DevOps

Martin Kellogg

Reading quiz: DevOps

Q1: **TRUE** or **FALSE**: Google intentionally hires people for SRE teams who will quickly become bored by performing tasks by hand

Q2: As an example of automation gone awry, the “Emergency Response” article discussed:

A. an automation test that accidentally triggered Diskerase requests for many servers

B. an engineer who accidentally removed 100% of capacity from a system (instead of 10%) by typing an extra “0”

C. a self-replicating AI system that took over a datacenter

Reading quiz: DevOps

Q1: **TRUE** or **FALSE**: Google intentionally hires people for SRE teams who will quickly become bored by performing tasks by hand

Q2: As an example of automation gone awry, the "Emergency Response" article discussed:

A. an automation test that accidentally triggered Diskerase requests for many servers

B. an engineer who accidentally removed 100% of capacity from a system (instead of 10%) by typing an extra "0"

C. a self-replicating AI system that took over a datacenter

Reading quiz: DevOps

Q1: **TRUE** or **FALSE**: Google intentionally hires people for SRE teams who will quickly become bored by performing tasks by hand

Q2: As an example of automation gone awry, the "Emergency Response" article discussed:

A. an automation test that accidentally triggered Diskerase requests for many servers
B. an engineer who accidentally removed 100% of capacity from a system (instead of 10%) by typing an extra "0"
C. a self-replicating AI system that took over a datacenter

Announcements

- Reminder: preliminary demos this week. Double-check the time of your team's demo with me.
  - Sorry to those of you who I've had to reschedule
- Team survey #2 will open this afternoon. You should fill it out after you've finished both of your preliminary demos but before the Thanksgiving holiday
- No class on November 26 (one week from today)
  - It's a Friday schedule

DevOps

Today's agenda:

- **Operations, Toil, and the DevOps philosophy**
- Achieving reliability
  - the service reliability hierarchy + SLAs/targets
  - monitoring and reliability testing
  - incident/emergency response
  - preventing problems before they occur
  - post-mortems + learning from failure

Operations

Definition: ***operations*** refers to anything that happens after the developers (think that they) are done building the software, including:

Operations

Definition: ***operations*** refers to anything that happens after the developers (think that they) are done building the software, including:

- setting up the servers that will run the software and installing the software on them

Operations

Definition: ***operations*** refers to anything that happens after the developers (think that they) are done building the software, including:

- setting up the servers that will run the software and installing the software on them
- conducting system/acceptance tests

Operations

Definition: ***operations*** refers to anything that happens after the developers (think that they) are done building the software, including:

- setting up the servers that will run the software and installing the software on them
- conducting system/acceptance tests
- running the software and keeping it running

Operations

Definition: ***operations*** refers to anything that happens after the developers (think that they) are done building the software, including:

- setting up the servers that will run the software and installing the software on them
- conducting system/acceptance tests
- running the software and keeping it running
- measuring the performance of the running software

Operations

Definition: ***operations*** refers to anything that happens after the developers (think that they) are done building the software, including:

- setting up the servers that will run the software and installing the software on them
- conducting system/acceptance tests
- running the software and keeping it running
- measuring the performance of the running software
- fixing any problems that arise while the software is running

Operations

Definition: ***operations*** refers to anything that happens after the developers (think that they) are done building the software, including:

- setting up the servers that will run the software and installing the software on them
- conducting system/acceptance tests
- running the software and keeping it running
- measuring the performance of the running software
- fixing any problems that arise while the software is running
- deploying new versions of the software

Operations: the traditional approach

Operations: the traditional approach

- traditionally, operations are mostly conducted by **system administrators** (or **sysadmins**) rather than by developers

Operations: the traditional approach

- traditionally, operations are mostly conducted by **system administrators** (or **sysadmins**) rather than by developers
  - sysadmins are specialists in specific tech stacks
    - e.g., experts at Linux or Windows, etc.

