



PACE INVADERS

Accelerating performance in software
development for rapid growth

CONTENTS

04	Introduction
05	A day in the life: Dave's Fiasco
09	What makes Dave cry at night?
10	Sprints: The wolf in sheep's clothing
15	Chris' Dilemma
17	Bottleneck logic
18	Case Study: Lufthansa Technik
22	Intro to Pace
27	Your next step

Imagine you lead a dev team...

The current approach to running your team causes: an increasing backlog, more and more bugs and the never ending need for more resources

OR

You founded a software business and the launch was great, but now you struggle to scale quickly (without compromising quality) as extra processes are required for your growing company to maintain quality, and these are stifling speed.

By looking into the current approach to development, revealing its fundamental flaws and exposing the damaging consequences it can have on businesses – this paper presents our preferred solution:

INTRODUCTION

In the following pages you'll read about the dilemma that affects Software businesses — the twin pressures of speed and quality!

There are two distinct methods used to deal with the complexities of managing software teams:

The first, is current standard practice; what we call the capacity-centric approach;

Sprints.

The second method, our recommended approach, is the bottleneck-centric approach;

Pace.

Capacity-Centric Approach

This approach proceeds from the assumption that smaller, planned iterations of work are more responsive and appropriate for faster and better quality software development.

As a result, planning occurs to short term due dates based on capacity (resulting in overloaded development teams, rushing to hit targets, and ultimately reducing quality.)

Bottleneck-Centric Approach

This approach recognizes that every team has a key individual or two, that gates the total output of a team, a bottleneck if you will.

It then follows that a 20% incremental increase in the bottleneck's capacity equals a 20% increase in the total output of the team.

If the argument presented in this paper is correct, the implications for the industry are considerable.

If, as argued, the traditional approach exacerbates the very problem it attempts to solve, it follows that this method is taking a tremendous toll on quality and speed of development!

Our recommended approach is one that teams can use to dramatically improve the quality and speed of development, while still retaining the responsiveness that customers demand.

It has the potential to provide Software businesses with a considerable competitive advantage and ongoing productivity improvements.

The background of the entire page is a solid dark blue. In the top-left corner, there is a small cluster of three white squares: one is positioned directly above the other two, which are side-by-side. In the bottom-right area, there is a larger, more complex arrangement of white and dark blue squares. This arrangement includes a vertical column of three white squares, a horizontal row of three white squares intersecting it, and several other squares of varying sizes and colors (white and dark blue) scattered around, creating a pixelated or mosaic-like effect.

A day in the life:

DAVE'S FIASCO

Dave doesn't know it yet, but he's about to have *'one of those days'...*

12.15pm

He's been at work less than an hour, and he's already dealing with an emergency.



1.30pm

The emergency is finally dealt with, but then an urgent email arrives.



Sally, one of the product owners, is on annual leave this week; however, she neglected to tell everyone (or just forgot) that a bug fix was promised to a major client by Monday. Unfortunately, it's now Tuesday morning and the bug fix wasn't planned into the sprint. Fifteen minutes ago, Dave took a frantic call from Derek in customer service, demanding to know what was going on.

"I have the customer waiting on the phone right now, and they're NOT happy!"

Dave begrudgingly promised to look into it immediately.

"I have absolutely no idea what's going on—it's Sally's

job to deal with customers, not mine," Dave told Ritika, a senior developer. "And I don't have the capacity to add more work—I have a backlog of my own bug fixes that I need to deal with today."

"Well, if we don't expedite this one, Sally's gonna rip into you when she's back," Ritika said. "Also, we could lose the customer. They're very upset—apparently, they expected the patch to be deployed first thing this morning. Until Sally gets back, you're going to have to deal with this."

Dave sighed, imagining the ticket was not even properly detailed in the system. "Okay, let me try and find it."

Dave has settled in with a cup of coffee, but before he has a chance to start catching up on his Slack messages, he's interrupted by an email alert from Derek.

"An urgent feature request has just come through from sales. Can you get the design spec'd today so I can get the signoff from the customer? Jenny over in sales promised we could get the design sorted today. If we drop the ball on this one, it won't look good."

Dave put down his cup of coffee, now cooling down, and wished they would just finish the work that had been started first. But that wasn't the company's way. Their unofficial motto was "Just get it done." It seemed as though both his coffee and the Slack messages were going to have to wait.

“

How can I make sure the next Sprint's planning session captures all these bug fixes and urgent features? If only I had a few more developers...

”



5.25pm

Dave has his fingers crossed; the last half hour has been free of interruptions.



7.45pm

Dave's been putting out fires all day.



He's been able to specify some urgent designs and actually get some coding done. If he can keep this up, he might actually finish work almost on time today. He's about to move to the next thing on his list when the phone rings, ruining his productive streak—it's the accounts department.

"We have a minor invoicing crisis," Henry from accounts said brusquely. "I need your help to sort out a mix-up."

