A Systems Approach to Software Product Delivery - a compilation

v1.1s

Richard Bown

August 2022

Introduction

This is a compilation of my favourite articles, written over the last six months or so. When I started out writing, for the website and the newsletter, I was in the mode of a more general systems and project approach to everyday software product transformation projects. Therefore the articles start with some useful background on Conway's Law and approaches to how systems work, what they are, why projects are complicated. During the course of the year, I've moved my attention very much back to my core subject which is around software delivery, product development and DevOps. The articles are reflected in chronological order and are all available on the website.

A moment of epiphany came in July 2022. I was starting to talk more about the meaning of software delivery - how it bridges needs - the needs of the customer and those of the developer - and how as engineers or product managers we often cross paths and have the same goals but we can trip over each other.

This is why I've called this collection "A Systems Approach to Software Product Delivery" because it covers systems, products and modern development practice in a way that is practical and not just about development or product. I hope it sheds some light on our various needs and behaviours and enables us all (as those who love building and delivering software products) to both create and more easily support better quality software products.

Looking back is sometimes an important thing to do - to see how far you've come. So I really hope that you enjoy these articles put together in one place for the first time.

I look forward to refining their message in the months and years to come.

Thank you for reading!

Richard Bown, August 2022

https://richardwbown.com

For the newsletter: https://softwaredelivery.club

For the podcast: https://podcast.softwaredelivery.club

TABLE OF CONTENTS

Thank You, Mr Conway

There is verifiable wisdom in Conway's Law. The pattern it describes has been accidentally repeated thousands if not millions of times in software projects all around the world.

So what is it? Who is Mr Conway and why should you care about this?

A New System Appears

Have you ever worked with a new computer system?

Does it work the same way as your old system?

Does it do everything you need it to do?

Perhaps it does most of what you expect but that final, vital piece of the puzzle is buried somewhere else or completely missing?

Does this leave you pulling your hair out?

Your System is Dumb

When coming to a new system you bring your knowledge and expectations built up over many years. Your new system however is dumb.

It doesn't know you.

It doesn't know how you work.

It was probably created by someone that you never met.

The person who created this system had a different set of expectations, instincts and thoughts in their head. Perhaps they even just guessed how it should work?

You have undoubtedly made no detailed contract with them about exactly how the system should work. Now you're confronted with a system that is

forcing you to change your way of working or worse still doesn't allow you to do what you need.

That is not much fun. It can be both a highly frustrating and highly expensive business to fix these issues.

What is Conway's Law?

Your expectations are not magically baked into any new system. A developer, marketer or product manager somewhere will have made assumptions about how you work to shortcut the contracting or delivery process. A project manager may help you understand what gaps exist and suggest improvements but all of this requires a high level of knowledge and attention to detail.

Conway's Law describes why projects often deliver questionable results and can be more expensive or take longer than they should.

"*Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.*" — Melvin E. Conway

Melvin Conway is a software engineer who made this statement back in 1967. Since then many others have restated it, reworked it, giving it extra relevance but the essence remains.

In some ways, Conway's Law is a warning but also an opportunity.

What it says is that from inception, the system you build or the project you run to build that system, is essentially going to be a mirror of your organisational layout.

You cannot avoid the human factor. It's in all aspects of your computer system from inception to delivery.

Conway's Law, also known as the mirroring hypothesis, shows how organisational communication pathways, arrangements and structure will always be reflected in the system that is delivered.

This can happen even if your project managers, your developers, your analysts and testers are 100% on their game. You may still get a poor copy of what you had before.

All of the sign-off, authorisation and administrative functions that are in your organisation can become represented in your new system. You may end up with a finance module, a marketing module, a customer support module, a membership system and the way you use them will mirror the human pathways in the org and also the assumptions made by your integrators or developers.

Therefore the tendency is for any new system to not improve or streamline your processes. It instead bakes them into a newer more brittle form.

The system become a formalisation of your existing working practice.

Due to Conway's Law and due to inattention to detail you've built something that firstly isn't quite yet as useful as your old system and secondly works in exactly the same way as your old system.

Does it need to be this way?

Although Conway's Law is inevitable there are things you can do to avoid its effects.

View your digital transformation – your new software project – as a human exercise rather than a technological exercise.

Ask yourself what people are involved in the process now? How do you want them to be involved in the future? Ensure that you dispassionately design something which will do the job you want to do and not the job you're doing right now.

The rest will flow from there.

Just remember to say thank you to Mr. Conway.

How to Know When Your (SaaS) Systems are Broken

The developer wants to get on with her work but the scrum master says that she needs to join a refinement session.

The product manager wants to prioritise something for the customer but the CTO has decided that technical debt has priority for the next few sprints.

The support engineer wants to be able to fix the broken email problem that he has for a bunch of his clients. It's now a software change. The infrastructure pipeline is codified and the CI/CD setup doesn't have enough of the right tests in place to ensure that changes can be rolled out safely and automatically.

The business owner is on the phone with her clients the whole day apologising for performance problems and never seeming to make any headway in the cycle of releases.

