POCKET-SIZED INSIGHTS FOR SOFTWARE TEAMS

TEAM GUIDE TO SOFTWARE OPERABILITY

Matthew Skelton, Alex Moore
& Rob Thatcher

1

conflux

Team Guide to Software Operability

Proven techniques for making software work well

Matthew Skelton, Alex Moore and Rob Thatcher

This book is for sale at http://leanpub.com/SoftwareOperability

This version was published on 2021-04-20

ISBN 978-1-912058-00-6

conflux

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2013 - 2021 Conflux Digital Ltd

Contents

Team Guides for Software

Pocket-sized insights for software teams

The *Team Guides for Software* series takes a *team-first approach* to software systems with the aim of **empowering whole teams** to build and operate software systems more effectively. The books are written and curated by experienced software practitioners and **emphasise the need for collaboration and learning, with the team at the centre**.

Titles in the *Team Guides for Software* series include:

1. *Software Operability* by Matthew Skelton, Alex Moore, and Rob Thatcher
2. *Metrics for Business Decisions* by Mattia Battiston and Chris Young
3. *Software Testability* by Ash Winter and Rob Meaney
4. *Software Releasability* by Manuel Pais and Chris O'Dell

Find out more about the *Team Guides for Software* series by visiting: http://teamguidesforsoftware.com/

Foreword

(Foreword is not available yet.)

Introduction

What is software operability and why should we care?

Software operability is the measure of **how well a software system works when operating** 'live' in production, whether that is the public cloud, a co-located datacentre, an embedded system, or a remote sensor forming part of an Internet of Things (IoT) network. We say that a software system with good operability works well and is *operable*. A highly operable software system is one that minimizes the time and effort needed for unplanned interventions (whether manual or automated) in order to keep the system running.

All too often teams and individuals ignore or downplay aspects of the live/production environment and operating procedures when building software, only for those aspects to cause problems when the software goes live. A focus on operability helps **address these operational concerns** and both **prevent problems happening** in the first place and also make our software systems **more resilient** to unexpected operating conditions.

> "The most amazing abstractions, cleanest code, or beautiful algorithms are meaningless if your code doesnâ€™t run well on production." David Copeland (@davetron5000) [Copeland2013]

Software with a high level of operability is **easy to deploy**, **test**, **and interrogate** in Production. It provides us with the right amount of **good-quality information about the state of the service being provided** and exhibits **predictable and non-catastrophic failure modes** when under high load or abnormal conditions, even if those conditions were never foreseen. Systems with good software operability also lend themselves to **rapid diagnosis and transparent recovery** following a problem, because they have been built with operational criteria as first-class concerns.

Operability is about making software that is reliable and handles failure gracefully in production (live).

Electric trains in Mallorca, Spain, built in 1929 by Siemens and still running in 2014: an example of a physical system with good operability - Photo Copyright (c) Matthew Skelton 2012

Software systems which follow operability good practice will tend to be **simpler to operate and maintain** and will make anticipation and **diagnosis of errors straightforward** [Crowley2012]. Good operability leads to a **reduced lifetime cost of ownership** and fewer operational problems compared to those software systems whose owners have prioritised functionality heavily over operational criteria.

Where can operability techniques be used?

The techniques and approaches we explore in this book work for many kinds of software systems: cloud-native, traditional on-premises enterprise IT, high-frequency/low-latency, mobile apps, desktop client-server, embedded systems, IoT devices, wearables, and industrial/medical applications. We're fairly certain that the techniques also apply to nuclear power stations and space rockets, but these sectors are out of our experience! The important thing here is that these techniques are not specific to any particular set of technologies but work in many different contexts (sometimes requiring some adjustments, sometimes not).

Where techniques differ for a certain kind of software, we identify this with a 'tip' box like this one.

How to use this book

Chapter 1 provides an overview of software operability: what it is, why we need to focus on operability, and what we can expect to gain from it.

Chapters 2-7 provide specific tried-and-tested techniques and practices for enhancing operability for teams building and running software systems.

Each chapter is readable independently, containing the necessary level of detail to be understood and actionable on its own, without requiring any of the other chapters in the book to be read first (although certainly reading the full book will provide a more comprehensive understanding of the concepts and practices and their inter-relations).

Practical entry points to the book:

1. Add 'hooks' into software components for operational checks
2. Have software development teams write a draft runbook
3. Avoid expensive Production-only tooling
4. Ensure that failure responses are gradual, graceful, and graphable
5. Treat logging as a first-class concern and a means of communication
6. Have the product owner and developers on call for Production incidents
7. Make operations activities more visible, for example using Kanban boards, ChatOps and graphing/alerting on operations (such as server restarts, load-balancer workarounds, etc.)

Terminology:

Terminology may seem a lesser concern but in fact extended use of incorrect terms can actually lead to the wrong values and assumptions setting in.

We recommend:

- Avoid the term 'non-functional requirements'; use 'operational features' instead

- Avoid a 'production-ization' or 'hardening' development phase; build in operational excellence from the beginning instead

Terminology and acronyms are explained in the *Terminology* section.

'Slow-burner' organisational changes:

While we've tried to provide very practical approaches to improve operability in this book, we also acknowledge the need to change organizational culture, in particular:

- Recognise that today's distributed, multi-component, multi-dependencies software systems demand team members with a deep skillset which takes time and dedication to either find or develop in-house
- Treat software operations as a high-skill, value-add activity, not support tasks

What is covered in this book

We have *not* tried to cover every possible detail related to software operability; for a comprehensive guide to software operability, we recommend reading the books *Patterns for Performance and Operability: Building and Testing Enterprise Software* by Ford et al (Ford2008), *Release It* by M. Nygard (Nygard2007), and *Continuous Delivery* by Jez Humble and Dave Farley (HumbleFarley2010).

What this guide *does* provide is a set of hands-on practices based on real-world, tried-and-tested experience across multiple organizations for teams to adopt (and adapt) in order to promote and enhance software operability.

In short, if you build or run software systems and care about how well they work, then this book is for you!

Why we wrote this book

Over the course of several years, working in both permanent roles and across numerous engagements with clients as consultants, it became increasingly obvious to us that many organisations were missing some almost intangible but fundamental understanding of what it takes to enhance the performance, deployability, reliability and all-round "working well"-ability of their software systems.

Many discussions and debates about the nuances of these ideas and concepts ensued, and shaped the content of this book over the course of several years. Some of these themes were reflected in the emerging 'DevOps' movement, but others are more related to critical thinking, engineering practice, and understanding human factors.

Approaching the question from our separate perspectives of IT Operations (Rob) and Software Development (Matthew) enabled us to consider a wide range of team drivers, concerns and needs, leading us (we hope) to greater understanding of the symbiosis of development and operations teams, and towards developing methods and techniques for enhancing this relationship.

In the end, we turned a lot of that discussion and thinking into this book and other articles in an effort to share some of the insights gained from our years of experience and to offer some help and hopefully practical advice to developers, operations staff, product managers, owners and others who were interested in getting the most usable software platforms they could.

Feedback and suggestions

We'd welcome feedback and suggestions for changes.

Please contact us at info@operabilitybook.com, via @Operability on Twitter, or on the Leanpub discussion at https://leanpub.com/SoftwareOperability

Matthew Skelton & Rob Thatcher - December 2016

1. What does good operability look like?

(This chapter is not available in this edition of the book.)

2. Core practices for good software operability

(This chapter is not available in this edition of the book.)

3. Use Run Book collaboration to increase operability and prevent operational issues

Key points

- **Use *Run Book collaboration*** as a technique for increasing operability
- **Use a Run Book template** as a common baseline for operational aspects
- **Flush out problems**: run whole-team workshops using Run Book dialogue sheets to highlight gaps in awareness and implementation
- **Own the operability as a team**: revisit operational aspects regularly

3.1 Operational aspects are very similar across many software systems

It's quite common for software teams to think that the systems they work on are unique or somehow special and that there is little they can learn from other systems. The reality is that the vast majority of business systems on which you might work as a software team have almost identical operational needs; the technologies and details change, certainly, but the make-or-break **operational requirements are remarkably similar across most systems**.