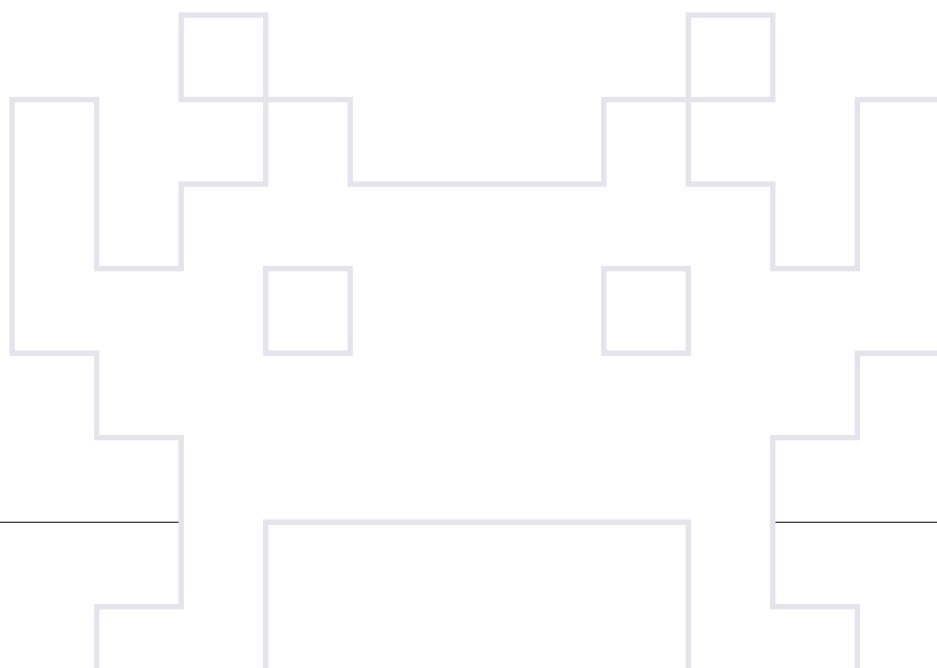
Dave groaned. He could tell from Henry's tone that this would consume the rest of his afternoon. "Mix-up? Did the team track their time improperly again? Never mind...who's the customer? I'll be right there."

He can't find time to get the urgent tasks processed, much less the non-urgent ones. Ritika dropped by his desk on her way out the door.

"Have you been able to review Sam's code yet?" she asked.

"You've got to be kidding," Dave answered without turning around. "I've had interruption after interruption since I arrived today. I've been putting out fires left, right, and center. I can't find time to get the urgent designs spec'd, much less do reviews."

Ritika nodded sympathetically. "Let me talk to Chris about it. Maybe there's something that can be done to ease the load."



WHAT MAKES DAVE CRY AT NIGHT?

The experienced developers necessary to do the complex work are often overloaded and difficult to find — it feels like a downward spiral.



Sadly, what Dave can't see is that the way he plans and executes Sprints is what is behind his pain...

In the software industry Dave and Chris' experience — although fictitious — is more the rule than the exception.

Almost every software business has its share of urgent requests, software disasters and unprofitably billed work.

Yet, like Chris, owners have little room in their cashflow to add devs. Margins are too tight, and the experienced devs necessary to process the complex work are difficult to find — and often costly to employ.

Of course, both owners and their staff are extremely diligent in their attempts to avoid mistakes that can cripple productivity for days.

Yet almost always — despite their best attempts to encourage better teamwork and communication — crises occur.

At best — as we saw with Dave — favors get called in and the work is expedited or apologies are made over and over.

At worst, you lose a customer or devs spend too many nights burning the midnight oil to fix things!

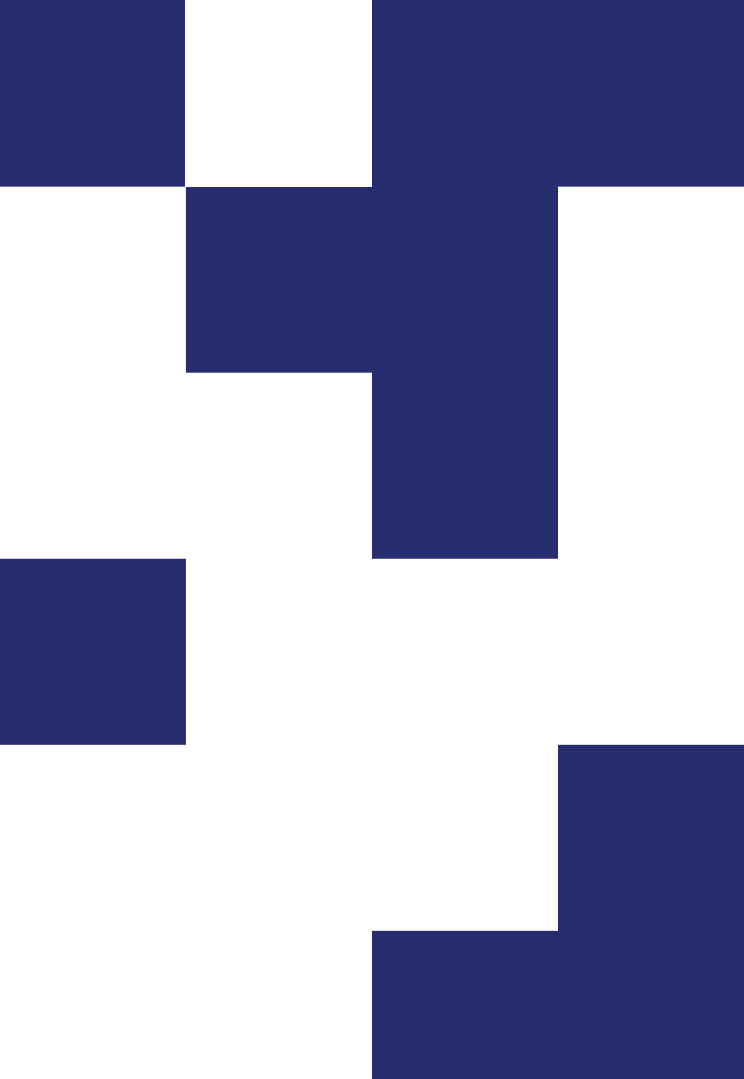
As a result, development tends to be characterized by panicked switching between fixes for quality and urgent delivery of new features, and ending up in a constant conflict between trying to slow down to get it right and speed up to keep customers happy.



