Software Platforms
10 Lessons

EXECUTIVE SUMMARY

Software platforms are incredibly powerful multipliers of R&D and strategy. Platforms are no longer just about reuse and efficiency. Platforms are key for enabling DevOps in your organization and critical for building an ecosystem on top of your portfolio.

Platforms may seem intuitively easy, but industrial practice shows that many mistakes are made that eliminate the potential benefits. Confusion about the strategic goal of the platform, poor architecture choices, accumulating technical debt, crushing complexity and drowning in commodity functionality are real and concrete dangers that companies need to avoid to effectively work with platforms

We discuss 10 key lessons for success with platforms, including focusing on speed, the superset platform approach, the three layer product model, variability management, handling technical debt and instrumentation.

TABLE OF CONTENTS

SOFTWARE PLATFORMS

In the field of software, few topics are discussed with such energy and passion as platforms. There are incredibly strong proponents and vocal detractors, lots of misunderstanding and even more outdated beliefs about platforms.

Having worked with platforms for the better part of three decades, I have seen the notion of platforms evolve over time and today we can distinguish at least three main goals that companies have for platforms, i.e. platforms for reuse, platforms for DevOps and platforms for an ecosystem.

The platform for reuse was the original proposition of software product lines. The idea was that a portfolio of products contains significant common functionality and we should bring all the common functionality together in a platform that the product teams can use to build their products faster and more efficiently. The perceived benefits of sharing resources and lower cost are certainly real, but in practice it turns out that there are lots and lots of ways to negate all the benefits of platforms. This especially happens in companies where the leadership does not understand how software platforms are fundamentally different from mechanical or electronics platforms.

The use of platforms for DevOps is a newer phenomenon driven by the increasing cost of manual configuration and customization of products built on top of a platform when releasing the product software more frequently. We can accept significant manual configuration, customization and testing effort when we release software once or twice per year, but if we push it out every month or more often, this becomes unacceptable. Platforms for DevOps include the superset of functionality of the entire product portfolio and each product is basically a subset created through automated configuration and test. That allows companies to frequently release new versions of the software without the associated release cost.

Third, the holy grail of platforms is to provide a software platform to a two- or multi-sided market where 3rd parties develop solutions on top of the platform and customers of the platform enjoy a rich ecosystem of solutions that increases value and stickiness significantly, making it virtually impossible to dislodge the platform provider from the comfortable perch. The challenge is that achieving this position is incredibly hard, even if the destination is very attractive.

Unfortunately, in my experience, most companies have a rather confused and contradictory strategy around platforms that is a mix of all three aforementioned types of platforms. In this post, I intend to share some of my learnings after working with platforms for many moons that I hope provides some benefit and informs the discussion. So, here goes ...

1. **Platforms should focus on speed, not efficiency**: Traditional thinking is that platforms are about efficiency. Product teams get a bunch of functionality for free from the platform and only have to build the remaining product specific functionality. In practice, the interface between products and the underlying platform tends to be incredibly complex and tends to result in a situation where the platform slows everyone down. To address this, we should design platforms and the ways of working around platforms to maximize speed. Speed to get new functionality out to customers. Speed to extend the platform with functionality required by products. Every activity and use case should be designed for speed. When doing so, interestingly, efficiency will follow.
2. **Avoid the platform/product dichotomy**: Over the last years, I have increasingly become convinced that for most companies, the best approach is to create a platform that contains the superset of all functionality in the product portfolio. Each product can then be derived from the platform using automated configuration. This avoid the tension and frustration that many organizations experienced, trying to align product and platform teams, deciding where to build functionality and struggling with increasing the frequency of release.
3. **Balance architecture and continuous integration/test**: A clean architecture with strong interfaces and decoupled functionality is great in that it simplifies testing because you can push most testing to the component and subsystem level and minimize system level testing. In practice, however, the architecture is always suffering from architectural technical debt and no matter what decomposition you choose, there will always be functionality and quality requirements that have cross cutting consequences. Consequently, we need to balance architecture work and investment in continuous integration and test. The architecture helps engineers avoid mistakes in the first place and continuous integration and test catches introduced errors early so that the cost for fixing these is minimal.
4. **Don't integrate new functionality too quickly**: Every product and platform requires innovation to stay relevant. Unfortunately, the nature of innovation is that many, if not the majority, of innovations fail to deliver the expected outcomes and should not be introduced into the platform or product. The good news is that the few innovations that are successful more than pay for all the failed ones. As most innovations fail, we should be careful to incorporate new functionality directly into the platform or product. Instead, it is better to first experiment and confirm with customers that the functionality adds value by "bolting on" the innovative functionality and keeping it at the other side of an API. If the innovation proves successful and a hit with customers, we can then plan the deep integration in the product or platform. If it isn't successful, we can easily ditch the functionality as it never got integrated in the first place. This avoids bloated products and platforms that contain tons of non-value adding functionality.
5. **Distinguish customer-unique and customer-first functionality**: There will always be functionality requested by customers. Some of this functionality will be useful for more customers and the customer initially requesting it was simply the first to ask for it. Other functionality is so specific to the unique context of the requesting customer that it will never be used by others. Although it is fine to incorporate customer-first functionality into the platform and then generalize it to other customers over time, we should be very careful to include customer-unique functionality. Ideally, we should provide that functionality outside

the platform, using an API, or decline the customer request. If we do decide to incorporate it, we should ensure that it makes financial sense as the majority of costs related to software are accrued after the initial development of the functionality.