Operations: the traditional approach

- traditionally, operations are mostly conducted by **system administrators** (or **sysadmins**) rather than by developers
  - sysadmins are specialists in specific tech stacks
    - e.g., experts at Linux or Windows, etc.
  - e.g., NJIT's IT undergrad degree program was (probably) originally intended as preparation for this kind of role

Operations: the traditional approach

- traditionally, operations are mostly conducted by **system administrators** (or **sysadmins**) rather than by developers
  - sysadmins are specialists in specific tech stacks
    - e.g., experts at Linux or Windows, etc.
  - e.g., NJIT's IT undergrad degree program was (probably) originally intended as preparation for this kind of role
- this approach is best when systems **change rarely**

Operations: the traditional approach

- traditionally, operations are mostly conducted by **system administrators** (or **sysadmins**) rather than by developers
  - sysadmins are specialists in specific tech stacks
    - e.g., experts at Linux or Windows, etc.
  - e.g., NJIT's IT undergrad degree program was (probably) originally intended as preparation for this kind of role
- this approach is best when systems **change rarely**
  - e.g., when software is released on physical media

Operations: the traditional approach

- traditionally, operations are mostly conducted by **system administrators** (or **sysadmins**) rather than by developers
  - sysadmins are specialists in specific tech stacks
    - e.g., experts at Linux or Windows, etc.
  - e.g., NJIT's IT undergrad degree program was (probably) originally intended as preparation for this kind of role
- this approach is best when systems **change rarely**
  - e.g., when software is released on physical media
  - other advantages: easy to staff for, off-the-shelf tooling, etc.

Traditional ops in different business models

- two business models:

Traditional ops in different business models

- two business models:
  - **services** (i.e., the developing organization runs the software and sells access to customers)

Traditional ops in different business models

- two business models:
  - **services** (i.e., the developing organization runs the software and sells access to customers)
    - service ops: need to set up the servers/machines on which the software will run, install the software + dependencies, configure firewalls, etc.

Traditional ops in different business models

- two business models:
  - **services** (i.e., the developing organization runs the software and sells access to customers)
    - service ops: need to set up the servers/machines on which the software will run, install the software + dependencies, configure firewalls, etc.
  - **products** (i.e., sell/lease the software to others to run)

Traditional ops in different business models

- two business models:
  - **services** (i.e., the developing organization runs the software and sells access to customers)
    - service ops: need to set up the servers/machines on which the software will run, install the software + dependencies, configure firewalls, etc.
  - **products** (i.e., sell/lease the software to others to run)
    - product ops: still need to system test in the anticipated operating environment(s), set up servers providing those environments, install the software + dependencies, etc.

Traditional ops in di

Traditional approach to operations can work in either of these models!

- two business models:
  - **services** (i.e., the developing organization runs the software and sells access to customers)
    - service ops: need to set up the servers/machines on which the software will run, install the software + dependencies, configure firewalls, etc.
  - **products** (i.e., sell/lease the software to others to run)
    - product ops: still need to system test in the anticipated operating environment(s), set up servers providing those environments, install the software + dependencies, etc.

Operations: the traditional approach

- However, the traditional sysadmin approach to operations has downsides, too:

Operations: the traditional approach

- However, the traditional sysadmin approach to operations has downsides, too:
  - for services, ops costs **scale with system load**: more users = must hire more sysadmins to administer more servers, etc.

Operations: the traditional approach

- However, the traditional sysadmin approach to operations has downsides, too:
  - for services, ops costs **scale with system load**: more users = must hire more sysadmins to administer more servers, etc.
  - separation of operations and development means developers are not **directly exposed** to the costs of poor design decisions
    - this is a misalignment of incentives

Operations: the traditional approach

- However, the traditional sysadmin approach to operations has downsides, too:
  - for services, ops costs **scale with system load**: more users = must hire more sysadmins to administer more servers, etc.
  - separation of operations and development means developers are not **directly exposed** to the costs of poor design decisions
    - this is a misalignment of incentives
  - developers and sysadmins have different backgrounds, terminology, etc., leading to **communication breakdowns**

Operations: the traditional approach

- However, the traditional
  downsides, too:
  - for services, ops c
    must hire more sy
  - separation of ope
    are not **directly e**
    - this is a misali
  - developers and sy
    terminology, etc., leading to **communication breakdowns**

These problems **do not** mean that the traditional approach to operations is bad in all circumstances!