SPRINTS:

The wolf in sheep's clothing

In light of traditional waterfall project management methods, impeding customer satisfaction, Sprints are meant to improve agility of software teams through responding to change and increasing customer collaboration...



Teams running sprints get caught in a number of conflicts that create a balancing point for them to either make a decision on, or manage regularly.

Let's start with one which is often decided early on in their implementation of SCRUM and doesn't change often after that. Sprint length.

If we make our sprints shorter, we plan more and more frequently and spend more time deciding what to work on, which decreases the time we can actually spend 'sprinting' and writing code. So we should make sprint lengths longer, however the longer the sprint the more things change during the sprint, and the more we have wasted effort on outputs from the team that aren't actually useful.

The longer we sprint, the longer we get to traditional project management and the issues that come with that in software development. The shorter the sprint the more we get the benefits of sprinting and the difficulties that come with it.

Teams that have a 'process of ongoing improvement mentality' end up at 1 week in most cases.

Those that find that a bit too hard, change back to a more comfortable 2 weeks.

We'll get into that more later as iteration length (sprinting or moving at pace) is a critical element. For now let's look at a few more of the common sprint dilemmas.

Planning for capacity, or planning for an outcome. We feel we should be focusing on an outcome, agile theory says so, it's what we've heard the top dogs in the industry do, but when we try to plan our work around a specific value drop we find it doesn't fit into our abstractly chosen 2 week deadline as neatly as we would like.

Who would have thought that modern day employment standards, based on the speed at which the earth rotates, crossed with a week based calendar designed over a thousand years ago, based on the waxing and waning of the moon, doesn't fit with the time it takes for a team of people to create a meaningful chunk of software?

So where does that leave us? With work that is either jammed into 2 weeks, expanded to fit 2 weeks, and with a few team members that aren't loaded up, having wasted capacity.



Our other team members are either unable to get all their work done, or in their efforts to be efficient, they're limiting the output of the expert resources, and therefore the team.

”

So we decide to plan based on everyone's individual capacity, but in doing that we haven't accounted for certain expert resources in the team who constrain the team's output (a lot more on that later). This means our other team members are either unable to get all their work done, or in their efforts to be efficient, they're limiting the output of the expert resources, and therefore the team.

This doesn't even take into consideration all the Grey Time the team has in their handovers, or completion of work in general. Ultimately we end up with undue stresses on the team at never meeting their sprint targets such that we feel pressure to go back to planning and revisit the sprint.

This balance will tip, based on the general ability of the people involved, to put aside efficiency metrics to deliver outcomes. Everyone being busy isn't a good thing (there's a reason emergency services have spare capacity and aren't measured on their 'productive' time). In software this will generally tip towards the agreed target outcomes in smaller start ups, and towards simply 'being busy' in larger businesses.

Which leads us to the dilemma of how to handle the work that isn't done during the sprint. We could ensure everyone in the team

has a bad Thursday night every fortnight, and depending on the severity of the poor sprint planning or execution, a bad Wednesday, Tuesday, Monday night, or even a 'what weekend?' scenario.

Dev's are far from stupid, and you don't need to be particularly intelligent to know that you can always gain time from cutting corners. Whether that means dropping quality reviews, or just shortening them to a quick skim, we get more done on time, but now we have rework that has to be dealt with. Whether it be caught by the team, caught by the customer as a bug in a few weeks, or just added to that technical debt piling up like rising ocean temperatures that we just turn a blind eye to.

If we've had enough of all that rework, we decide to start rolling some or all unfinished sprint work into the next sprint. This is a great way to show that deadlines don't mean anything, they're just something we have to pay lip service to because we're not really sure how else to track work progress against something.

This of course means the team can slow down, until someone gets fed up with the slower rate of progress and decides 'drop dead' means 'drop dead' when it comes to due dates.

Erratic decisions about how to plan and execute sprint work don't get much more erratic than when the unplanned work tries to be squeezed into a current sprint. The team is sprinting away pretty well, until the shadow of 'that' manager appears at the end of the aisle.

'That' manager might be from sales and has just promised some great new feature, customisation, or integration to land an important new sale. The manager hopes you'll see this from the company's point of view, and not think about the fresh commission check about to land in their wallet.

'That' manager could also be from customer service, and is just sick of being yelled at by a certain customer. Even worse, 'that' manager could be the founder who just wants to make his dearest, oldest customer happy.

Either way the interruption breaks any hope of completing the sprint on time and on spec. Something will have to give, and the turbulence generated by putting down half completed work to be picked up again at a later date will never be recovered by the team.

In some teams they have the freedom to decide the outcome of this conflict themselves, in others someone else in the business makes the decision.

On one hand, shoving work into the current sprint causes interruptions, which in turn increases multitasking, work-in-progress, and touch-time per task.

On the other hand, not shoving work into the current sprint means 'urgent' requests from 'important' customers take too long to action. This, in turn, raises stress levels of 'those' managers, as well as the resentment levels of the dev team. This puts the pressure back on shoving work in, with the dreaded "just get it done" request.

A similar situation, where the team has more control, is the volume of bugs to squish in each sprint. When more bug fixes are planned in, perhaps because the team feels there are too

many outstanding, or perhaps there were more serious level tickets raised in the previous few weeks, it displaces business as usual, whether that be new features, customer developments, or roadmap items. This in turn slows down progress of product development.

Slowing of progress is of course a slippery slope, as there are so many important things that can chip away at time spent until we are making little progress, and the market is starting to catch up or leave us behind.