6. **Control platform variability**: Most of the platforms I have been involved in have thousands, tens of thousands or even more than a hundred thousand variation points. The complexity resulting from this is phenomenal and it causes significant extraneous costs when adding features and when testing the platform. Many variation points have dependencies on other variation points, resulting in a combinatorial explosion of configurations that often cause post-deployment issues as it is impossible to test all variations. Platform variability needs to be carefully controlled, meaning that variation points that no longer provide sufficient business value need to be removed and new variation points introduced only when the business case, including all the secondary cost, is significant.
7. **Constantly optimize commodity for TCO**: Platforms have three layers of functionality, i.e. innovative and experimental functionality, differentiating functionality and commodity functionality. The latter category tends to constitute the vast majority and tends to gobble up 80-90% of the R&D resources. In order to ensure we can invest sufficient R&D resources in innovation and differentiation, we need to continuously look for ways to reduce total cost of ownership of commodity functionality. We can do this by replace bespoke functionality with commercial or open source components, simply removing functionality, freezing functionality, meaning that we stop accepting change requests, etc.
8. **Instrument your platform for data-driven decisions**: Edwards Demming, the American who helped Japan rebuild itself after the 2nd world war, famously said: In God we trust; all others must bring data. This is still a lesson most companies have not fully incorporated. Also around platforms, there are many decisions that get taken without much evidence, purely based on beliefs and earlier experiences by key decision makers. One reason is that often the data is not available and hard to collect. For instance, most companies that I work with can not answer basic questions concerning feature usage in their platform. How do you prioritize R&D resources if you don't even know whether the features you already have built are even used? Hence, platforms need to be instrumented to facilitate data-driven decisions and should be easily extensible with new instrumentation when required.
9. **Be careful to open up to 3rd parties**: Every platform company I have worked with would love to open up their platform for third parties and get "free" functionality extensions. In my experience, if something sounds too good to be true, it typically is and this is no exception. 3rd parties often put constraints on the platform such as requiring interface stability, requests for functionality that helps them, rather than your customer, etc. Also, the 3rd party has an inherent ambition to build differentiating functionality themselves and push the platform into commodity as much as possible. Consequently, opening up should be based on a carefully crafted strategy and initially be driven by experiments that allow you to shut down the initiative if the consequences are not as desired.
10. **One ecosystem platform stakeholder at a time**: Ecosystem platforms serve, by definition, two- or multi-sided markets. Getting a multi-sided market to the ignition point where the ecosystem fuels itself without constant investment by the platform provider is extremely demanding and expensive. Rather than trying to build all sides of the market simultaneously, the better approach is to build the market one stakeholder category at a time. In practice, this means first building up a customer base of sufficient proportions based purely on the platform functionality and only then opening up to building up a second stakeholder category.

PLATFORMS SHOULD FOCUS ON SPEED, NOT EFFICIENCY

Traditional thinking is that platforms are about efficiency through reuse. Product teams get a bunch of functionality for free from the platform and only have to build the remaining product-specific functionality. The interesting thing about software reuse is that it's been extremely successful at the inter-company level. The amount of software that's being reused through operating systems, commercial software products, the open-source community and software ecosystems is phenomenal. Compared to the situation in the 1990s, when I came of age in the software community, it's incredible what we've accomplished.

For many companies, however, the sharing and reuse of software developed in the context of the company itself, ie intra-organizational, still proves to be a challenging and uphill battle. The conceptual idea is to develop a platform that all product teams can use to build their respective products. The reasoning is that by focusing on building common functionality only once, we can save development effort by sharing this common functionality through a platform.

In practice, unfortunately, the interface between products and the underlying platform tends to be incredibly complex and tends to result in a situation where the platform slows everyone down. There are several reasons for this, but the main one is that software is different from mechanics. Although the functionality in the platform might be common, very often product teams find out that they need a different flavor or that some part is missing. The product teams need to request this functionality to be developed by the platform team, which often gets overwhelmed by all the requests. The consequence is that everyone waits for the slow-moving platform team.

This is a specific instance of a generic pattern I see time and again: when companies focus on efficiency, the consequence tends to be that everything slows down. Whether it's approval processes, bottlenecks, reviews or coordination problems, the behavior in service of efficiency tends to put hurdles and delays in the ways of working, causing delays everywhere.

To address this, we should design platforms and the ways of working around platforms to maximize speed. Speed to get new functionality out to customers. Speed to extend the platform with the functionality required by products. Every activity and use case should be designed for speed. To achieve this, there are at least three strategies I've seen companies use in practice.

First, make the use of the platform optional. In many traditional platform setups, product teams are forced to use the platform and this tends to make the platform team prioritize its own efficiency over the efficiency of the product teams. By making the platform optional to use by product teams, the priorities of the platform team tend to change in favor of serving the product teams.

Second, allow product teams to change the platform code. In traditional software development, only the team responsible for a software asset can make changes in it. However, with the increasing adoption of continuous integration and test, it's much less risky to let others touch the code as their mistakes are captured immediately. As platform teams tend to be overloaded with change requests, allowing the product teams to make the changes in the platform code that they urgently need can remove the bottleneck in development.

Third, it's possible to do away with the product/platform dichotomy altogether by creating a configurable product base where the superset of all functionality is in one code base and each product simply becomes a configuration of that superset. This will be the topic of the next lesson.

When companies focus on speed, interestingly, efficiency is a side effect. To gain speed, teams tend to look for ways to automate and in other ways reduce the amount of time needed to accomplish various outcomes. The consequence is a significant increase in efficiency. The takeaway is that focusing on efficiency slows you down and focusing on speed makes you more efficient! So, focus on speed and get everything else as part of the deal.

AVOID THE PLATFORM/PRODUCT DICHOTOMY

The traditional view on platforms is one where the platform provides generic functionality used by multiple products in a portfolio and family. These products can then simply take the platform as a lego brick and build their product-specific functionality on top of it. Like software from external providers, you simply integrate the platform as a component in your product and you're good to go.

The reality is rather different. As the platform is company-internal software, it's much easier for the product and platform teams to build significant interactions and dependencies that are beneficial in the short term but easily increase coordination costs in the long run. Also, external software is typically not concerned with the application domain in which you're operating whereas the platform is – deeply. That causes deeper interfaces between platforms and products through variation points, extension interfaces, optional components, and so on.