Despite doing all the "right things" i.e. doing "Agile" and doing "CI/CD" and doing "DevOps", a SaaS business, or indeed any business, can be a battleground when there is a lack of clarity and a mess of systems getting in the way.

Priorities

Sometimes there is a divide between product (i.e. what the customer wants) and the technology (i.e providing that thing for the customer). Occasionally there are processes in place which are only really there to provide busy work for those who feel disenfranchised from the creative process. Couple these with a lack of leadership intent and you have a recipe for nothing – for stagnation – for arguments – for embarrassment.

Do you recognise any of these symptoms?

Do you wish you could make all the systems just go away and actually make things happen?

When you lead an organisation where these things are happening you need to ask yourself a few hard questions. Like:

- Who are we doing this for?
- Despite our problems now, what do I want to see in the future for our product?
- How do I want this company to be remembered?

Remind yourself that any company, any enterprise, should benefit the customer and benefit the shareholder (or owner) while also making the employee happy.

Leadership is required to make decisions that will bring about clarity. Decisions that will enable your company to start delivering on your promises – to yourself, to your customer, to your people.

Fewer Systems, Fewer Processes

Systems and processes are merely tools to help us achieve our goals. Increasingly the lines blur between processes, systems and tools. We lose clarity.

Has a look at this short shopping list:

Agile, DevOps, CI/CD, Jira, Azure DevOps, GitHub, QuickBooks, HubSpot, Salesforce, SAP, PeopleSoft, Scrum, SaFE, AWS, Kubernetes.

You shouldn’t need to worry about them. They are the distraction.

Worry about your customer, your staff and your bottom line.

Don’t worry about the systems.

Sunblock vs Suncream

Both sunscreen and sunblock protect us from the sun's UV rays. However they're not the same thing and they work in different ways.

Sunscreen is designed to be absorbed into the skin and needs ten minutes to get working. Sunblock works immediately – it has minerals in it which reflect or absorb the sun's ray.

Which is right for you? Immediate application and total block or wait ten minutes and get a tan?

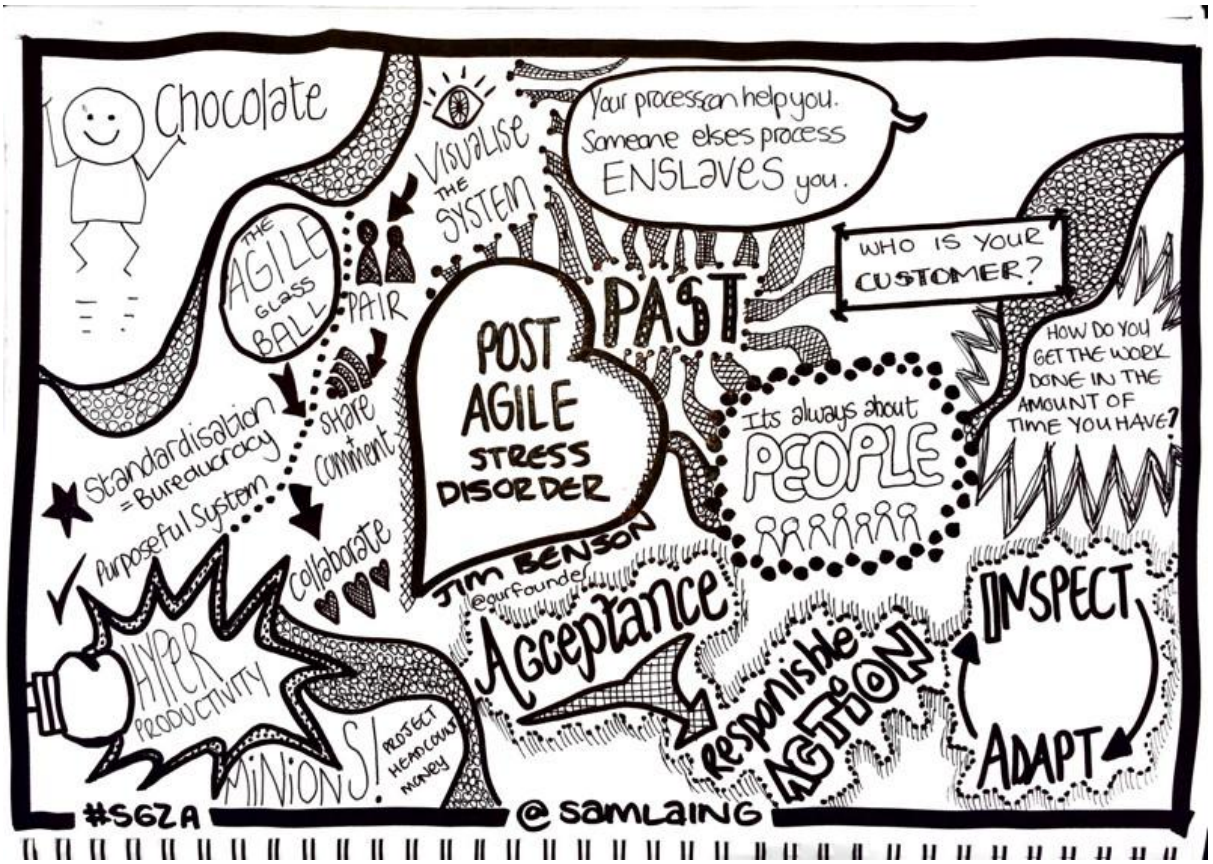
Like most things, there's a choice. You'll only find out how effective either choice is once you're tried it out.

