Changing the engines without landing the plane

Robert Bentall, January 2022

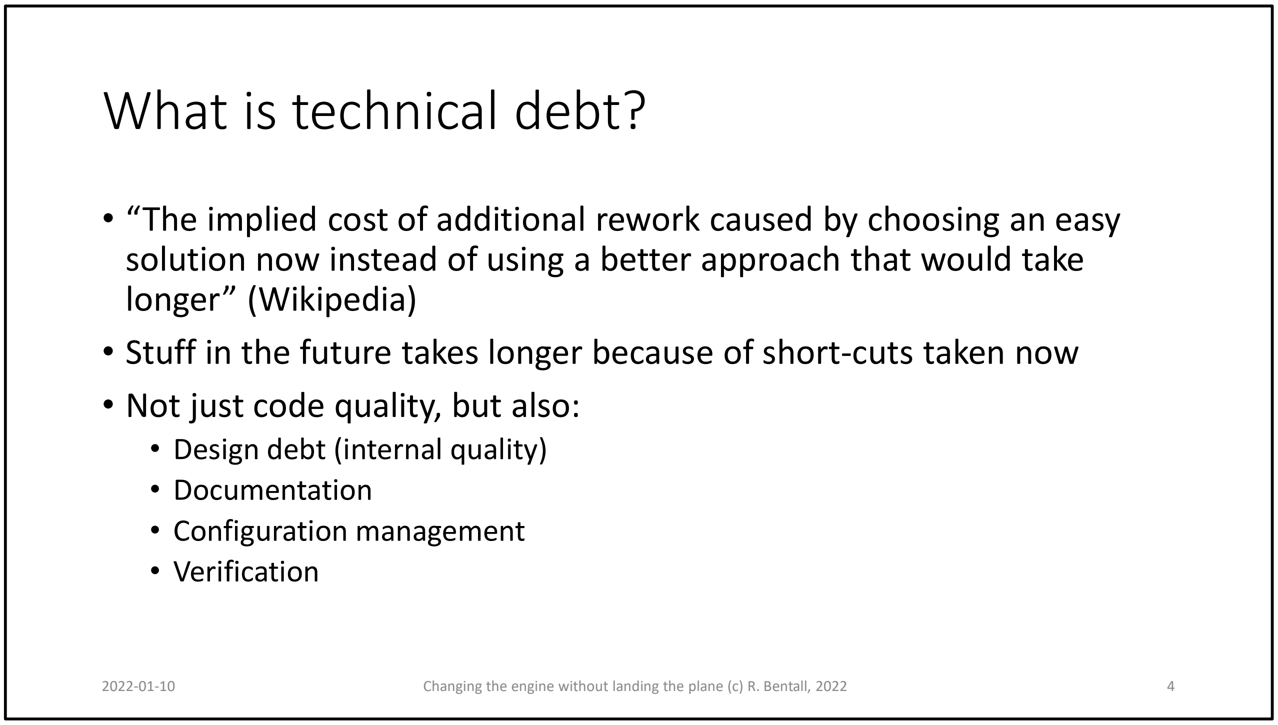
About me

- I’ve worked as a software engineer across a range of domains, organisations, technology stacks
- I am currently a Principal Software Engineer at Martin-Baker, where my team works on software used in ejection seats
- I’m fascinated by the process of software engineering
- Especially how to produce higher quality software within tighter schedules and cost constraints

In this talk…

- What is technical debt?
- Case studies
- Framework
- Antipatterns
- Antidotes
- The future

None of the case studies or data in this presentation relates to Martin-Baker



What is technical debt?

- "The implied cost of additional rework caused by choosing an easy solution now instead of using a better approach that would take longer" (Wikipedia)
- Stuff in the future takes longer because of short-cuts taken now
- Not just code quality, but also:
  - Design debt (internal quality)
  - Documentation
  - Configuration management
  - Verification

2022-01-10 Changing the engine without landing the plane (c) R. Bentall, 2022 4

It can occur for lots of reasons:

- Schedule pressure

- Skill (of engineers + customers)
- Just because the application is long-lived & has been enhanced over the years

I. "Unintentional Debt." Debt incurred unintentionally due to low quality work

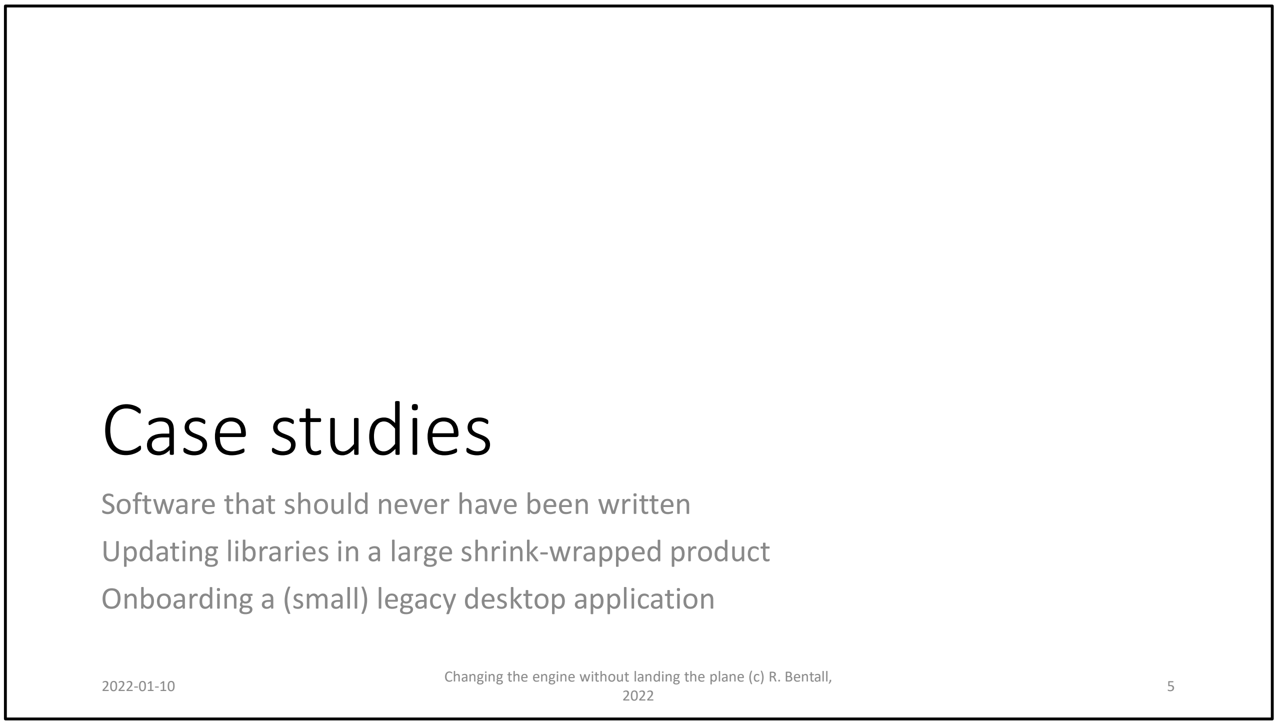
II. "Intentional Debt." Debt incurred intentionally

II.A. "Short-Term Debt." Short-term debt, usually incurred reactively, for tactical reasons

II.A.1. "Focused Short-Term Debt." Individually identifiable shortcuts (like a car loan)

II.A.2. "Unfocused Short-Term Debt." Numerous tiny shortcuts (like credit card debt)

II.B. "Long-Term Debt." Long-term debt, usually incurred proactively, for strategic reasons



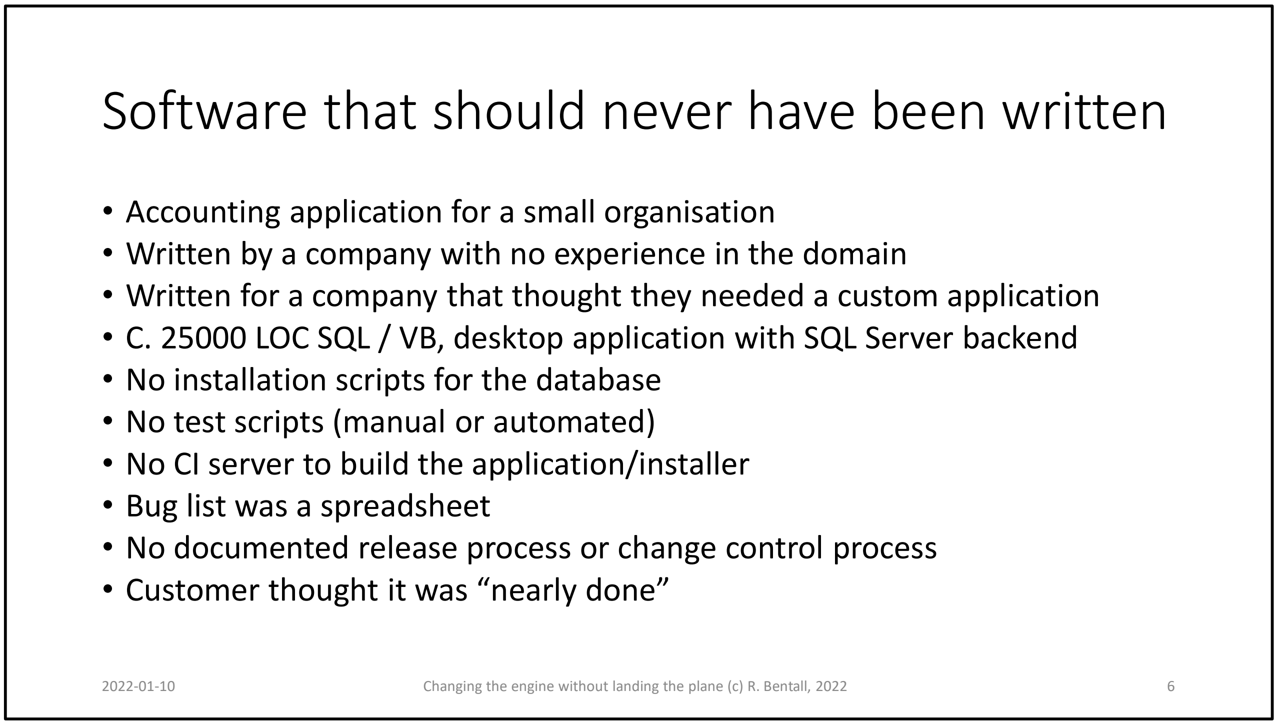
Case studies

Software that should never have been written

Updating libraries in a large shrink-wrapped product

Onboarding a (small) legacy desktop application

The case studies are some examples I’ve found particularly interesting over the years, relating specifically to technical debt.



Software that should never have been written

- Accounting application for a small organisation
- Written by a company with no experience in the domain
- Written for a company that thought they needed a custom application
- C. 25000 LOC SQL / VB, desktop application with SQL Server backend
- No installation scripts for the database
- No test scripts (manual or automated)
- No CI server to build the application/installer
- Bug list was a spreadsheet
- No documented release process or change control process
- Customer thought it was “nearly done”

2022-01-10 Changing the engine without landing the plane (c) R. Bentall, 2022 6

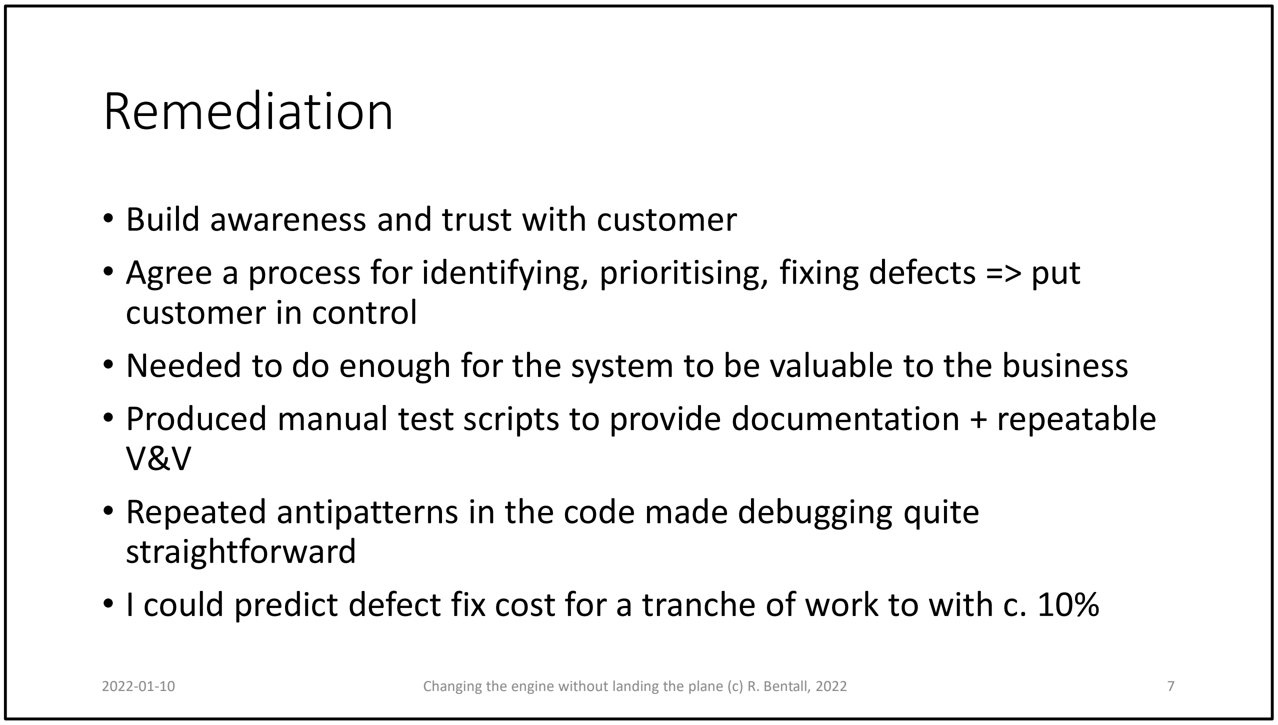
A company had purchased a custom accounting application from the firm that had produced their website. The firm had no experience of application development, and the development of the accounting application bankrupted them. The company had then approached another software company to “finish off” the application for them.