The result can easily be a situation where an overloaded platform team, receiving requests from multiple product teams, deprioritizes functionality until a product team is ready to partner on the development and the interface between product and platform can be established collaboratively. This then delays the introduction of innovative functionality that products desperately need but the platform team doesn't prioritize.

This is based on a fundamental implication of platforms: you need to decide on the scope of the platform and draw a line between the platform and the rest of the software assets. In many ways, this is an artificial line from a company perspective as you need the functionality anyway and it doesn't really matter where it lives. It can be in the platform with the advantage that every product has access to it, but it may require more R&D effort as the functionality may need to be constructed more

generically. It may be implemented on the product side, which may be faster but with the disadvantage that other products may need the same functionality later.

The boundary between the platforms and the products built on top of them isn't a static one but is continuously floating, shifting and evolving. It's also subject to company politics as product teams have a vested interest in having the platform team take on functionality on its backlog as they'll get it 'for free.' This causes the platform team to be wary of requests by their customers, the product teams, as they never know if it's real generic functionality that this product team just happens to be the first to request (a customer-first feature) or if it's a 'customer-unique feature' that only this product will require. The first is good to incorporate in the platform, even if only one product uses it for now, whereas the latter should stay in the product-specific code. The amount of time that can be spent on these questions can be quite phenomenal.

A second challenge with maintaining the boundaries between products and the platform is the rigidity of resource allocation. The platform and each of the products will have a team associated with them and the resources can't quickly be moved around. So, if one of the products or the platform has high-priority requirements to meet, the team is on its own and can't easily get help from the other teams.

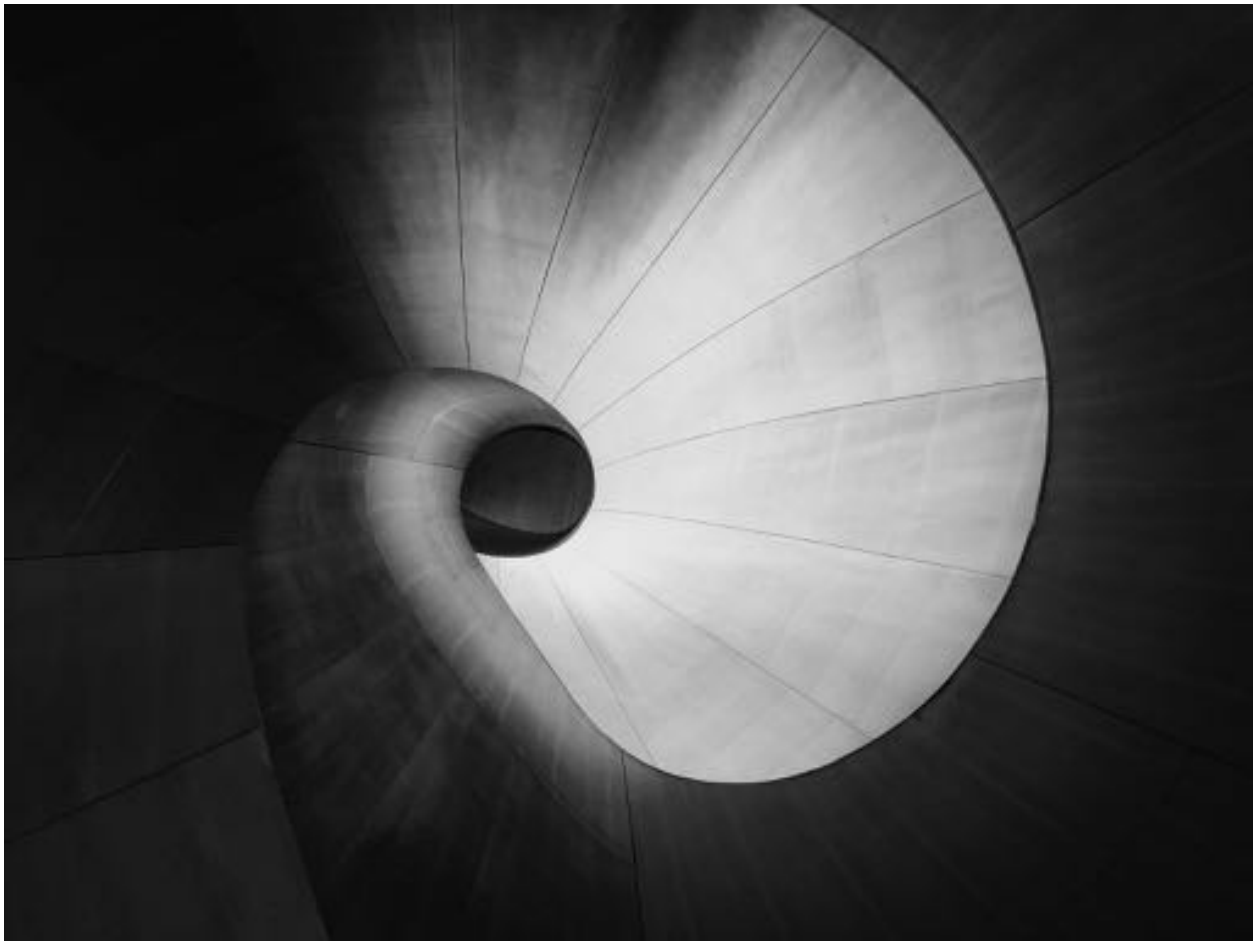
The third challenge is that although many R&D teams have worked hard to develop a build and test pipeline for themselves, it often is hard to include the platform in this infrastructure. The consequence is that moving towards DevOps or even just more frequent releases is more difficult as the latest versions of the platform and the product software tend to be integrated late in the process, leaving most of the testing until the integration and release testing stages. As the interface between the product and platform tends to be wide and complex, this typically results in many issues being found late in the process.

To address these challenges, over the last years, I've increasingly become convinced that for most companies, the best approach is to create a platform that contains the superset of all functionality in the product portfolio. Each product can then be derived from the platform using automated configuration. This avoids the tension and frustration that many organizations experience, trying to align product and platform teams, deciding where to build functionality and struggling with increasing the frequency of release.