For example, all software systems have service level requirements, even if these are not identified or articulated. By extension, these systems also have an implicit or explicit SLA, even if it's just a (physics-defying) agreement that "the system needs to work all the time without failure". Interestingly, non-trivial systems generally have several run-time dependencies (not all of which might be known), and these systems all need to be monitored for healthy operation (especially when one or more of those dependencies fail). Thankfully, this means we can **reuse existing approaches across different systems**!

> *Matthew writes:*
>
> Some years ago, I led a team working on a major system for a well-known financial organisation in London. We were one supplier out of four contributing to the system, and we ran fortnightly, extended 'dress rehearsal' sessions together to test the operational readiness of all parts of the system. I remember that many of the software developers (myself included) were surprised at the things we needed to test: SAN controller failovers, content switch rulesets, SSL offloading, a separate management NIC & networks, scheduled data cleardown, etc.

> As a result of jointly testing for operability like this, we were able to co-design the system and build in operability well before the system went live. Working directly alongside Ops people (network engineers, data architects, etc.) really helped to highlight the operability considerations to the software developers, and we addressed many potential operational problems early on.

At a high level, to make typical software systems work in Production, we need to consider things like:

- The **purpose and remit** of the system - who 'owns' the system?
- The **characteristics** of the system (data flows, network topology, etc.)
- Required **resources** (CPU, storage, etc.)
- **Security** (access, encryption, etc.)
- System **configuration**
- **Backup** and restore of data
- **Monitoring & alerting**
- Regular **operational tasks** (including troubleshooting)
- **Patching** and cleardown
- **Failover** and recovery

Some systems have smaller requirements in some areas compared to other systems, but we need to consider all of these things (and more), whether the system is a cloud-based flight-booking system, an on-premise industrial control unit, or an array of IoT devices forming part of a Smart City deployment. Many software teams find this list of operational aspects quite daunting, but don't worry - the techniques in this chapter will help you to understand and deal with all these operational concerns.

By addressing operational concerns early on we improve operability and therefore how well the system works in Production. Collaborating on operational features builds a lasting trust between Dev and Ops teams and helps achieve better software over an extended period of time, improving the operational readiness of Dev teams.

In modern software systems, **we expect almost all operational tasks to be automated**. Old-style run books with step-by-step instructions for a low-skilled IT support person to follow have no place in today's software systems (see Goldschrafe2011). When we say 'Run Book' here, we mean a **set of common checks and prompts based on sound industry experience - plus high-level answers** - that help teams to discover operational features of their software systems and thereby improve operability, **not a giant document which might replace or even prevent automation and monitoring**. We explore some examples of such *common checks and prompts* in the next section and in the Appendix (Run Book template).

3.2 Use a Run Book template as a common baseline for operational aspects

A good way to start improving operability is by collaborating on filling out a *Run Book template*. A Run Book template is a set of headings and questions that together cover the operational concerns that will need to be addressed in most business software systems. For instance, there are typically headings like these:

Service Level Agreements (SLAs)

What explicit or implicit expectations are there from users or clients about the availability of the service or system?

(e.g. Contractual 99.9% service availability outside of the 03:00-05:00 maintenance window)

Service owner

Which team owns and runs this service or system?

*(e.g. The **Sneaky Sharks** team (Bangalore) develops and runs this service: Telephone Ext. 9265 / _sneaky.sharks@company.com / **#sneaky-sharks** on Slack)*

The point of this Run Book collaboration is *not* to produce a document but to *explore and understand* the expected runtime behaviour of the system.

You can use the Run Book template like this:

1. Start by an overview of the service or system, then clarify with the team what each section in the template means.
2. Fill in some details for sections that you know about.
3. If you don't know the details for a section, mark it as "not defined" - these areas of the system are likely to have operability gaps. Discuss these sections with people outside the team.
4. If any section is simply not relevant for your system, mark it as "not relevant".

Be suspicious of too many sections marked as "not relevant", because there is probably some need for that operational task or

feature, even if it is handled by a third party supplier. Check with members of the operations team before you assume that something is not relevant!

Software developers and testers complete the first draft of the Run Book

You'll probably get the best outcomes from having the *software development team* own and drive the activities around the Run Book information, seeking input from IT Ops people (and others in the organization with operational awareness) to fill in gaps in knowledge.

Traditionally, the IT Ops team defined the Run Book information, but this often leads to 'hacky' workarounds, such as scheduling application or machine restarts every 2 hours in order to work around a memory leak. Above all, it leads to a lack of operational thinking in the development team.

Because the software development team needs to collaborate with the operations team in order to define and complete the various Run Book details, **the operations team also gains early insight into the software**. This helps to set up channels and patterns of communication, trust, and collaboration, which help to improve the quality and operability of the software system early on in the process.

In practice, after filling in the Run Book template, you'll want to automate many of the checks and procedures (rather than leaving them in a wiki or document) using techniques such as Deployment Verification Tests or DVTs - see Chapter 2. The Run Book collaboration activities should help you and your team discover things about how the system will work, along with any gaps in knowledge, implementation, or operational readiness that need to be addressed.

This work on operational aspects should be a regular team activity - see Chapter 7 for more details on how to make operational features a normal part of your team's work. By working on operational aspects of the software every sprint/iteration/week, we keep ourselves 'tuned in' to the practical consequences of product decisions, avoiding a build-up of poor design for the future, and so reducing runtime risk and 'feature friction'.

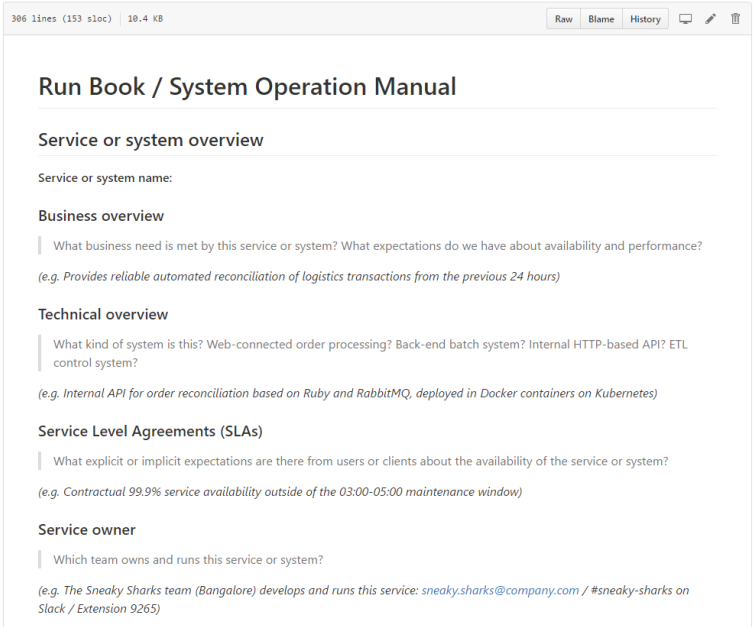
By insisting that **software developers and testers** need to define the draft Run Book details, we help to encourage collaboration between people building new features and people running the live systems.

Many developers will not have full knowledge of the details in the Run Book template or dialogue sheet, so they will need to ask operations people for help.

The purpose of Run Book collaboration is discussion and discovery, not documentation!

Run Book template example

During our work with many different organisations and teams over the years, we have gathered a set of common operational concerns that span many different kinds of software systems across many industry sectors. This *Run Book template* is available on Github or via the shortlink *runbooktemplate.info*:



Run Book template on Github - *runbooktemplate.info*

The template contains more than 50 headings that relate to the operation of modern software systems, along with some sample responses. You can print the template directly from Github, fork the repository to modify it for your own organisation, or use a Run Book Dialogue Sheet. However you use the template, treat the resulting information as a starting point for discussions about operational readiness, not as a finished document.