Ouch.

The application was c. 25000 LOC of SQL stored procedures and VB code, running as a desktop application with a SQL Server backend. There was no installation script for the database, no test script (manual or automated) for exercising the application. No CI server to build the application. No bug list beyond a spreadsheet that was being maintained by the developer working on it. No documented release process or change management process.

The customer thought it was “nearly done”.

I worked on this for about a year all told, for roughly half of my working week.



Remediation

- Build awareness and trust with customer
- Agree a process for identifying, prioritising, fixing defects => put customer in control
- Needed to do enough for the system to be valuable to the business
- Produced manual test scripts to provide documentation + repeatable V&V
- Repeated antipatterns in the code made debugging quite straightforward
- I could predict defect fix cost for a tranche of work to with c. 10%

2022-01-10 Changing the engine without landing the plane (c) R. Bentall, 2022 7

The starting point was to agree a process for identifying, prioritising and fixing defects. This gave the customer a mechanism for directing and controlling the work, and it gave us a mechanism for agreeing that we'd done what we said we would do for each tranche of work.

They didn't have the money or motivation to fix everything, but needed enough to be fixed for the system to be reliable and maintainable, and for the system to add value to the business.

As fixes were made, manual test scripts were written to act as documentation of system behaviour, providing the supplier and customer with an independent record of what it was supposed to do.

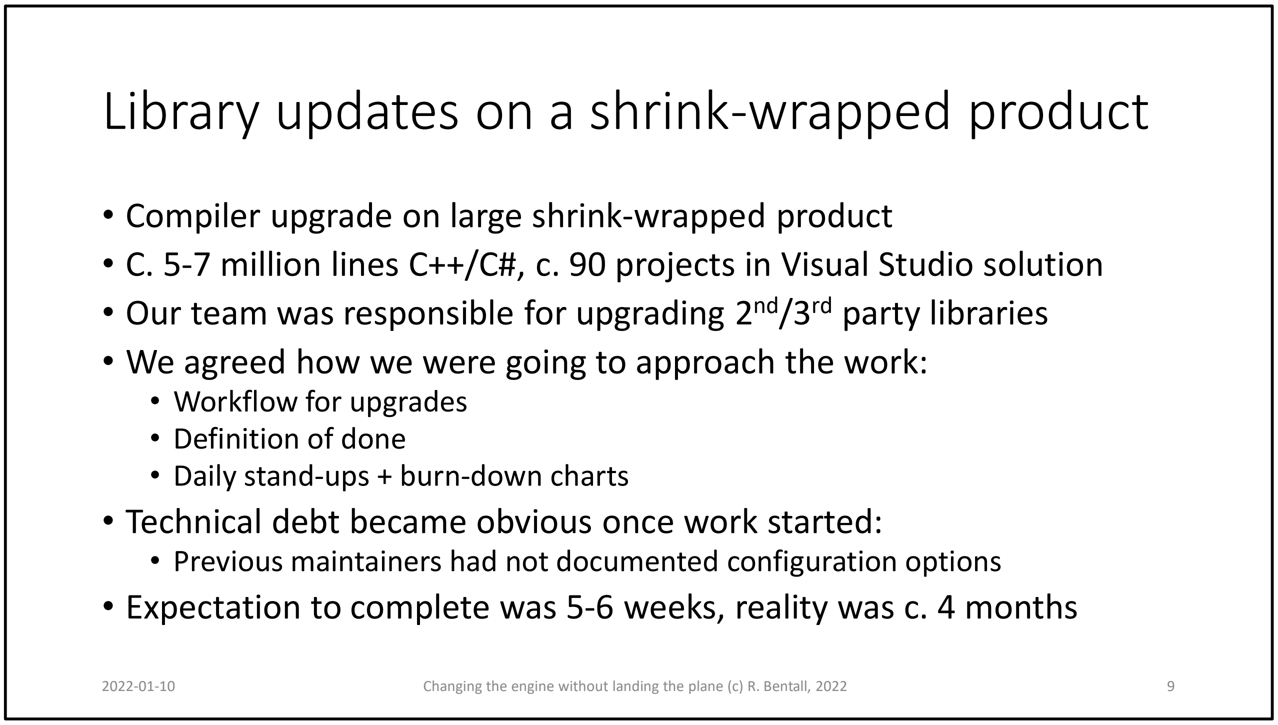
Whilst working on it, I got to the point where I could predict defect fix cost to within c. 10%. That tells you quite a lot about the structure and quality of the application – there were repeated antipatterns running through the application which made it quite straightforward to debug once you'd got your eye in:

- Empty "try-catch" blocks instead of systematic error handling
- Lots of copy-paste code between modules
- Lack of separation of concerns – presentation logic mixed up with business logic and accounting rules

For every defect the customer raised, I'd expect to find at least two or three more myself once I started debugging that area of the application.

Reflections

- Three problems:
  - A customer that thought they needed a custom system
  - A supplier that thought they knew how to build it, but only got partway through
  - The customer thought it was nearly finished
- Lesson?
  - Don’t do custom software unless you (a) need to, (b) know what you are doing



Library updates on a shrink-wrapped product

- Compiler upgrade on large shrink-wrapped product
- C. 5-7 million lines C++/C#, c. 90 projects in Visual Studio solution
- Our team was responsible for upgrading 2nd/3rd party libraries
- We agreed how we were going to approach the work:
  - Workflow for upgrades
  - Definition of done
  - Daily stand-ups + burn-down charts
- Technical debt became obvious once work started:
  - Previous maintainers had not documented configuration options
- Expectation to complete was 5-6 weeks, reality was c. 4 months

2022-01-10 Changing the engine without landing the plane (c) R. Bentall, 2022 9

My second example is from a large product that needed a compiler upgrade. By way of context, “large” means c. 5-7 million lines of C++/C# code. The visual studio solution contained something like 90 individual projects for separate DLLs. With practice you could get the build cycle down to c. 20 minutes on a fast developer machine, by only compiling the bits you were interested in. The application was developed by multiple teams on different continents.

So here we’ve got a mature, successful product, and engineering teams/management that know what they are doing.

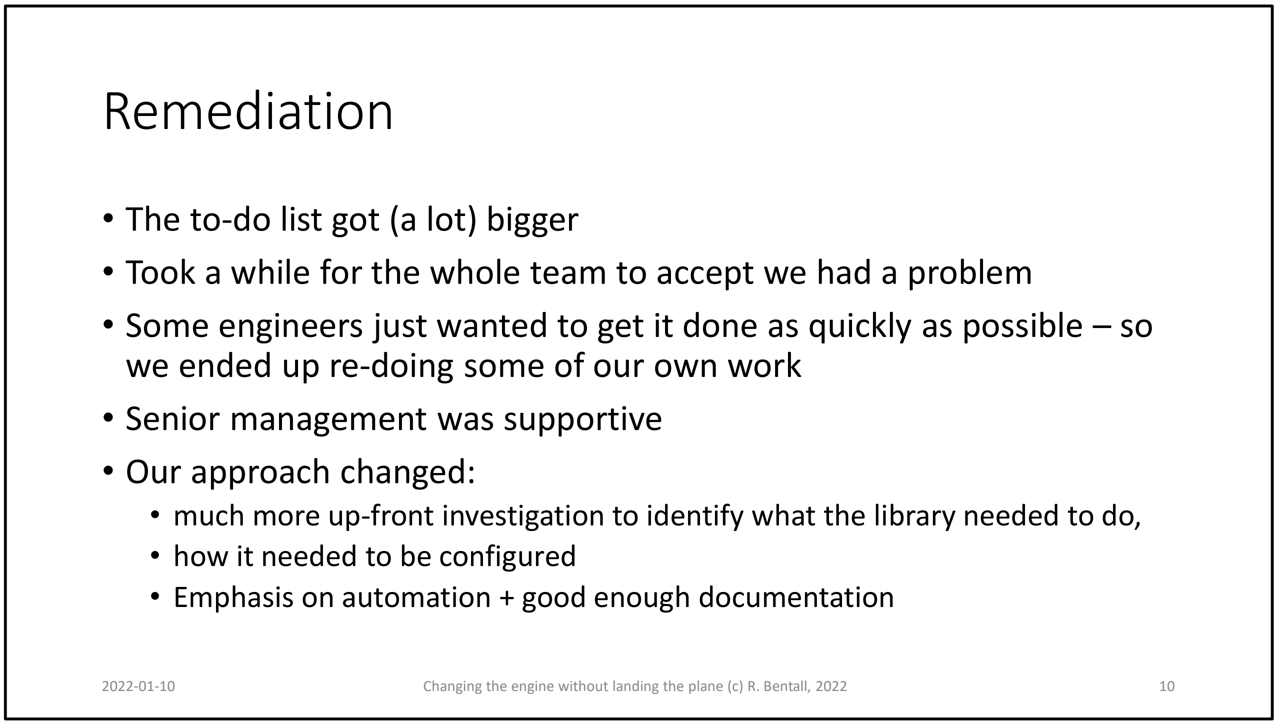
I was part of a team that worked on a compiler upgrade for this project. We were responsible for upgrading a range of second- and third-party libraries that were consumed by the application. It was expected that we’d take 5-6 weeks… It took closer to four months.

As a team, we knew what the scope was (which libraries needed upgrading). We also defined a workflow that guided our work, something like:

- Download the latest source code for the library,
- Check licensing changes
- Check functional changes
- Rebuild for whichever platforms were required
- Verify the new binaries worked with our product
- Review
- Integrate changes to the mainline

There were a couple of gotchas here…

As we worked through the individual libraries, we found that previous maintainers had not bothered to document configuration options, compiler settings etc. for each library. This meant that we had to redo these from scratch.



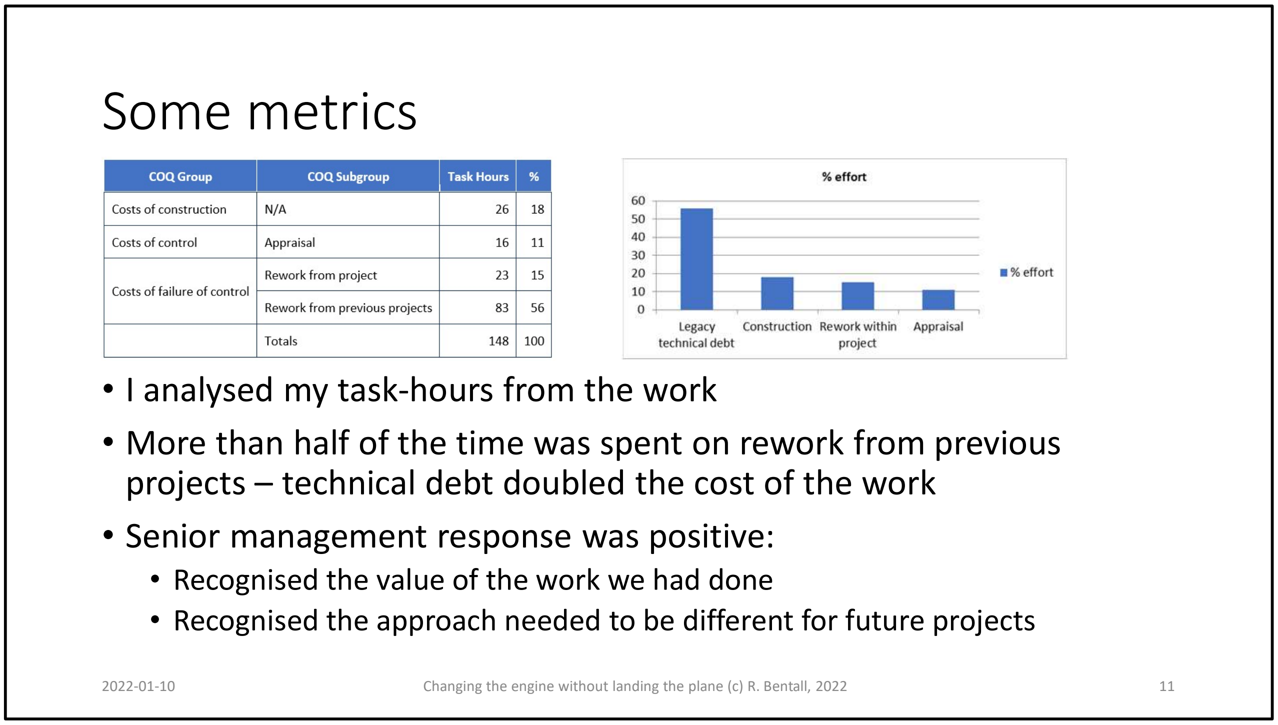
Remediation

- The to-do list got (a lot) bigger
- Took a while for the whole team to accept we had a problem
- Some engineers just wanted to get it done as quickly as possible – so we ended up re-doing some of our own work
- Senior management was supportive
- Our approach changed:
  - much more up-front investigation to identify what the library needed to do,
  - how it needed to be configured
  - Emphasis on automation + good enough documentation

This can be a very time-consuming business. You need to understand how the library's behaviour is controlled by the various build options, and you need to understand what parts of the library are being consumed by your application. Putting these together gives you a fighting chance of producing something that is likely to work.