Agile vs Waterfall Doesn't Matter: The Three Degrees of Perfect Projects

We are in a post Waterfall, post Agile world. So where does that leave us with our software and IT projects?

Agile started as a reaction to the Waterfall methodology. Waterfall said that software needed to be built from specification, tested to specification, released to specification. However Waterfall was always some kind of ideal which never truly existed. Likewise Agile is an ideal, a mixture of good practices and rules which have been spun into a theory of work. Neither of these 'ways of working' are hard and fast so why is so much time and energy spent debating the merits?

This amazing sketch from Samantha Laing after a talk by Jim Benson for me sums it up:



Post Agile Stress Disorder by Samantha Laing

Poor Old Agile

Agile is certainly not a new way of working. It's over 20 years old. The Agile Manifesto came about because frustrated developers, stakeholders and project managers said "we can do this better, we can work together better". It was also a reaction to the perceived inefficiency of existing project management and software development methodologies. What we have 20 years later is exasperation from many stakeholders and product managers that they still can't get what they needed built using Agile. Agile has spread to not only software development but how infrastructure, product discovery, and all sorts of other types of projects are implemented. The world that inspired Agile has moved on, but we're still think it's a good idea.

What do we need?

I've started to think recently that perhaps the fact that we think we need a methodology at all is actually over-complicating things.

I recently wrote the below comment on this excellent article:

The approach I've always tried to take with Agile is to sell it to developers as them being front and centre when it comes to running the project. This works for bullish devs who rate their ability and want to accomplish great things. For many who just see development as a paycheck there is no advantage or disadvantage to either method. What is more important is that everyone communicates better — rather than Agile vs Waterfall — build trust, communicate often, enjoy working together. Software projects shouldn't be a grind so let's work to not make them so.

Software projects shouldn't be a grind. But why so often are they?

We start projects full of hope and ambition. We spend hours, days, weeks, months — no time is too long, no effort is too great, no money is too much. However the point of any project management methodology is to help validate ideas as we go, to reduce risk, to reduce uncertainty. Waterfall, Lean, Agile, MVP. All words that we recognise and understand but how often do we truly validate the ideas or the project we're building?

In truth, everyone is too busy to validate. We trust it will work rather than planning to make it work.

Ideas have to be validated by people. Validation is a social contract.

The social barometer is the biggest indicator you have that your project is working. Checking everyone is ‘happy’ by taking a roll call or checking off a list is not really listening to your stakeholders.

So how do we ensure that our ideas are being built and delivered correctly?

We must make all stakeholders responsible for sharing the burden. And we do this with three simple things.

Trust, communication and alignment.

The three sides of the perfect project management triangle.

Trust — everyone in the project team from sponsor, stakeholders, project managers, testers, developers, infrastructure. Everyone must have trust and faith in the project and it must be demonstrated and felt by those taking part in it regularly. Where does trust come from? Transparency, professionalism, honesty, openness. No fear of hiding mistakes or blaming.

Communication — timing of communication, regular, open, explicit, clear. This works as much for internal communication as it does external. When stakeholders need to provide information to the project or are keen to show it off — they must show discipline and faith and trust in the project unless it’s agreed by everyone that the project isn’t working. This comes through…

Alignment — all parties in the project need to be aware of what’s happening in the project. A timeline, an agreed schedule of work, a prioritised backlog. Whatever you want to call it.

Essentially all of these three elements combine in providing us a continuous dialogue of the project. Keep it front of mind, in front of people, and make sure it completes on time, on budget and to the relief of everyone — it completes achieving everything you set out for it.

These three components — Trust, Communication, Alignment — are for me the most important parts of any design or project management philosophy. And you will see that they all find a place in any methodology already existing.

These however emphasise the need for us to recognise our humanness in the process. We are not machines, we have emotions. Projects, even good ones, are always an emotional roller-coaster.

Recognise that humans need to be catered for in the process, prioritise this, and make sure above all you show Trust, you Communicate clearly and often and you provide Alignment. For details on how you do those things — see the details of your favourite project management philosophy.

After all, Agile says we should value People above Process, right?

Engineering vs Product Management

The engineer has a mindset which leaps on solutions.

Because of the way that we are wired, our solution-orientation makes us sometimes harder to approach and more easily distracted by shiny things which to others might seem disconnected or unimportant.

Sometimes the leaps we make in our minds as engineers can help our business. In fast moving firms and startups, where it is nimble enough to pivot, the product can follow the technology.

Most businesses move slowly, because staying on message is important for any business. Consistent messaging driven through marketing, makes it easier for the customer to understand, trust and eventually buy from the business.

Once business has some traction, the ability for it to change becomes a potential risk to existing revenue.

As a leader, with your engineers hat on, when you push through your digital change project, how will it directly benefit your current customer?

