SCTE•ISBE CABLE-TEC EXPO® VIRTUAL EXPERIENCE » OCTOBER 12-15 2020

2020 Fall Technical Forum SCTE·ISBE • NCTA • CABLELABS®

# Expediting New Product Deployments with Agile Operations and DevOps

A Technical Paper prepared for SCTE•ISBE by

**Andrew Frederick**
Principal Engineer
Comcast
4100 E Dry Creek Rd, Littleton, CO 80122
(303) 881-6103
andrew_frederick@comcast.com

 

# Table of Contents



 

# 1. Introduction

Deployment Engineering in the cable sector is where the rubber meets the road. These groups are responsible for taking products from the prototype stage to a scaled solution across the entire enterprise. Deployment Engineering is an in-house specialty that creates the cloud resource experience and delivers that to the local headend. An idea begins much like how a single raindrop falls thousands of miles upstream, then gathers and runs to the ocean. This is a useful analogy for the lifecycle of a product, too. By the time it reaches the ocean – in this analogy, our customers -- it reshapes the environments that it passes through. Cable products can be based on web applications, but when dealing with real time video or streaming video, it becomes impractical to distribute content centrally, and must be decentralized in order to maximize efficiency. One can simply look to the Content Distribution Networks (CDNs) that the industry's Multiple Systems Operators (MSOs) have spent billions of dollars building to move video to the edge as evidence of this truth.

Deployment Engineering holds the unique perspective of being at the nexus of product creation and customer integration. Over the years, the methodologies change (waterfall, agile, lean, scrum, Kanban, CI/CD, etc.) to embrace new ways of tackling problem spaces, and offer meaningful ways of reinventing the wheel. Such methodologies will continue to change and evolve with time – however, the problem space of deploying products at an enterprise scale has a core foundation of challenges that are common to nearly all products. Therefore, it becomes more critical for engineering leads, architects, and executives alike to understand the environment and problem spaces than it is to be masterful of any given tool used on the same old problem. Whenever shifts occur to adopt the latest methodology, as an industry, we put our momentum at risk and can be prone to recreating and reliving the same mistakes. We sacrifice the valuable experiences and lessons learned to move to a new methodology, and in doing so tend to be more focused with how we're operating in the new system rather than staying focused on the big picture. This paper aims to reinforce the common elements within new products and deemphasize the tools used to build them. In other words, technique matters more than the actual tools used to build something.

## 1.1. Traditional CI/CD Model Overview

It's helpful to take a moment to acknowledge the landscape of contemporary software and hardware implementations in an MSO. An appreciable trend over the past decade is to build your own system, using off-the-shelf vendor hardware. In some cases, that extends to the development of in-house and at-scale technology solutions that can be syndicated to business partners. It's a high risk, high reward endeavor, and being the first to market can be the difference on that investment paying out or not. To that end, the current method of maximizing efficiency and speed is the Continuous Integration/Continuous Development (CI/CD) pipeline. Some define this acronym as Continuous Integration and Continuous Deployment. Others, to be tongue in cheek, call it Continuous Integration and Continuous Disruption. In any case, and if not managed properly, CI/CD can become a dangerous feedback loop that pulls on products like a black hole -- and can grind a product's market introduction to a halt.

Why are cloud services successful to the point of creating a $150B per year industry? Because the cloud offers the most direct and rapid route to putting bits on the pipe. The ability to instantly scale one's computing needs is insanely attractive, but it doesn't come for free. For web services, where traffic is bursty and light, and done over a Transmission Control Protocol (TCP) connection with retries, it's relatively affordable to purchase CPU, memory, and disk in a datacenter, spin up an application, and you're off to the races. In the cable video portfolio, however, it's not always possible to create a good user experience by running applications out of a hosted datacenter, and the need to move large volumes of data can be cost prohibitive in leased space. Often the locations of national and regional data centers can be hundreds of kilometers from customers, which makes them impractical. This begets the need to build