There is a copy of the Run Book template at the end of this book.

In our experience, you will need to address the majority of the points in the Run Book template, if only to confirm that "this section definitely does not apply here" - a valuable realisation. Each section has a description to set the context and explain why it's needed.

Checklists for quality and safety

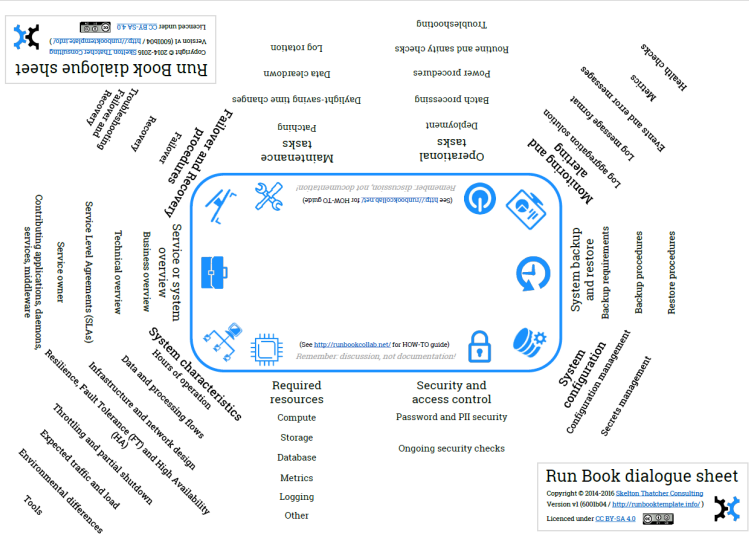
The items in the draft Run Book are analogous to basic coding standard checks such as throwing a NullReferenceException at the start of a method, or boundary checks on array access. Somewhat boring activities that nevertheless can prevent damaging errors in Production.

There is empirical evidence to supporting the use of checklists for maintaining quality (and safety). As reported in the book *The Checklist Manifesto* by Atul Gawande (Gawande2011), a simple surgical checklist from the World Health Organization was recently heralded as "the biggest clinical invention in thirty years".

3.3 Use a Run Book Dialogue Sheet to facilitate discovery and avoid 'documentation fallacy'

We have found that a very effective way of discovering the operational aspects of a software system is to use a *Run Book dialogue sheet*[1]. These are large (A1 size) sheets of paper that cover a good-sized table and show all the headings from the Run Book template, enabling the whole team to interact with the sheet around the table:

[1]We were inspired by Allan Kelly's *Dialogue Sheets* for agile retrospectives and recommend that you try them out! Thanks to Allan for his input to the Run Book dialogue sheets. See Kelly2016.



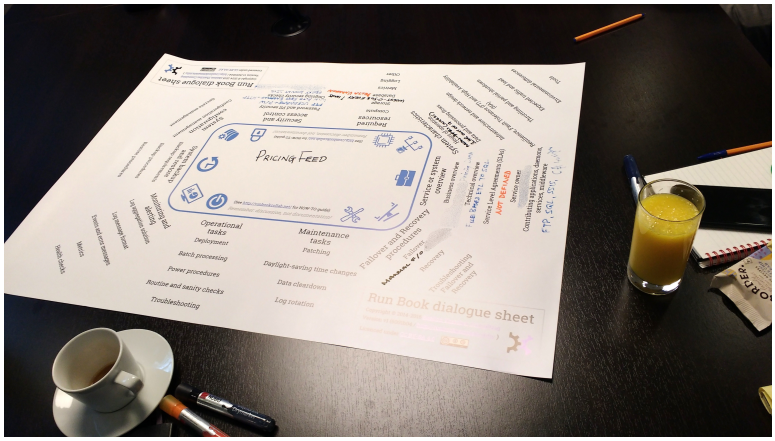
A Run Book dialogue sheet - download from runbooktemplate.info

This particular example is based on the Run Book template hosted at runbooktemplate.info - we've used a Creative Commons SA license on the dialogue sheet so you can modify it and create your own version if you like.

You can use the Run Book dialogue sheet like this:

1. Find the most recent version of the A1-size PDF at runbooktemplate.info.
2. Save and print the PDF at A1 size.
3. Bring together the whole team (Product Owner, UX, Developers, Testers, Build & Release, and Operations people) in the same room.
4. Talk through each heading on the sheet, capturing useful details using a marker pen on the dialogue sheet. We find that it's best to start with the *Service or system overview* section so that the **purpose of the system** is well-understood by everyone.

5. Continue with other headings on the sheet until all sections have been either:
    - Completed
    - or Marked as “Not relevant”
    - or Marked as “Not defined”
6. Take a photograph of the finished dialogue sheet for reference - you will want to return to the details at a later date!
7. (Optional, but recommended) Place the dialogue sheet on the wall next to your team area so it can stimulate discussions with people who walk past it or when new team members join the team.



A Run Book dialogue sheet in action - note that some answers are explicitly *not defined*, indicating possible operability gaps

We recommend spending between 4 hours and 2 days on the Run Book dialogue sheet *per system* to begin with; some systems may need even more time. One very valuable output from these sessions is to discover the headings in the dialogue sheet that end up being *not defined* (rather than *not relevant*). You likely have gaps in operability if an operational aspect of the system is *not defined*, so there is a good chance that errors or bugs will arise around this

aspect. Use this as a 'signal' that more clarity is needed (by asking someone external to the team, for example).

Using Run Book dialogue sheets helps to emphasise the importance of discovering (and rediscovering) operational aspects of the system in a collaborative way. If the whole team has helped shape the operability of the system, there is less need to begin 'documenting' operational requirements in a static wiki or Word document that will quickly become outdated (while providing a false sense of security). The 'documentation fallacy' happens when people rely on those static documents as a safety net, only to find out in the worst moment (during an incident) that those documents have long drifted from the reality of our ever evolving systems.

Invite a facilitator (anyone who knows how to facilitate, regardless of role in the organization) to the all-team Run Book dialogue sheet session. The facilitator can help to assess when to mark a heading as *not defined* and move the discussion on. This helps keep the discussion fresh and avoid too-detailed arguments.

3.4 Assess operability on a regular basis: every sprint, iteration, or week

To maintain a high state of operational readiness in our software, we need to assess operability on a regular basis within the team. A straightforward way to do this is to include a brief exercise in every sprint/iteration planning meeting or retrospective:

1. Place the most recently-completed Run Book dialogue sheet on the table (or print out a blank Run Book template)

2. Assign one person to each of the 10 or so areas of focus on the dialogue sheet:
    - Service or system overview
    - System characteristics
    - Required resources
    - etc.
3. Give each person a different coloured sticky note (or combination of sticky note and letter/shape drawn on the sticky note)
4. For each story card completed (or planned), give everyone (say) 1 minute to assess whether the feature described by the story might affect their area of focus.
5. If someone thinks that their area might be affected, they place a sticky note (plus annotation if needed) onto the story card.

Any story cards with sticky notes are then discussed and (if necessary) taken to other teams to discuss operability concerns:

Checking operability of recent changes using the Run Book dialogue sheet and stickies

Doing this operability assessment exercise should take around 5-10 mins each time once team members are familiar with the

criteria and motivations for doing it. If you run the exercise during planning sessions you will get *leading indicators* of possible operability problems (but with uncertainty), whereas if you run this during a retrospective, you will get *lagging indicators*, although with more detail. Running the exercise during both retrospectives *and* planning sessions gives the best outcomes; you can compare expected effects on operability with the actual!

You can also run the exercise on a weekly or bi-weekly basis using Kanban approaches; just take the cards representing 'done' or 'waiting' (or both!) and assess operability against these tasks.