Ask yourself if the cost of change will deliver direct benefit or if you're just making busywork? Then ask yourself how could that change provide benefit?

Suddenly you're thinking with a product manager's hat on.

How to get DevOps Insights You Can Trust

Throughout my career, I've built engineering toolchains to help software development teams deliver higher quality software, more quickly, with less fuss. I've also worked hard to improve feature throughput with pragmatic application of (Agile) project management.

Many of the engineering leaders I've worked with have realised the importance of tooling, but few have the luxury of someone dedicated to implement it. Being a software geek looking for an automation angle, I was always looking to optimize the development experience for those I worked with. So I stepped up. The happy accident of that meant that teams could focus on delivering more features and worry less about building, packaging and deploying the software. The boring stuff that ensures insight, quality and repeatability – the cornerstones of resilience.

After many years of building bespoke automations using scripts, they invented a word for all of this – DevOps. Off the back of the movement came a ton of products which help us now every day. The merging of Development work with Operations work which meant that faster feedback and faster fixing of failing or incorrect software. Nowadays there is a sea of automation and tooling options for all engineers to use off-the-shelf, but this doesn't always mean that we go faster or deliver more value.

Why?

Because we have too much choice in our toolchains, too many ways of doing things differently and it's very hard to see what's going on. Sometimes we even have multiple implementations of the same tool within our enterprise. How many engineering leaders end up building their own bespoke observability platforms? Too many.

Wanting to give engineers the freedom to play and experiment is wonderful, but how can you combine that with a vision, oversight and ensure quality and speed at the same time?

That, my friends, takes work. Break it out. Make it a work-stream. Treat it as a product in its own right.

It’s vital, so treat it with respect.

Building Better Software through Observability

Increasingly, when we talk about software systems we actually mean distributed software systems. They live on the network, in the cloud. They are not standalone.

Distributed software systems quickly scale to a point where it gets complex to manage them or know what's going on inside.

How do we gain more insight and ensure reliable software?

The key word is 'Observability'.

Observability platforms are everywhere. A few examples: New Relic, Honeycomb, Splunk, Datadog, Azure Application Insights, Elastic/ELK stack.

Observability is built upon the three pillars of Observability: Metrics, Traces and Logs.

But what does THAT mean?

Essentially three layers of data interpretation, some of it labelled, some of it unstructured.

Metrics – tagged with metadata, measures which enable an easy way of determining overall health and performance of the system.

Traces – giving more detailed indication of performance and functional insights into the application. How users are using it and how it's responding.

Logs – analysis of the raw logging information that comes from all parts of the systems and application. Used for remediation and failure recovery.

Observability platforms provide a way of ingesting and visualising data. Support teams and SREs (Site Reliability Engineers) can receive alerts as well as respond to incidents.

All production-quality (i.e. end user) software needs good metrics and logging in order for it to be fully supportable. The only way (well, IMO the best way) you can achieve reliable software is by gathering feedback – through observability.

When it comes to building software, don't leave monitoring and logging as an afterthought. It's an essential part of the feedback process of making your application more fit for purpose and better at fulfilling your customer's needs.

Cult of the Developer

Developer relations, developer retention, developer worship. Despite talk of mental health challenges, long hours, pressure to deliver – the developer is still king or queen.

I know, and like, a great many developers. Heck, I am one, although I've not developed full-time now for years. Maybe I'm not the best, but I speak the language(s) and I can hack the hacks and I have the experience of spending a lot of time building and supporting stuff that matters to people.

So, I can sidle up to the bar of development and order a drink and not look too foolish.

And where have we gone with this cult of the developer?

The world is in love and a little bit afraid of the developer. In awe of their prowess, in fear that they will not understand what we say and laugh at us, in terror of us deciding what to do and what not to do.

Yet, despite salaries being higher than ever and developers picking and choosing, there is almost so little choice in the world of development. While there are more languages and more frameworks than ever, something feels a little bland.

We have the 12 factor app – a way to build and deploy reliably.

Because the wheel has been reinvented so many times, in so many places, the world is awash with software. Open source, propelled by GitHub among other repositories, is there for us to pick and choose from. And that is now the role of the developer, to pick a framework, to pick a platform, and to glue them together.

It was ever thus. But then something like GitHub Copilot comes along and suddenly perhaps the balance is tilting towards systems rather than creativity? How much longer will the developer be a developer? And what does that mean in 2022?

What Kind of a Business Are You?

A while back I had the great idea of writing a web app that would do easy and secure video and PDF embedding for websites. I even got so far as getting a prototype working based on AWS cloudfront, mediaconvert and a React app. You login, upload your video or PDF and get a secure link back which is embeddable in your website.

While it's potentially useful, there are quite a few ways you can do this with other commercial solutions and they're not necessarily expensive (vimeo for example). So while it was an interesting problem to play around with, commercially probably not a starter while support-wise a potentially headache.

Before you commit significant resources to a development project, ask yourself, what kind of a business am I? Am I a business that would benefit from investing in the development of a bespoke system, or can I buy off the shelf?