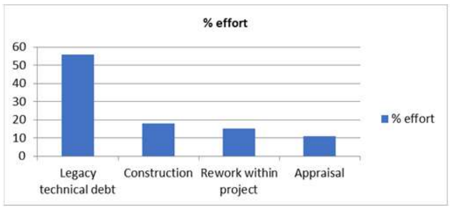
Our customers (the people consuming the libraries) were also on multiple continents, so getting answers to questions was often a time-consuming business.

Early on, there were also different views within the team as to what constituted an acceptable approach to this work. Some engineers just "made it build" without necessarily checking what the end-user required. This meant we had to do some more rework – due to stuff we'd done incorrectly during the project. As we progressed, we managed to converge on a reasonable "definition of done".



Some metrics

COQ Group	COQ Subgroup	Task Hours	%
Costs of construction	N/A	26	18
Costs of control	Appraisal	16	11
Costs of failure of control	Rework from project	23	15
	Rework from previous projects	83	56
	Totals	148	100



- I analysed my task-hours from the work
- More than half of the time was spent on rework from previous projects – technical debt doubled the cost of the work
- Senior management response was positive:
  - Recognised the value of the work we had done
  - Recognised the approach needed to be different for future projects

I analysed the data for my tasking on this project, so just a subset of the total set of libraries being upgraded. Splitting the rework into two categories made it easy to identify the technical debt ("mistakes made last week" vs "mistakes made 5 years ago").
More than half of my effort was spent on dealing with stuff that was done incorrectly "5 years ago".

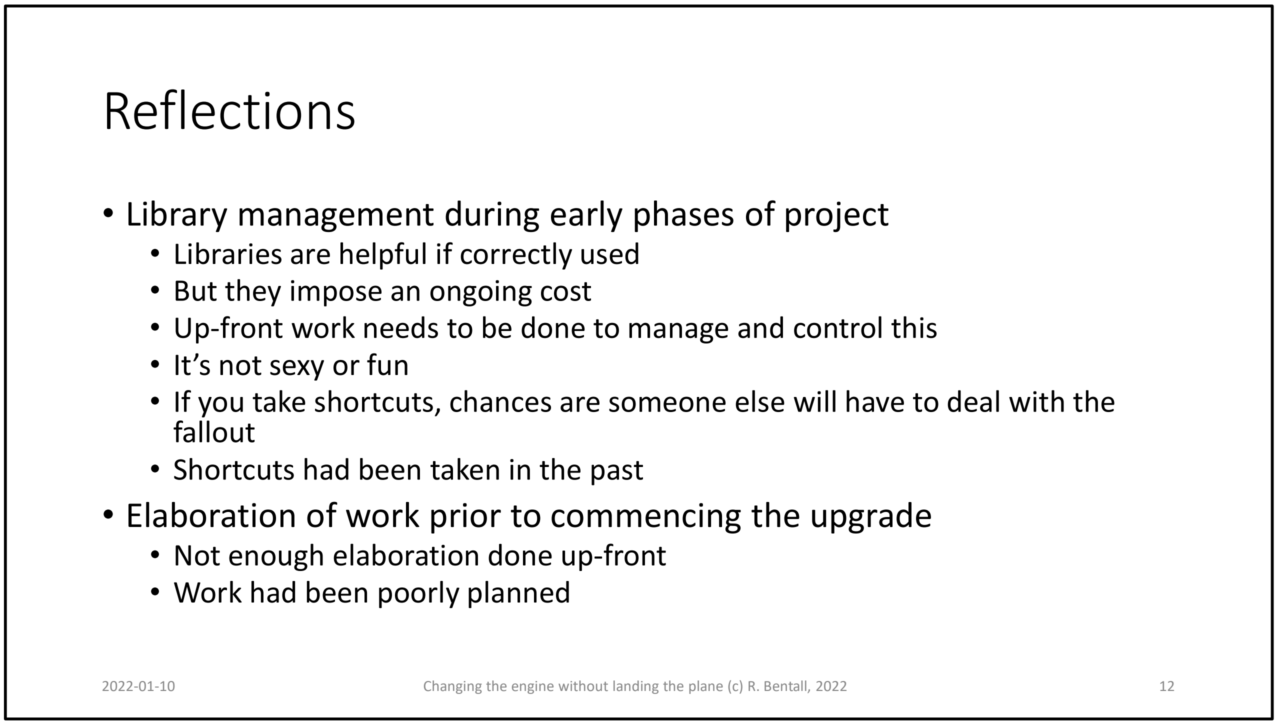
The consensus around this data was that it was likely to be similar for all engineers within the team – the technical debt wasn't known about in advance, and most libraries had some – so you'd expect to see a similar profile across the whole project.

This sort of ratio – the doubling of a project's cost due to technical debt – is something I've seen on other programs I've worked on over the years.

As an aside, the technical debt work was much harder to estimate accurately – my estimates were much worse when dealing with components which had significant technical debt in.

If you are always having to deal with technical debt, you can expect to find lots of "discovered tasks" during your project, and you can expect the cost and schedule to be much more unpredictable.

The view of the senior managers in response news was positive – they recognised that the next compiler upgrade needed much better planning, and also that the work that had been done to get the infrastructure in a good place on this project was very valuable.



Reflections

- Library management during early phases of project
  - Libraries are helpful if correctly used
  - But they impose an ongoing cost
  - Up-front work needs to be done to manage and control this
  - It’s not sexy or fun
  - If you take shortcuts, chances are someone else will have to deal with the fallout
  - Shortcuts had been taken in the past
- Elaboration of work prior to commencing the upgrade
  - Not enough elaboration done up-front
  - Work had been poorly planned

2022-01-10 Changing the engine without landing the plane (c) R. Bentall, 2022 12

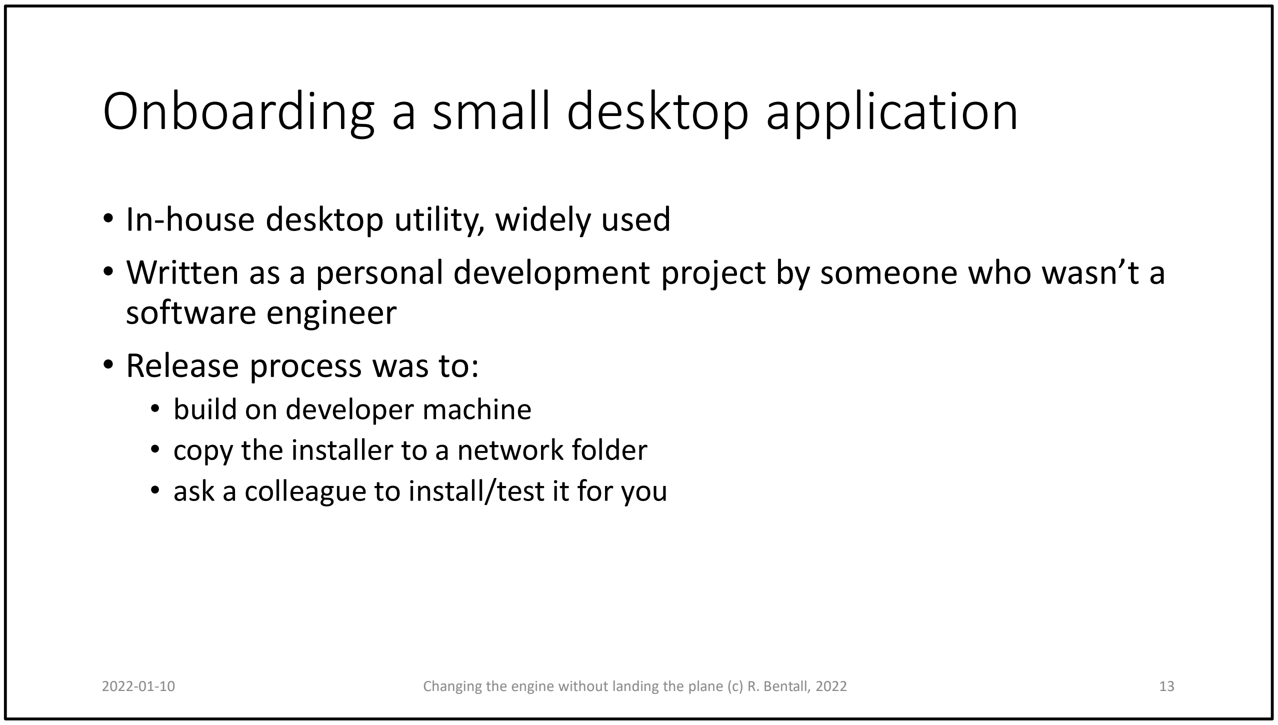
So what are the lessons? I think they group into two areas – what was done “5 years ago” and what was done “on the project now”:

- Library management during earlier phases of the project
- Elaboration of work prior to commencing the upgrade

Ideally the technical debt should not have been injected in the first place. Skimping on library management in earlier releases had a significant knock-on effect on the project when it came to the compiler upgrade years later. Adding a third-party library is a good idea if you really need the functionality. But you are committing future maintainers of your application to (potentially significant) maintenance costs years down the line, so the decision shouldn’t be taken lightly. And once you’ve decided to include a library, you really need to make sure you’ve understood, automated and documented the build and release process for your library.

Generalising this slightly, if *you* take a shortcut during the project, there is a good chance *someone else* will have to deal with the consequences later on.

When it came to managing the compiler upgrade project itself, it was clear that not enough elaboration activity had been done to produce realistic estimates of the required activity, the risks being run and so on.



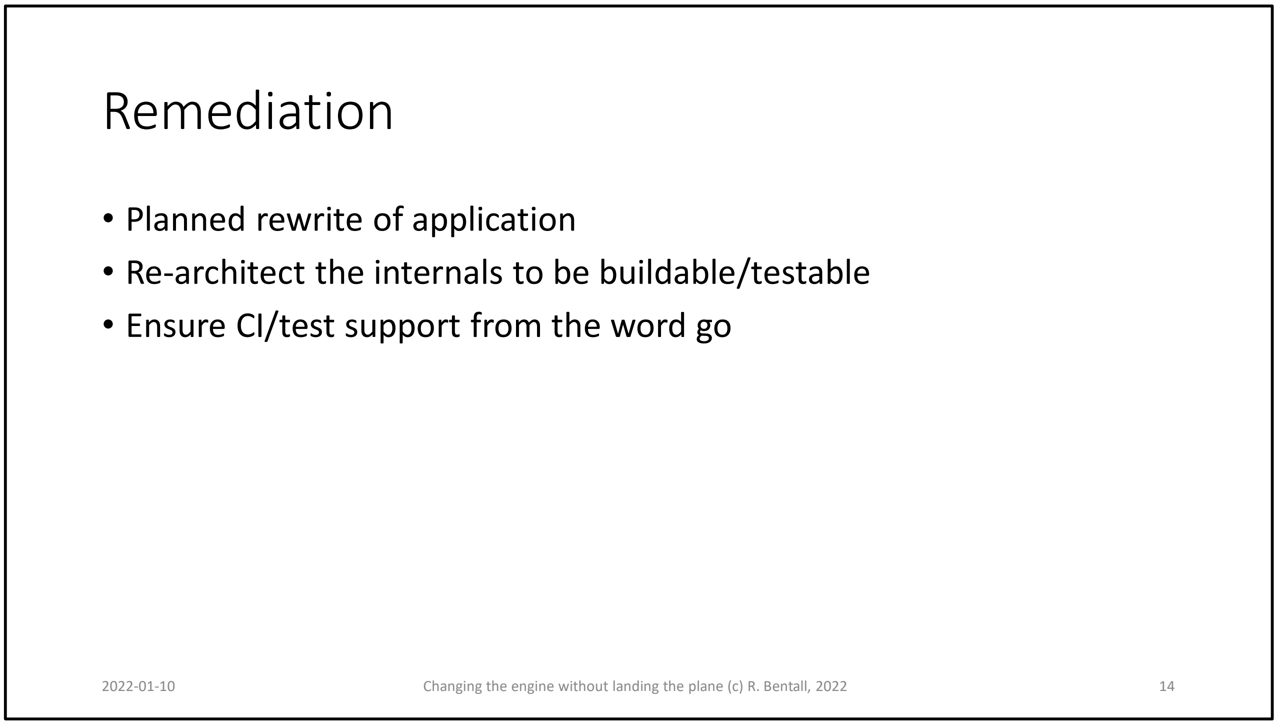
Onboarding a small desktop application

- In-house desktop utility, widely used
- Written as a personal development project by someone who wasn’t a software engineer
- Release process was to:
  - build on developer machine
  - copy the installer to a network folder
  - ask a colleague to install/test it for you

2022-01-10 Changing the engine without landing the plane (c) R. Bentall, 2022 13

My third example will be familiar to anyone who has had to maintain an “in-house” written custom application.

I said onboarding deliberately. The example in question was a small desktop utility widely used within the organisation. It had originally been written as a personal development project by an engineer who wasn’t a software engineer by training, but was reasonably smart and capable. The release process for this tool was to build it on the developer machine, give it to a colleague to test, and then copy the resulting installer to the relevant shared folder on the network. This was ironic, given the individual in question later went on to work in dev-ops.