3.5 Summary

The operability of a software system should be the responsibility of the team building the software. Usually, this means that the team needs to collaborate with operations people (or with operational experience) in order to explore and define the operational characteristics of the system. Proven techniques like *Run Book collaboration* can really help to bring together all the people needed to identify these operational criteria, as well as build trust between teams. Practical tools like *Run Book templates* and *Run Book dialogue sheets* provide a light framework for teams to discover and assess operability in their software on an ongoing basis.

4. Use modern log aggregation for deep operational insights

(This chapter is not available in this edition of the book.)

5. Use Deployment Verification Tests and Endpoint Healthchecks for rapid feedback on environments

(This chapter is not available in this edition of the book.)

6. Use information radiators and dashboards to drive effective behaviour and good psychological responses

(This chapter is not available in this edition of the book.)

7. Make operability part of the software product

(This chapter is not available in this edition of the book.)

8. Appendix

8.1 Adapt your logging techniques to the technology characteristics

Broadly speaking, there are four main dimensions that shape the detail of how we use logging and metrics:

1. The end-to-end connection quality (bandwidth, reliability) between the originating node and the collecting/ingest system - "how much data can we transmit?"
2. The processing power and storage on the originating node - "how much data can we store and forward from the node?"
3. What permissions we have to transmit data from the originating node - "how far can we push the data?"
4. What is the cost to transmit data - "how much per Gigabyte?"

For example, consider a **stock control system** for a large manufacturer running in a secure data centre. Although the software system has access to Gigabit-speed internet bandwidth via fiber connectivity (high connection quality - #1 above), and runs on x86 VMs (good processing power - #2 above), the company refuses to allow log data and metrics to be shipped outside the data centre (limited ability to "push" the data - #3). In this case, the log aggregation system and metrics collector need to be run inside the same data centre and likely have some resource constraints (storage

and processing power) so the emission of log and metrics data needs to be controlled.

Conversely, a **building management system** that uses smart sensors deployed in offices that communicate with a central web-based system using the customers' internet connections as part of the provided service would have very different constraints: the sensors might be reasonably-capable ARM-based devices with plenty of onboard storage, and we can assume that the network connection is rapid, but we should take care not to swamp the customers' connections with too much data, as this may cost them additional fees.

A SaaS-based online **shopping website** running in 2018 likely has excellent bandwidth, capable runtime hardware nodes, permission to send data to a collector/aggregator, and reasonable bandwiidth costs.

Guidelines for logging and metrics

Whatever runtime system you are working with, you should adhere to some basic guidelines for logging and metrics:

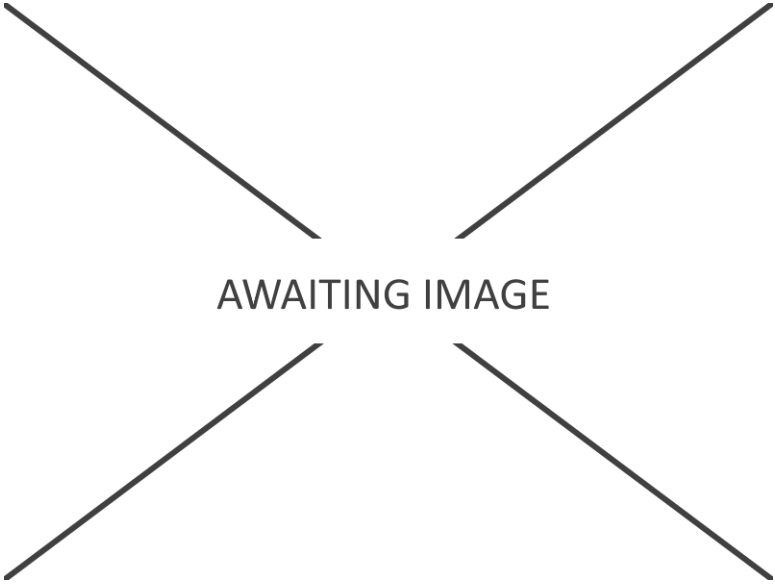
1. Obfuscate sensitive information when logging, such as credit card numbers
2. Encode raw user input before writing to a log - this avoids "log forging" (see *9 Logging Sins* by Eugen Parachiv)
3. Avoid excessive logging and metrics output. Through collaboration with other teams, output only the data that is likely to be needed. Use structured output, not raw string data.
4. Make log messages and metrics names specific. Do not assume that other people will understand the context for that message or metric.

See the article *Lies My Parents Told Me (About Logs)* for more detailed guidance on logging in particular.

Logging and metrics for applications running on x86-based machines

If your software runs directly on a machine that resembles a commodity x86-based server or desktop machine (whether 32-bit or 64-bit, physical or virtual), then it is usually safe to assume that:

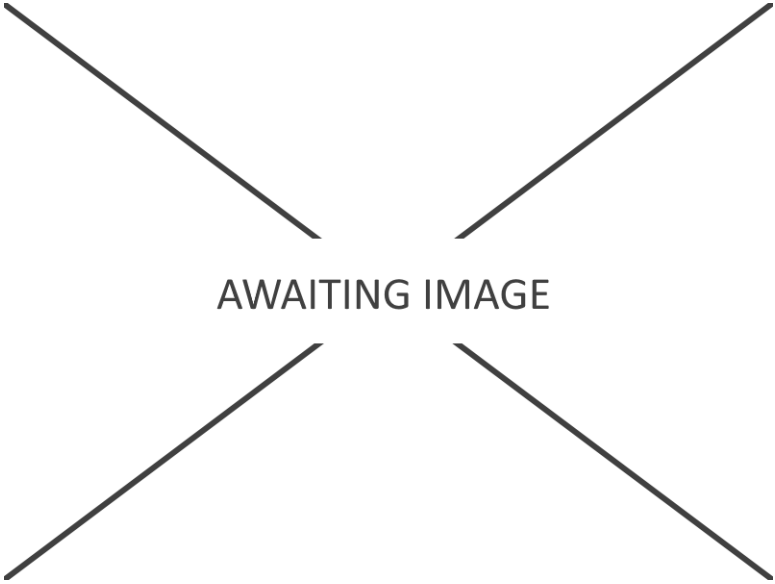
- the machine has plenty of RAM and CPU power, and probably multiple CPU cores
- there is ample local storage for files (including log files)
- the machine has a fast Ethernet connection (100MB)



In this context, a reasonable model for dealing with log data and metrics is to use a locally-installed log agent (a separate daemon/service) that watches log files written to local storage and then forwards these logs to a central location. Metrics can be derived from log data or can be emitted directly from the application with an assumption that the metrics will be collected more or less immediately by the central server.

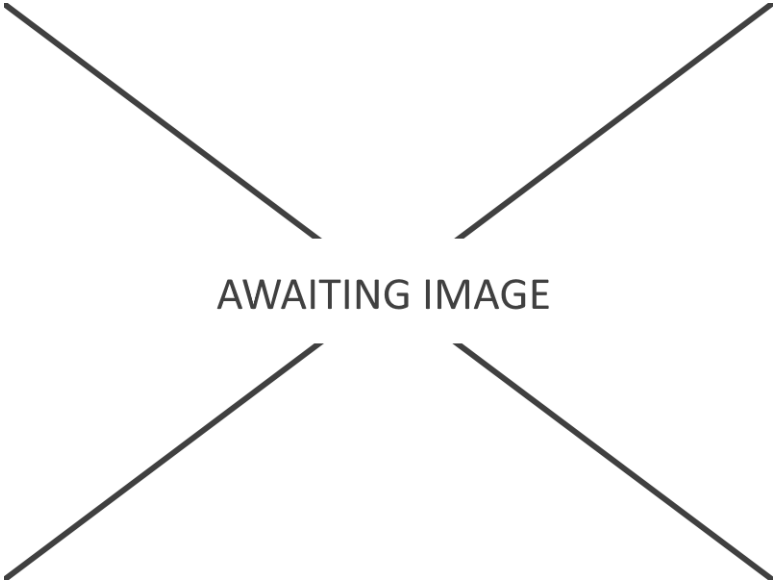
Logging and metrics for containers and containerized applications

If you software runs in a container (Docker, LXC, etc.), good practice is to avoid writing to the filesystem locally inside the container. Storage for containers is heavily virtualised and not optimal for use by many hundreds of containers simultaneously.