Most tools are business enablers, not commercial opportunities. Focus on your core business and let the tools you buy and implement help you.

Are You Still Ignoring the Cloud

Amazon Web Services (AWS) is twenty years old (well, officially the version we use today is only 16 years old ... but the name is 20). We are two decades into public cloud and the confusion is still real. But why?

What does "going to the cloud" mean? And what skills do we need to work with it effectively?

Cloud means a large amount of IT compute and network services, many different things, but all things that we can buy right now with a credit card and start deploying straight away.

So the power of public cloud, is immediacy. Your business can grow, can change, can scale without having to wait for someone to raise a purchase order for a piece of hardware, or wait for a third party to get back to us about when they can rack it, or network it, or install it with software.

Public Cloud has revolutionised how the world does business because it allows for flexibility, speed and control.

Of course, because we are in the driving seat, we need to have the skills to be able to work with it. Networking, system administration, configuration management, security, compliance, identity and access management, plus many more concerns become ours, and only ours, to be responsible for.

While is sounds scary, a few people with the right skills can stand up and run and entire organisations IT infrastructure without every having to leave their cloud console.

So, what are you waiting for? Another twenty years to pass or is time to hop on?

Delivery Wins

You can choose whatever you like from the options.

Languages, tools, methodologies, ways of working, ways of being, reacting over planning, intricate refinement, gantt charts, kanban, extreme programming, SQL, NOSQL, cloud, on-prem, kubernetes, desktop, mobile, angular, vue, nextjs.

None of it matters without customers.

Once you have customers, you'd better keep delivering.

So remember.

It doesn't matter.

Just keep delivering.

Doing the Minimal Viable Product

Much is made of the MVP – the Minimal Viable Product – but so rarely is it executed in a meaningful and valuable way.

So how do you achieve that?

Take an idea, form it quickly and work closely with a developer, a designer and a marketer to realise it as soon as possible. Sounds simple enough but how hard this is in reality will depends on your circumstances. If you are independent and non-corporate then there are zero barriers to entry, so good luck and stay focussed.

If you have rules in the workplace about infrastructure, about coding standards, about security baselining, then you're going to struggle to make your product as minimal as you'd like. But perhaps your environment has thought about that and you can used a paved road.

Whatever path is open to you, your MVP won't deliver unless it has users. So along with speed you must ensure you make some noise.

Create a landing page for your app – describe what it's going to do. Get some users signed up to a mailing list.

The point of an MVP is prove that there is a need for something. To realise it you need technology, product design and product marketing.

The commitment of users wanting to use your MVP will drive you forward. Don't leave the marketing to last.

How the MVP Has Lost Its Meaning

Following up from yesterday's post on Minimal Viable Product, I received some great feedback on just how the MVP has lost its way.

A couple of classic warning signs are the use of multiple MVPs in a project, or using it interchangeably with a Proof of Concept (PoC). Neither of these get to the heart of the purpose of an MVP which is to deliver minimal functionality to the customer as soon as possible.

A Proof of Concept is often just that – showing how something could work and often tied to a particular technology. Multiple, planned MVPs in delivery cadence makes no at all – either you have a product or you don't. If you're trying out multiple MVPs then you're trying out different products.

Use of the MVP should be targeted on specific functionality and a specific product over a specific timeframe. Aim to deliver something that proves or disproves something for your customer.

Often the best way to achieve this is to focus on a particular end-to-end process for a customer. A vertical slice through the whole application.

Prove functionality throughout the whole functional and technical chain and you can easily determine if this adds value, while also proving the technology.

Release Cadence Considered a Poor Quality Metric

If you're doing DevOps and releasing frequently you may be aware of the DORA (DevOps Research and Assessment) metrics. These have been identified as four key measures for success in deployment of software changes and are summarised as follows:

- Deployment Frequency—How often an organisation successfully releases to production
- Lead Time for Changes—The amount of time it takes a commit to get into production
- Change Failure Rate—The percentage of deployments causing a failure in production
- Time to Restore Service—How long it takes an organisation to recover from a failure in production

These are good measures and helpfully take away the argument of what to measure, because they are industry standard. They are also surprisingly simple to implement if you have your CI/CD tools set-up sensibly *

However are they all useful? And what do they show exactly? Is it how quick we are at responding to a situation or is it more about how valuable are changes are to the customer?

I don't consider deployment frequency to be a good measure of the effectiveness of an engineering organisation. An effective engineering organisation should be able to release at a cadence that can guarantee a level of service quality which is deemed acceptable for the customer. No more, no less.

It's tempting to purely focus on engineering prowess, readiness and speed however what does this mean from a customer experience perspective?

Use DORA as a baseline set of metrics to measure your engineering capability but don't be tempted to consider this to be "job done". Engineering change is only effective if it delivers value to a customer.

How you measure the delivery of value to a customer depends on your specific product and your specific customer.

Refining the Big Scary Story

Firstly, let's get two things straight:

- Refinement == Design
- Design == Drawing

Secondly, you might have noticed that a whiteboard session doesn't need to use an actual whiteboard anymore.