These pressures present themselves in different ways for investors, sales and management, and the result is a negative impact on roadmap

milestone bonuses. Ultimately these competing forces lead to increased time spent on progress work, and less time on bug fixes. Fewer bug fixes means there will be more bugs, and more bugs make the users mad, which means the next sprint will be planned with more bug fixing.

All this work needs to be managed somehow. A simple and common way to increase visibility of work progress and therefore sprint execution management is with a task board. Referred to as Kanban boards by most software companies, they will show the planned tasks for a sprint and their current state broken across three categories, 'not started, working, done'.



The detail beyond this basic setup is where the 'more' vs 'less' dilemma arises for teams. The boards can have more detailed information on them, such as updating estimates once started, showing which tasks are being actively worked on, having some kind of progress vs planned indication (such as a burndown chart), and many more options.

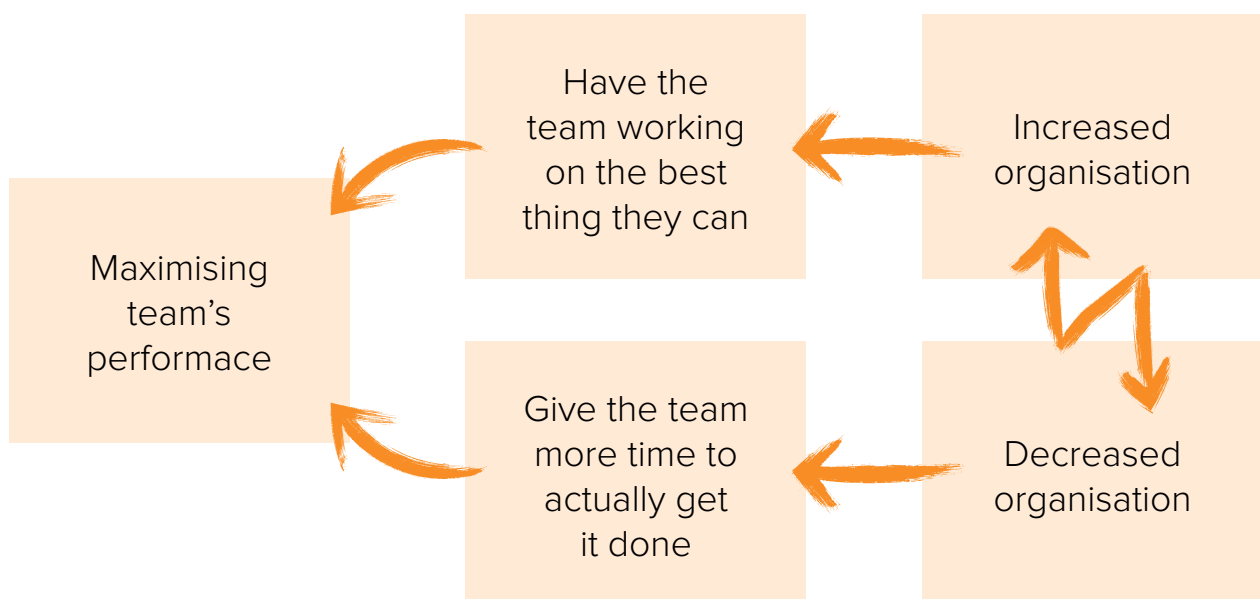
The more detailed the boards are, the more visibility and assistance in effective sprint execution they bring, however that comes with the need for more frequent updates. At a certain point these updates become a frustrating admin burden which take time

and interrupt other activities that ultimately mean less quality dev time. This leads teams to decrease the update frequency and detail, and in doing that, decrease visibility; which makes it harder to deliver at a smooth consistent pace.

These common dilemmas and many more take place, some daily, some monthly, some only once, in software teams around the world. They all experience this balancing act, which we refer to as a 'Figure 8 Loop'. Figure 8 Loop is visual representation of the loop we get stuck in when trying to do less of action X, which unwittingly gives us less Y, which puts

pressure on us to go back and do more of X, to get more Y, ... and on and on it goes.

There are a finite number of Figure 8 Loops that are shared by business in software development. For example, Figure 8 Loops centered on process and productivity come down to, 'increase organisation to have the team working on the best thing they can, vs 'decrease organisation to give the team more time to actually get it done'. Of course 'increased organisation' and 'decreased organisation' are in direct conflict, but both are in pursuit of maximising team performance. A simple diagram to show this looks like:



The background of the slide is a solid dark blue. In the top-left corner, there is a small cluster of three white squares arranged in a 2x2 grid with the bottom-right square missing. In the bottom-right area, there is a larger, more complex arrangement of white and dark blue squares, resembling a staircase or a large, irregular geometric shape. The title 'CHRIS' DILEMMA' is centered in a white horizontal band.

CHRIS' DILEMMA

The frustrating thing is, no matter how much “hoop jumping” Chris does, by expediting a new feature the customer desperately needs, his clients show little gratitude.

”

Remember Dave? If you felt sorry for him, make sure you also spare a thought for his boss, Chris. Chris is the hardworking owner of the company Xyphyr. He works long hours, and so does his team. In almost any other industry, they would all be reaping rich rewards; however, endless competition and never-ending customer demands constantly pressure his cashflow, making it a constant headache to attempt to turn a profit. Chris thinks, We have no other options — we either drop prices, or we increase the feature offering to match what other companies are doing!

In addition to dealing with unexpected emergencies, Xyphyr also has to jump through hoops to win new customers. Not only do those new customers want bug-free software, but they also demand ever-increasing amounts of new features and customization. One customer demands tailored calendar integration for its HR processes. Another wants specialized email tracking. And yet another demands accounting integration from an obscure company.