Operations: the traditional approach

- However, the traditional
  downsides, too:
  - for services, ops c
    must hire more sy
  - separation of ope
    are not **directly e**
    - this is a misali
  - developers and sy
    terminology, etc., leading to **communication breakdowns**

These problems **do not** mean that the traditional approach to operations is bad in all circumstances!

- But, they are serious concerns for modern systems with high release cadences, especially those that are:
  - microservices
  - delivered via the web
  - use “continuous delivery”

Operations: the DevOps approach

Operations: the DevOps approach

Key idea: combine the development and operations teams

Operations: the DevOps approach

Key idea: combine the development and operations teams

- “DevOps” is a portmanteau of “developers” + “operators”

Operations: the DevOps approach

Key idea: combine the development and operations teams

- “DevOps” is a portmanteau of “developers” + “operators”
- DevOps teams are organized around services/projects

Operations: the DevOps approach

Key idea: combine the development and operations teams

- “DevOps” is a portmanteau of “developers” + “operators”
- DevOps teams are organized around services/projects
    - similar to organizational motivation for **microservices**

Operations: the DevOps approach

Key idea: combine the development and operations teams

- “DevOps” is a portmanteau of “developers” + “operators”
- DevOps teams are organized around services/projects
  - similar to organizational motivation for **microservices**
- operational burden is **shared** by the developers who are building the system

Operations: the DevOps approach

Key idea: combine the development and operations teams

- "DevOps" is a portmanteau of "developers" + "operators"
- DevOps teams are organized around services/projects
  - similar to organizational motivation for **microservices**
- operational burden is **shared** by the developers who are building the system
  - better alignment of incentives between developers and operators, since same people perform both roles

Operations: the DevOps approach

Key idea: combine the development and operations teams

- “DevOps” is a portmanteau of “developers” + “operators”
- DevOps teams are organized around services/projects
  - similar to organizational motivation for **microservices**
- operational burden is **shared** by the developers who are building the system
  - better alignment of incentives between developers and operators, since same people perform both roles
- encourage operators to automate **toil**

Operations: the DevOps approach

Key idea: combine the development and operations teams

- "DevOps" is a portmanteau of "developers" + "operators"
- DevOps teams are organized around services/projects
  - similar to organizational motivation for **microservices**
- operational burden is **shared** by the developers who are building the system
  - better alignment of incentives between developers and operators, since same people perform both roles
- encourage operators to automate **toil**
- may still have some dedicated ops roles (e.g., SREs at Google)