 

custom hardware solutions at the edge. These products are typically homed at the edge, or use tiered caching models to distribute efficiencies of storage and transport. One of the main trappings of CI/CD is that it can be "loved to death," meaning it can be so appealing that it becomes overused and negatively impacts the outcome. This vulnerability that presents itself with many modern methodologies, including scrum, agile, lean, etc. CI/CD must be mindful about the state of the platform and the ability to make architectural changes at scale. Without this perspective, if the sole focus is on constant delivery, it's easy to create more work than resources can support, and accumulate unmanageable volumes of technical debt. Technical debt is usually accumulated when decisions are made that favor short-term and easy solutions, rather than using a better approach that would take longer. Tech debt, like financial debt, carries interest and makes it harder to implement changes to a product. For best results, a careful balance between change and progress must be struck that emphasizes the maximal efficiency of this model, rather than the act of change and flexibility itself.

### 1.2. Multitasking and Task Changing

It's no secret that there's benefit to being able to simultaneously handle multiple workflows. Modern CPUs are a fantastic example of being able to handle large workloads and change tasks, for virtually an unlimited amount of time. A CPU doesn't care what kind of processing it's doing -- it marches along endlessly forever. It feels obvious to say, but human beings are not CPUs. A certain amount of task switching and busy work is helpful, but taken to extremes, it can quickly overwhelm teams of humans who aren't able to work around the clock. To use an example, if you were to write a sentence ten times on a piece of paper ("I am a very good hexadecimal mathematician.") using two different methods and timed this exercise. It takes a lot longer to write one letter on each of the ten lines serially, rather than writing the complete sentence ten different times. Task changing can be done so rapidly that all meaning is lost in the work, and the rush is placed on the quantity of the output, not the quality.

Taking this example a step further, operating teams in this manner, where humans and projects are multitasking and changing tasks constantly, is counterproductive. When a product must be re-architected or a code base refactored, it's a common trapping to forget all the newly accumulated technical debt. A tendency is to keep pushing forward, without being mindful of the train we're pulling behind ourselves. A train can't stop and take a 90-degree corner, and it's not practical to pick every train car up and reseat it on the tracks when a product takes a turn. Good product owners and engineers know that it's important to get all the work done before the train arrives. You can't lay track under the train and you can't survey your route one mile at a time. This paper doesn't assert that we make less changes or stop enhancing and evolving our products, but rather that a longer view is necessary when considering the prioritization of features and functionalities that yield the best long-term outcome. All the short-term planning in the world can't make up for a poorly managed long-term goal, or a product that isn't ready for customers. Directional changes need to be planned and managed, especially including solutions for moving the platform and keeping it together through these transitions. Very often the focus is on the locomotive doing all the pulling up front, but if the cars fall off the tracks behind it, any first mover advantage is jeopardized. Maybe the next train can survive without derailment, but isn't it more important for customers to get it right the first time?

### 1.3. The Eighty Percent Rule

To create a construct to help balance progress and stability, the Eighty Percent Rule is a good tool to evaluate both micro and macro levels of decision making related to industrial innovation. It's instructive to not get caught up "chasing the nines" when building a proof of concept, but rather to start out with a much looser framework. "Chasing the nines" as it relates to product reliability and uptime is something that comes later on down the development path of mature products. In early product stages, the Eighty

Percent Rule paves the way to make active decisions about overall readiness for providing the nines of reliability. Only after obtaining this benchmark is it appropriate to begin working towards that plateau of of uptime and reliability.

The Eighty Percent Rule can be loosely defined as a means to strike a balance between efficiency and speed. The guiding idea is that there is a need for product owners to honestly rate the product experience, from the position of the consumer, and knowing its technical shortcomings. Simply put, if you wouldn't give your product an honest B rating, then it's not ready to be put in front of customers, let alone mass deployment and exposure. If a feature isn't going to function at 80% efficiency or more, it should probably hold.