This of course removes the need to separate between product and platform teams as all teams are concerned with adding functionality to the "superset platform." It also removes the challenges of integrating build and test pipelines as all software is in the one source code control system. And it brings more benefits as the company typically realizes that instead of selling a limited portfolio of products, the configurable platform can be used to create hundreds, thousands or even millions of product variants at virtually no or at least very low cost. This then allows the company to provide mass customization of its product portfolio, resulting in a situation where customers can request close to "engineered-to-order" or bespoke products while the company still has a product and platform-centric cost structure where the R&D costs are amortized of many customers and product instances.

The dichotomy between product and platform that often is present in platform organizations causes all kinds of challenges that can be remediated by removing that dichotomy through the adoption of a superset platform (I've also referred to this as a "configurable product base") where the platform contains the superset of the entire product portfolio. That then allows for the automated derivation of products by selecting a subset of the functionality for inclusion in the product. The build and test toolchain can automatically create and test each product continuously to ensure that the entire portfolio is at production quality at all times. It also allows the company to offer a much broader set

of products or a much more configurable product so that each customer can to a large extent get the product he or she wants.



BALANCE ARCHITECTURE AND CONTINUOUS INTEGRATION/TEST

A key challenge for any software product is quality. Before we can do anything to experiment with customers and improve the product, we need to make sure that the product does what it's supposed to. Although no software is free from defects, there being too many of them hurts the perception of the product and the value it is to deliver.

A key enabler of quality is test automation. Running tests every time new code is checked in as well as more advanced system-level tests periodically is one of the most effective ways to ensure that the functionality that used to work is still working. Of course, this needs to be complemented with test-driven development and some amount of manual exploratory testing.

The best way to set up a test infrastructure and what to test when are subjects that have received vast amounts of research, and every software engineer has an opinion about them. I'll not go into details here, but in an earlier post, I discussed the CIVIT model as an effective approach to visualize all the test activities that are conducted end-to-end for a product. This helps understand the current state, define the desired state and prioritize improvements.

The challenge for complex software is that you can't test your way to quality. Especially for platforms, the number of configurations and connections between different parts of the system is so high that it's simply impossible, or at least prohibitively expensive, to test everything. This is made worse for

platform-based products that allow for significant configuration of each product instance at the customer site. The result may be a very high number of issues being reported by customers with little commonality between them.

The best answer to deal with this challenge is not to test more, although that may be required, but rather to focus on refactoring the platform and product architecture. A clean architecture with strong interfaces and decoupled functionality is great in that it simplifies testing as most testing can be pushed to the component and subsystem level and system-level testing can be minimized. For platforms, this means that it should be possible to test them independently of the products. This, of course, means a defined API that the products need to use. This then also means that products can be tested, at least to some extent, without the platform.

In practice, however, the architecture is always suffering from architectural technical debt and no matter what decomposition, there will always be functionality and quality requirements that have cross-cutting consequences. Consequently, we need to balance architecture work and investment in continuous integration and test. The architecture helps engineers avoid mistakes in the first place and applying continuous integration and test catches introduced errors early so that the cost for fixing these is minimal.

With more and more products being connected, an additional factor is the use of the post-deployment stage to identify quality issues. Of course, the main functionality needs to be confirmed before deployment, but more minor and rare quality issues can be identified post-deployment by monitoring system behavior and detecting deviations from the baseline established by the previous software version. I know of several cases where companies used this to detect issues at customers, after which they rolled out a fix before the customer even noticed the issue.

As more and more products with high-reliability requirements are adopting DevOps, ensuring quality becomes increasingly important. The knee-jerk reaction in many companies is to simply test more. However, in complex systems, such as platform-based products, you can’t test yourself to quality; the number of configurations often is prohibitively large. A complementary architecture refactoring initiative is required. Decoupling components and minimizing interaction, for example through a message bus and microservices, is a powerful way to focus test effort on the components and reduce system-level testing. Remember, sometimes the obvious is the wrong thing to do. Fix root causes, not symptoms.

DON'T INTEGRATE NEW FUNCTIONALITY TOO QUICKLY

Any solution aiming to stay relevant needs to continuously integrate new, innovative functionality. The main reason is that functionality commoditizes over time so that if we wouldn't add new features, the entire product or platform would commoditize and become irrelevant. Commodity software isn't necessarily useless, as proven by quite a bit of open-source software, but it's hard to monetize software stacks that don't provide clear differentiating and innovative functionality.

One of the models I often use in this context is the Three Layer Product Model (3LPM). It organizes functionality into three layers. At the bottom, we have the commodity functionality layer where we aim to optimize for minimizing the total cost of ownership. On top of that, we find the differentiating functionality layer with the functionality that's really liked and appreciated by our customers. Here, we optimize for maximizing customer value, meaning that we provide ample configuration and variation points to ensure that customers can optimally exploit the functionality. The top layer is the innovation and experimentation layer where we try out new functionality that we hope will provide future differentiation. Here, we optimize for maximizing the number of experiments that we can run.

We can use the 3LPM as a conceptual model, but it's even feasible to use it as the basis for the architecture of the system. In this case, we define interfaces between the three layers to minimize dependencies and clearly classify functionality into different categories. This is important since, especially, commodity functionality is often still treated as differentiating, as a consequence of which it receives far too much R&D resources.