Operations: the DevOps approach

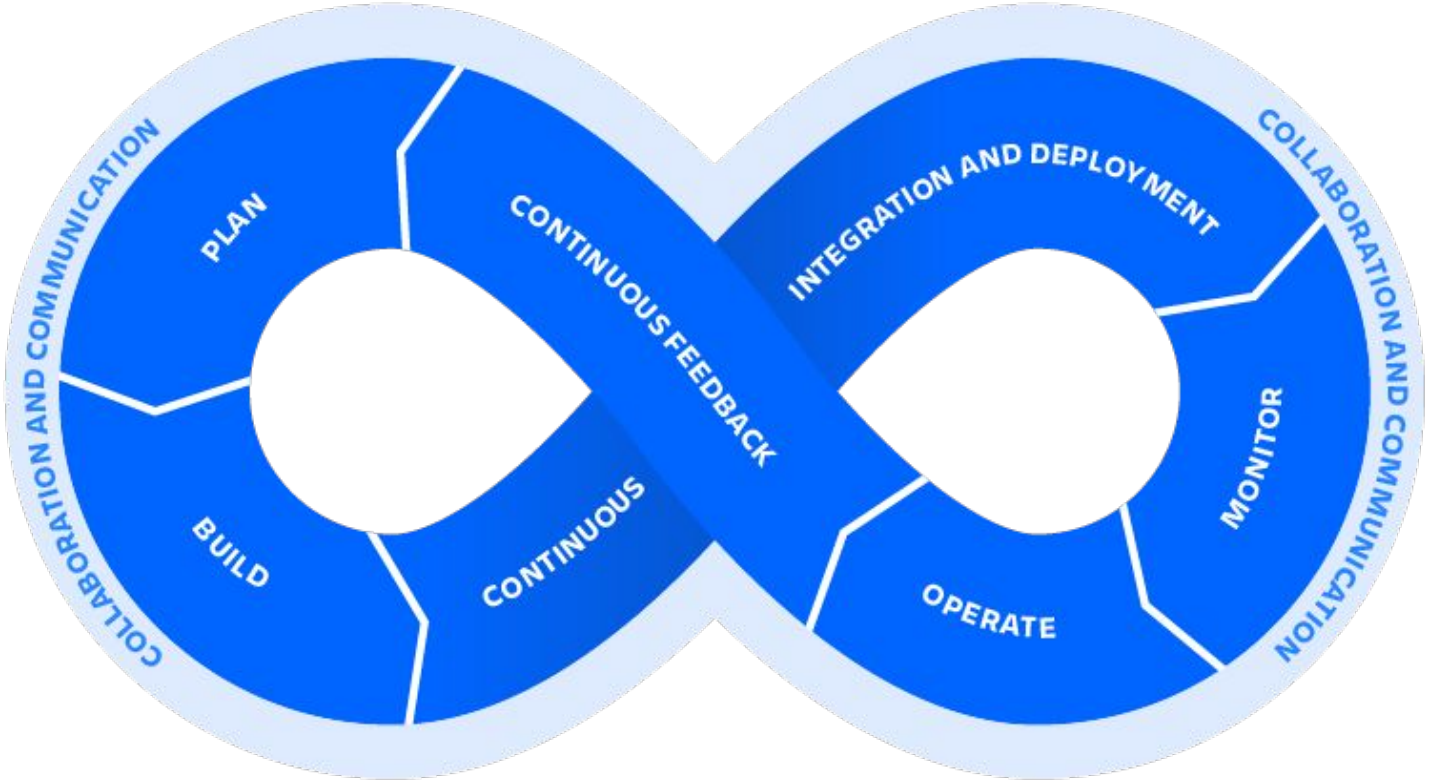


figure credit: Atlassian

Operations: toil

> *If a human operator needs to touch your system during normal operations, you have a bug. The definition of normal changes as your systems grow.*
>
> Carla Geisser, Google SRE

Operations: toil

Definition: ***toil*** is the kind of work tied to running a production service that tends to be manual, repetitive, automatable, tactical, devoid of enduring value, and that scales linearly as a service grows

Operations: toil

Definition: ***toil*** is the kind of work tied to running a production service that tends to be manual, repetitive, automatable, tactical, devoid of enduring value, and that scales linearly as a service grows

A key advantage of DevOps is that it encourages **removing** toil

- if operators are separate from devs, devs have no incentive to avoid toil

Operations: toil

Definition: ***toil*** is the kind of work tied to running a production service that tends to be manual, repetitive, automatable, tactical, devoid of enduring value, and that scales linearly as a service grows

- **manual**: includes work such as manually running a script that automates some task (typing the command itself is toil!)

Operations: toil

Definition: ***toil*** is the kind of work tied to running a production service that tends to be manual, repetitive, automatable, tactical, devoid of enduring value, and that scales linearly as a service grows

- **manual**: includes work such as manually running a script that automates some task (typing the command itself is toil!)
- **repetitive**: if you're performing a task for the first time ever, or even the second time, this work is not toil

Operations: toil

Definition: ***toil*** is the kind of work tied to running a production service that tends to be manual, repetitive, automatable, tactical, devoid of enduring value, and that scales linearly as a service grows

- **manual**: includes work such as manually running a script that automates some task (typing the command itself is toil!)
- **repetitive**: if you're performing a task for the first time ever, or even the second time, this work is not toil
- **automatable**: if human judgment is essential for the task, there's a good chance it's not toil