Instead, containerized applications should write logs to STDOUT / STDERR, relying on the container fabric or helper containers to listen for this STDOUT output and forward the log messages to a central collector. Generally, the log listeners will decorate the log messages with details of the container ID and type, providing more context for the log messages when they appear in the central aggregator.

Time-series metrics for containerized applications can rapidly saturate a network due to the large number of running containers and the subsequent network traffic that can ensure from metrics transmission. In these cases, metrics can be most effectively managed by using a pre-aggregation helper:



Log aggregation on the container host - *Sysdig*

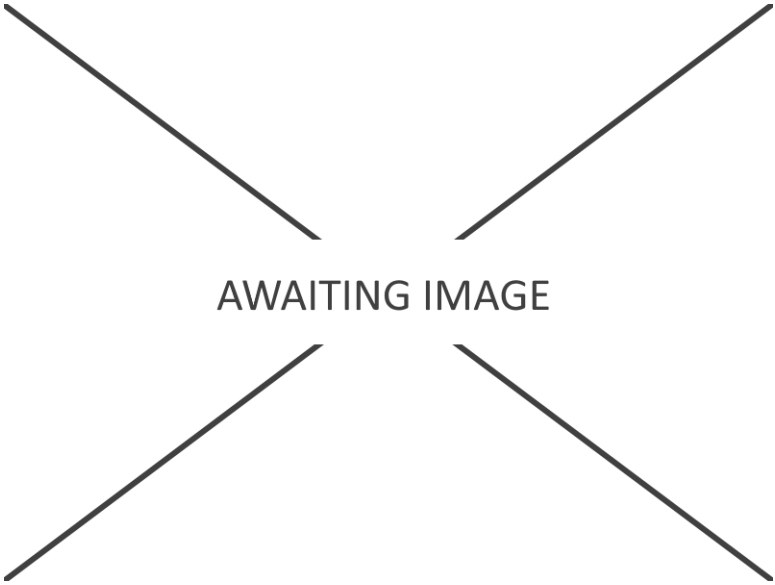
The containerized applications send their metrics to a aggregator container running on the same host (thereby using only the host's network card and not the switched network itself). The aggregator then decorates the application metrics with system-level metrics and (possibly) compresses or collates the application metrics before sending them to the central collector.

Operational Considerations for Containers

For an excellent overview of many operational aspects of containers, see this InfoQ talk by Chris Swan in 2017.

Logging and metrics for Serverless / Function-as-a-Service (FaaS) and Platform-as-a-Service (PaaS)

For Serverless or Function-as-a-Service applications (such as AWS Lambda, Azure Functions, and Google Cloud Functions) and applications using a Platform-as-a-Service framework like CloudFoundry or OpenShift, the logging and metrics options are typically (deliberately) restricted, with the platform or runtime hiding the implementation details.



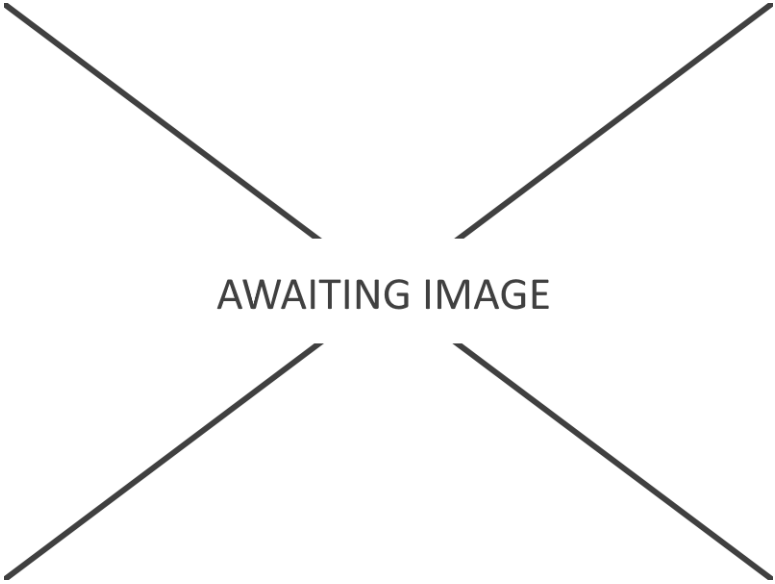
For FaaS and PaaS applications, you will typically either use a library specific to the platform, or send data to a default endpoint (STDOUT or localhost) for the platform to collect. For Serverless/FaaS systems, a "log reflector" approach using one or more additional helping functions seems to work well.

Logging and metrics for IoT / connected devices and embedded systems

With the increase in internet-connected devices ("Internet of Things"), autonomous vehicles, medical and agricultural devices, building sensors, and industrial automation, we need to consider sensible approaches to log aggregation and metrics for IoT devices. Such systems are typically characterised by:

- Limited compute resources (slow CPU, limited RAM, storage that is slow and/or limited in size, limited bandwidth)
- Only occasional network connectivity, often by design for some devices
- Protocols other than HTTP as the initial transport mechanism (MQTT, LoRaWAN, etc.)

In these cases, we need to adopt a pragmatic approach to logging and metrics that still gives us the benefits of rich information and invaluable insights into the operational effectiveness of the device or system *without* compromising CPU/RAM/network limits.



We should be highly selective about what data we send and when we send it from the remote device to any data collection endpoint; if possible, send only essential information in a compressed form. On some IoT networks - such as LoRaWAN - data transmission bandwidth is extremely restricted, so use the bandwidth wisely. It is also important to limit the size of on-device storage used for log and metrics data: overwrite old data if necessary.

Logging and metrics for mobile apps

Personal mobile devices (phones, tablets, and similar devices) sit between x86 machines and IoT devices in terms of storage and processing capabilities. When generating and sending log data and metrics from mobile devices, we need to expect that:

- The network connectivity is intermittent
- The network connectivity is expensive for the user
- The device has limited storage space remaining

This means we should be quite selective about what data we send and when we send it from the mobile device to any remote data collection endpoint; wait for a wifi connection (rather than cellular data) to send log data and metrics, and even then send only essential information in a compressed form. It is also important to limit the size of on-device storage used for log and metrics data, overwriting old data if necessary.

8.2 Understand how the complexity of modern distributed systems drives a need for a focus on operability

> "As systems become more complex, this reductionist way of understanding them fails; they behave in ways that cannot feasibly be predicted from under-standing of the individual parts, or were not expected by the system designer who assembled the parts, or both." – Jeffrey Mogul, p.293

Modern computer systems are typically highly-distributed, multi-node systems that act as part of a larger service. In order to cope with the many different failure modes of these kind of systems, we need to address operability as first-class concern.

Many professional and college/university courses in software engineering, computer science, and programming have only recently begun to tackle the distributed nature of modern software, leaving many software professionals unaware of the need for operability as a foundation for effective distributed software systems.

This effect has been exacerbated by the way in which people have misinterpreted the lack of an operability focus in the Agile Manifesto to mean that operability (and operational concerns in

general) are not important for Agile software development. The Agile manifesto was written at a time when most software development was for user-installed desktop PC software whose operational needs were very simple (a single computer). Software in 2018 and beyond needs an additional focus on operability above and beyond the user focus of the Agile Manifesto.