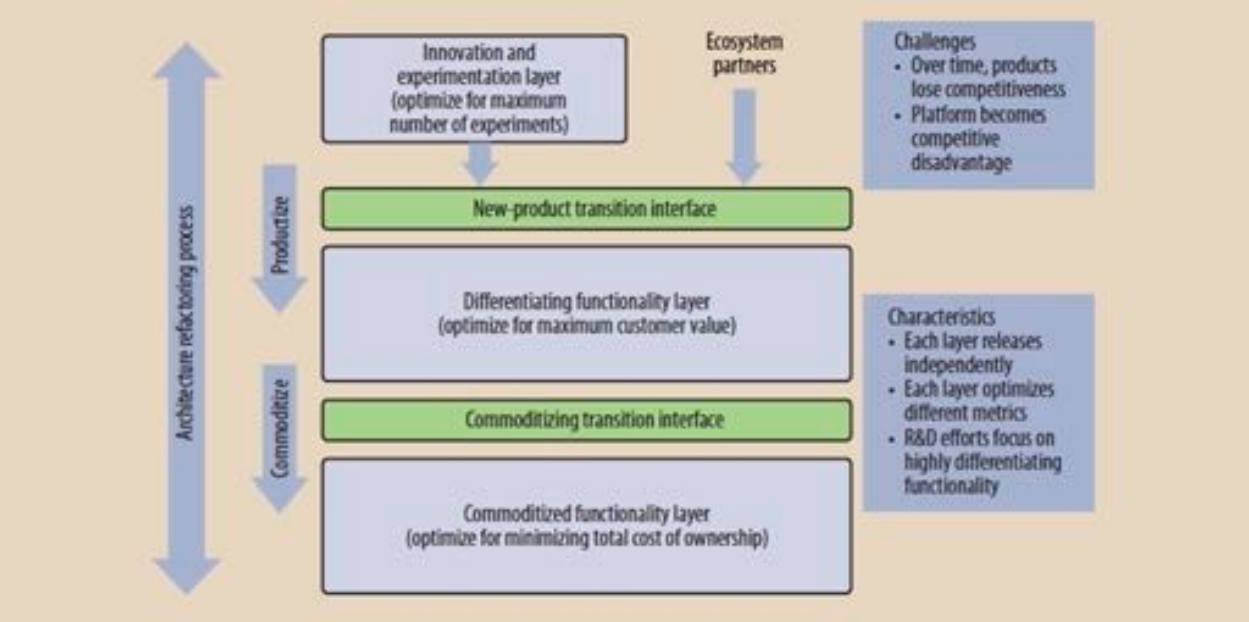


Figure: Three Layer Product Model

The focus here is the interface between innovative and differentiating functionality. The nature of innovation is that many, if not most, innovations fail to deliver the expected outcomes and shouldn't be included in the platform or product. The good news is that the few innovations that are successful more than pay for all the failed ones, but we need to make sure that we minimize the negative consequences of these failures.

In many of the companies I work with, the software stacks contain vast amounts of functionality of which few people know whether customers use and appreciate it or not. The challenge is that most companies fail to instrument their software in such a way that they have a good understanding of the frequency of use of their features. As our research shows that half or more of all features are never or hardly ever used, it's clear that a typical software stack contains significant amounts of functionality that shouldn't have been included in the first place.

Rather than including innovative features immediately in the product or platform, we should first experiment and confirm with customers that the functionality adds value by 'bolting on' the innovative functionality and keeping it at the other side of an API. If it proves a hit with customers, we can then plan the deep integration into the product or platform. If it isn't successful, we can easily ditch it as it never got integrated in the first place. This avoids bloated products and platforms that contain tons of non-value-adding functionality.

Although this is a generic problem for all software, it's especially relevant for platforms as there always is a tendency in R&D organizations to include new functionality directly in the platform as it then is available to all products immediately. This is one of the many ideas in software engineering that makes a lot of sense on the surface but proves to be false when studied in more detail. Many companies struggle to remove commodity features and functionality that are no longer used from existing software, so let's make sure we don't integrate features that nobody (or too few) will use.

Software products and platforms need to continuously incorporate new functionality and features to stay relevant. The challenge with these new features is that many will prove to be irrelevant for our customers. This means that we shouldn't integrate new features into the platform immediately but rather build them outside the product or platform, validate that customers actually use them (and not

only say they do) and integrate only after we have evidence that each feature indeed adds sufficient value.

In the end, we care about effectiveness, not efficiency. Cranking out tons of features that nobody ends up using is a colossal waste of R&D resources. Instead, focus on finding what customers really care about and then invest heavily in those features. Even if you worry about some rework when a successful feature needs to be integrated into the platform, your overall R&D effectiveness will be vastly greater.

DISTINGUISH CUSTOMER-UNIQUE AND CUSTOMER-FIRST FUNCTIONALITY

The idea behind platforms is to share functionality between different products and make it available to as many customers as efficiently and quickly as possible. This is one driver for continuously incorporating new features into the platform. Incorporating new features can be done using a product-first approach or a platform-first approach.

In the product-first approach, the products built on top of the platform first incorporate new functionality in a product-specific fashion. Once two, three or more products have it, the functionality makes it to the platform's backlog and a generalized, generic version of it is incorporated into the platform, replacing the product-specific versions. When using the platform-first approach, new features are immediately incorporated into the platform and made available for all products.

Even though the product-first approach can easily be viewed as less efficient, there's a caveat: in practice, it's often quite unclear what the best way is to realize new functionality. Therefore, using the product-first approach, the functionality is first realized for each specific product and once it has matured to a point where it's clear how it should be realized, it can be generalized and incorporated into the platform.

The risk with the platform-first approach is that it may easily result in significant amounts of rework. Feedback from customers and the product teams will often require the platform team to make several changes to the functionality. Of course, the team can decide to deprioritize these requests, but that results in a suboptimal realization of features.

Many companies aspire to have roadmap development, ie functionality driven by their strategic priorities. In practice, however, much of the features on the roadmap are the consequence of customer requests.

Few companies are successful without being responsive to their customers. Consequently, many companies and their salespeople, product managers and senior leaders tend to say yes whenever a customer asks for something. Although this is fine when you're building bespoke systems and customers are paying you for your time, it easily becomes problematic when selling a product or a platform and there are many, potentially conflicting, requests from customers.