Operations: toil

Definition: ***toil*** is the kind of work tied to running a production service that tends to be manual, repetitive, automatable, tactical, devoid of enduring value, and that scales linearly as a service grows

- **tactical**: toil is usually interrupt-driven and reactive

Operations: toil

Definition: ***toil*** is the kind of work tied to running a production service that tends to be manual, repetitive, automatable, tactical, devoid of enduring value, and that scales linearly as a service grows

- **tactical**: toil is usually interrupt-driven and reactive
- **no enduring value**: if your service remains in the same state after you have finished a task, the task was probably toil

Operations: toil

Definition: ***toil*** is the kind of work tied to running a production service that tends to be manual, repetitive, automatable, tactical, devoid of enduring value, and that scales linearly as a service grows

- **tactical**: toil is usually interrupt-driven and reactive
- **no enduring value**: if your service remains in the same state after you have finished a task, the task was probably toil
- **O(n) with service growth**: if the work involved in a task scales up linearly with *service size*, *traffic volume*, or *user count*, that task is probably toil

Operations: toil

Definition: ***toil*** is the kind of work tied to running a production service that tends to be manual, repetitive, automatable, tactical, devoid of enduring value, and that scales linearly as a service grows

- **tactical**: toi
- **no enduri** e after you have fi
- **O(n) with s** les up linearly wit sk is probably toil

A task doesn't need to have **all** of these attributes to be toil. But, the more closely work matches one or more of these descriptors, the **more likely** it is to be toil.

Operations: toil

Things that **aren’t** toil:

Operations: toil

Things that **aren’t** toil:

- work you don’t like to do is **not always** toil

Operations: toil

Things that **aren't** toil:

- work you don't like to do is **not always** toil
  - useful, productive work can be unpleasant
    - e.g., cleaning up the entire alerting configuration for your service and removing clutter may not be fun, but it's not toil

Operations: toil

Things that **aren't** toil:

- work you don't like to do is **not always** toil
  - useful, productive work can be unpleasant
    - e.g., cleaning up the entire alerting configuration for your service and removing clutter may not be fun, but it's not toil
  - but most toil is unpleasant

Operations: toil

Things that **aren't** toil:

- work you don't like to do is **not always** toil
  - useful, productive work can be unpleasant
    - e.g., cleaning up the entire alerting configuration for your service and removing clutter may not be fun, but it's not toil
  - but most toil is unpleasant
- **overhead** is also different than toil

Operations: toil

Things that **aren't** toil:

- work you don't like to do is **not always** toil
  - useful, productive work can be unpleasant
    - e.g., cleaning up the entire alerting configuration for your service and removing clutter may not be fun, but it's not toil
  - but most toil is unpleasant
- **overhead** is also different than toil
  - tasks like team meetings, setting and grading goals, and HR paperwork (that are not tied to operations) are overhead

Operations: toil

What’s **so bad** about toil?

Operations: toil

What's **so bad** about toil?

- career stagnation (it doesn't get you promoted)
- lowers morale (it's boring)
- creates confusion (easy to forget to do a manual task!)
- slows progress (could be doing useful work instead)
- sets precedent (avoid letting toil become normal!)
- promotes attrition ("I want to work on something interesting!")

Operations: toil

What's **so bad** about toil?

- career stagnation (it doesn't get you promoted)
- lowers morale (it's boring)
- creates co
- slows prog
- sets prece
- promotes esting!")

Despite all this, a **little bit** of toil is often okay. After all, engineers only have so many productive hours in every day, and sometimes a **mental break** is nice :)

DevOps example: Google SREs

DevOps example: Google SREs

- SRE teams are a mix of:
  - software engineers
  - software-inclined sysadmins

DevOps example: Google SREs

- SRE teams are a mix of:
  - software engineers
  - software-inclined sysadmins
- goal: SRE teams should spend at least 50% of their time on “development” work and at most 50% on toil

DevOps example: Google SREs

- SRE teams are a mix of:
  - software engineers
  - software-inclined sysadmins
- goal: SRE teams should spend at least 50% of their time on “development” work and at most 50% on toil
- SRE teams are assigned to a collection of related “SWE” (i.e., software engineering/development) teams, each of which works on one of the systems
  - SRE team manages ops for all of these systems