The Eighty Percent Rule is also helpful for resolving discrepancies – because it's usually impractical to think in terms of getting something absolutely right. Corrections are expected as part of the process, so the aim is to get it mostly right, in order to resolve gridlock and move forward. This also serves as a nice goalpost to evaluate forward progress. The Eighty Percent Rule is designed to favor and reward long-term planning. It would be a foolish task to think we're going to show up and win a marathon when we haven't done any preparation for the race, so it's important to know where you're going in order to steer your product farther down the road. One of the tradeoffs of the Eighty Percent Rule is that it can take time to build the necessary momentum to get a product launched; it's important to keep in mind that this construct doesn't necessarily include a fast start, and it takes time to move with precision. If products are built with these key concepts, the time investment to moving quickly is minimized.

### 1.4. Hitting the Moving Target

The final piece of the puzzle requires our leaders to think in three dimensions. At a critical junction, a product goes from a small patch of trial sites to a multi-million-dollar production across dozens of geographic locations. At this point scale becomes a more heavily-weighted factor when making decisions about a product. The more knobs turned, the longer corrections will take to execute -- but oftentimes product leaders can inadvertently forget about reality testing decisions that can get made in haste. Was sufficient time budgeted for changes to be cascaded throughout the system? Was time budgeted to pay off technical debts, or did the trajectory lead straight into the next development hurdle? This methodology can appear to be cumbersome on the surface. In practice, however, managing low technical debt prevents a product or a team from falling into the abyss. Changes that are more deliberate and paced tend to yield better decisions about future planning. A useful goal is to consider the best decisions for the month, not the moment.

A premature commitment to a product build can have the same effect as changing a product too rapidly. Essentially, it's two sides of the same issue: Scaling an enterprise product is not assembly line work, and the best way to do that, especially with a lean team, is to keep the product flexible. When the product reaches this phase, it's useful to focus on the realities of expontential scale. It's no longer trivial to make changes to a few sites by hand anymore, or to rebuild the system from scratch. Any change is multiplied by the size of the deployment footprint. Making an architectural change at scale? That's going to be costly to go back and redo all those sites a second time. Need to make a global configuration change? Even a simple task can take hours when multiplied over a large footprint, let alone complex changes. Have to make several changes to a product? Validating that all those changes were made correctly can be error-prone, especially if a human must do all that swivel chair work. Incredible amounts of cost and resource losses can accrue very quickly at this stage -- because hitting a moving target is difficult. We have to be mindful about predicting where we're going to be upon impact. A simple miscalculation can cause a missed target -- which results in a wasted first attempt, as well as time lost to take another shot.

When time to market is critical, making the first shot will yield the best return on investment. Customers won't be impressed by a half-baked product, and first impressions are critical to new product adoption.

# 2. Launching A New Product

## 2.1. Universal Truths

The information outlined in subsequent sections of this paper intend to reinforce or structuralize what it takes to rapidly deploy a new product to the market in the cable sector. These tenets are reasonably ubiquitous, regardless of work portfolio, and can be considered common regardless across frameworks.

## 2.2. Challenge of Scale

### *2.2.1. Inventory and Accountability*

In 2020, some take for granted that modern cloud computing products have built-in reporting, monitoring, or telemetry features. In some products, the physical layer has been completely abstracted, so that consumers don't have to manage that layer of infrastructure. But when a company sets out to build a new product at the edge, a lot of those efficiencies stop at remote access, and each hardware manufacturer has its own interface and operating system. There may be a common management interface, or SNMP trapping, but it's worth recognizing that to the operations product owners, it looks like a blank slate. If the hardware platform is an internal build, it's most likely going to require specific firmware revisions, or specific driver requirements. When this hardware platform serves dozens or thousands of devices, visibility across the footprint is necessary, to audit for discrepancies. At a minimum, telemetry should be built into the product, to inventory critical platform components, such as: installed RAM, SSD wear rate, drivers, firmware, BIOS, fan status, power supply status, and temperature alarms, among others. Such visibility is a minimum requirement -- for further efficiencies, auditing capabilities can be added to actively seek out exceptions and raise their visibility, resulting in corrective action. Mastery of cloud components includes the ability to not just audit the footprint, but also to take components out of service for patching with minimal to zero intervention. After all, once the product is running, it's likely going to be run by a lean team, so this functionality will realize a vast amount of utility through the end of the lifecycle of the product. It's important to be able to build into a product the ability, early on, to know how many exist, where they are, and their current state. In order to move faster, having the ability to orchestrate the hardware layer is an essential foundation to any enterprise product. For best results, it is one of the first components at the integration layer, built after the proof of concept is working.