You can collaborate via 'whiteboard' in Teams, you can do it in Zoom, you can get plugins for miro and figma or mural or clickup. There are many options and many overlapping tools – but how often do you actually use them?

How often do you sit down with a backlog for example and discuss a single, scary story and elaborate it in a refinement session across your entire product?

Often we get obsessed with ticking off lists, whether they be in Jira or Azure DevOps or wherever they are. We get obsessed with the tool driving our habits. We get lazy or we get impassive and we just want the lists to go away so we Click Click Click.

What about flipping this all on its head? Try taking the work out of the tool for a change and using the whiteboard (whether in person or virtual) to elaborate.

Take a single story, a big one, one that your team has been dreading doing, and refine it fully.

Draw it out on your favourite collaboration tool – the design, the architecture, the understanding that you have. Discuss the implications of the choices you have made in the past and the ones you might make in the future.

Do not fear refinement. Do not fear the whiteboard. Do not fear the mistakes you made in the past or the ones you will undoubtedly make in the future. Instead, be bold, discuss ideas, dare to dream.

Those big scary stories need writing down, they need drawing out, they need you to spend some time with them. And they don't need to be so scary anymore.

Collaboration isn't the software engineer or designer's default mode. However we have the tools. We have the ability to share. We have very few excuses not to do something different.

So just try it. And be sure to let me know how you get on.

When The Weekend Isn’t Your Own

As an engineer you often have to do on-call – this means covering a release on a weekend, or working later during the week or starting early. Getting a release in on time and making sure it works.

Seemingly you have little choice in your organisation – you let your team-mates down, you let your boss down, you let their boss down.

An expectation flows through the company that if one person is on call, the rest need to be on call.

As engineers become managers this expectation remains. The most experienced engineers fix production issues. Managers get twitchy when the most experienced people aren’t available. The whole company becomes a reactive – on-call related company – rather than a place where there is control and expectation around release moments.

Do you continue on this path and never get your weekend back or do you do something about this? You have two choices.

You can either leave the company or you can change the expectation.

Both are possible.

The solution is up to you.

Creativity Not Always Required

Developers or programmers like to think that building software is a creative pursuit.

Creativity implies freedom to do what you like. However the way that we control the output of the software delivery process requires formality, procedure, repeatability in order to be successful.

Formality works against our creative instincts. However getting something into production gives us a feeling of completion and satisfaction.

“I did something and it helped someone”

Two parts of the puzzle.

1. I did something.
2. It helped someone.

We can be cynical about development. We can want things to be more chaotic and more creative but any true software engineer enjoys working code (a problem solved) over a broken mess.

Creativity is a small but vital part of what we actually do as professional software engineers. If you’re on the more creative side sometimes it can be hard to just “park” that instinct and think about what delivering software should look like.

But by focussing on simplicity, elegance, self-commenting code, or even heavily commented code which explains our thinking – we give a chance to the next developer, to the next deployer of our software, to the next SRE, to be able to deliver and support quality code in production.

That after all should be our goal, right?

The Developer Mindset

When we have traditional IT, we rely on the IT crowd to help us out.

When we have cloud systems bought from a credit card – we take on the personal responsibility to make sure we're using it wisely.

With flexibility comes power. With power comes great responsibility.

This is the developer's mantra – the developer mindset. Before we make a change which we know might cause something to break, we check, we double check, if in doubt we ask someone else for an opinion.

I Wanna Be Adored

I find myself sometimes saying that I'm pragmatic when it comes to engineering decisions but the more I think about it, the more I guess I'm just lazy.

And thought of course reminds me of what Larry Wall (creator of Perl) said:

"The three chief virtues of a programmer are: Laziness, Impatience and Hubris."

Now I loved Perl, and I'd probably still use it these days but nowadays I'm probably more of a Python guy (because dependencies) however I think Larry still nailed it when he said.

Laziness:

It makes you go to great effort to reduce overall energy expenditure.

It makes you write labour-saving programs that other people will find useful, and document what you wrote so you don't have to answer so many questions about it.

Impatience:

The anger you feel when the computer is being lazy.

This makes you write programs that don't just react to your needs, but actually anticipate them. Or at least pretend to.

Hubris:

Excessive pride, the sort of thing Zeus zaps you for.

Also the quality that makes you write (and maintain) programs that other people won't want to say bad things about.

I think these principles tap into a more honest side of the reasons why people program.

We do it to be loved. Like a rock star, when we’re not rock stars.

We have control, we are lazy (we make it looks effortless), we want to be adored.

Do You Need The Default Clause?

Just out of university with a degree in EE and a masters in EE about software process engineering behind me, I didn't know what I didn't know. I didn't presume to know anything, I was humble. I hope I still am.

My first proper software job was working as a "Software Designer" (I loved that title) for Nortel Networks/Bell Northern Research. A huge engineering organisation with incredible hardware and software resources. I worked on network management software for trunk multiplexers. Nortel had its own source control management system called PLS which was pretty advanced and it used inner source widely. This was the year 1995.

I needed to use a piece of someone else's code for a change one day. And while testing noted with dismay that the code – a switch statement – had no default clause and this caused my code to fail in a way I wasn't expecting.