DevOps example: Google SREs

- SRE teams are a mix of:
  - software engineers
  - software-inclined sysadmins
- goal: SRE teams should spend at least 50% of their time on “development” work and at most 50% on toil
- SRE teams are assigned to a collection of related “SWE” (i.e., software engineering/development) teams, each of which works on one of the systems
  - SRE team manages ops for all of these systems
- SRE motto: “Hope is not a strategy”

Another DevOps example: AWS

Another DevOps example: AWS

- unlike Google, AWS does **not** have dedicated ops teams

Another DevOps example: AWS

- unlike Google, AWS does **not** have dedicated ops teams
- all development teams are **solely responsible** for the operations of their own services

Another DevOps example: AWS

- unlike Google, AWS does **not** have dedicated ops teams
- all development teams are **solely responsible** for the operations of their own services
  - teams are also small (“two-pizza”) and usually organized around a single microservice

Another DevOps example: AWS

- unlike Google, AWS does **not** have dedicated ops teams
- all development teams are **solely responsible** for the operations of their own services
  - teams are also small (“two-pizza”) and usually organized around a single microservice
- this setup is **leaner** (no need to staff SRE teams!)

Another DevOps example: AWS

- unlike Google, AWS does **not** have dedicated ops teams
- all development teams are **solely responsible** for the operations of their own services
  - teams are also small (“two-pizza”) and usually organized around a single microservice
- this setup is **leaner** (no need to staff SRE teams!)
  - but means teams must **choose** between delivering new features and reducing operational burden

Another DevOps example: AWS

- unlike Google, AWS does **not** have dedicated ops teams
- all development teams are **solely responsible** for the operations of their own services
  - teams are also small (“two-pizza”) and usually organized around a single microservice
- this setup is **leaner** (no need to staff SRE teams!)
  - but means teams must **choose** between delivering new features and reducing operational burden
    - makes technical debt riskier to take on (why?)

DevOps

Today's agenda:

- Operations, Toil, and the DevOps philosophy
- **Achieving reliability**
  - the service reliability hierarchy + SLAs/targets
  - monitoring and reliability testing
  - incident/emergency response
  - preventing problems before they occur
  - post-mortems + learning from failure

Achieving reliability

- DevOps teams usually have a goal: make their service **reliable**

Achieving reliability

- DevOps teams usually have a goal: make their service **reliable**
- a reliable service is:

Achieving reliability

- DevOps teams usually have a goal: make their service **reliable**
- a reliable service is:
  - **available** (i.e., when a client calls it, it responds)

Achieving reliability

- DevOps teams usually have a goal: make their service **reliable**
- a reliable service is:
  - **available** (i.e., when a client calls it, it responds)
  - **correct** (i.e., client requests get the right results)

Achieving reliability

- DevOps teams usually have a goal: make their service **reliable**
- a reliable service is:
  - **available** (i.e., when a client calls it, it responds)
  - **correct** (i.e., client requests get the right results)
- these two properties are related: an unavailable service **cannot** be correct

Achieving reliability

- DevOps teams usually have a goal: make their service **reliable**
- a reliable service is:
  - **available** (i.e., when a client calls it, it responds)
  - **correct** (i.e., client requests get the right results)
- these two properties are related: an unavailable service **cannot** be correct
  - so, availability is the first thing we need to worry about when trying to make a service reliable

Reliability: setting expectations

- To determine if your system is behaving reliably, you need **metrics** that approximate whether it does what your users expect

Reliability: setting expectations

- To determine if your system is behaving reliably, you need **metrics** that approximate whether it does what your users expect
  - **availability** is often a good metric to start with

Reliability: setting expectations

- To determine if your system is behaving reliably, you need **metrics** that approximate whether it does what your users expect
  - **availability** is often a good metric to start with
  - other metrics will depend on the **meaning** of “correct” in your service’s context. Possible metrics:

Reliability: setting expectations

- To determine if your system is behaving reliably, you need **metrics** that approximate whether it does what your users expect
  - **availability** is often a good metric to start with
  - other metrics will depend on the **meaning** of “correct” in your service’s context. Possible metrics:
    - **latency** (time it takes to serve client requests)

Reliability: setting expectations

- To determine if your system is behaving reliably, you need **metrics** that approximate whether it does what your users expect
  - **availability** is often a good metric to start with
  - other metrics will depend on the **meaning** of “correct” in your service’s context. Possible metrics:
    - **latency** (time it takes to serve client requests)
    - **throughput** (how many requests can you serve per hour)

Reliability: setting expectations

- To determine if your system is behaving reliably, you need **metrics** that approximate whether it does what your users expect
  - **availability** is often a good metric to start with
  - other metrics will depend on the **meaning** of “correct” in your service’s context. Possible metrics:
    - **latency** (time it takes to serve client requests)
    - **throughput** (how many requests can you serve per hour)
    - **durability** (how much of your data can you still retrieve after a fixed time has passed)

Reliability: setting expectations

For a given service, here is a playbook for defining reliability:

Reliability: setting expectations

For a given service, here is a playbook for defining reliability:

1. decide what your users care about (call these “**objectives**”)

Reliability: setting expectations

For a given service, here is a playbook for defining reliability:

1. decide what your users care about (call these “**objectives**”)
2. map those objectives to one or more **metrics**

Reliability: setting expectations

For a given service, here is a playbook for defining reliability:

1. decide what your users care about (call these "**objectives**")
2. map those objectives to one or more **metrics**
    a. it might not be possible to match each objective to easy-to-collect metrics. In that case, choose metrics that **approximate** the objective

Reliability: setting expectations

For a given service, here is a playbook for defining reliability:

1. decide what your users care about (call these "**objectives**")
2. map those objectives to one or more **metrics**
    a. it might not be possible to match each objective to easy-to-collect metrics. In that case, choose metrics that **approximate** the objective
3. define the levels of those metrics that your service **should meet**, in order to meet user expectations

Reliability: setting expectations

For a given service, here is a playbook for defining reliability:

1. decide what your users care about (call these "**objectives**")
2. map those objectives to one or more **metrics**
    a. it might not be possible to match each objective to easy-to-collect metrics. In that case, choose metrics that **approximate** the objective
3. define the levels of those metrics that your service **should meet**, in order to meet user expectations
    a. optionally, publish these as a **service level agreement** ("**SLA**")

Reliability: setting expectations

For a given service, here is a playbook for defining reliability:

1. decide what your users care about (call these "**objectives**")
2. map those objectives to one or more **metrics**
   a. it might not be possible to match each objective to easy-to-collect metrics. **approximate** the objecti
3. define the levels of those me in order to meet user expectations
   a. optionally, publish these as a **service level agreement** ("**SLA**")

Sometimes SLAs are written into contracts with your customers!

Aside: subtleties in metrics

Aside: subtleties in metrics

- For simplicity and usability, we often aggregate raw measurements. This needs to be done **carefully**.

Aside: subtleties in metrics

- For simplicity and usability, we often aggregate raw measurements. This needs to be done **carefully**.
- e.g., consider “the number of requests per second served”

Aside: subtleties in metrics

- For simplicity and usability, we often aggregate raw measurements. This needs to be done **carefully**.
- e.g., consider “the number of requests per second served”
  - even this apparently straightforward measurement **implicitly aggregates** data over the measurement window

Aside: subtleties in metrics

- For simplicity and usability, we often aggregate raw measurements. This needs to be done **carefully**.
- e.g., consider “the number of requests per second served”
  - even this apparently straightforward measurement **implicitly aggregates** data over the measurement window
- We need to consider questions like “Is the measurement obtained once a second, or by averaging requests over a minute?”