### *2.2.2. Monitoring*

Monitoring is another component that is often overlooked until the product gets closer to launching to customers. There are advantages to beginning to monitor the hardware installations coincident with the system being built, but be careful -- this can also be taken a bit too far. Example: A product launch involving SSD drives, which, in older installations, had completely worn themselves out -- before it was ever used for customers. Had this been monitored earlier, the issue could have been identified with an immediate impact assessment because of good inventory practices. This kind of surprise usually happens near the moment when the equipment is lit up with customers, and it throws another technical barrier in front of the product launch. And because the solution usually involves coordinating either with a vendor on site or asking for help from local site personnel, it's not exactly the fastest procedure, and usually ends up involving multiple resources. At this point there are usually a host of other critical issues being

 

addressed before launching the product -- a good project lead will work to identify the platform's critical components, and build systems to monitor and assess system health so that a lean team can operate the platform with maximum efficiency.

While it is critical to build monitoring early and often, it is also helpful to suppress or mute production alarms while the system is being built. If the system is in a full production monitoring mode before customers are on the equipment, that usually means change management tickets need to be coordinated to make updates to the platform. This creates additional overhead by means of adding additional layers of process to the workflow, and across many sites this can add up quickly. While the system is greenfield (in this case, meaning no customers are on it), the platform may need to change frequently to keep up with the pace of the developers. For best results, build alarming into the system early and schedule regular reviews of the alarms to make sure the system is behaving as expected. But don't paint your teams into a corner by creating additional administrative overhead for anyone making changes to the platform when there is no exposure to customers. A good monitoring system, when implemented early, can help identify problems in the platform and surface them before they become critical. Maintaining a platform, at scale, while building it, carries a higher up-front cost, but the payoff resembles that data center experience that every product wants to operate in. By conquering the hardware footprint with good inventory management, and having the ability to automate upgrades and by monitoring those investments, products are in a very good position to sustain a much more rapid CI/CD model.

## 2.3. Challenge of Change

### *2.3.1. Data In, Data Out*

This paper will conclude with some insight about users and user experiences, as it relates to expediting customer hardware deployments. At this stage, it's useful to keep in mind that it is when products scale that the most can be learned about the platforms that were designed. It's unacceptable to just have the data if it can't also be managed. CI/CD wants to always be moving. In order to stay highly effective, which is to say moving without stumbling, any management systems that are built should prioritize consideration of the teams who will ultimately run or use the platform, day after day after day. Ideally, both back end and front end teams coordinate on the architecture and data flows, to make the language and technology decisions. After that, how do the teams go about building the system and adding components? Do they build a UI that allows for manual data entry, so configurations can be managed in a central database? Or do they build a protocol and an interface to onboard devices to an endpoint, and have devices send their configurations into a central repository? The best answers here depend on how the data needs to be maintained, and there may not be a wrong answer. Now that this large data set has been created, do the teams have the ability to make bulk updates across the platform? Not everything needs to be mutable, but building it such that the most common data points that can change are scalable is useful. Also: Does the system expect that it can be handled by a database admin, or are end users also empowered to manage the data?

