DevOps from Scratch (Organisational)

Mark Bools

November 14, 2020

Last Modified: 2022-08-03

Contents

Chapter 1

How to...

1.1 ...read this book

If I was being flippant I might say, "with your eyes" but I'm bigger than that so instead I will suggest some ways you might use this material.

This book is organised as a single narrative course centered around the technology that supports DevOps. If you start on page one, read through each page and follow along with all the material, you should end up being proficient with the entire IT System Lifecycle Management.

Don't have time to follow along from the start? Perhaps you already know enough about networks and fell confident skipping that material. No problem. At the start of each section you will find details of how to create an appropriate environment for that section. These 'checkpoints' also mean that if you mess up you can simply throw your environment away, recreate it using the closest checkpoint and continue with the course.

Having difficulty? No problem. Ask for help. Someone in the community may help and since I am in the community I can also help clarify things. If enough people are confused by the material then obviously I messed up and need to rewrite that material to be more clear; it shall be done.

Other books are available with more detailed material. Trying to cover all of the many complex topics under the IT System Lifecycle Management rubric would make this book not only much larger but also less focussed. My solution is to write books to deep-dive into related topics and reference those books from this one where appropriate. This way I hope you will find all the guidance you need either here or in one of the supporting books.

All of these books undergo constant maintenance (hopefully improvement), my only goal is to make the material more clear and more accessible over time.

1.2 ...get the most from this book

Firstly, forget DevOps. Seriously, ignore it. Although this book uses the term DevOps (mostly for marketing reasons) it is really more general, it is about

how to do IT System Lifecycle Management more effectively. DevOps is a distraction at worst and only a small part of sucessful IT System Lifecycle Management at best.

I'm assuming you're primarily interested in the technical parts of this book. If so, *do the examples*! You'll get much more from the material by following along and investigating for yourself. As for the non-technical parts of the book, read them as general advice and observations, not hard and fast rules. More people fail with IT System Lifecycle Management simply because they try to implement DevOps, ITIL, etc. as hard and fast rules rather than guidance.

Chapter 2

Core Concepts

There are several concepts that crop up across the design and management of IT that are so universally applicable that it is worth learning them regardless of your specific interests.

This chapter introduces these core concepts.

2.1 Cohesion

Cohesion refers to how closely elements are related to one another. In generally it is desirable to keep related things together and unrelated things separate.

2.2 Coupling

Coupling refers to how tightly to elements depend upon one another. In general we want elements to be as independent as practicable.

2.3 Abstraction

Abstraction is the process of extracting the essential from the incidental.

2.4 Separation of Concerns

Separation of concerns is a general principal that employs abstraction to increased cohesion and reduce coupling. The general idea is for elements to 'mind their own business', performing a well bounded function (or set of factions).

2.5 Scope

Scope is the 'range of applicability' of an element.

2.6 Context

Everything operates in a context. Most of the time in IT the context is well defined.

2.7 Contingency

Probably best summarised as 'it depends', contingency is the idea that we often must account for things changing.

2.8 Entropy

'Things degrade over time', or perhaps more accurately 'without concerted effort to prevent it, thing get worse over time'. In software circles this is colloquially known as 'bit rot', the idea that without specific work to avoid it a software system's structure will, over time, become more complex, more difficult to maintain, and more prone to error.

2.9 Parsimony

The 'KISS' (Keep It Simple, Stupid) principle. Do not make things more complex than required. The more parts a system has the more opportunity the more things there are to go wrong, so it makes sense to use as few things as possible.

Chapter 3

DevOps from 20,000 feet

This is the obligatory 'what is DevOps' chapter. The problem is DevOps has been so abused as a term that it is largely useless but here goes.

The core idea of DevOps pre-dates the term DevOps, indeed at the start of computing there was no significant distinction between user, developers, or operators as users were also the developers and the operators. As computers matured from academic and military environments into more commercial settings users split off into a clearly distinct group[1]. The people developing systems also started to separate from the people operating the systems, particularly in the mainframe days where these vast machines required the constant attention of operators who loaded punched cards and paper tape, and removes printouts for delivery to developers or users. The advent of Teletype terminals and subsequently video display terminals made it possible for user to interact with systems blissfully ignorant of the operators behind the scenes. Similarly developers became increasingly independent of the operators, able to write, compile, and run their creations independent of the operators keeping things running behind the scenes.

Once developers has completed their work they would hand over the finished product to operators who would then take over the loading and operation of the system on behalf of users. This generally worked well in the early days simply because both developers and operators worked within the same organisation. The increased commodification of software though increased the distance between developer and operator. Indeed operators would often take a software product and install it knowing only that it was an IBM product, never knowing or interacting with the developer. Because this gap developed between commercial suppliers of software and those who operated the systems delivering value to customers so too there tended to be an increasing gap between developers and operators even within a single organisation; why have one rule for operators when working with an external supplier and another when working with internal developers.

[1] Yes, I am massively over simplifying things here, but I think the point stands.

This divorce between developers and operators made a certain sense when delivering packages software, that is software that was a simple stand-alone system. Developers had little concern about working well with others. The operating system ensured the various programs running on the computer were separated from one another so operators had few concerns. Products tended to go through long life-cycles and this allowed for extensive testing (and testing was somewhat simpler as the software operated largely on its own).

This situation did not last long though. Increased complexity meant increased specialisation and consequently increased interaction between components of a system. No longer would a development team write a system from largely from scratch but they would take various commercial products and build them into a system. Each building block would be treated as a black-box and development teams where hostage to the delivery cycle of the vendor to fix problems (resulting in code being developed by these teams to 'work around' perceived deficiencies in the commercial packages).

Long story short, the increased complexity of systems also increased the complexity of delivering systems from development teams into operation. No longer a simple 'here it is' the delivery became a complex set of operations the install various off-the-shelf systems, configure them, then install the custom components, test they worked, etc. All this complexity resulted in either longer delivery times (and wicked documents describing—hopefully—how the installation should be performed), or more commonly development projects would build out the first operational system and deliver the entire system to the operations team (this saving the complexity of delivering build instructions to operators). This second approach often became the norm.

Maintaining development teams after the initial system was delivered starts to become expensive as more and more systems are delivered. Consequently it is common for operations teams to take over maintenance of systems. Problems arise when knowledge held by the developers evaporates with the development team, seldom being effectively communicated to the operations team. This leads to poor understanding of the system and consequently a struggle to diagnose and correct problems.

This problem is exacerbated by the advent of systems supporting web based products and services. The rapid development and delivery of these systems means constant rapid change.

Enter DevOps, a call to return to a culture in which developer and operators work closely together in both developing and maintaining systems. DevOps philosophy complements the Agile movement which encourages close work between users and developers.

DevOps is about creating as frictionless a cycle as possible for the development, validation, delivery, and monitoring of systems.

3.0.1 The DevOps Infinite Cycle

No doubt you've come across the DevOps steps illustrated as a sort of infinity symbols. This suggests that DevOps is an endless cycles of these steps.

- Plan
- Code
- Build
- Test
- Release
- Deploy
- Operate
- Monitor

Chapter 4

The Meta-project

Author Note

This is an early draft, little more than notes and initial thoughts.

Every project has an associated 'meta-project' that deals with all the activities necessary to support the main project.

A Brief History of “devops”