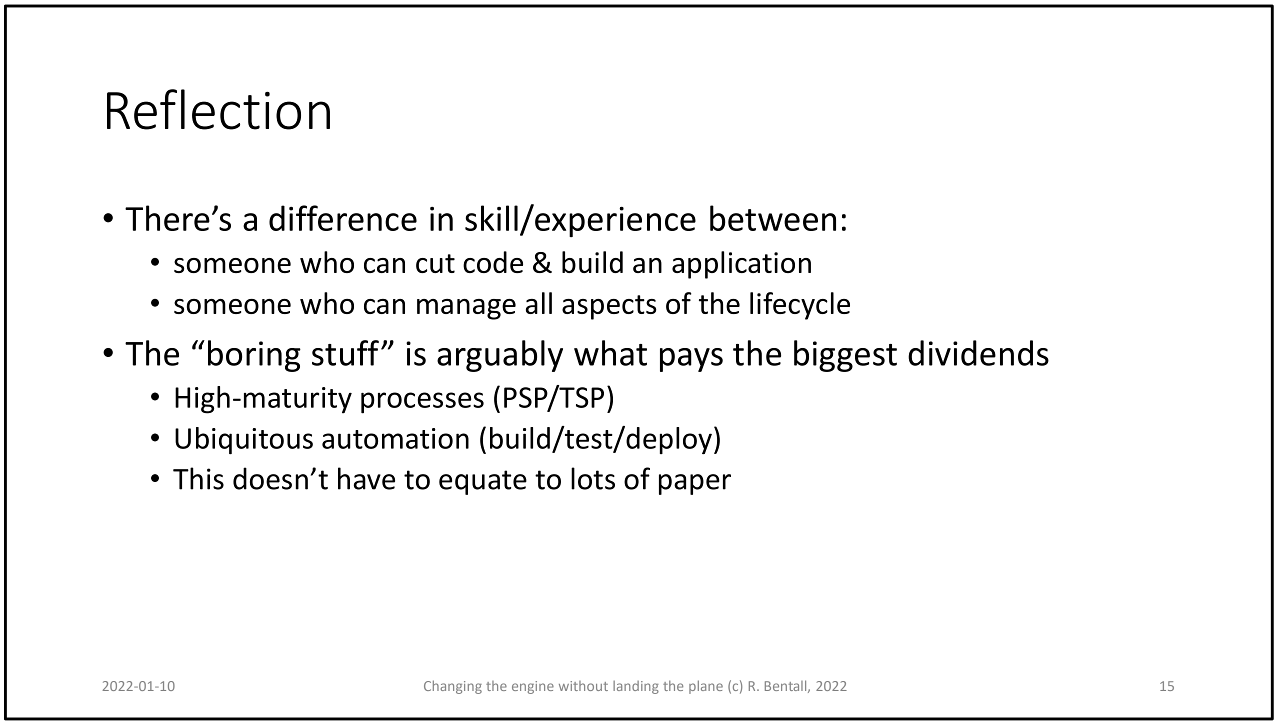


Remediation

- Planned rewrite of application
- Re-architect the internals to be buildable/testable
- Ensure CI/test support from the word go

2022-01-10 Changing the engine without landing the plane (c) R. Bentall, 2022 14

The solution here was some fairly radical surgery under the hood. This was undertaken as a planned upgrade for the legacy application, so it wasn't like the original development project had doubled in scope. It was straightforward to re-architect the guts of the application to be more buildable & testable, and the development of an automated test suite was reasonably straightforward.



Reflection

- There’s a difference in skill/experience between:
  - someone who can cut code & build an application
  - someone who can manage all aspects of the lifecycle
- The “boring stuff” is arguably what pays the biggest dividends
  - High-maturity processes (PSP/TSP)
  - Ubiquitous automation (build/test/deploy)
  - This doesn’t have to equate to lots of paper

2022-01-10 Changing the engine without landing the plane (c) R. Bentall, 2022 15

What’s interesting about this example is that it illustrates the gap in skill & awareness between someone who can code an application, and a software engineer who can manage the development and maintenance of an application throughout its lifecycle.

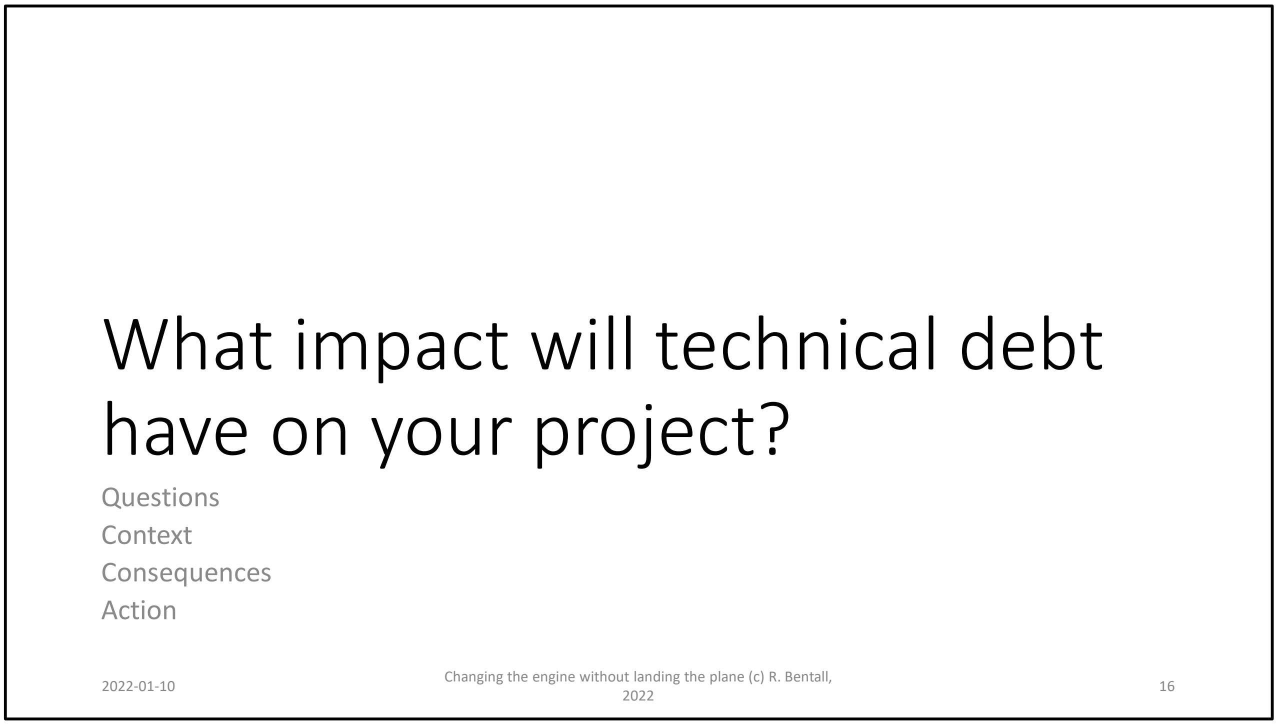
It’s easy to think that if you can cut code, you can be a software engineer, but the reality is that there is much more to it than that (and the software engineers aren’t just being precious about their job titles).

I’ve seen this issue on multiple occasions, where individuals with some domain expertise have developed an application functionally, but have not had the skill to make it maintainable (and therefore valuable) for the longer term.

It was one of the contributing factors in the previous case study, where the application had grown quickly by integrating plugins from different companies

It was on this project I found myself thinking about the sequence of steps I went through to make stuff more robust:

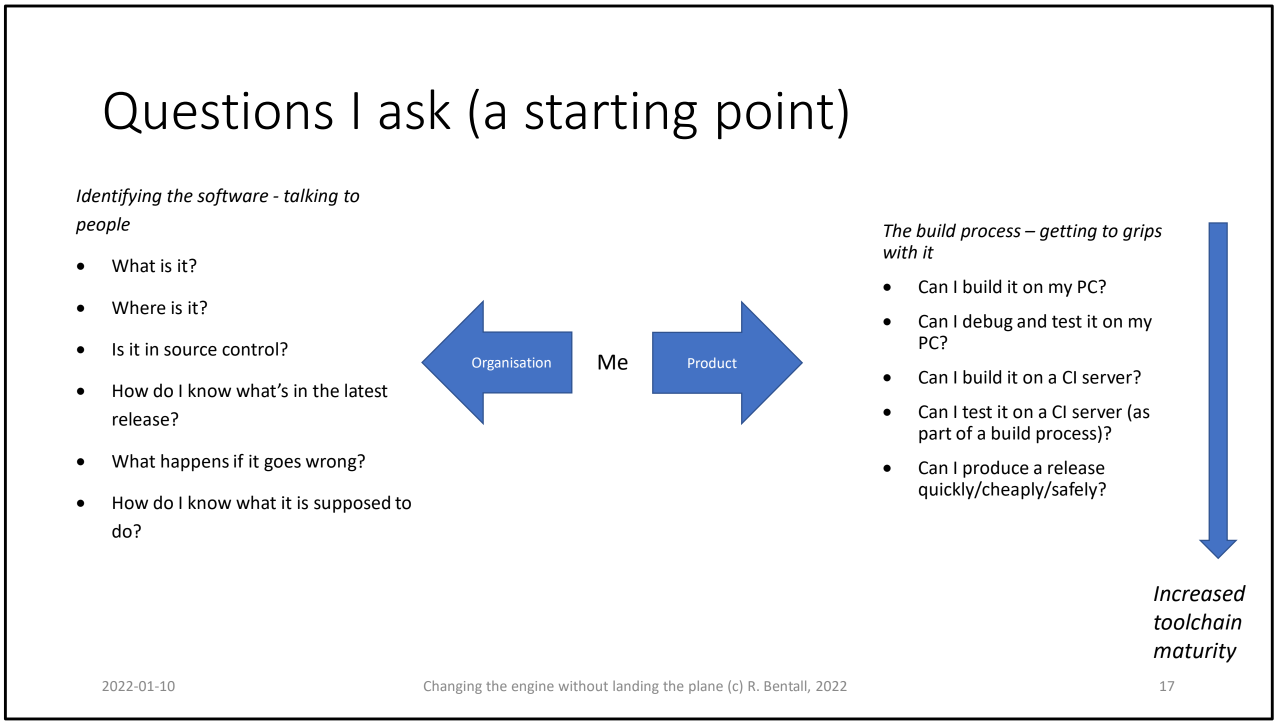
- Getting it built on the developer machine was OK
- Getting it built on a CI server proved challenging – because of the architecture of the application
- Running a test suite on a developer machine was challenging – because there wasn’t a test suite, just a word doc with some screenshots
- Running a test suite on a CI server was impossible given the previous two points.



What impact will technical debt have on your project?

Questions
Context
Consequences
Action

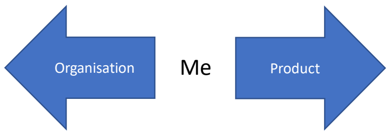
“how I approach problems with technical debt”
Most of the time I’ve had to deal with technical debt is when it has been discovered during the life of the project I was working on.
In other words, it wasn’t being pro-actively tracked and managed by the organisation, but was waiting to be discovered by the unfortunate engineer as they started out on their assignment.



Questions I ask (a starting point)

Identifying the software - talking to people

- What is it?
- Where is it?
- Is it in source control?
- How do I know what's in the latest release?
- What happens if it goes wrong?
- How do I know what it is supposed to do?



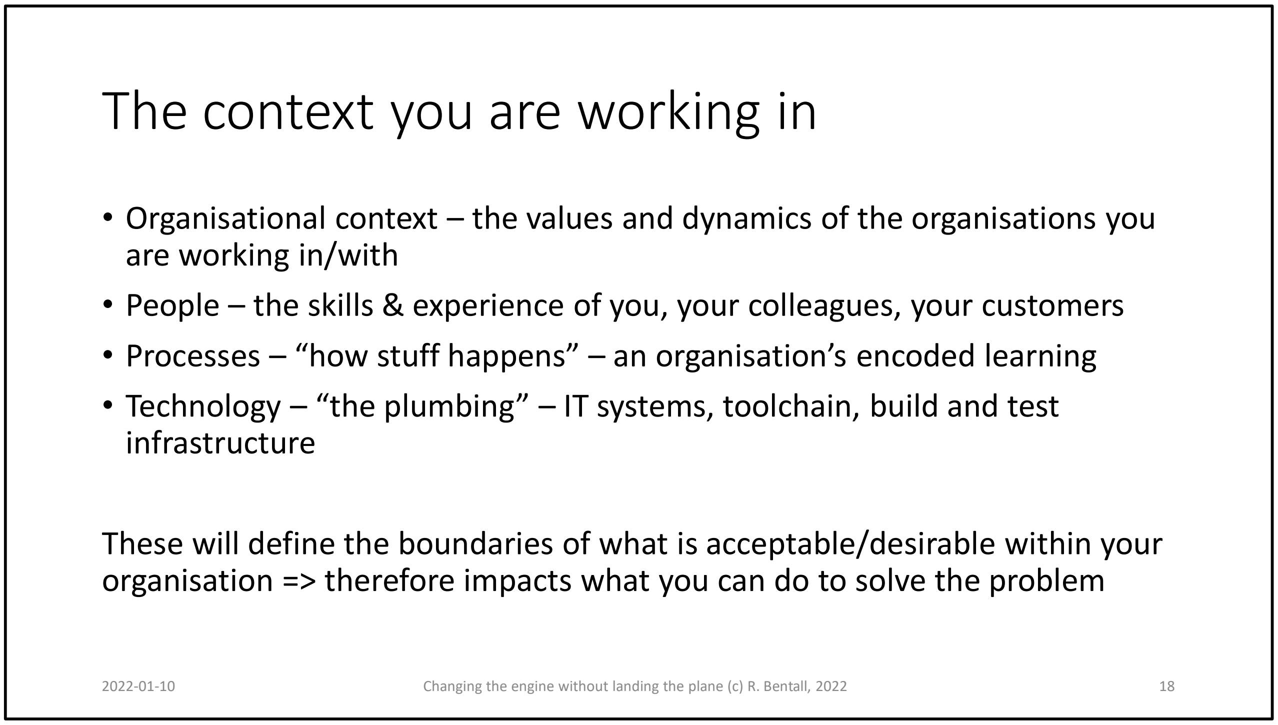
The build process – getting to grips with it

- Can I build it on my PC?
- Can I debug and test it on my PC?
- Can I build it on a CI server?
- Can I test it on a CI server (as part of a build process)?
- Can I produce a release quickly/cheaply/safely?