Building on the previous tenets of inventory and monitoring, data is the next piece of the system that must be reinforced. This user interface may account for the needs of users for several years to come, so this step is critical to getting right. The cold reality is that once a product is launched and there are customers on it, operators tend to become risk averse. This can sometimes mean having to settle for a mediocre UI. A good timeline to keep in mind is about ten years. If the platform can be put to good use for about a decade, before having to rebuild it, that's a good system. Such timelines are also useful for future planning: can any shortcomings of the system be tolerated for the next ten years? Or will running the platform always require more resourcing? How does one not only support the platform, but any syndication partners, too? What kind of example does this set within a company if its enterprise tools are

 

subjected to compromise and work-arounds, just to perform daily functions? Put another way, does the platform inspire your teams to do their best work?

#### *2.3.2. The Hardware Lifecycle*

Nothing in the hardware computing world lasts forever. In 2020, a best-case scenario for a syndicated product is a lifespan of about a ten years. Some older systems in the industry, built in the early 2000s, could have easily been designed to last ten years -- but as the product space for today's technology gets more crowded, stagnation becomes unprofitable. Old equipment and legacy technologies will end up costing money in the long run. Because the pace of innovation and competition is relentlessly increasing, hardware platforms are unlikely to have the same lifespan they enjoyed twenty years ago.

It may not be possible to know what a replacement hardware platform will be, but when building a platform, it's useful to ask suppliers questions about how long that model of equipment will be in service. There are no guarantees, and it can happen that a custom hardware product may be use equipment that is near the end of the manufacturer's hardware refresh cycle. Or, the opposite can happen, in that the appliance could enjoy a foreseeably favorable lifecycle. Gathering data points that can inform an estimate about your hardware's expected life span will help guide any decision making about how much time and energy are invested into the platform. While we don't want to withhold functionality or usability from the users, it may be prudent to investigate what compromises should be made if the platform is going to have to be rebuilt for technical reasons and in short order.

### 2.4. Meltdown Intel Security Vulnerability – Retrospective

Catastrophic events in the digital world are uncommon, but they aren't impossible. Looking at a fairly recent example, the Meltdown Intel security vulnerability, discovered in 2018, triggered major impacts on the hardware and software world – especially for delivering video. When this vulnerability was disclosed, it brought home a very real problem to an internal project involving thousands of Intel Xenon CPU processors, distributed across dozens of locations, to do real time linear encryption and ad insertion. The performance of the product, at the time, was calculated based on a a certain density of the hardware solution which met the product's architectural needs for streaming throughput. After the Meltdown vulnerability was published, the resulting software patching of this vulnerability generally resulted in a performance loss of the CPU. If your product is very CPU dependent, a performance loss can have major impacts on its ability to service the needs of your customers. These kinds of incidents can have major architectural impacts to custom hardware products. This kind of vulnerability doesn't happen often, but that such a sizeable one did within the last five years is a good reminder about diligence in system design.

If you have a CPU intensive product, and something like Meltdown happens again, how do you quickly evaluate what options are available? How does your product recover and move forward? What if it costs money because in choosing not to compromise on security, you trade off being able to support the service levels of syndication contracts? Is it possible or cost effective to throw more CPU at the problem to make up for the shortfall? Do you focus on gaining efficiencies from the software and try to get that to be more performant? Or does the product team need to look at other options, to determine whether there is a critical tipping point that will need to be crossed in the design? As the investment into a product increases, it behooves product owners to be able to immediately materialize specific information about their platform to make the most informed decisions possible. If another Meltdown happens tomorrow, does the product have the telemetry in place to support good decision making? With so much invested into building these products, it's borderline negligent to build products without telemetry. Product owners should be able to assemble a data set of information about their platform in time for their next meeting, not in a few days. This same principal applies to other consumables for your product -- whether

 

it's CPU cycles or network bandwidth, it's going to be critical to have real data about your product to make the best decision to build it on or ahead of schedule.