The challenge is that each request needs to be triaged into two or three different buckets. The primary question is whether a particular request is customer-first or customer-unique. Customer-first refers to functionality that one customer happens to be the first to ask for but that over time will become relevant for other customers as well. This functionality should of course be incorporated into the platform over time. Customer-unique refers to functionality that's unique for that specific customer and that won't be relevant for other customers at any point in the future.

The first question concerning customer-unique functionality is whether it's possible to say no to the request without significant consequences to the customer relationship. If the functionality needs to be built, in the ideal case, we can do so at the other side of a customization API. Thus, neither the product nor the platform will be affected. The alternative is to include the functionality in the product or platform and then use feature toggles to activate or deactivate it. Most companies use some form of licensing mechanism to configure, typically at startup or runtime, what functionality each customer has access to.

The problem with customer-specific functionality in the platform is that it drives up variability and the number of variation points in the system. When many customer-unique features are included, the dependencies between variation points as well as the variants for each variation point explode. This makes it exponentially harder to predict the implications of changes and additions to the platform code and has at least two significant negative consequences. First, developer productivity tends to plummet as so much analysis and design have to be done to ensure that additions don't break legacy functionality. Second, despite this effort, many quality issues will slip through anyway, resulting in dissatisfied customers and many problems being identified in the field rather than during testing. This often leads to a third consequence: a slowing down of the release cycle as the company tries to compensate for the quality issues in the field by investing more and more time and effort into testing and a staged release process where new versions are tested longer at lead customers before being made available for all customers.

There will always be functionality requested by customers. Some of this functionality will be useful for more customers – the customer initially requesting it was simply the first to ask for it. Other functionality is so specific to the unique context of the requesting customer that it will never be used by others. Although it's fine to incorporate customer-first functionality into the platform and then generalize it to other customers over time, we should be very careful to include customer-unique functionality. Ideally, we should provide that functionality outside the platform on the other side of an API or decline the customer request. If we do decide to incorporate it, we should ensure that it makes financial sense as the majority of costs related to software are accrued after the initial development of the functionality. Remember, being customer centered doesn't mean accepting any customer request but rather carefully balancing all customer requests and your roadmap development to ensure maximum value for all customers.

CONTROL PLATFORM VARIABILITY

Software platforms by their very nature support multiple products. Typically, these products are used in different contexts and configurations. As a consequence, the platform has to offer variation points that allow each product and customer to use the platform in the way that best suits their purposes. Each variation point then has two or more variants that can be selected in that variation point or has configuration settings that allow the user of the platform to influence platform behavior. In addition, the platform often offers extension points where the product team or the customer can build custom extensions.

Although a simple and beautiful concept, it's very easy to cause significant practical challenges. Most platforms I've been involved in have thousands, tens of thousands or even more than a hundred thousand variation points. There are several reasons for this. First, when a customer asks for a point of variation and it becomes part of a sales negotiation, it's very tempting to accept the request and just carry the cost of introducing the variation point as part of the "cost of goods sold" line item. Second, product teams can easily demand that certain variation points be included as a precondition for getting or staying on the platform. Third, engineers are by their nature optimizers and identify, for each feature or functionality, the parameters that can be tuned. That often results in several variation points to be added for each feature in the platform.

When left unchecked, the number of variation points tends to explode. The resulting complexity is phenomenal and causes significant extraneous costs when adding features and when testing the platform. Many variation points have dependencies on other variation points, resulting in a combinatorial explosion of configurations that often cause post-deployment issues as it's impossible

to test all variations. Most companies only focus on the cost of introducing a variation point but fail to recognize the downstream cost for maintenance, testing and managing inter-dependencies.

One model that's very helpful in controlling platform variability is the Three Layer Product Model (3LPM). It organizes functionality into a commodity layer, a differentiation layer and an innovation and experimentation layer. The rule of thumb is that we seek to keep innovative and experimentation functionality outside the platform until it has proven itself as being valuable. In the differentiation layer, we allow for value-adding variation points to be included, whether these are configuration parameters, variants or extension points, as we seek to maximize the value for the customers of the platform. Finally, in the commodity layer, we seek to actively reduce the number of variation points. This includes the refusal to add new variation points in that functionality but also the removal of variation points that have outlived their usefulness.

The largest contributor, in my experience, to unmanageable amounts of variation points is the absence of variation point removal efforts. Naively, one may easily assume that an existing variation point, once introduced, has no associated cost. In practice, however, for each variation point, a constant 'tax' is paid that has to be outweighed by the value generated for customers and the company. For commodity functionality, this value is often missing or significantly reduced. Consequently, as part of technical debt management, we need to continuously allocate resources for the removal of variation points that have outlived their usefulness.

The challenge with removing variation points, as well as features and other functionality, is that the company often doesn't know whether they're in use by customers. This is why instrumentation of the platform is so important. It allows us to make data-driven decisions on variation points as well as features that are no longer used. We can then proactively work with the few customers who are still using them to help them transition.

Platform variability needs to be carefully controlled as it leads to significant costs and many post-deployment quality issues due to the combinatorial explosion of inter-dependent variation points, variants, configuration parameters and extensions. Controlling platform variability has two main activities. First, the introduction of new variation points needs to be carefully evaluated and only focused on the differentiating functionality. Second, we need to proactively remove variation points that no longer provide sufficient business value. In many cases, the quote of Henry T. Ford provides the right answer: "You can have any color you want, as long as it's black."

CONSTANTLY OPTIMIZE COMMODITY FOR TCO

In earlier posts, I've introduced the Three Layer Product Model (3LPM). Similar to all other software, platforms have three layers of functionality: innovative and experimental, differentiating and commodity. Functionality typically starts as innovative and, when it resonates with customers, becomes differentiating. The differentiating functionality drives sales and market share, causing competitors to develop similar functionality in response. Also, customers often get used to functionality. As a consequence, it commoditizes over time.

Functionality should be treated differently, depending on the lifecycle stage it's in. During the innovation stage, the focus should be on testing many variations with customers to ensure that we have the best possible realization. Once the functionality is differentiating, we should focus on maximizing the value for customers in various ways, including adding variation points and variants. However, once it commoditizes, the focus should be on minimizing the total cost of ownership (TCO).