Increased toolchain maturity

2022-01-10 Changing the engine without landing the plane (c) R. Bentall, 2022 17

The net effect of having done this quite a few times is that I have a series of questions that I ask when confronted with some work on a piece of software I've never seen before.
The interesting thing about these is that process of answering each one will tell you quite a lot about the software you are dealing with, the environment that produced it, the values and capabilities of your customer, and the political environment of the organisation you are working in.
On the left-hand side, I'm looking out at the organisation, the context, trying to understand what the organisation wants & needs
On the right-hand side, I'm looking at the product – trying to figure out how mature it is.
The more mature your toolchain is, the further down the list you will already be.

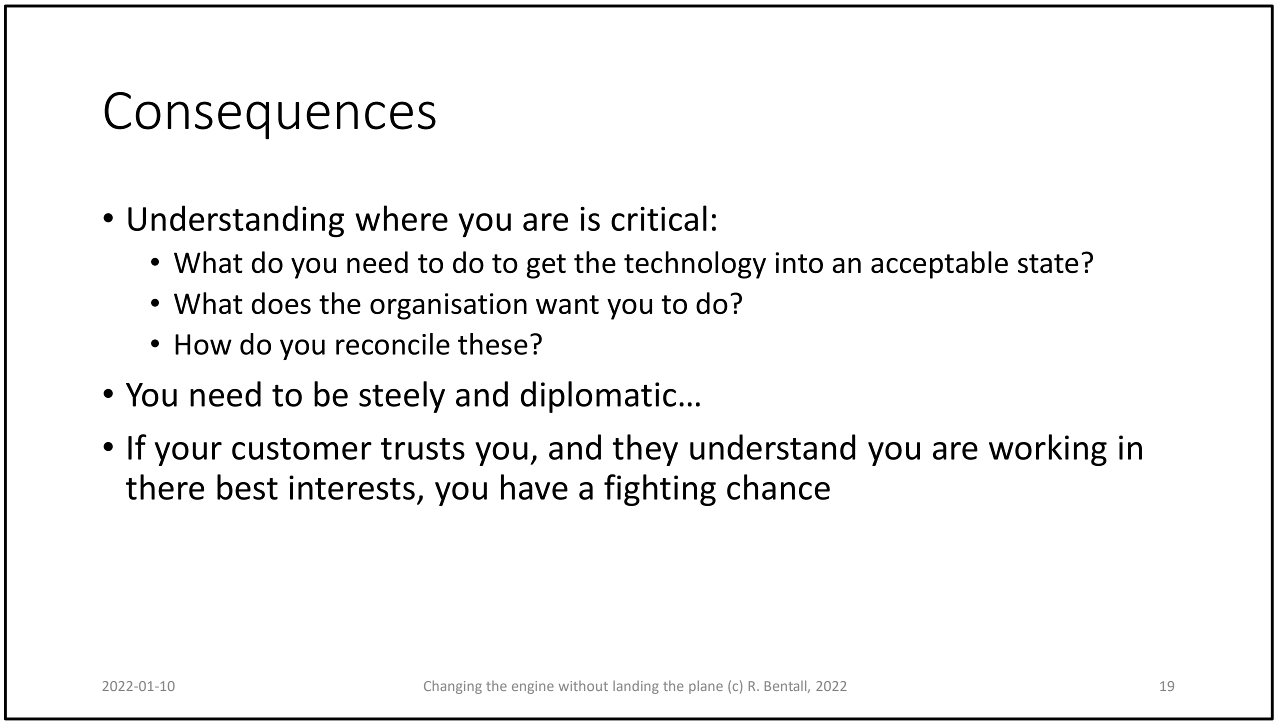


The context you are working in

- Organisational context – the values and dynamics of the organisations you are working in/with
- People – the skills & experience of you, your colleagues, your customers
- Processes – "how stuff happens" – an organisation's encoded learning
- Technology – "the plumbing" – IT systems, toolchain, build and test infrastructure

These will define the boundaries of what is acceptable/desirable within your organisation => therefore impacts what you can do to solve the problem

The context stuff is important – it defines the boundaries of what is acceptable or desirable within your organisation, and therefore will have a big impact on what you can do to solve the problem.

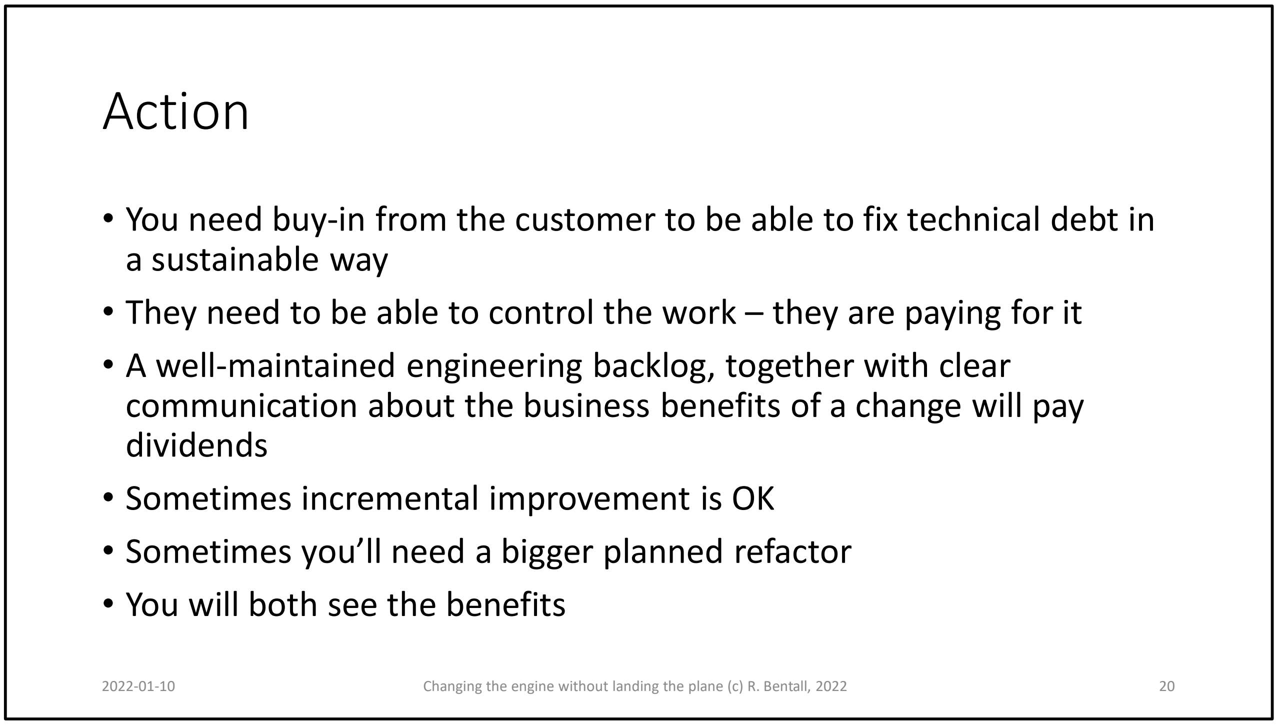


Consequences

- Understanding where you are is critical:
  - What do you need to do to get the technology into an acceptable state?
  - What does the organisation want you to do?
  - How do you reconcile these?
- You need to be steely and diplomatic…
- If your customer trusts you, and they understand you are working in there best interests, you have a fighting chance

2022-01-10 Changing the engine without landing the plane (c) R. Bentall, 2022 19

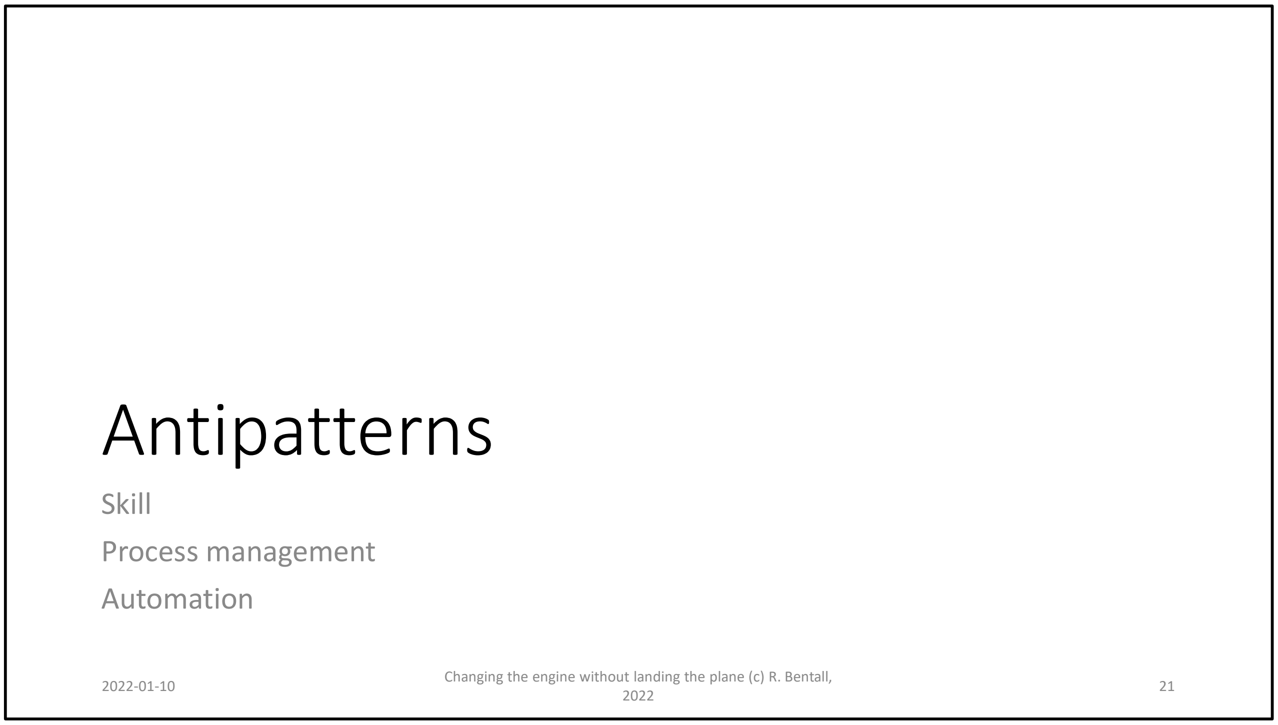
Crucially, if you are discovering problems as you go (when your boss and your customer weren’t expecting any), you are going to need to be steely and diplomatic.



Action

- You need buy-in from the customer to be able to fix technical debt in a sustainable way
- They need to be able to control the work – they are paying for it
- A well-maintained engineering backlog, together with clear communication about the business benefits of a change will pay dividends
- Sometimes incremental improvement is OK
- Sometimes you'll need a bigger planned refactor
- You will both see the benefits

2022-01-10 Changing the engine without landing the plane (c) R. Bentall, 2022 20



Antipatterns

Skill

Process management

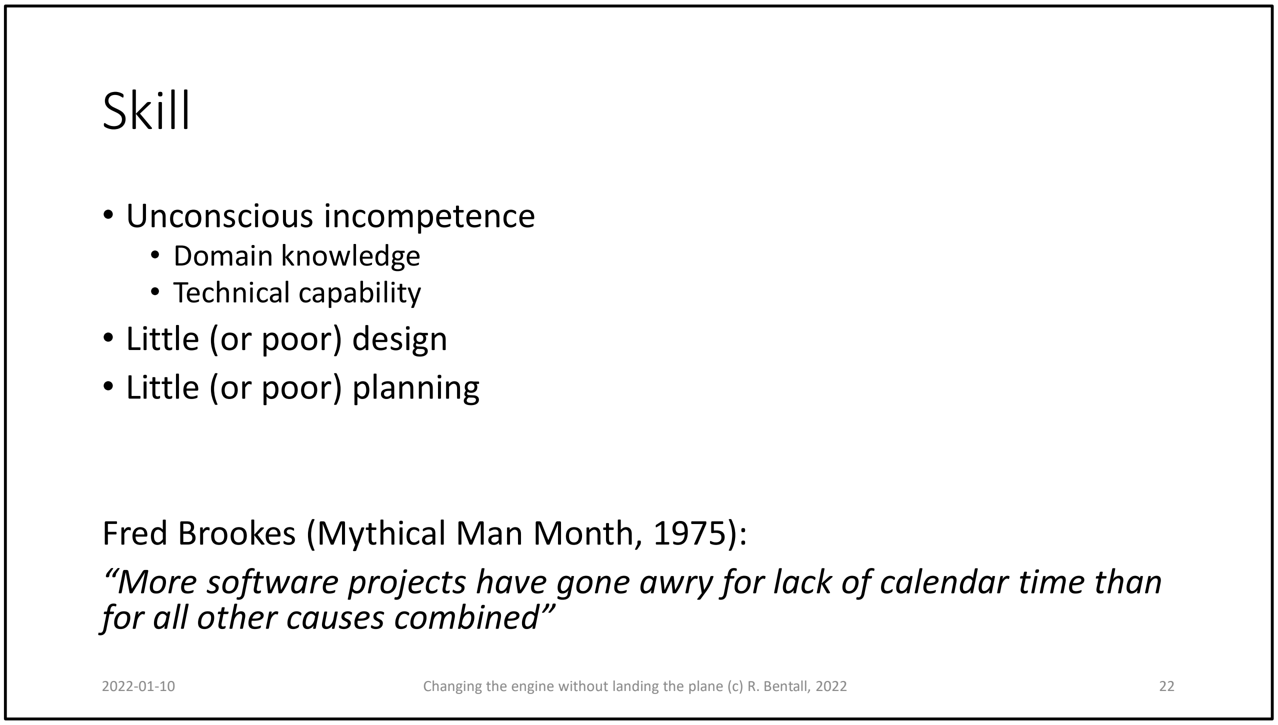
Automation

2022-01-10 Changing the engine without landing the plane (c) R. Bentall, 2022 21

After reflecting on these, I found myself thinking of some antipatterns.