### 2.5. Who Can Move My Cheese?

Each product is different, and with this uniqueness comes a host of common and individual characteristics that are important to manage for any given system integration. The previous tenets of this paper discussed how essential inventory and monitoring are to building a good product. If there is an outage and teams are escalated to troubleshoot a problem, time is money, and finding the problem becomes a race. A well-built product, that has visibility into all the code bases across the footprint, can quickly help rule out problems and steer troubleshooting in a positive direction. If the product can detect that there's a firmware mismatch, or that a node failed an upgrade, this can lead the production teams to the source much more efficiently. On the other hand, if there is a good auditing system and all the basic health checks for the system are coming back clean, then we can assume the problem lies elsewhere and quickly rule out the first layer(s) of the product. When systems grow large and there are thousands of data points to look at, humans are going to be not well equipped to spot those small exceptions.

On top of the physical layer, enterprise products are typically built with redundancy in mind. Oftentimes modern products are geo-redundant (meaning there are multiple co-locations where the loss of a physical facility can be carried by another peer). Is it possible to build this logical layer into the management of the product, so it can understand product states? Can the platform be upgraded without requiring a skilled operator who understands the specifics of failing the systems over, in order to perform work on the platform? The reminder here is that operations teams are generally lean, so anything we can do as product builders to make common tasks easier, or find a way to decrease the amount of time and skill required to accomplish maintenance work, is beneficial. A core theme for this paper is that to encourage and promote CI/CD, we MUST be designing products platforms that are easy to upgrade and efficient to operate. We need to eliminate the technical debt so efficiency is baked into the system. There's a benefit to being able to build and scale a platform with small teams, so the small teams of operators are best equipped for this task. Time and time again in cable we make the same mistakes by pushing unsupported products to market too quickly. We fail to recognize that the lean teams handling the build, the customer transition, and the operation of the platform are not going to be the best suited to find time and necessary skills to work backwards through the hardware stack to create a useful orchestration layer, after a product has been built. This is the crux of the issue -- products are created in a vacuum, then pushed out the door without proper supportive tooling. They change too frequently before getting in front of customers, and lack a good orchestration layer to manage these changes. Products end up behind schedule because they aren't built in a way that helps operators get them to customers faster. We must collectively stop this behavior if we expect to improve our time to market. We must stop believing that as leads, our job is done once the proof of concept is working in the lab. We must be invested in the management of the platform and product that comes after this idea.

## 3. System Users

### 3.1. Someone Else's Problem

Traditional product lifecycles dedicate much of their development to creating a working product, and frequently this comes at the expense of the end users. As developers, we often lose sight of the fact that the product isn't complete until we have system users and managers. The R&D road to successfully solving a new problem is often a challenging journey, and it's a common trapping to collapse across the

 

wrong finish line. So much time is focused on the problem that we forget to also deliver a solution that works for the end users. Engineers can become hyper-focused with building a working proof of concept and are allowed to completely ignore the complexity of the system they've built. Very often, the mindset of being aware or caring about the next phases of a project differs from the traditional build mindset. Often the product builders don't have any connection to the UI, the users of the system, or the operations of the system, which become a problem for "someone else". Or, more formal barriers emerge and takes the incarnation of a handoff between product development teams and operations teams, further exacerbating the problem. Building a management and orchestration layer then becomes automatically inherited technical debt. Operations teams aren't typically staffed or funded to build a UI that meets the complex needs of its users. Each of us in the industry, not to mention our end customers, can probably think of a product we worked on which had a compromised UI. The product itself works just fine but using the system to manage the product can come with a lot of work arounds or hang ups. This isn't the hallmark of an enterprise or world-class product, and as project leads aiming to improve our speed to market, it will be imperative to think through these challenges in a new way. The team who built the prototype typically has the most working experience building the product. Why wouldn't we leverage this experience and have this team construct the UI or the basic foundations for manipulating data within the system? Instead we allow these experienced teams to discard this knowledge and then the snowflake becomes a snowball rolling down the hill. The last barrier for innovation and correction of a management platform is exposing the product to live customers. The appetite for product owners to make big changes to their platform once customers are stably onboard approaches zero in a short time span. The risk vs reward construct comes into play, and with paying customers on your product, that is the most prudent business decision. We then find ourselves in a place where customer migrations override any needs for the operations teams to manage the product, and the window for making changes to the platform closes like a feedback loop, underscoring the importance of good tooling.

