is another question: "do you need it now?" Platforms can become clogged with hypothetically sharable services that are, in fact, never actually shared. If no other teams need the service now, the team about to develop it might want to wait until another team needs it before trying to develop something general-purpose. Sometimes it's better to develop something usable by one team and learn some things about what worked and what didn't, before trying to make it reusable by many.

In one sense, the architecture of the common platform is self-organizing, driven by the needs of the team it serves, but it is not wholly self-organizing and needs frequent review and pruning. Periodic architectural reviews focused on refactoring and evaluating actual reuse can help with decisions about whether to put things into the platform. Similar services that show up in many applications are good candidates for refactoring into the common platform. Likewise, services that were contributed to the common platform by one team that are used by no other teams are candidates for moving back into the application. These reviews are best conducted by the teams that are actually using the platform, sometimes assisted by developer-architects who can look across the needs of many teams.

How platforms help scaling

When teams use a platform, they get the benefit of sharing work with other teams without having to wait on those teams to get around to having time to help. If a team needs to solve a problem now, they are not blocked from doing it because some other team "owns" that part of the solution. And when they do solve a problem, other teams benefit from that solution. As a result, teams are freed from waiting for other teams while still benefiting from solutions that other teams create.

While an innersourced platform can sound like it allows anyone to do anything, potentially breaking lots of applications in the process, a tightly controlled automated continuous delivery approach that employs stringent and robust automated testing to make sure that changes don't break other applications actually provides more control that the typical approach where teams own components but lack stringent automated testing. Manual oversight is slow and inconsistent and fails to deliver the kind of results that organizations seeking agility need.

A platform approach can also solve a major problem that organizations using third-party and open source frameworks face: project funding models don't provide for ongoing upgrades of frameworks to newer versions. Among other problems, this creates security vulnerabilities when older versions of a framework have known security holes that can be exploited by malicious parties. A platform approach means that third-party frameworks only need to be upgraded and tested in one place, not in many different applications.

Platforms also improve the sustainability of applications that use the platform by reflecting the experiences of all the teams that contribute to the platform. Individual application teams don't have to figure out the best way to solve the problem if many other teams have already had experience solving it, and if a team encounters a new problem that other teams share, they can be confident that the solution will improve over time, guided by experience.

This helps the organization to evolve the MVAs for applications - each team can do just what it needs for the moment, but over time, as teams mature their applications and the associated MVAs based on feedback, they will also improve the parts of the platform that support the applications based on that same feedback.

Conclusion

Cross-team dependencies kill enterprise agility. When teams have to wait on other teams, they lose momentum and the organization loses the benefits of agility. A common shared platform removes many of these cross-team dependencies, but only if managed using an innersource approach, coupled with a robust automated testing strategy that ensures that platform changes don't break existing applications.

When platform services do need to change, teams should create a new version rather than changing an existing service so that existing applications that use the service won't be broken; application teams can upgrade to the new version when it fits into their priorities.

Platforms also provide an environment that encourages and enables organizations to make their products smaller and more independent by focusing on a narrower set of customer needs. This helps the organization serve those customers faster by reducing complexity and removing cross-team dependencies.

Finally, platforms have their own MVAs; a group of teams does not need to step back and spend lots of time on an upfront platform design before they can get started, they just need to identify a small set of common services that they need and that they believe other teams will need as well. In this sense, the MVA of the platform evolves alongside the MVA of the application.