What is interesting is that most of them are related to human factors, not technical factors.

And the technical factors are definitely at the less sexy end of the spectrum (who wants to sort out an aging build system?)



Skill

- Unconscious incompetence
  - Domain knowledge
  - Technical capability
- Little (or poor) design
- Little (or poor) planning

Fred Brookes (Mythical Man Month, 1975):
“More software projects have gone awry for lack of calendar time than for all other causes combined”

2022-01-10 Changing the engine without landing the plane (c) R. Bentall, 2022 22

These three often occur together.

If you don’t know what you are doing with the technology, you won’t be able to develop a credible design, and therefore you won’t be able to develop a credible plan.

Effectively you’ll be coding a series of prototypes until you’ve got something that “sort of works”.

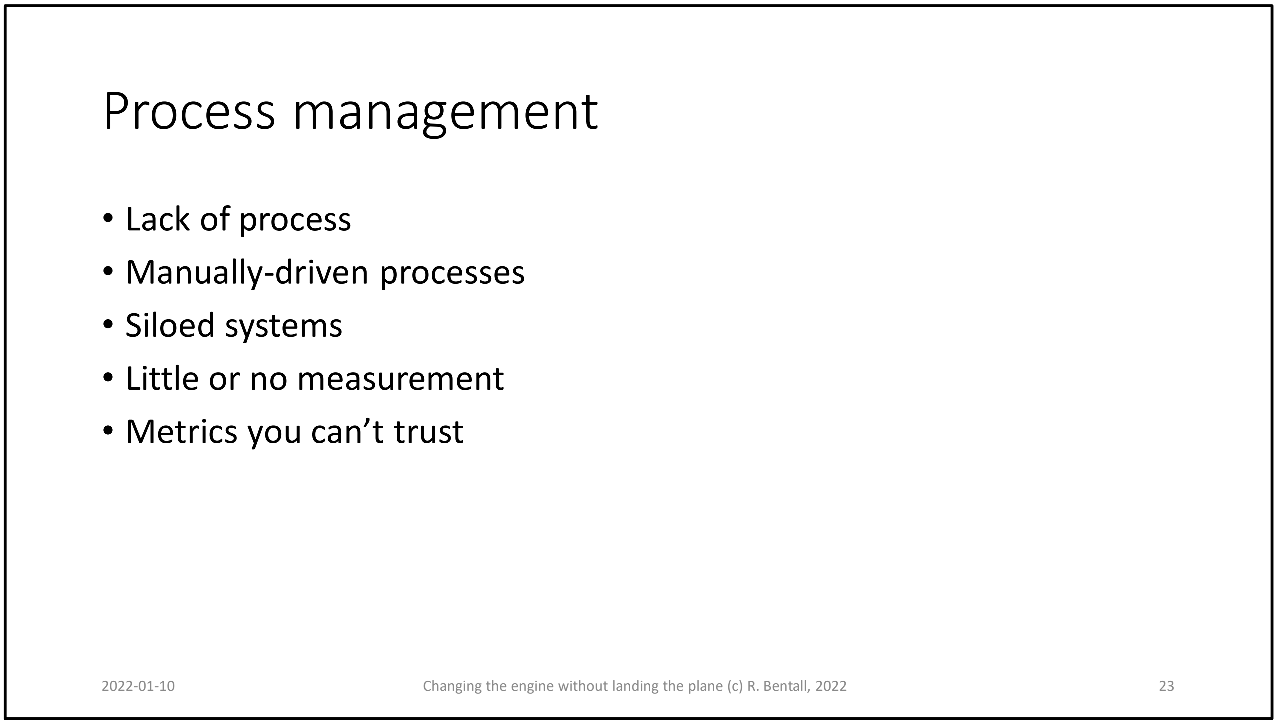
If you don’t understand the domain properly, you’ll take longer to produce an acceptable solution, even though you understand the technology.

And if you don’t understand either the technology or the domain, you are stuffed.

The cone of uncertainty (popularised by steve mcconnell communicates how some of these factors will influence your project;

Barry Boehm’s work on Cocomo ii provided the raw data and analytics that it was based on.

It’s worth remembering that the great Fred Brookes observed (in 1975…) that “more software projects have gone awry for lack of calendar time than for all other causes combined”.



Process management

- Lack of process
- Manually-driven processes
- Siloed systems
- Little or no measurement
- Metrics you can’t trust

2022-01-10 Changing the engine without landing the plane (c) R. Bentall, 2022 23

Processes provide a way of encoding knowledge, allowing you to focus on the novel stuff in a given situation.

Good processes reduce the information load you need to deal with on a day-to-day basis, and excellent processes make it much easier to do the “right thing”.

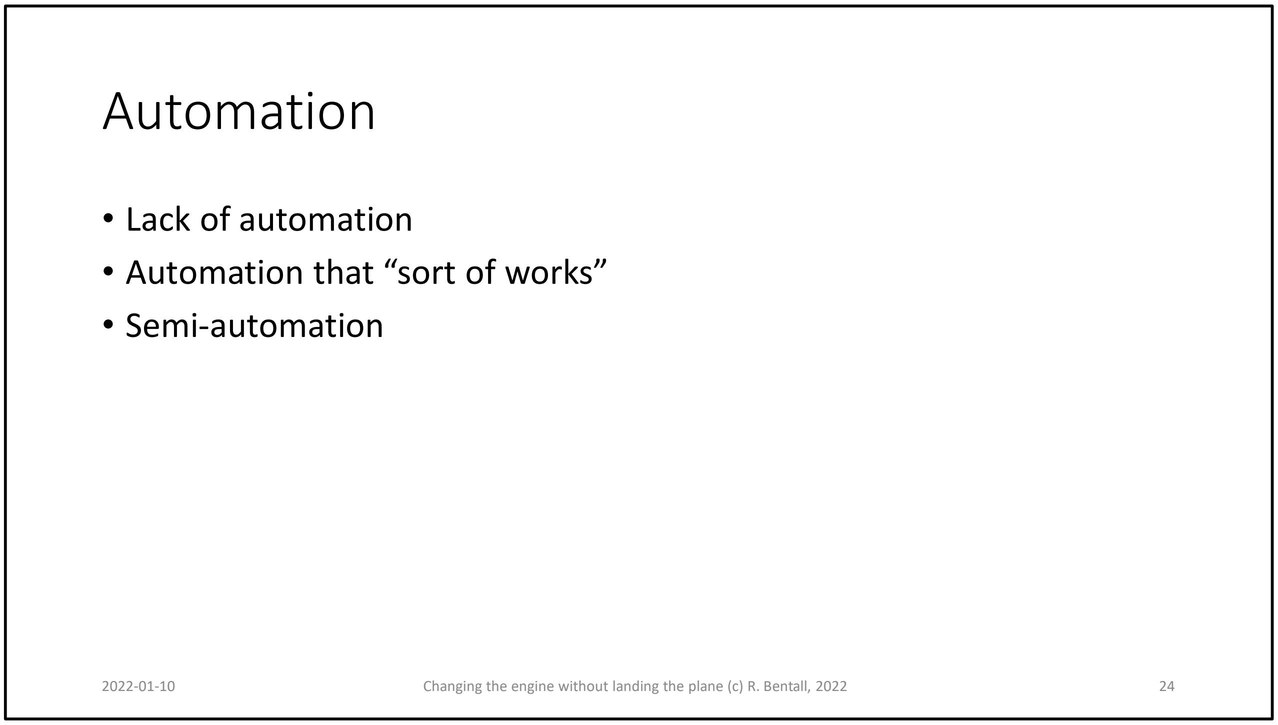
A side-effect of well-designed and implemented processes is that they are “information radiators” – they shout about how well you are doing.

Not for nothing do SPC practitioners talk about the “voice of the process”

“Manually-driven processes” are where you don’t have effective tool support – and therefore stuff gets run using spreadsheets, powerpoints, word docs, email chains etc. These increase the cost of using the process, so detract from their value & effectiveness to the business.

A consequence of using manually-driven processes is that they don’t radiate information. You get little or no data off them without putting in substantial work, so any metrics you have are likely to be less valuable – you won’t be able to use them to make timely and effective decisions.

A close relative of manually-driven processes is where you have separate systems for different functions (bug trackers, project planning tools and so on), but without ease of access to the data they contain. Each tool will contain some kind of model of your product, and may contain some automation for a particular part of your process (think the defect resolution workflow). In that case, it can be quite time-consuming to pull together an overall view of how effective your processes are - you’ll end up messing around with spreadsheets and reports to try and get the metrics you need.
All of these put barriers in the way of measurement – making the acquisition of data more expensive & reducing your trust in the metrics you get.



Automation

- Lack of automation
- Automation that “sort of works”
- Semi-automation

2022-01-10 Changing the engine without landing the plane (c) R. Bentall, 2022 24

Good automation is incredibly valuable.

It eliminates variability, and drives design decisions in a direction that will provide more value to the business.

Like good processes, automation allows you to focus on stuff that really matters, rather than trying to remember whether you actually *did* copy the installer you built into the correct location.

Good automation also gives you the possibility to capture and analyse more process data – the process is more straightforward to instrument.

Unreliable automation is a pain. If you are dealing with a legacy application that you are responsible for maintaining, hopefully this will be temporary state – you should certainly be striving for that to be the case.

In the same bucket as unreliable automation I’d include “fragile” automation:

you have monster build scripts that are too fragile to update quickly and safely – so they just grow and grow, becoming a bigger source of unreliability.

I think my least-favourite antipattern is semi-automation…

By that I mean some steps are automated, but there are manual steps interspersed. For example, you run a script to generate some code,

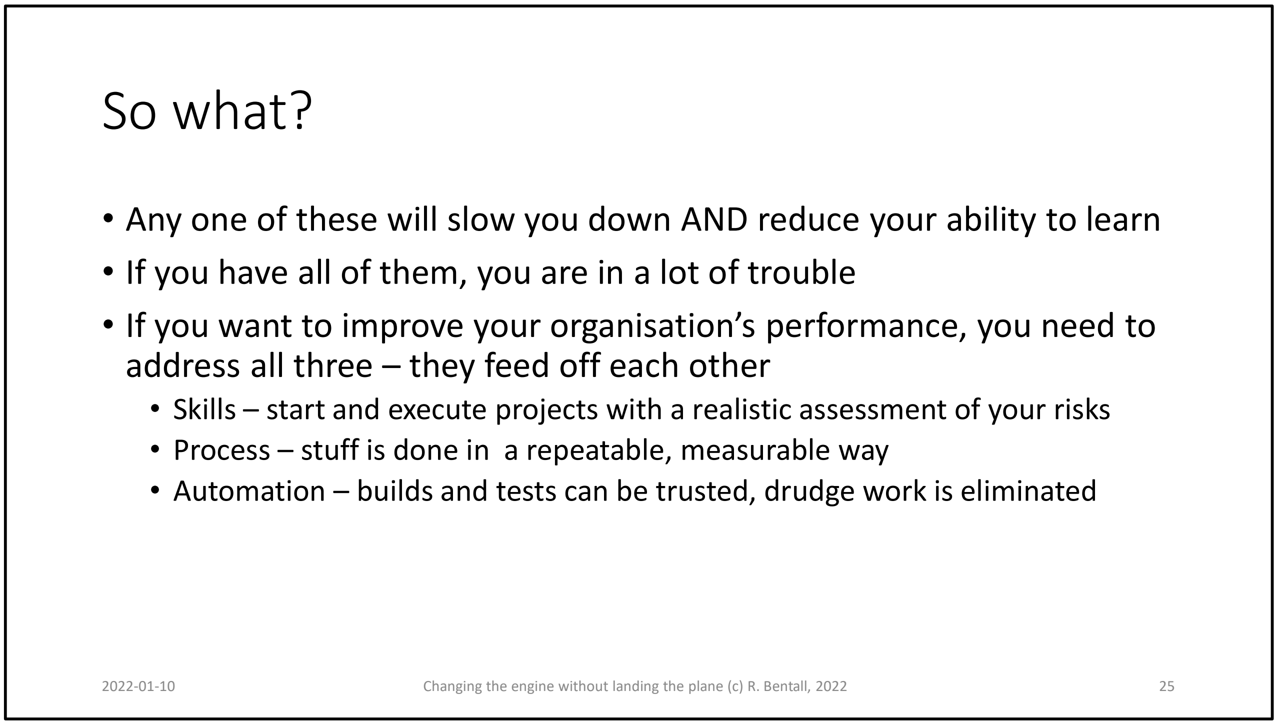
Then you manually modify the generated code

And then you run the next step in the process…

I can think of an example like this where an upgrade to a third-party tool resulted in the code-gen step producing source code that would no longer compile. The developer dealing with this just used the “previous” version of the generated file in place of the one that wouldn’t compile (and didn’t document that this was the case).