This not only adds to the cost of doing business (placing even more pressure on his margins), it increases the pressure on Chris’ already overworked development team. And the frustrating thing is, no matter how much “hoop jumping” Chris does, by expediting

a new feature the customer desperately needs, his clients show little gratitude. They’ll probably be back on the phone in a week, demanding another bug fix.

It’s this kind of persistent problem that is currently the subject of a heated discussion between Chris and Ritika. Everyone else has finally gone home — even the cruelly overstretched Dave.

“We need this new feature,” Chris stressed to Ritika. “Desperately.”

“Chris, I’m telling you — development has absolutely no capacity to deliver it. With Sally away, Dave was completely overwhelmed today. I’ve been hoping he would get a chance to design that new feature for the customer that Jenny won the other day, but he spent most of the day putting out fires. It’s not going to be much better, even when Sally gets back.”

“Well, that will change once Jenny starts losing clients because we can’t match what InstaHR or FaceHR are now offering to every client,” Chris pointed out.

“Discount again?” Ritika suggested helplessly.

“You’ve seen the P&L for the last six months—there’s absolutely no fat left on our bottom line to discount any further.” Chris sighed. “We have two tools in our kit—a hammer and a screwdriver. That’s all.

We have no other options — we either drop prices or we promise customizations to get ahead of what others are doing. And we can’t do any better on price than we’re currently doing. If we do nothing, we’ll bleed the regular customers who keep us even marginally profitable. And perhaps if things go well, we’ll be able to add more developers to cope with any increased backlog. But there’s no way I can afford to add anyone else right now.”

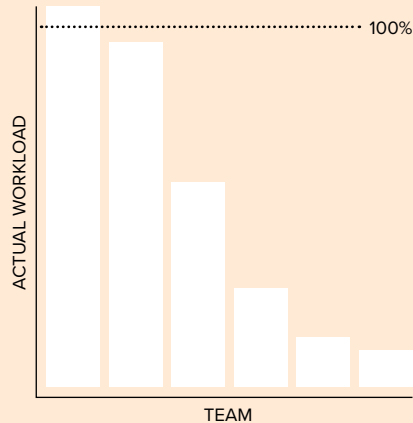
Ritika was lost for words. She knew that it was wrong to add yet more customizations to her already overworked development team. Intuitively, she felt that it would be a disaster — further increasing already slow delivery times, adding to the number of critical bugs and near misses, and leading to more and more unhappy clients. And the way reputation tended to spread online, it would make it harder to win clients in the long run, not easier!

She also knew that, even though Chris understood exactly what she meant, she would never convince him that offering the extra features or customizations was actually counterproductive. They were simply stuck between a rock and a hard place. As she left for the night, Ritika wondered how on earth she was going to explain this to Dave and the others the next morning...

BOTTLENECK LOGIC

When more work has to go through the same people that are already working at, or near capacity, there is inevitably going to be an overload.

- ▶ Output is slow
- ▶ Capacity is constrained
- ▶ Stress levels of the overloaded individuals are high



In any environment where you have a mixture of skills, you are going to have the most experienced or the most highly productive people on the team (you know who they are!).

Most software dev teams have a couple of top people that are loaded more so than anyone else on the team. We refer to them as 'bottlenecks'. Then there's a few more, that are decent devs, but are limited because so much stuff has to go through the top two. Things like tests, reviews, anything particularly tricky etc.

Then you will have, let's say, a couple of fairly junior people. While it looks like they are productive, their actual output can be very low because they are limited by all the others. The others still have their own work to do, and they need to interact with these more junior resources.

So our top two devs are going to be fully loaded. In fact if you visualise it, they're going to be overloaded (There probably isn't a software team in the world where these guys or gals work 40-hour weeks). They are nearly always doing a bit (or a lot) more than that.

And, they are essentially gating the productivity of the team as a whole. This is hard to see because we get the situation where, if there isn't much work on, people slow down, they find other things to keep themselves busy, and generally find ways of looking and being busy, but busy doesn't mean productive.

It can be difficult to see what junior people are actually working on. One of the worst scenarios is juniors finding and working on extra work that is not what has been planned into the iteration. Juniors, like the rest of the team, want to be productive so they resist being idle, and will look for other work to do if none is given to them. What they don't take into account is that, quite often, this out-of-iteration work further loads the already overloaded experienced devs. This extra work, while it may keep the juniors busy, may also have to be reviewed or designed by the more experienced devs (the

bottlenecks), thereby increasing their workload.

The bottleneck people can only do so much work. You are now wasting some of their time with stuff that is not necessarily valuable or relevant, or not the right place or time for them to be working on it. So, you start limiting their capacity, and because of the way the team works, the moment you limit the bottlenecks' capacity, you are limiting the entire output of the team.

So, the juniors can end up thinking they are being unproductive, and usually take the initiative of "I will go and find something to do." And this is one of the major issues with the way teams plan sprint work. They plan based on the capacity of the individuals, as opposed to the potential output of the team; they fall into this trap. The junior devs are woefully underloaded, so we find stuff to jam into their workload, but this actually loads up our most productive resources with work that is not necessarily the best thing for the team to be doing at the time.

The background is a solid dark blue. In the top-left corner, there is a small cluster of two white squares. In the bottom-left area, there is a larger, more complex arrangement of white squares and rectangles of various sizes, some overlapping, creating a stepped or architectural look.

CASE STUDY:

Lufthansa Technik

Companies that use Critical Chain Project Management (CCPM) manage bottlenecks well.