As the Internet of Things (IoT) drives a proliferation of network-connected devices for both consumer and industrial applications, the potential for unpredictable side-effects in software interactions increases (see Leveson2017a). A strong commitment to operability as a core aspect of modern software is crucial to ensuring that these interconnected systems work effectively.

Terminology

- API: Application Programming Interface - a way to interact with software from other software (rather than via a screen)
- blast radius: the extent of systems affected by a fault
- CAB: Change Advisory Board - a change management group that is supposed to advise on service changes
- FaaS: Function as a Service
- Feature Friction: a term characterised by Matthew Skelton, this is the (usually increasing) "effort needed to add a feature non-invasively" (Michael Feathers). Sometimes (incorrectly) called *Technical Debt*.
- GB: Gigabyte
- IaaS: Infrastructure as a Service
- IoT: Internet of Things
- MB: Megabyte
- NIC: Network Interface Card
- Operability: a measure of how well (software) works in a Production/Live system.
- PaaS: Platform as a Service
- RAM: Random Access Memory
- RAS model: the measures of operability used by IBM since the 1960s of Reliability, Availability, Serviceability
- RBAC: role-based access control
- Run Book: Collection of details about how a software system works in Production. Also known as *System Operation Manual*.
- SaaS: Software as a Service

- SLA: Service Level Agreement
- TB: Terabyte
- UX: User Experience
- Zero-Day Vulnerability: a public security flaw announced with 0 days' warning to vendors

References and further reading

Introduction

Ford2008 - C. Ford, I. Gileadi, S. Purba, and M. Moerman, *Patterns for Performance and Operability: Building and Testing Enterprise Software*. Boca Raton: Auerbach Publications, 2008.

HumbleFarley2010 - J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, 1 edition. Upper Saddle River, NJ: Addison Wesley, 2010.

Nygard2007 - M. T. Nygard, *Release It!: Design and Deploy Production-Ready Software*, 1 edition. Raleigh, N.C: Pragmatic Bookshelf, 2007.

Chapter 1 - What does good operability look like?

Beyer2016 - Beyer, Betsy, Jennifer Petoff, Chris Jones, and Niall Richard Murphy. Site Reliability Engineering. 1 edition. Beijing; Boston: O'Reilly, 2016.

Black2012 - Black, Edwin. "IBM's Role in the Holocaust – What the New Documents Reveal." Huffington Post (blog), February

27, 2012. https://www.huffingtonpost.com/edwin-black/ibm-holocaust_b_1301691.html

Ford2008 - Ford, Chris, Ido Gileadi, Sanjiv Purba, and Mike Moerman. Patterns for Performance and Operability: Building and Testing Enterprise Software. 1 edition. Boca Raton: Auerbach Publications, 2008.

Gawande2011 - A. Gawande, *The Checklist Manifesto: How to Get Things Right*, Reprint edition. New York: Picador, 2011.

Leveson2017b - Leveson, Nancy G. 2017. Engineering a Safer World: Systems Thinking Applied to Safety. Reprint edition. Cambridge, Massachusetts London, Englang: MIT Press.

Nygard2018 - Nygard, Michael T. Release It! Design and Deploy Production-Ready Software. 2nd ed. edition. Raleigh, North Carolina: O'Reilly, 2018.

Chapter 2 - Core Operability Practices

Allspaw2010 - John Allspaw, and Jesse Robbins. 2010. *Web Operations*. O'Reilly Media. http://shop.oreilly.com/product/0636920000136.do

Binette2018 - Binette, Elisa. 2018. "Your Guide to Setting SLOs and SLIs." New Relic Blog (blog). October 31, 2018. https://blog.newrelic.com/engineering practices-for-setting-slos-and-slis-for-modern-complex-systems/

Cohn2008 - M. Cohn, 'Non-functional Requirements as User Stories', 21-Nov-2008. [Online]. Available: http://www.mountaingoatsoftware.com/blog functional-requirements-as-user-stories [Accessed: 12-May-2014].

Cohn2016 - Cohn, Mike. "What Are Story Points?" Mountain Goat Software. August 23, 2016. https://www.mountaingoatsoftware.com/blog/what-are-story-points

Davies2010 - R. Davies, 'Non-Functional Requirements: Do User

Stories Really Help?', 2010. [Online]. Available: http://www.methodsandtools.com/a
[Accessed: 12-May-2014]

Gilb2009 - T. Gilb, 'Tom Gilb & Kai Gilb - Helping you deliver Value to your Stakeholders | Are non-functional requirements functional? : Tom Gilb and Kai Gilb's blog', 18-Jan-2009. [Online]. Available: http://www.gilb.com/blogpost70-Are-non-functional-requirements-functional [Accessed: 12-May-2014].

Humble2010b - Humble, Jez, and David Farley. 2010. "Continuous Delivery: Anatomy of the Deployment Pipeline." http://www.informit.com/articles/a

Kubernetes2018 - Kubernetes. 2018. "Configure Liveness and Readiness Probes." July 2018. https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes/

LonWorks2009 - Echelon Corporation. 2009. "Introduction to the LonWorks® Platform." http://downloads.echelon.com/support/documentation/manu
0183-01B_Intro_to_LonWorks_Rev_2.pdf

Mitchell2017 - Ian Mitchell, "Walking Through a Definition of Done." Scrum.Org. May 17, 2017. http://www.scrum.org/resources/blog/walking-through-definition-done

Murphy2016 - Murphy, Niall, Betsy Beyer, Chris Jones, and Jennifer Petoff. 2016. Site Reliability Engineering. O'Reilly Media. http://shop.oreilly.com/product/0636920041528.do

SAFe2014 - 'Scaled Agile Framework'. [Online]. Available: http://scaledagileframew
[Accessed: 12-May-2014].

Chapter 3 - Use Run Book collaboration to increase operability and prevent operational issues

Gawande2011b - A. Gawande, *The Checklist Manifesto: How to Get Things Right*, Reprint edition. New York: Picador, 2011.

Goldschrafe2011 - J. Goldschrafe, 2011 'Runbooks are stupid and you're doing them wrong' 19-08-2011. [Online] http://holyhandgrenade.org/blog/201 are-stupid-and-youre-doing-them-wrong/ [Accessed 27-Oct-2016]

Kelly2016 - A. Kelly, 2016 'Dialogue Sheets'. [Online] https://www.softwarestrategy. sheets/ [Accessed 24-Oct-2016]

Chapter 4 - Use modern log aggregation for deep operational and insights

Black2016 - David Black, 2016 - "A "Log Reflector" for AWS Lambda" http://blog.davidablack.net/2016/08/17/a-log-reflector-for-aws-lambda/

Boswell2017 - Drew Boswell, 2017 - 'A Million Metrics per Second' [Online] https://medium.com/swissquote-engineering/a-million-metrics-per-second-17a4c7274062 [Accessed 28-Jan-2018]

Bourgon2017 - Peter Bourgon, 2017 - 'Metrics, tracing, and logging' http://peter.bourgon.org/blog/2017/02/21/metrics-tracing-and-logging.html

Cheney2015 - Dave Cheney, 2015 - "Let's Talk About Logging" https://dave.cheney.net/2015/11/05/lets-talk-about-logging

Cui2017 - Yan Cui, 2017 - "Yubl's Road to Serverless, Part 3 - Ops" https://hackernoon.com/yubls-road-to-serverless-part-3-ops-

6c82139bb7ee

DeBortoli2017 - Alberto De Bortoli, 2017 - "A better local and remote logging on iOS with JustLog" https://tech.just-eat.com/2017/01/18/a-better-local-and-remote-logging-on-ios-with-justlog/

Degioanni2015 - Degioanni, Loris. 2015. "How to Collect StatsD Metrics in Containers." *Sysdig*. June 3. https://sysdig.com/blog/how-to-collect-statsd-metrics-in-containers/