I took to an internal mailing list to voice my disapproval for this – because I believed that best practice was (and still is) to include a default clause in a switch statement.

The (very senior) guy who has written the code replied with a massive flame email which left me quite badly burnt but also pretty hurt. I replied in probably quite a hurt and defensive manner. The point I made was, that while calling this out was the correct course of events, perhaps the manner I'd done it wasn't the right way. The discourse was the important bit even if it didn't end up with a code change.

A few *weeks* later he replied, saying yes, he was sorry, and he should've not reacted in that way and it had made him think deeply about his interactions in future. I think we both learned something. It doesn't matter what end of the experience spectrum – the newb or the super-senior – be respectful.

Communication, people. It's important. Let's keep doing it, keep pushing back, keep it respectful. If something isn't right then call it out and enquire.

I’m probably older now than that guy was back then. I will still call out things that don’t make sense but I will always admit when I’m wrong.

I’m not perfect but I’m learning every day.

Users: The Hidden Opportunity in SaaS

Get ready for an obvious statement: things have changed a lot over the last 30 years of IT.

How did IT used to be organised? How it is now?

Long ago, when a company wanted a new system is that they would look around, or more likely use a consultancy to help them decide. A ‘computer system’ would be selected and the vendor would ‘put it in’. This could literally mean buying in physical computers as well as software. Traditional IT would be installed and maintained by a third party and perhaps you would have an on-site technical person or IT manager who would make sure that things worked day-to-day and also be able to escalate (and also who to escalate to) in the event of a problem.

In the event of serious problems it was a call back to the vendor to get a resolution.

This was Software Delivery in the long ago.

Software was often tied to specific hardware, updates were slow, changes were expensive to make, difficult to arrange and systems were hard to fix when they went wrong. If you wanted anything specific changing for your company then that would require additional development which would cost extra money.

So whats the difference, what happens nowadays?

Traditional IT is increasingly being replaced with commodity Software as a Service (SaaS) offerings and of course public cloud has sped this transformation. Commodity hardware or BYOD (Bring Your Own Device) has meant that you can access company IT services from anywhere and on many types of devices. Additionally software services can be bought on short term contracts and even on a credit card. The current mode of IT services is anarchy in comparison to decades previously.

This freedom comes though, at the cost of users having to deal with more third parties and more intermediaries.

The onus is increasingly on the individual user to be aware of what to do with certain systems.

As a user of a modern system, you have to be ready to google stuff, find things out, be self-aware and help yourself. This also impacts how we, as developers and operators of systems, present our software to the world. We have a responsibility to make things clearer and more obvious. We have a responsibility to document things well and provide a mechanism for support in case things break.

Product Opportunity

So the user gets more freedom, shorter contracts, increasing speed and choice when implementing systems but relies more on the product to provide high quality, well-documented and well-supported software.

The user has to take more responsibility, has to bring more knowledge in-house to ensure that they know what their systems are doing.

The smart SaaS product manager can use this as an opportunity to make their product stand out in the market. A product can differentiate itself by making sure that this truth is acknowledged and that users are not only attracted to great solutions but also well supported at an individual level.

How To Know When The Team is Broken

We talk about high performing teams. But what about broken teams? How can you tell when a team isn't performing?

How do you measure what it means for a team to perform?

Do you measure it in story points? In stories or tasks completed? In delivered software artifacts? In sprints done? In refinement sessions? In happy customers? In reviews? Individually or together?

How can you spot a broken team?

Here's a few things that I've noticed:

- Broken teams often don't want to talk to each other.
- However, broken teams often don't want to be split up.
- Broken teams often work harder than high performing teams but with fewer results.
- Broken teams are unhappy at their work and sometimes also in their personal lives. The line between work and life can blur.
- Broken teams can polarise the rest of your engineering organisation around their opinions. This can be highly dangerous.

None of these can be measured by your engineering metrics or your dashboards. Broken teams are working just like the rest, they might be working harder, but they are not happy, they are rigid, they are inefficient and they are bad for your organisation.

The first, and perhaps shocking step, is to realise that you, as a leader, are a big contributor to their unhappiness. Either through action or inaction, you have let them down. Your role is to make sure that you provide a framework which enables all of your engineers to work happily and effectively.

Acknowledging that will help you understand how you can start help to rebuild.

When The Team is Scared

The other day I talked about the Big Scary Story, but sometimes it's not the story that's scary or even broken. The team is scared.

How do you stop a team from being scared of a problem? How do you get them into a state where they feel ready to tackle a big story (which may or may not be scary)?

If your team is scared, then the biggest problem with your team is undoubtedly you. You need to start leading your team in a way that makes them feel that anything is possible.

How can you do that? Well for starters give them a break, some time off the hard stuff, some time away.

Secondly, show them that you trust them. Spend some time with all the individuals in the team to understand their approach, their strong suit, their fears, and their thoughts on how things are going. Be honest.

Thirdly, give them time and space to come back to a situation where they feel that they can engage. How long this is depends on how damaged they are from previous work. If it's been years of different managers, all pushing the same message, then it might take months or even years to rebuild trust and faith. If its relatively new and recent hurt, then it might only take a week or so. I've seen both situations.