Managing bottlenecks is not relevant to industries that deal with one-off projects (which by default

do not suit the iteration approach), or industries such as manufacturing, with large, fixed projects, with known steps completed over and over again. However, there is an industry where CCPM is more comparable to software; the Aircraft Maintenance, Repair, and Overhaul (MRO) industry. MRO is appropriate because the

flow, or task types, are the same in each project, but the unknown variability is extremely high. Let's look at an example, then figure out how we can maintain the principles while scaling down the project length into iterations which maintain the benefits we already have from sprints.

LUFTHANSA TECHNIK

Lufthansa Technik is the MRO subsidiary of the Lufthansa Group. They complete what are known as 'checks' on Aircrafts for both Lufthansa and many other airline customers.

There are primarily two types of checks that the MRO does, the A checks, and the C checks. The A checks are more frequent, smaller checks, similar to getting an annual service at your car dealer. C checks however, are significantly more involved and require pulling apart the aircraft and engines to go over it with a fine tooth comb.



By breaking the work into smaller packages, the team can get better visibility of progress and much faster handover of tasks.

”

In any check, but especially a C check the variability of what needs to be done is massive, and the project teams have no idea going in how long or complicated each aircraft check might be. This is preventative maintenance, rather, than fixing something not working, but the similarities to debugging are there.

An engineer looks into a part of the aircraft only to find something more that needs fixing, and something more after that. To add to this, not only is the aviation market growing rapidly, so too are different requirements from various new aircraft types entering the market. (This of course mimics the software environment perfectly, with its rapid new developments).

This is a daily headache for many MRO project managers, and leads to firefighting being the norm in the industry.

Going into a project most MRO teams will find it hard to know where the issues are going to arise, and what bottleneck is going to slow down each aircraft in particular. This leads to a lot of multi tasking among the teams, from jumping around various tasks, to even worse, wasting the use of the highly skilled staff.

Estimating the time work takes becomes a wild guess as the Grey Time waste from pick up/put down, multi tasking, and

waiting on other people destroys any hope of accurate handover or completion times.

So what are the rules Lufthansa Technik uses from CCPM that give them, what they refer to as the ‘Magic Formula’ which frees them from all this chaos?

Low work in progress

Lufthansa break work down into smaller packages, with fewer tasks in each work section, contrasting this with the use of major milestones in traditional project management. By breaking the work into smaller packages, the team can get better visibility of progress and much faster handover of tasks. Once this work is broken down the emphasis is on quick handovers to the next team, and quick starts on work that is handed over.

In any environment with deadlines or major project points, people tend to put off completing the work until they need to (known as The Student Syndrome). The way Lufthansa Technik combats this is in two ways.

Firstly, people are set up to be ready to receive work and start it immediately. If someone in a chain of tasks finishes early but the next person is not ready to receive the handover, or start the next step, the time gain is lost. If the first person in the chain is late, and hands over late, there is

clearly loss on the project. This is true across every step, meaning that gains are always lost and losses always accumulate.

Secondly, when planning and estimating work, the buffers people naturally build into tasks are taken out and put at the end of the project. This is particularly beneficial when people are making the most of this extra time with slick handovers. The total buffer can then be used to visualise and manage the project which is the next major rule.

Buffer management

The buffer is used for tasks that need it, allowing for things to take longer than expected and for even the odd blow out without pushing the project beyond its overall time plan. Project Managers then monitor the status of the buffer and react to a rising buffer. They don’t need to look at every task’s progress but instead focus only on the tasks which are consuming the buffer. The team can then give their own attention to resolve the difficulties at these steps and act proactively as tasks increase in risk, rather than reactively after the project has gone off the rails.

This new reporting and focusing method gives new attention to management reports and productivity focus shifts away from ensuring high levels of individual productivity (which is the norm in MROs) to focusing

on the lead time instead. The time an individual takes to complete work is irrelevant, the time the whole team takes to complete the project is what matters.

All this visibility allows both the internal team and the external customer to have a clear and accurate view of the progress of the project, so everyone is kept up to date.

Don't start what can't be finished

The third of the major rules is to ensure all the pre-requisites are cleared before any work is started. This is implemented by breaking the projects into two stages. In the first stage, called Full Kit 1 the team will depannel the plane for inspection, and go through a thorough evaluation of the state of the airframe and engines. Routine tasks are then carried out. Then the team preps for the second stage by getting Full Kit 2 ready, based on actual findings for that individual aircraft.

Traditionally MROs would rip into any non-routine work following the age old principal of the sooner you start the sooner you finish. But this just increases work in progress and multitasking when the materials and tools needed aren't available. The two stages to the project allows the team to clear all the pre-requisites to be able to complete every task in quick

succession and hand over smoothly to the next engineer.

How this applies to the Software industry:

There are two primary similarities between MRO and Software development. Firstly, the work is consistent in the tasks themselves that are executed. That is, you can define the steps of tasks for a standard operations procedure in an MRO project, and a Software iteration. And the variability is high and unknown in both, prior to work starting.

The second similarity is the presence of highly skilled people, being Engineers in MRO and Software Engineers in Software. As we know from what we discussed earlier, any environment with highly skilled people working in a team will have natural bottlenecking by the skilled people.

They are the experienced ones, so most likely to be loaded with tasks, if anything goes wrong or is harder than normal. They are the ones who will be called on to fix any issues, and the less experienced resources will rely on their input in such a way that giving more work to the less experienced people loads up the experienced people and slows down the whole team. So like Lufthansa Technik, we need to move away from a system that focuses on individual's inputs, and focus instead on team output. To do that we will need to implement



the same rules, although the mechanics of how the rules function will be different in our environment.