GrahamCumming2017 - John Graham-Cumming, 2017 'Incident report on memory leak caused by Cloudflare parser bug' https://blog.cloudflare.com/report-on-memory-leak-caused-by-cloudflare-parser-bug/

Golubenco2016 - Tudor Golubenco, 2016 "Structured logging with Filebeat" https://www.elastic.co/blog/structured-logging-filebeat

Kreps2013 - Jay Kreps, 2013 'The Log: What every software engineer should know about real-time data's unifying abstraction' https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying

Majors2017 - Charity Majors, 2017. "Lies My Parents Told Me (About Logs)" [Online] https://honeycomb.io/blog/2017/04/lies-my-parents-told-me-about-logs/ [Accessed 28-Jan-2018]

OpenTracing2017 - OpenTracing. 2017. "Introduction · Opentracing." 2017. http://opentracing.io/documentation/#what-is-a-trace

Paraschiv2017 - Eugen Paraschiv, 2017 '9 Logging Sins in Your Java Applications'. [Online] https://dzone.com/articles/9-logging-sins-in-your-java-applications [Accessed 28-Jan-2018]

Pivotal2017 - Pivotal, 2017 "Monitoring and Troubleshooting Apps with PCF Metrics | Pivotal Web Services Docs." *Pivotal Web Services Documentation*. http://docs.run.pivotal.io/metrics/using.html#trace.

Rapid72016 - Rapid7, 2016 "Logging Mosquitto Server logs (from Raspberry Pi) to Logentries" https://blog.rapid7.com/2016/10/07/logging-mosquitto-server-logs-from-raspberry-pi-to-logentries/

Reselman2016 - Bob Reselman, 2016 'The Value of Correlation IDs'

https://blog.logentries.com/2016/12/the-value-of-correlation-ids/

Skelton2012 - Matthew Skelton, 2012 "Tune Logging Levels In Production Without Recompiling Code" https://blog.matthewskelton.net/2012/12/05/tur logging-levels-in-production-without-recompiling-code/

Sridharan2017 - Cindy Sridharan, 2017 - "Logs and Metrics" https://medium.com/@ and-metrics-6d34d3026e38

Swan2017 - Chris Swan, 2017 "Operational Considerations for Containers". [Online] https://www.infoq.com/presentations/containers-operations [Accessed 28-Jan-2018]

Turnbull2015 - James Turnbull, 2015 'Structuted Logging'. [Online] https://kartar.net/2015/12/structured-logging/ [Accessed 28-Jan-2018]

Chapter 5 - Use Deployment Verification Tests and Endpoint Healthchecks for rapid feedback on environments

TBC

Chapter 6 - Run operational checks within a deployment pipeline to gain rapid feedback and increased collaboration

TBC

Chapter 7 - Use information radiators and dashboards to drive effective behaviour and good psychological responses

TBC

Chapter 8 - Use operability as a differentiating aspect of your software

TBC

Appendix

AgileManifesto2001 - *Manifesto for Agile Software Development.* 2001. http://agilemanifesto.org/

Leveson2017a - Leveson, Nancy G. 2017. Engineering a Safer World: Systems Thinking Applied to Safety. Reprint edition. Cambridge, Massachusetts London, Englang: MIT Press.

Mogul2006 - Mogul, Jeffrey C. 2006. "Emergent (Mis)Behavior vs. Complex Software Systems." In Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006, 293–304. EuroSys '06. New York, NY, USA: ACM. https://doi.org/10.1145/1217935.1217964

Run Book template

This sample Run Book template is taken from runbooktemplate.info - used with permission of Skelton Thatcher Consulting Ltd.

Licenced under CC by SA - see https://creativecommons.org/licenses/by-sa/4.0/

Revision: 01bfe09

Service or system overview

Service or system name:

Business overview

> What business need is met by this service or system? What expectations do we have about availability and performance?

(e.g. Provides reliable automated reconciliation of logistics transactions from the previous 24 hours)

Technical overview

> What kind of system is this? Web-connected order processing? Back-end batch system? Internal HTTP-based API? ETL control system?

(e.g. Internal API for order reconcilation based on Ruby and RabbitMQ, deployed in Docker containers on Kubernetes)

Service Level Agreements (SLAs)

> What explicit or implicit expectations are there from users or clients about the availability of the service or system?

(e.g. Contractual 99.9% service availability outside of the 03:00-05:00 maintenance window)

Service owner

> Which team owns and runs this service or system?

*(e.g. The *Sneaky Sharks* team (Bangalore) develops and runs this service: sneaky.sharks@company.com / *#sneaky-sharks* on Slack / Extension 9265)*

Contributing applications, daemons, services, middleware

> Which distinct software applications, daemons, services, etc. make up the service or system? What external dependencies does it have?

(e.g. Ruby app + RabbitMQ for source messages + PostgreSQL for reconciled transactions)

System characteristics

Hours of operation

> During what hours does the service or system actually need to operate? Can portions or features of the system be unavailable at times if needed?

Hours of operation - core features

(e.g. 03:00-01:00 GMT+0)

Hours of operation - secondary features

(e.g. 07:00-23:00 GMT+0)

Data and processing flows

> How and where does data flow through the system? What controls or triggers data flows?

(e.g. mobile requests / scheduled batch jobs / inbound IoT sensor data)

Infrastructure and network design

> What servers, containers, schedulers, devices, vLANs, firewalls, etc. are needed?

(e.g. '10+ Ubuntu 14 VMs on AWS IaaS + 2 AWS Regions + 2 VPCs per Region + Route53')

Resilience, Fault Tolerance (FT) and High Availability (HA)

> How is the system resilient to failure? What mechanisms for tolerating faults are implemented? How is the system/service made highly available?

(e.g. 2 Active-Active data centres across two cities + two or more nodes at each layer)

Throttling and partial shutdown

> How can the system be throttled or partially shut down e.g. to avoid flooding other dependent systems? Can the throughput be limited to (say) 100 requests per second? etc. What kind of connection back-off schemes are in place?

Throttling and partial shutdown - external requests

(e.g. Commercial API gateway allows throttling control)

Throttling and partial shutdown - internal components

(e.g. Exponential backoff on all HTTP-based services + `/health` healthcheck endpoints on all services)

Expected traffic and load

> Details of the expected throughput/traffic: call volumes, peak periods, quiet periods. What factors drive the load: bookings, page views, number of items in Basket, etc.)

(e.g. Max: 1000 requests per second with 400 concurrent users - Friday @ 16:00 to Sunday @ 18:00, driven by likelihood of barbecue activity in the neighborhood)

Hot or peak periods

_

Warm periods

_

Cool or quiet periods

_

Environmental differences

> What are the main differences between Production/Live and other environments? What kinds of things might therefore not be tested in upstream environments?

(e.g. Self-signed HTTPS certificates in Pre-Production - certificate expiry may not be detected properly in Production)

Tools

> What tools are available to help operate the system?

(e.g. Use the `queue-cleardown.sh` script to safely cleardown the processing queue nightly)

Required resources

> What compute, storage, database, metrics, logging, and scaling resources are needed? What are the minimum and expected maximum sizes (in CPU cores, RAM, GB disk space, GBit/sec, etc.)?

Required resources - compute

(e.g. Min: 4 VMs with 2 vCPU each. Max: around 40 VMs)

Required resources - storage

(e.g. Min: 10GB Azure blob storage. Max: around 500GB Azure blob storage)

Required resources - database

(e.g. Min: 500GB Standard Tier RDS. Max: around 2TB Standard Tier RDS)

Required resources - metrics

(e.g. Min: 100 metrics per node per minute. Max: around 6000 metrics per node per minute)

Required resources - logging

(e.g. Min: 60 log lines per node per minute (100KB). Max: around 6000 log lines per node per minute (1MB))

Required resources - other

(e.g. Min: 10 encryption requests per node per minute. Max: around 100 encryption requests per node per minute)

Security and access control

Password and PII security

What kind of security is in place for passwords and Personally Identifiable Information (PII)? Are the passwords hashed with a strong hash function and salted?