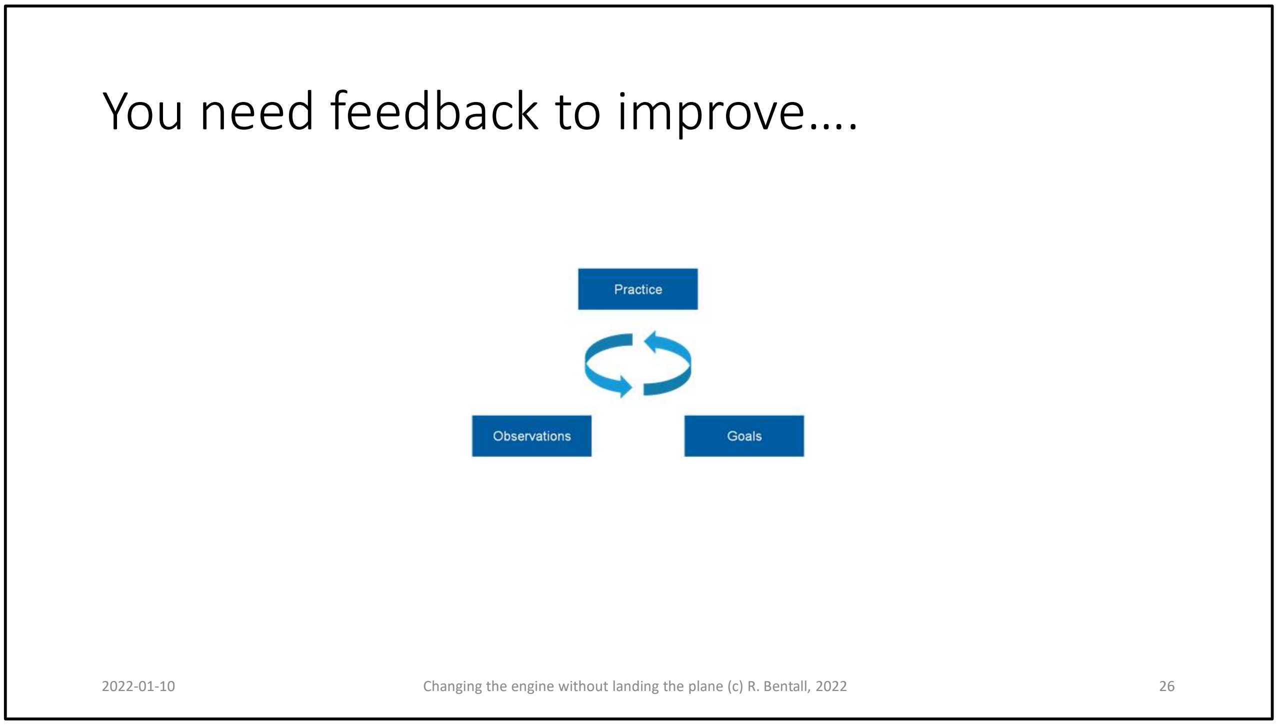
I kid you not….

It took several weeks to get to the bottom of this….

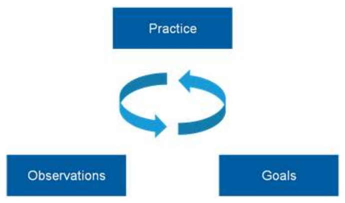


So what?

- Any one of these will slow you down AND reduce your ability to learn
- If you have all of them, you are in a lot of trouble
- If you want to improve your organisation’s performance, you need to address all three – they feed off each other
  - Skills – start and execute projects with a realistic assessment of your risks
  - Process – stuff is done in a repeatable, measurable way
  - Automation – builds and tests can be trusted, drudge work is eliminated

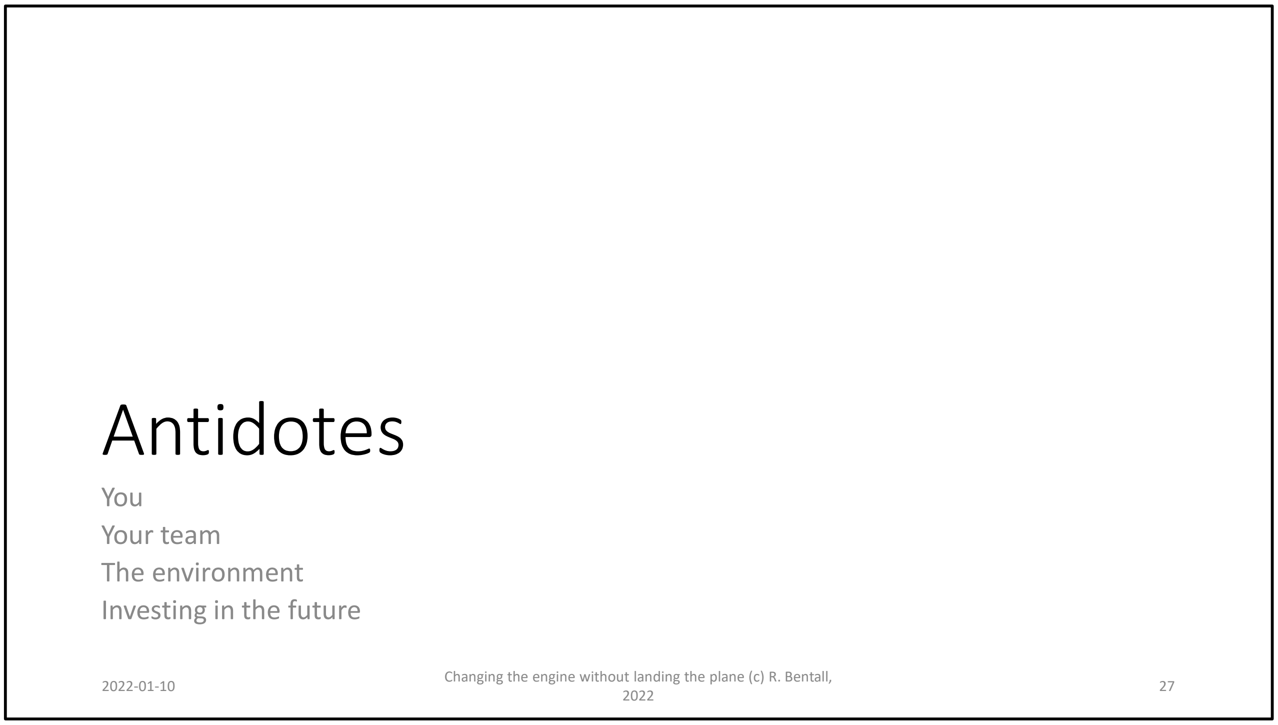


You need feedback to improve….



- If you want to get better, you need to practice (or at least engage with what you are doing).
- You need to watch/listen to yourself/your team/people you admire (what are the high performers doing?)
- You need an idea of what perfection looks like (or at least good enough)
- You need to try different things in order to improve.

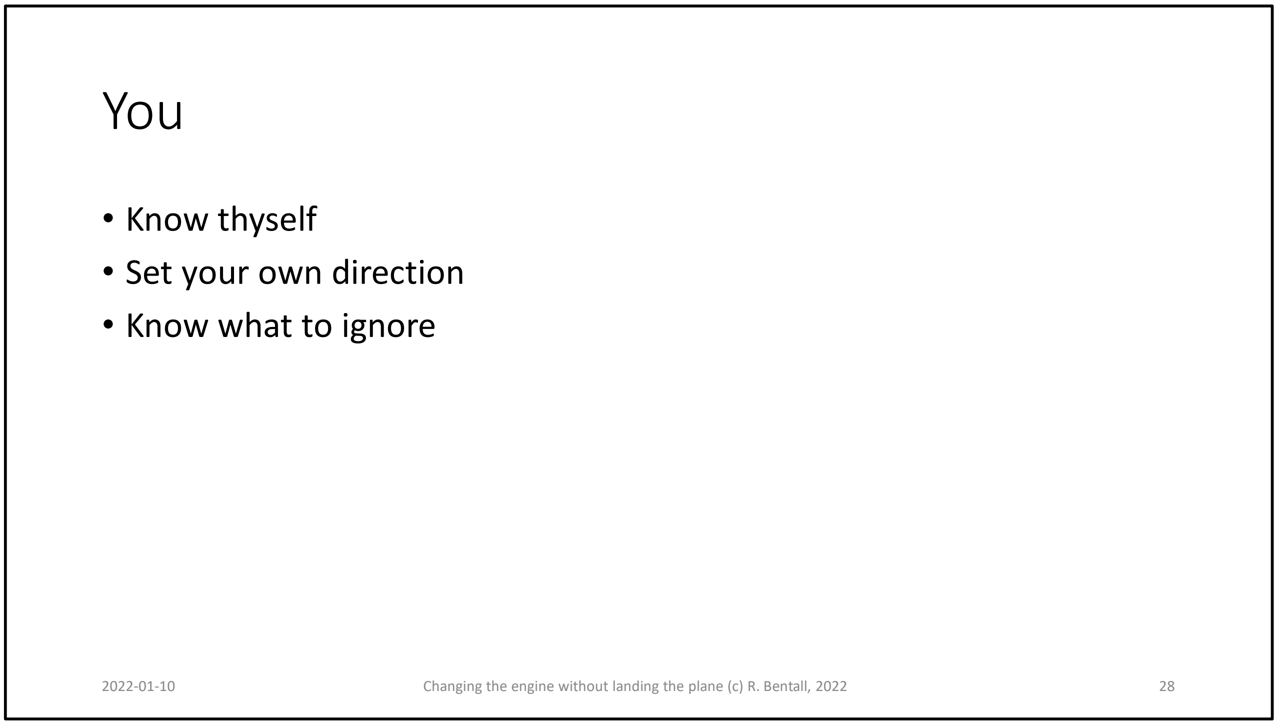
All of these antipatterns get in the way of the learning process, so stop you from learning / going faster.



Antidotes

You
Your team
The environment
Investing in the future

I guess the subtitle for this section is “how I stay sane”…
Or at least tolerate the discomfort….
I don’t have a silver bullet – and what follows sounds a bit like motherhood and apple pie…
But here goes.



You

- Know thyself
- Set your own direction
- Know what to ignore

2022-01-10 Changing the engine without landing the plane (c) R. Bentall, 2022 28

Do you want an easy life? Then turn off your brain and fit in with the organisation's norms.

At some level you are going to need to swim against the tide. That requires some ability to tolerate discomfort and stress, and to persist.

You also need to balance confidence and humility – a lot of the time you are going to be wrong, and in most jobs I've had, I've spent quite a bit of time disagreeing with my boss.

Know your own mind. What really matters?

When you work in an organisation, there will be lots of things outside your control – the constraints of the project, the organisation's structures + capabilities, the customer's needs.

That doesn't prevent you from finding areas for manoeuvre – something where you can try to make a difference beyond the original brief.

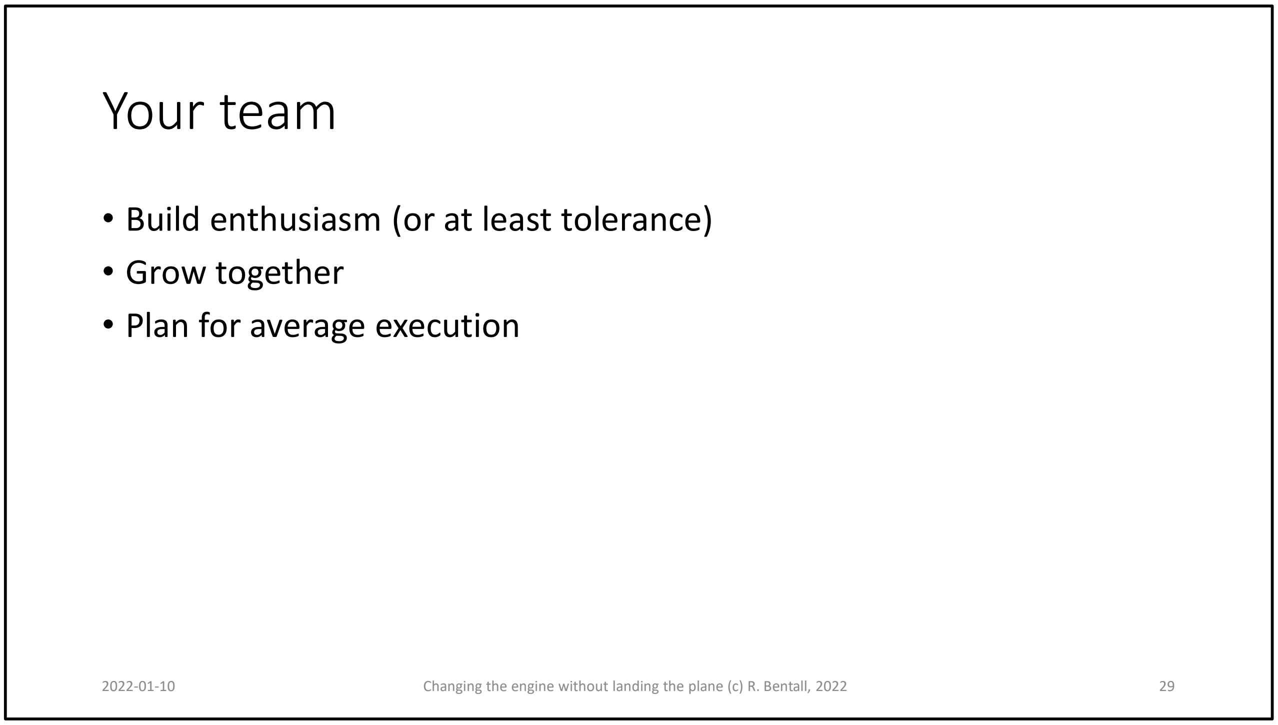
Unfortunately that often seems to involve using the midnight to 3am slot (the R&D budget…)

You aren't going to be able to fix everything, so you will need to accept there is some stuff that just isn't the way you'd like it.