We need to focus on the flow of work through the bottleneck, which is our most experienced resources. Much like Lufthansa Technik, there are huge benefits to reducing work package lengths and the batching effects that go with it, moving our focus to buffer management of very small, but fast flowing pieces of work. Let's look at how we can apply these principles by changing our current standard practice approach of sprints.



INTRO TO PACE

a bottleneck-focused
alternative to Sprints



Shorter length sprints bring with them a lot of benefit, however, also bring more downsides in other areas.

Essentially the shorter the sprint is the more 'agile' the team is. This means the team experiences the extremes of agile, both the positives and the negatives. What if we took it to the extreme of a single-day sprint, as an exercise to emphasize the impact of the positives and negatives.

As we reduce the length of the sprint we increase the accuracy of estimates, as they are smaller lengths of time to estimate. We also don't have the sequence issue of estimating future tasks days or weeks in the future, which are contingent on the tasks before them. This causes any estimating errors to ripple into future tasks and the negative effects of this to compound.

With the shortened sprint length quality steps, such as reviews, can be introduced more frequently. When quality steps are delayed or batched together they are less effective for two main reasons. Firstly, people are more attached to the work they have done when it has taken them longer time or more effort to complete, so they are less likely to make changes

that increase the quality of the code, whether this is increasing the likelihood and number of bugs, or contributing to the technical debt.

Secondly, the review will be less thorough as the reviewer will tend to rush through the review just wanting to get it done as the review length increases.

With smaller and more frequent quality steps comes less rework, work is more likely to be on the right track and the rework that is done is smaller and more contained. As the length of the sprint decreases, the interruptions also decrease.

People are less likely to feel they need to interrupt the sprint if they can just put their 'urgent' tasks in tomorrow's release of work. The longer the sprint is, the more likely it is that pushy sales managers, or founders who are excited to help out their oldest customer, will want to, get it done now!

Shorter sprints also allow for deployments into the code more frequently and consistently, which for some is of extremely high value. Sprints create much smaller more frequent due dates for work, when compared with traditional project management.



The idea of short sprints brings huge benefits at even greater costs. What if there was a way to plan and execute dev work in a different way with more upside and less downside!



The longer the due dates are, the more the potential damage from surging is, in the sense that devs have more time for their work to get away on them. The longer the time gets away, the more there is to catch up on, in a surge of effort to try and hit the due date.

With those considerations in place, the negatives of agile will also impact the team when taken to the extreme. In fact the impact must be significant as it outweighs the positives we've just looked at. I say this because I'm yet to hear of a sprint that is less than five business days, and most software companies have fallen back to the two-week standard sprint.

Planning becomes far more difficult for managers as the plan changes essentially every day, with backlogs being chipped away at day by day. This would give both managers and people from other parts of the business relying on development very little visibility of what will be coming up in the future even if that future is only a few days.

Will that bug be fixed by tomorrow, within our service level agreement? Teams and devs already tend to be adverse to planning, so taking the first section of each day to plan, in place of a normal standup, might encounter significant resistance. Not to mention that all the devs would have to be at work at the

same time, which would destroy the beneficial flexible culture that many software businesses promote and enjoy.

Worse than that we all know how brief meetings blow out to become anything but, meaning our planning session could easily take a very large chunk of the day.

Considering we discussed bottlenecks in the previous chapter, it's important to consider how they would be affected, or affect others. With very short sprints it's likely that most people, other than the person doing the task, will be sitting around waiting for their turn. While we could go ahead and load in more tasks, the bottlenecks will probably limit what can be done in a day, leaving others to be doing very little, as they cannot work on 'tomorrows' sprint to get work ready for the bottlenecks.

Finally the short time frames with a daily deadline allows for almost no flexibility in timeframes for when estimates go to plan meaning people will come in early and have nothing left to do, or more often come in late on the estimate and the pressure to skip other steps including quality steps will be high.

To conclude, the idea of short sprints brings huge benefits at even greater costs. But what if there was a way to get those

benefits without the costs? If we could plan and execute dev work in a different way with more upside and less downside!

Many of the issues brought about from sprints come from the fact that work is batched together into a chunk (normally one or two weeks, worth), this then has a deadline to be met. Essentially this is using traditional project management (i.e. Waterfall) but in shorter batches or iterations to reap the benefits of such. But it's the hang overs from this traditional method that are causing the issues.

Even considering the name, sprint, hints a strenuous burst of activity (one that can't be maintained) towards a short term goal, that require us to stop, take a breather, and the do it all over again. More appropriate is a smooth and steady pace which is sustainable, and doesn't have deadlines. Deadlines, which are in place to drive time management, are largely arbitrary. Who says a feature set takes 40 or 80 hours for a team to produce just because that's how many hours we decide to work in a day or a week?

Why the focus on deadlines? Deadlines contribute to two productivity killers that are built into human nature. First being 'Parkinson's law' being the adage that "work expands so as to fill the time available for its completion". The longer we have, the longer we take to do something. Whether



we slow down or we take more time to polish our work, this is a well observed behavioural pattern. The second being 'The Student Syndrome' which is when people start tasks as late as possible and eliminate any of the positive benefits of buffers built in to planning and estimates, and put themselves under unnecessary pressure and stress.


What if instead of this we had multiple 'sprints' operating at any time, and as one finishes the team rolls on to the next one. This allows the planning of the length of the sprint to be appropriate to whatever the workload is for that sprint. To address the issue I mentioned earlier of people not having work to do we would have

more than one sprint taking place at once. We could have a few depending on how many people we have in our team and how long a sprint naturally ends up being for us.