### 3.2. Accessing, Viewing, and Changing Data

Big changes don't typically happen to a product after it has been launched to customers, but in the beginning stages of a product, large changes and disruptions are much more common to the landscape. Couple this with productivity methodologies like CI/CD, which aim to continually keep making changes, and we have a recipe for creating a lot of technical debt. Often, we find that our engineering assumptions don't always happen in the real world, and we're forced to make pivots to designs and plans when new information presents itself. This is a common theme for prototyping – we make enough assumptions to move forward and reserve resources for when that doesn't always work out. When we must make these corrections, how will the middle and end users be able to access, view, and manipulate the product's data? At scale, changing data by hand is inefficient and impractical, so we need methods of importing and exporting data if they can't be manipulated in the tool directly. How will other users or teams consume the data your product will generate? Will other users or systems have access to the data? It's imperative to be able to make scale changes to a product, especially in the early stages of its deployment. Product leads need to be finding ways to minimize technical debt from the earliest stages of a product's lifecycle in order to sustain the changes needed to mature the product.

Balancing the needs of the product's users and the development cycles required to achieve those goals requires a delicate balance to be struck- products can't wait forever before they need to be exposed to real world challenges. With some foresight and experience, we can anticipate what our user's needs will be and find compromises in functionality that help us achieve rapid product penetration.

 

### 3.3. Why Did We Wait This Long?

Waiting to add users in to the closing stages of your product's development is an open invitation for rework, and by nature of what's been discussed in this paper, a challenging exercise with diminishing opportunities and resources. There is logic to building a prototype first before creating a UI, but we must acknowledge that there is a lot of development time ahead that need to be spent transforming a prototype into a world-class product that's not only fit for your customers, but valuable enough to market to syndication partners. There is a flexible window in building a product to introduce your users and start building for their needs. But when prototyping new products, we should be doing the basic research to anticipate what the users will need, and what data is going to be valuable, ahead of time, then bake that into the fabric of the product. We need to be building platforms that have the full vision for what the product needs to do, how it should behave, and how it's going to be used, from the earliest stages of conception. The users are what bring the product alive -- without them, it's just a bunch of ones and zeros.

### 3.4. Build a User Interface That is Meaningful; Build Your UI in a Meaningful Way

When you interact with something that is designed well, it feels natural and intuitive for how to accomplish a given task. UIs that don't feel obvious leave users frustrated and can create barriers to adoption. It's important to represent your company's investment by creating a UI that works well and works well for your users. A UI will seldom be perfect, as that is an objective measure, but having a UI that is in agreement with what your users want and need is central to the long-term success of the product. A UI represents your product, and a misrepresented UI can leave a user speculating about the effectiveness of the underlying technology. It's the paint job on the car, or the finishing materials selected in the interior. A sloppy build on the surface will be an implied reflection on the underlying technology. The UI should do all the talking and speak for itself, leaving all explanations absent.

As important it is to have a natural interface, it's also crucial to make sure that the UI can do all the heavy lifting for your users. If large, bulk updates must be made to the platform in order to maintain it, the UI needs to have a way to orchestrate these large changes without needing to involve a back-end user to facilitate the change. End users must be empowered to care for and manage their platform. If doing so is hard to use and time consuming, the userbase will not be inclined to maintain non-critical data. Working on the product will become tedious, and the employees supporting it will only do the minimum required to "keep the lights on."