Fourthly, when they decide they are ready, make sure you are ready to back them and their individual ways of working with everything you have.

So, the four stages to fixing your broken team:

1. Give Them a Break
2. Show You Understand Them
3. Give Them Time and Space
4. Back Them in Everything They Do

This is not something you can achieve overnight.

Trust is easily lost, hard to regain and change starts with taking a look at your own behaviour. It's your own experiences and emotions that are driving your interactions with others. So look for support around you to give you a solid understanding of your own motivations and fears and use that to shape your interactions and expectations.

Software development is supposed to be fun, right? Let's all try to remember that.

How To HumanOps

It's hard not to be cynical in the software development business. It seems almost a default for some consultants and some developers.

Cynicism comes about when you are able to predict the outcomes of certain behaviours within a few percent. Our job as software engineers, SREs, testers, consultants, engineering leaders, isn't to be cynical about the tools, processes and formal parts. It's up to us to enjoy the small parts of creativity and enjoy the result of deploying working code to users who are happy to have it.

How do we make things better?

- Challenge the norms. Do you need to do everything on your list today – what happens if you don't?
- Automate the parts of your work that you can automate – did you ever apply your analytical skills to the work you do every day?
- Research new languages, tools, ways of working – grow as an individual and bring enthusiasm to your work.
- If those are too hard – then just EXPLAIN how you feel to those around you. Perhaps you don't need to change, perhaps it's the world around you?

Ever heard of HumanOps?

HumanOps is a set of principles which focus on the human aspects of running infrastructure.

It deliberately highlights the importance of the teams running systems, not just the systems themselves.

The health of your infrastructure is not just about hardware, software, automations and uptime – it also includes the health and wellbeing of your team.

The goal of HumanOps is to improve and maintain the good health of your team: easing communication, reducing fatigue and reducing stress.

Let’s bring back the happiness to software development and make less of the Death March about it. Work is supposed to be fun, right?

Say Yes to Different

There is an oft-misattributed quote thats goes:

The definition of insanity is doing the same thing over and over and expecting different results – *Narcotics Anonymous 12-step programme*

Many people think it's by Albert Einstein or by Benjamin Franklin but it's actually in the 12-step rehabilitation programme of Narcotics Anonymous.

The quote resonates with us all. Deep down we know we need to balance the repetitive nature of work with the rewards it brings. Sometimes we convince ourselves that repetition ad nauseam is actually something that is good for us.

Sometimes it feels good to hang on to ideas that worked before. Looking back at success it is always a unique combination of factors. The bosses, the colleagues, the technology. They all play a role.

Those situations however are all unique.

Don't expect to be able to replicate your past successes by doing the same over and over again. The moment we repeat ourselves we are doomed to become frustrated – ever reaching for that previous high which is no longer attainable.

The idea of replicating success is an addiction. The only way you'll truly find clarity and happiness in your job is by looking to change the challenge that you commit to.

Don't rely on a team, on a boss, on a technology, on an organisation. Rely on yourself. You are the thing that brings uniqueness to your role.

Don't say yes to the next thing because you think it's easy – even if it comes with a fat pay check or a promotion. Say yes because it's hard and different and will allow you to grow. Say yes because it takes you outside of your comfort zone. Say yes because it makes you feel like you are in charge of the situation rather than the other way around.

Say yes because you know. You know deep down, that you won't commit fully to something if you don't believe there is enough of something different in it, to challenge you every day.

Pull Requests Cause Friction

Think about how we can improve our interactions in the sometimes emotional world of software engineering. Do Pull Requests make sense in our modern way of working? How do they affect not just software quality but also our interactions with each other?

Having to review PRs is dreadful, as is chasing people to review PRs.

Many commentators doubt the Pull Request mechanism for a variety of reasons, all valid as far as I can see in a commercial context(*). For example, a PR always happens from a branch or a forked repo. Why would we branch at all if we're looking to perform proper CI (Continuous Integration) every day? Many CI implementations only rebuild the main branch and perhaps only run once a day due to long running tests, flaky tests etc. By not integrating to main, and not testing our long lived feature branches, we risk a bigger bang of breakage on merging further down the line.

Branches would have to be short lived (less than a day) so you might as well just check-in to main? i.e. Trunk Based Development.

The other side effect of 'early merging' (or making sure we do proper CI) is that our interactions with each other around merging and our communication needs to be way better. So we need to work at it, but it's preferable to just shooting in a PR.

About ten years ago I was working in a small, high-performing team, building business rules ETL transformations for a bank. Many changes were need to core code and our source control was unsophisticated but effective. We had no branches, we just kept informing each other of the changes. Didn't always work but our integration tests ran regularly on the whole product. Also we all knew when breaking changes were on the horizon.

Don't overcomplicate your branching strategy just because the tools almost enforce it for you or you decided gitflow or even github flow was the way.

Pull Requests cause friction both in our integration strategy and in our working relationships.

If you don’t need to do it, then don’t do it.

* – for open-source projects or projects that are more asynchronous in nature, PRs and forked repos/branches make a lot more sense.