(e.g. Passwords are hashed with a 10-character salt and SHA265)

Ongoing security checks

How will the system be monitored for security issues?

(e.g. The ABC tool scans for reported CVE issues and reports via the ABC dashboard)

System configuration

Configuration management

How is configuration managed for the system?

(e.g. CloudInit bootstraps the installation of Puppet - Puppet then drives all system and application level configuration except for the XYZ service which is configured via `App.config` files in Subversion)

Secrets management

How are configuration secrets managed?

(e.g. Secrets are managed with Hashicorp Vault with 3 shards for the master key)

System backup and restore

Backup requirements

Which parts of the system need to be backed up?

(e.g. Only the CoreTransactions database in PostgreSQL and the Puppet master database need to be backed up)

Backup procedures

How does backup happen? Is service affected? Should the system be [partially] shut down first?

(e.g. Backup happens from the read replica - live service is not affected)

Restore procedures

How does restore happen? Is service affected? Should the system be [partially] shut down first?

(e.g. The Booking service must be switched off before Restore happens otherwise transactions will be lost)

Monitoring and alerting

Log aggregation solution

What log aggregation & search solution will be used?

(e.g. The system will use the existng in-house ELK cluster. 2000-6000 messages per minute expected at normal load levels)

Log message format

> What kind of log message format will be used? Structured logging with JSON? `log4j` style single-line output?

(e.g. Log messages will use log4j compatible single-line format with wrapped stack traces)

Events and error messages

> What significant events, state transitions and error events may be logged?

(e.g. IDs 1000-1999: Database events; IDs 2000-2999: message bus events; IDs 3000-3999: user-initiated action events; ...)

Metrics

> What significant metrics will be generated?

(e.g. Usual VM stats (CPU, disk, threads, etc.) + around 200 application technical metrics + around 400 user-level metrics)

Health checks

> How is the health of dependencies (components and systems) assessed? How does the system report its own health?

Health of dependencies

(e.g. Use `/health` HTTP endpoint for internal components that expose it. Other systems and external endpoints: typically HTTP 200 but some synthetic checks for some services)

Health of service

(e.g. Provide `/health` HTTP endpoint: 200 –> basic health, 500 –> bad configuration + `/health/deps` for checking dependencies)

Operational tasks

Deployment

How is the software deployed? How does roll-back happen?

(e.g. We use GoCD to coordinate deployments, triggering a Chef run pulling RPMs from the internal yum repo)

Batch processing

What kind of batch processing takes place?

(e.g. Files are pushed via SFTP to the media server. The system processes up to 100 of these per hour on a `cron` schedule)

Power procedures

What needs to happen when machines are power-cycled (or cold-rebooted)?

*(e.g. *** WARNING: we have not investigated this scenario yet! ***)*

Routine and sanity checks

What kind of checks need to happen on a regular basis?

(e.g. All `/health` endpoints should be checked every 60secs plus the synthetic transaction checks run every 5 mins via Pingdom)

Troubleshooting

How should troubleshooting happen? What tools are available?

(e.g. Use a combination of the `/health` endpoint checks and the `abc-*.sh` scripts for diagnosing typical problems)

Maintenance tasks

Patching

How should patches be deployed and tested?

Normal patch cycle

(e.g. Use the standard OS patch test cycle together with deployment via Jenkins and Capistrano)

Zero-day vulnerabilities

(e.g. Use the early-warning notifications from UpGuard plus deployment via Jenkins and Capistrano)

Daylight-saving time changes

Is the software affected by daylight-saving time changes (both client and server)?

(e.g. Server clocks all set to UTC+0. All date/time data converted to UTC with offset before processing)

Data cleardown

> Which data needs to be cleared down? How often? Which tools or scripts control cleardown?

(e.g. Use `abc-cleardown.ps1` run nightly to clear down the document cache)

Log rotation

> Is log rotation needed? How is it controlled?

*(e.g. The Windows Event Log *ABC Service* is set to a maximum size of 512MB)*

Failover and Recovery procedures

> What needs to happen when parts of the system are failed over to standby systems? What needs to during recovery?

Failover

_

Recovery

_

Troubleshooting Failover and Recovery

> What tools or scripts are available to troubleshoot failover and recovery operations?

*(e.g. Start with running `SELECT state__desc FROM sys.database_-_mirroring__endpoints` on the PRIMARY node and then use the scripts in the *db-failover* Git repo)*

About the authors

Matthew Skelton

Matthew Skelton

Matthew Skelton has been building, deploying, and operating commercial software systems since 1998. Head of Consulting at Conflux (confluxdigital.net), he specialises in Continuous Delivery, operability and organisation design for software in manufacturing, ecommerce, and online services, including cloud, IoT, and embedded software. Matthew also co-founded pioneering DevOps consultancy Skelton Thatcher Consulting, which led industry thinking around operability and team design. Matthew is co-author with Chris O'Dell of Continuous Delivery with Windows and .NET (O'Reilly, 2016), and instigated the first conference in Europe dedicated to Continuous Delivery, PIPELINE Conference.

Alex Moore

Alex Moore

System Build Engineer at TransUnion UK, Alex Moore is a passionate advocate of all things DevOps. Originally from an Operational background, she now works closely with Product Development Teams to improve practice, flow and automation. She is proud to have been part of the DevOps transformation within her own organisation and constantly participates in the wider DevOps community, learning and communicating. She will not rest until Ops and Operability are seen as intrinsic to software development.

Rob Thatcher

Rob Thatcher

Co-founder at Skelton Thatcher Consulting, Rob Thatcher has substantial experience helping organisations to build effective technical operations, support, and delivery teams, and design and operate

effective IT architectures. With Director-level experience in the financial services sector, his focus is on building high availability and high performance environments in on-premise deployments, hosted private cloud environments, and public cloud services.

Conflux Books

Books for technologists by technologists

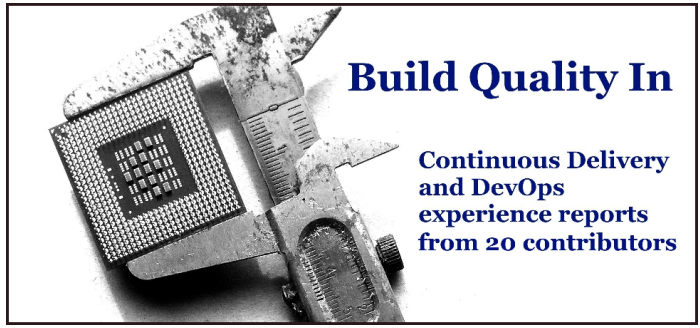
Our books help to accelerate and deepen your learning in the field of software systems. We focus on subjects that don't go out of date: fundamental software principles & practices, team interactions, and technology-independent skills.

Current and planned titles in the *Conflux Books* series include:

1. *Build Quality In* edited by Steve Smith and Matthew Skelton (B01)
2. *Better Whiteboard Sketches* by Matthew Skelton (B02)
3. *Internal Tech Conferences* by Victoria Morgan-Smith and Matthew Skelton (B03)
4. *Technical Writing for Blogs and Articles* by Matthew Skelton (B04)

Find out more about the *Conflux Books* series by visiting: confluxbooks.com

conflux



Build Quality In - a book of Continuous Delivery and DevOps experience reports. Edited by Steve Smith and Matthew Skelton. Conflux Books, April 2015

Internal Tech Conferences by Victoria Morgan-Smith and Matthew Skelton. Conflux Books, April 2019

Better Whiteboard Sketches by Matthew Skelton. Conflux Books, August 2019

Team Guides for Software: *Software Operability, Metrics for Business Decisions, Software Testability, Software Releasability.* Skelton Thatcher Publications, 2016-2019