Tough. There are only so many hours in the day.

That isn't to say you ignore or sweep stuff under the carpet – identifying and escalating risks is a crucial part of any engineers job.

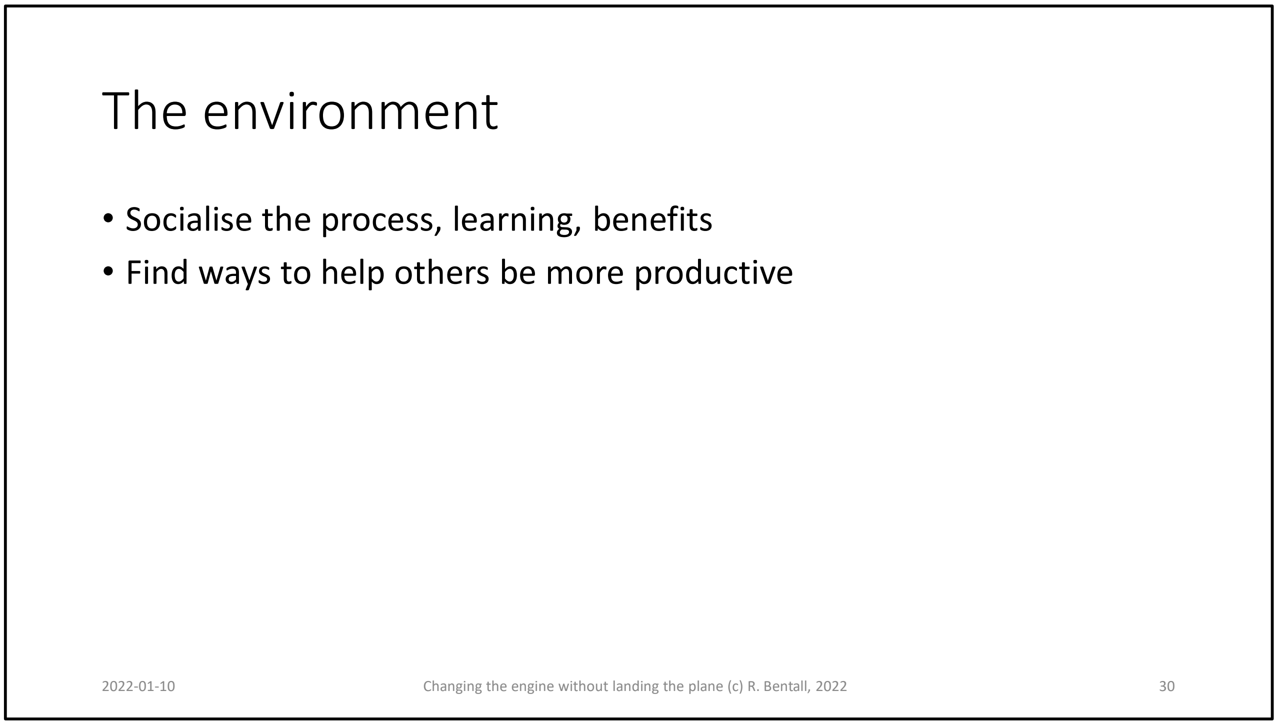
Once you've raised a concern, if the company chooses to ignore that, then fine – you've made your point. You need to know when to stop.



Your team

- Build enthusiasm (or at least tolerance)
- Grow together
- Plan for average execution

Software engineering is a social as well as a technical business – most systems are too big for one person to conceive of and execute.
So in spite of your enthusiasm for doing things better, at some level you'll need to take your colleagues and your team with you if you want process improvement to be anything other than an academic exercise.
Your colleagues may not welcome your enthusiasm for process measurement and improvement….
For one thing, if they believe that any data being captured is going to be used as part of a performance appraisal, the system will get gamed…
But hopefully you can persuade them that it's an endeavour worth pursuing.
With time, you'll learn as a group. Good ideas start as a minority of one, but it takes teams to change things.
To quote Margaret Mead:
Never doubt that a small group of thoughtful, committed citizens can change the world; indeed, it's the only thing that ever has.
As you progress, the team will talk more about what they notice in their own work, what they see in the project/organisation, and what they can do about it…
It's no longer completely reliant on you.
You also need to remember that people have families, lives outside work – every software engineering manager wants super-motivated elite performers (old head, young shoulders)……
But what you've got is probably average, and that's OK.
You just need to remember to plan for average, and then know when to push yourself and your team for more.

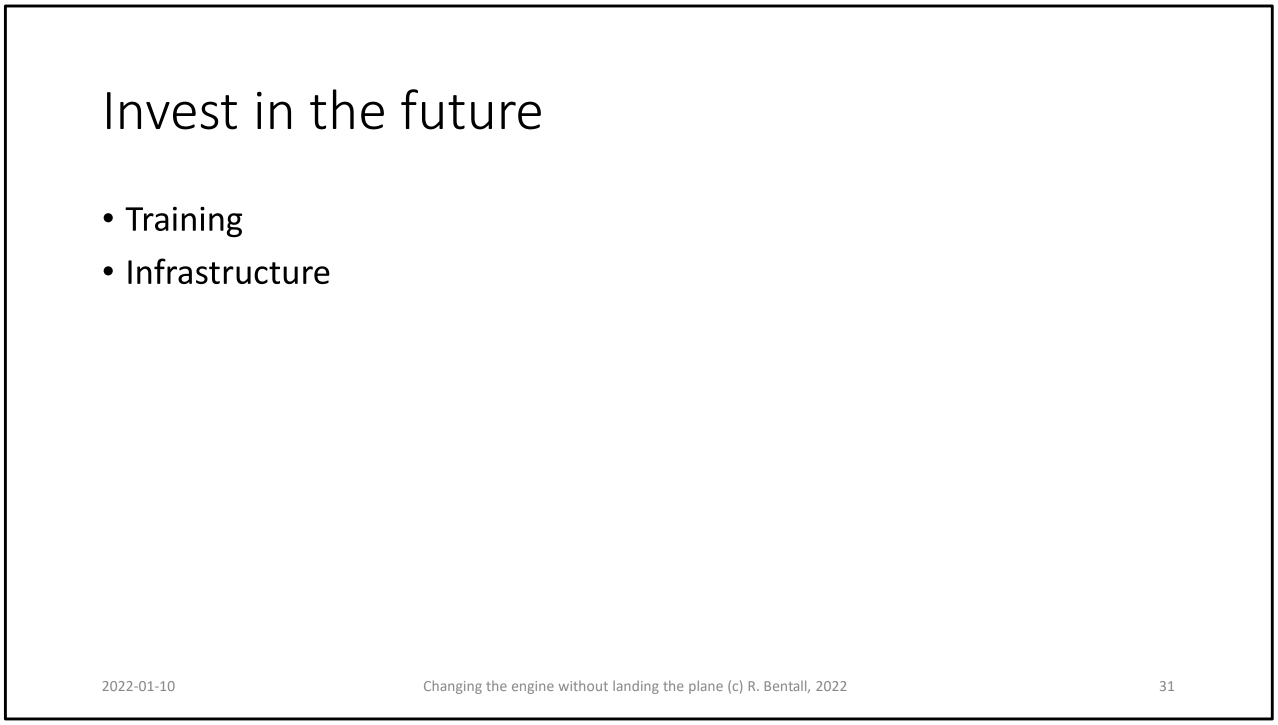


The environment

- Socialise the process, learning, benefits
- Find ways to help others be more productive

2022-01-10 Changing the engine without landing the plane (c) R. Bentall, 2022 30

As you go out beyond your team (to your boss, or your boss's boss), it's more about the big-picture themes – here's what we are trying, here's what we learned, here's what we are planning on doing next.
"Socialising the process" is about communicating values, and crucially listening to what other people, other groups have to say.
There's a good chance they may be thinking about the same sorts of issues, and you can help each other.

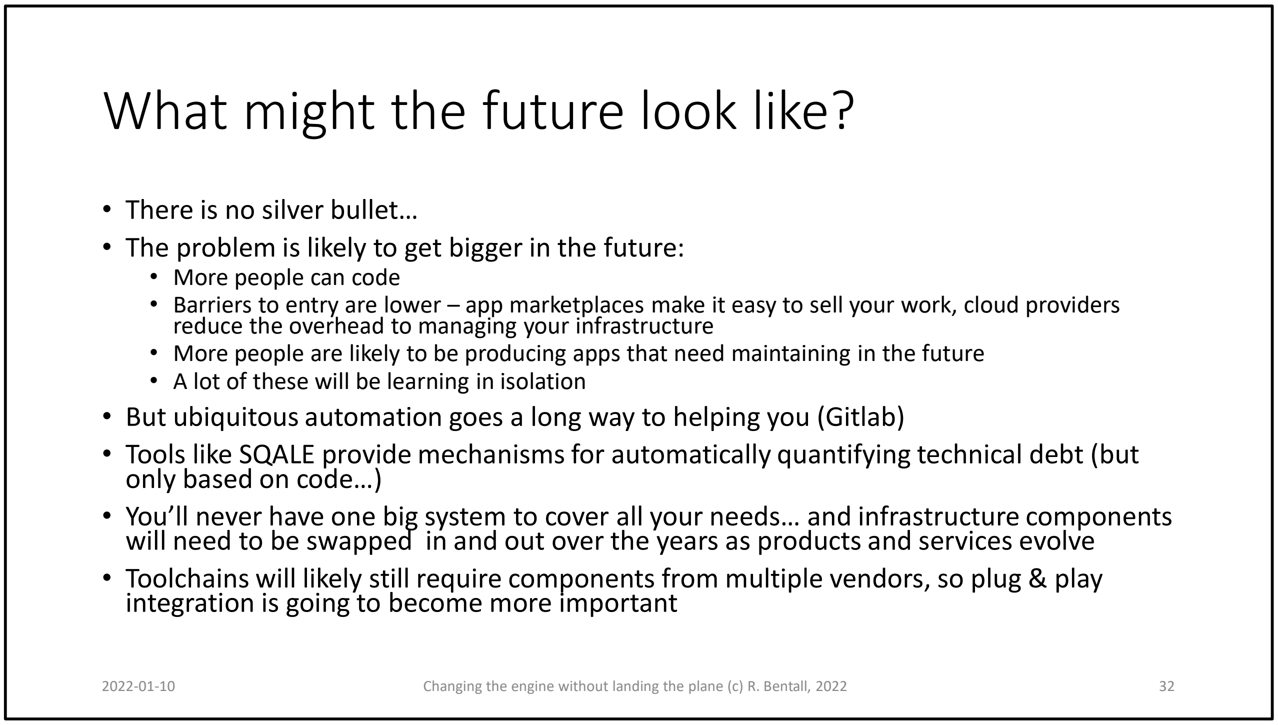


Invest in the future

- Training
- Infrastructure

2022-01-10 Changing the engine without landing the plane (c) R. Bentall, 2022 31

Trying to make the future better than the past is a good antidote to despair….
I still find learning new technology stacks interesting (and I think I'm due for a new one sometime soon…) but there are other areas that are also important.
Can you find some infrastructure in your immediate environment that you can do to improve things a little?
Most bug-trackers and source control systems have the machinery to handle plugins & extensions – does one exist that will help you? Can you write one that will take some drudgery out of your day?
And can you persuade your boss that a particular training course is a good idea?
You won't know until you try.



What might the future look like?

- There is no silver bullet…
- The problem is likely to get bigger in the future:
  - More people can code
  - Barriers to entry are lower – app marketplaces make it easy to sell your work, cloud providers reduce the overhead to managing your infrastructure
  - More people are likely to be producing apps that need maintaining in the future
  - A lot of these will be learning in isolation
- But ubiquitous automation goes a long way to helping you (Gitlab)
- Tools like SQALE provide mechanisms for automatically quantifying technical debt (but only based on code…)
- You'll never have one big system to cover all your needs… and infrastructure components will need to be swapped in and out over the years as products and services evolve
- Toolchains will likely still require components from multiple vendors, so plug & play integration is going to become more important

2022-01-10 Changing the engine without landing the plane (c) R. Bentall, 2022 32

Finally some crystal ball gazing….
As Fred Brookes said, there is no silver bullet…
But there are a lot of ways in which we can make things better.

Thank you

- Any questions?