The focus on TCO is critically important as our research shows that the vast majority of resources, up to 80-90 percent, are spent on commodity functionality. There are several reasons for this. First, as it tends to cover the majority of the code base, commodity has a high likelihood of being affected by change requests. Second, companies tend to consider functionality to be differentiating much longer than customers do. Consequently, we tend to continue to invest in functionality even when it has turned into commodity in the eyes of our customers. Third, the commodity functionality tends to sit closest to externally provided software, such as operating systems, databases and web servers. As these external components are continuously being updated, there frequently are implications for the commodity software as well. Finally, especially in leaders without a software background, there's a

general belief that software is free once it has been built and that there are no costs or investments required to keep the legacy software up to date.

The main risk in platforms is that, over time, a larger and larger portion of the R&D resources are consumed by commodity functionality. As a consequence, the platform as a whole may start to commoditize and offer fewer and fewer reasons for product teams and external complementors to build on top of it.

We need to engage in proactive efforts to reduce the TCO of commodity functionality. There are at least three mechanisms I've seen used successfully in industry. First, between product management and R&D, we need to explicitly label components as commodity once these reach that stage. Based on that, we need to stop, or at least minimize as much as possible, accepting change requests for these components. Rejecting change requests affecting commodity components is often difficult in the beginning but tends to lead to impactful conversations about the platform, the product portfolio and what constitutes differentiation.

Second, where feasible, we should replace bespoke functionality with commercial or open-source components. In many cases, functionality is first built in-house as no external alternatives are available. However, over time, these alternatives may start to emerge and then the principle should be to replace internally built functionality with external commercial or open-source components to remove the necessity of future investments in these components.

Finally, I tend to encourage companies to instrument their products with data collection functionality to be able to know which functionality in the product is actually used in practice. Based on that information, we can decide to simply remove functionality that no or few customers are using. Although hard to implement in practice, often due to the sunk cost fallacy, this is by far the lowest-cost alternative.

All these mechanisms require architecture refactoring to separate commodity functionality from functionality that's in earlier stages of the lifecycle. It can be hard to justify these refactoring efforts, but failing to invest in this will, in the long run, kill the platform for sure as it will become irrelevant.

Over time, the commodity functionality in platforms tends to consume a larger and larger part of the R&D resources. As a counter, we need to proactively engage in activities to reduce the total cost of ownership for the commodity. To accomplish this, we can stop accepting change requests for this functionality, replace commodity components with commercial or open-source components or remove functionality altogether. As Kay Yow is quoted to say: "Don't let the urgent get in the way of the important!"

INSTRUMENT YOUR PLATFORM FOR DATA-DRIVEN DECISIONS

William Edwards Demming, the American who helped Japan rebuild itself after World War II, famously said: “In God we trust; all others must bring data.” This is still a lesson most companies haven’t fully incorporated. Once a platform gets a certain amount of traction, the opportunity to make data-driven decisions presents itself. This is incredibly important as it allows for much higher-quality decision-making than is possible with opinions or qualitative data (what customers said). The challenge is that in many platforms, the architects never spent much time thinking about instrumenting the platform with data collection capabilities. As a consequence, the platform has limited, if any, data collection built-in.

When there’s no data available, many decisions are made without much evidence, purely based on beliefs and earlier experiences from key decision-makers. One reason is that it’s often hard to collect the data post-hoc. As a consequence, most companies that I work with are unable to answer basic questions concerning feature usage in their platform. How do you prioritize R&D resources if you don’t know whether the features you’ve already built are even used? And if you do know, do you then also know who’s using what features so you can do a proper segmentation of your customer base?

Moreover, when data gets collected, in many cases it’s the wrong data. Engineers tend to focus on basic quality data that allows them to verify that specific features actually work in testbeds and in the field, but this data doesn’t contain the right information for strategic decision-making about features, platform boundaries and customer segmentation.

The key lesson is that platforms need to be instrumented to facilitate data-driven decisions. This requires proactive and early thinking about the categories of questions we want answered about the platform and the resultant data that should be collected to ensure that we can answer these

questions. This data tends to be much more concerned with customer benefits and KPIs rather than functional correctness.

The second lesson, however, is that it's impossible to predict all the questions that could be asked about the platform. It should therefore be possible to easily extend the platform with new instrumentation when needed. This typically requires a "data fabric" layer in the architecture overlaying the functionality so that it's easy to insert "probes" into different parts of the system for data collection. The challenge is, of course, that this typically requires some form of DevOps to be present allowing updates to the software to be pushed out to extend the data collection functionality.

In systems where updating the software is prohibitively expensive, eg because of certification issues or high cost of updating software, one strategy that I've seen used is to create an approach where everything can be measured, but data collection is turned off by default. When certain data is required, collection can be turned on through configuration (changing a parameter setting). This avoids the need to update the software when new information needs arise but requires even more clairvoyance on the likely data needs.

Taking things one step further, we can see the first signs in several industries that the physical products are commoditizing and the value is shifting towards data generated by them. This is a completely new viewpoint for many product people, but it only reinforces the need to think carefully about instrumentation, data collection and data aggregation. Imagine a situation where your product is nothing but a vehicle for the data collected through it. How would that change the way you architect it?

The initial focus with platforms tends to be on getting the functionality to a point where users will adopt the platform. Consequently, instrumentation tends to not be in focus and we easily end up in a situation where many decisions need to be made based on opinions, rather than data, which of course leads to lower-quality decision-making. So, remember Edwards Deming and require everyone to bring data.

BE CAREFUL TO OPEN UP TO THIRD PARTIES

very platform company I've worked with would love to open up their platform to third parties and get 'free' functionality extensions. Especially the idea of a multi-sided platform where different parties exchange value with each other and you collect a nice slice of each transaction comes across as a highly desirable state of being where free money is simply flowing into your coffers.