What do we do about the deadlines? They would become rather difficult to manage in this system, and as we've looked at already, they cause a lot of negative behaviours. So why have deadlines at all? The purpose is to help ourselves, as individuals, and managers, to manage progress against a benchmark. With no time management, everything would take a long time. To replace the deadlines, we would use the estimated time plus a 'time buffer'. It is unrealistic to think everything we get done meets estimates and also unrealistic to think a single time and date matter to a single piece of work (except where a commercial deadline exists externally).

Adding a time buffer and tracking the progress of time from giving the iteration to the team, to when the team starts working on it, gives an appropriate time-bracket in which the iteration is likely to be completed in. A time-bracket makes more sense than a hard due date.

Given the multiple iterations being worked on by the team, some iterations will take longer than estimated, and some will



come in on the lower end of the estimate, as normal variation is experienced in software development.

Over the multiple iterations, these will average out to be a smooth and consistent rate of development. This pace of development is then used in place of 'sprints' for planning and delivery estimates. With no due dates to plan iterations around, work in play, against the capacity the team has, is used to determine if more iterations should be in play for the team or they have sufficient work to work on.

For example if you have a 5 day buffer of work to complete iterations in and there is a person or capability in the team that has 35 hours' worth of work already, any new iteration could only be given to the team if there is 5 hours of work or less for them in the new iteration. This system maintains an appropriate number of tasks to keep the team productive, however not so many to allow work in progress to build, or for multi-tasking to occur.

The process of planning the iterations, estimating, and assigning the tasks becomes decoupled from starting on the iterations. Unlike a sprint, teams no longer plan the iteration then immediately begin work on it. Planning of iterations can take place whenever best suits the team. This might be weekly, it could be individually when a dev gets, or needs, a break, or,

in some teams we even use a co-ordinator role to plan iterations and queue them in front of the team, ready to go as the capacity availability becomes available.

Many of the other rituals that teams use around sprints and SCRUM exist in Pace. Daily stand-ups are implemented to discuss the status of tasks, primarily around the ones at risk of running over, discussing what is being done to escalate them, or if there is a way other members of the team can help. In some cases this is just clearing what they are doing to be ready for the handover, minimising the downtime of the iteration and ensuring it is finished as quickly as possible.

Handovers on a task whose iteration is in the 'at risk' zone and handovers to a constraint capability or person should be performed like a relay race. The next person should know the handover is coming and both have everything they need to receive it and run with it. This is another step to eliminate WIP piling up or people multi tasking.

A team working at Pace has a steady stream of work being completed and an equivalent amount of work being approved to be in play for the team. Anything that would once be an interrupt can now be at the front of the release queue, to be the next iteration for the team to work on.

A focus on 'chunking' down iterations to the smallest appropriate

size means the quality steps around code being written are smaller and more frequent. In other words, these smaller quality checks have a larger impact as people are more likely to be thorough, and more willing to make the necessary changes, because they haven't invested as much time in the code, that may make them resistant to changing it.

Importantly the workload on the team is balanced around the bottlenecks. The bottlenecks shouldn't be overloaded by others' work that requires their attention, and other capabilities can choose non bottleneck iterations to fill up their available capacity or just be ready to start and handover completed tasks quickly.



YOUR NEXT STEP

Pace Invaders is just an introduction to *Pace*—a new software development methodology that builds upon the principles of Agile to turn your software development pipeline into a highly-efficient machine for rapid output and growth.

If *Pace* resonates for you, here a few other resources you may also enjoy.

BOOK

Pace: Accelerating Performance in Software Development for Rapid Growth

This is Peter Cronin's comprehensive book on *Pace*.

Pace contains the blueprint for a new way to execute software development—one that finally brings planning and execution under control, so you can smoothly increase output without sacrificing code quality. You'll discover how to use *Pace*—a new system for software development made up of 12 counterintuitive but proven rules that lets you scale without losing the magic. And you'll learn a simple decision-making toolset for making snap decisions that harness growth—even in the face of imperfect information.

Pace is available in hard copy and Kindle on Amazon (search *Pace* by Peter Cronin).

WEBINAR

Developing at Pace

This is a no-cost, no-pressure, live webinar where you can start exploring the implications of the concepts presented in *Pace Invaders*. The webinar is facilitated by Peter Cronin, the author of *Pace Invaders* and *Pace*.

Peter will lead a deep dive into three underlying principles of *Pace*, and their practical application to software development teams. You'll come away with practical ideas that you can take away and apply for immediate business benefits.

You can read more and view dates at devpaceworkshop.com

PETER'S PRODUCTIVITY BLOG

Peter regularly writes about the challenges of software development and how teams need to reboot performance improvement discussions so they can scale at *pace*.

You can easily follow these discussions on his blog. Go to viagointernational.com and click on the Productivity Blog button.

PACE WEBSITE

This website houses everything related to *Pace*. Read Peter's 12 Rules of *Pace*—a set of breakthrough and counter intuitive rules and decision-making tools that encourages rapid growth without sacrificing quality.

And watch a collection of video classes where Peter Cronin breaks down the rules of *Pace* and explains how to implement them within your software development step-by-step.

Go to:

pacesoftwaredevelopment.com

PACE EXECUTIVE BRIEFING

If you are in a rush to get a handle on your software development problem and fast-track rapid, sustainable year-on-year scaling, this may be a fantastic 40-minute investment of your time.

It will give you a solid grasp of how the ideas presented in *Pace Invaders* apply to your organisation. Just email us at sales@viago.com.au and we'll work with you to set up a time that suits.



VI **ViAGO[®]**
International

www.viagointernational.com