## 4. Conclusion

Whatever productivity suite we're operating in for the moment is only as effective as the experience it's being built on top of. If we lack the vision and experience of working on a product through all stages of its life, we can compartmentalize deficits and defer responsibility without consequences. All this comes at a cost, and these assumptions we make about other teams who will pick up the slack usually means that there's not a coherent plan or vision for the lifecycle of the product. If we have no common vision or goal, and no means to get there quickly, we must expect that projects will meander through time and will eventually arrive in front of our customers. If we want to empower ourselves to take control of these timelines, we must start by being honest about planning around the full spectrum of the product's lifecycle at every stage with every team involved. We can't unfurl the Mission Accomplished banner across the windshield just because we solved the prototype challenge -- there's a long road ahead to taking this fledgling product to millions of customers. We find that when projects are run well and have this vision in mind from the beginning, they often start slowly, but finish quickly. It takes time to invest

 

in orchestration, management, and user requirements, and when platforms are being built, they are often done so through lots of learning (see: failure) and chaotic change. Product leads need to be mindful to create and build orchestration and management into the platform so that navigating all these changes and managing technical debt is factored into the platform. Our modern productivity suites emphasize rapid and dynamic changes, and without a proper support system for this disruption, we can quickly find our products in the middle of a vortex.

Building a product should be done so with the least technical and experienced users in mind. This is a great pipeline for engineering talent -- more experienced engineers design and build a platform, and more junior engineers can learn how to operate and support the platform. It's a great way to get to know your product and business from the inside out. As these engineers grow, they accumulate experience and knowledge of how a good product is built and operated, thus helping to create the next crop of engineers and leaders in the company. To do so, we must create meaningful UIs that make the operators passionate about their product and give them a sense of authority and accomplishment. We can't expect every end user to be a database administrator, therefore we need to build our systems for the most common denominator and prioritize functionality that will empower our engineers to be a force multiplier, especially considering that lean teams are typically "keeping the lights on" for these products.

By nature, prototyping a new product means to boldly go where no person has gone before, and building new world class products doesn't happen overnight. Often, it takes years of work to find the balance between budgets, expectations, technology, and resources to craft a meaningful middle layer to manage your architecture. Being able to keep an enterprise platform running at scale with a lean team demands that all the hard work be done up front and most of the challenges and desires of the users can be met autonomously. Platforms are born, and they are born to change. Flexibility, inventory, and scale are critical components to master when building and finishing building your platform. Hardware is going to change due to natural or unnatural causes; additional technology disruptors like those that surfaced in recent history are practically inevitable. The most effective product leads know how critical it is to account for consumables in real time and are robust enough to support large changes across the footprint in a short period of time.

In conclusion, project leads shouldn't wait too long to invite users into your system to find out all the ways your they will need to access and consume data from the platform. Establish relationships early and engage users directly about how they think the system should behave and what their pain points are with the existing system. For best results, build a user interface that removes technical burden from the operations teams so they can be left to an already important task of running, monitoring, troubleshooting, and repairing their platform twenty-four hours per day, 365 days per year. Product leads need to be mindful about not leaning in so far to the Continuous Delivery that we forget that the other half of the feedback loop is Continuous Integration. We can't simply ignore the process for the sake of meeting deadlines and pushing an unsatisfactory product out to our customers too early. Product leads need to incorporate tooling from the ground up and across team handoffs in order to build a product that can get to market the fastest and most direct route possible.

# Abbreviations

| | |
|---|---|
| CDN | Content Delivery Network |
| CI/CD | Continuous Integration / Continuous Delivery |
| CPU | Central Processing Unit |

 

| | |
|---|---|
| MSO | Multiple Systems Operator |
| TCP | Transmission Control Protocol |
| UI | User Interface |

# Bibliography & References

CI/CD Web Article https://www.infoworld.com/article/3271126/what-is-cicd-continuous-integration-and-continuous-delivery-explained.html

Cloud Market Share 2020 https://kinsta.com/blog/cloud-market-share/

Meltdown and Spectre https://meltdownattack.com/

Still Fighting Meltdown and Spectre https://www.wired.com/story/intel-meltdown-spectre-storm/

Shane Pape, Mgr 2, Prodt Dev Engineering, Comcast. Interview.

 