In my experience, if something sounds too good to be true, it typically is and this is no exception, for at least three reasons. First, there's the mistaken belief of many that if you build it, they will come. Simply opening up your platform for third parties doesn't at all mean that anyone will show up to the party. Also, external parties, or complementors, have a business to run, so they'll only invest resources when there's a viable business case. For complementors, that business case typically requires you to have a large customer base for which they have a sufficiently large addressable market slice.

Second, complementors often put constraints on the platform such as interface stability, requests for functionality that helps them rather than your customer, and so on. So, there's a significant risk that your ability to innovate and extend your platform is restricted and that the simple fact of opening up causes you to slow down, reducing competitiveness.

Third, complementors have an inherent ambition to build differentiating functionality themselves as they need to monetize. As a consequence, they seek to push the platform into commodity as much as possible. From your side, you need to yield areas of functionality to complementors to allow them to have a business on your platform. Especially for companies evolving from products to platforms, this tends to be a difficult transition because instead of addressing customer requests that can be viewed as highly lucrative you need to give space to your complementors.

If opening up the platform isn't as easy as it seems, how should we move forward? To me, it helps to think in three phases: preparation, experimentation and scaling. In the first phase, you need to prepare the platform for opening up. This isn't a technical challenge but a business challenge. You need to build a sufficiently large customer base to ensure that complementors have a business case to partner with you. So, in the preparation stage, you focus on one stakeholder group, the customers, and, for now, ignore others such as complementors.

In the experimentation phase, you experiment with opening up based on a carefully crafted strategy. Rather than blindly executing the strategy, the goal is to perform experiments to learn about the behaviors of customers and complementors so that you can open up in the way that leads to the 'ignition point' the fastest while ensuring that you have a control point in the ecosystem. This is to avoid having your complementors and customers take off on their own without you and your platform. Each experiment is intended for learning purposes and should allow you to shut down the initiative if the consequences aren't as desired.

Once you have a sufficiently well-founded hypothesis of how to open up, it's time to execute and to drive the scaling of the ecosystem around your platform. In this stage, you'll still run experiments, but the goal is to accelerate growth and optimize conversion.

Although many platforms benefit from opening up, it's important to be careful and strategic. The risks are that there's no engagement, making you look weak, you get slowed down by the demands of complementors or you get pushed into commodity as the complementors demand space for their contributions. In my experience, grow your customer base to a sufficient size first before experimenting with different approaches to opening up and only then hitting the accelerator. Everyone wants a platform business, but there are so few real ones as it's incredibly hard to get there. So, don't get burned in the process.

ONE ECOSYSTEM PLATFORM STAKEHOLDER AT A TIME

Although platforms can be used purely for internal purposes, many reach a point where they're opened up to third parties, becoming an ecosystem platform. Ecosystem platforms serve, by definition, two- or multi-sided markets. This means that you have multiple stakeholder groups to support to make the platform successful.

When thinking about ecosystem platforms, I often distinguish between the 'operating system' class of ecosystems and the application-centric class. An operating system ecosystem needs to be built up by encouraging app developers to provide content and users to use the content. But both stakeholder groups have no interest in the platform until the other group is sufficiently large. In this case, you have no choice but to fuel the ecosystem by heavy investments in paying app developers to create content for a tiny user base in the hope that this content will entice new users to join and existing users to stay. As you can imagine, it's very hard to achieve this stage and there's a winner-takes-all pattern in the industry as the network effects are extremely strong in these types of ecosystems once you get to the 'ignition point' where users and app developers join the ecosystem without needing to be encouraged by you.

The application-centric ecosystem, on the other hand, provides value to users even without third parties providing content. In general, the best way to create an ecosystem platform is to first build the customer base to a level that makes it sufficiently attractive for third parties to provide extensions.

The challenge is to know when the time is right to start opening up. In my experience, it's best to wait until you get inbound signals. These can be requests from customers or third parties for APIs or it might be signs of others trying to integrate new functionality into your system using creative means. For on-premise software, this might include third-party developers intercepting calls to OS

functionality and inserting functionality there. For online software, this often becomes visible through atypical calling patterns of backend APIs.

Once there are clear signals that there's an interest in opening up the platform, you need to carefully design the APIs, deal with proper authentication, set up proper security measures, and so on. However, the main factor for most companies is to create a sufficiently large addressable market for third parties. In many companies using a platform for a portfolio of products, the variation in functionality and the capabilities of the products can be significant. This can easily lead to a fragmented market where the ecosystem APIs aren't available for all products or the behavior behind each API is different between products. That decreases the ease with which a third party can develop complementing functionality for a sizable group of users. Harmonizing the public APIs for the entire portfolio is very important as an enabler for creating a successful ecosystem platform but will initially be experienced as countercultural in the company.

Getting a multi-sided market to the ignition point where the ecosystem fuels itself without constant investment by the platform provider is extremely demanding and expensive. Rather than trying to build all sides of the market simultaneously, the better approach is to build the market one stakeholder category at a time. In practice, this means first building up a customer base of sufficient proportions based purely on the platform functionality and only then opening up to building up a second stakeholder category of third-party complementers.

CONCLUSION

Platforms are no longer just about reuse and efficiency. Platforms are key for enabling DevOps in your organization and critical for building an ecosystem on top of your portfolio.

Platforms may seem intuitively easy, but industrial practice shows that many mistakes are made that eliminate the potential benefits. Confusion about the strategic goal of the platform, poor architecture choices, accumulating technical debt, crushing complexity and drowning in commodity functionality are real and concrete dangers that companies need to avoid to effectively work with platforms

We discussed 10 key lessons for success with platforms, including focusing on speed, the superset platform approach, the three-layer product model, variability management, handling technical debt and instrumentation.

Platforms are beautiful and bring great benefit if done right. Use these lessons to your advantage. Remember, a smart person learns from one's own mistakes; a wise person learns from other people's mistakes!