Aside: subtleties in metrics

- For simplicity and usability, we often aggregate raw measurements. This needs to be done **carefully**.
- e.g., consider “the number of requests per second served”
  - even this apparently straightforward measurement **implicitly aggregates** data over the measurement window
- We need to consider questions like “Is the measurement obtained once a second, or by averaging requests over a minute?”
  - The latter may **hide** much higher instantaneous request rates in bursts that last for only a few seconds

Aside: subtleties in metrics

- For simplicity and usability, y
  measurements. This needs t
- e.g., consider "the number o
  - even this apparently stra
    aggregates data over th
- We need to consider questi
  once a second, or by averagi
  - The latter may **hide** much higher instantaneous request rates in bursts that last for only a few seconds

E.g., consider two systems:

- system A serves 200 requests in every even-numbered second, and 0 requests in every odd-numbered second
- system B serves 100 requests every second

Aside: subtleties in metrics

- It is better to view metrics as **distributions** (as in statistics) rather than as averages
  - this avoids hiding details like the example on the last slide

Aside: subtleties in metrics

- It is better to view metrics as **distributions** (as in statistics) rather than as averages
  - this avoids hiding details like the example on the last slide

Aside: subtleties in metrics

- It is better to view metrics as **distributions** (as in statistics) rather than as averages
  - this avoids hiding details like the example on the last slide

purple is 50th % latency

Aside: subtleties in metrics

- It is better to view metrics as **distributions** (as in statistics) rather than as averages
  - this avoids hiding details like the example on the last slide

green is 85th % latency

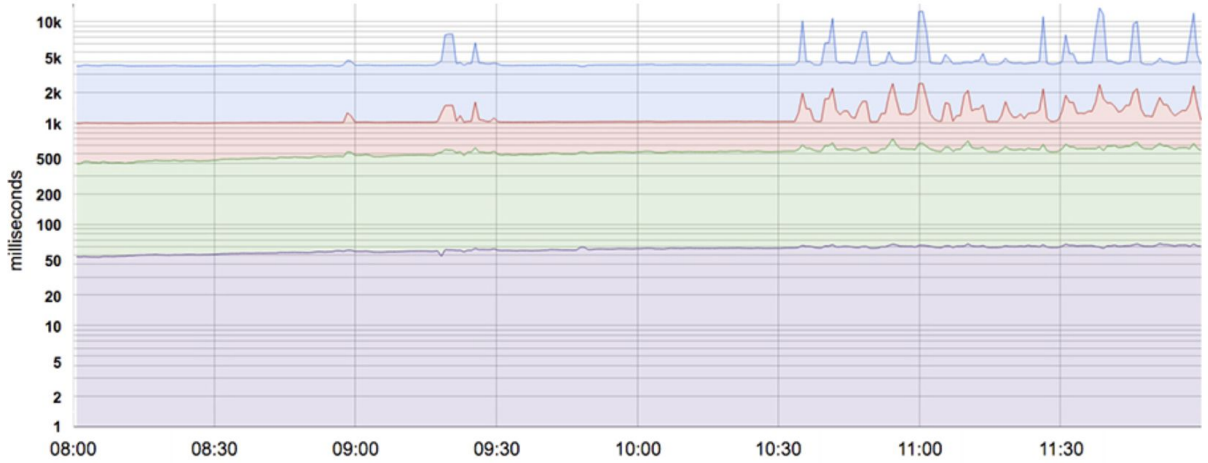
Aside: subtleties in metrics

- It is better to view metrics as **distributions** (as in statistics) rather than as averages
  - this avoids hiding details like the example on the last slide

red is 95th % latency

Aside: subtleties in metrics

- It is better to view metrics as **distributions** (as in statistics) rather than as averages
  - this avoids hiding details like the example on the last slide



blue is 99th % latency

Advice: choosing metrics

Advice: choosing metrics

- don’t pick target metrics based on **current system performance**
  - this just enshrines the status quo
  - instead, focus on what your users need

Advice: choosing metrics

- don’t pick target metrics based on **current system performance**
  - this just enshrines the status quo
  - instead, focus on what your users need
- keep it **simple**
  - SLAs, especially, should avoid mentioning complex aggregations of metrics (which are hard to reason about)

Advice: choosing metrics

- don't pick target metrics based on **current system performance**
  - this just enshrines the status quo
  - instead, focus on what your users need
- keep it **simple**
  - SLAs, especially, should avoid mentioning complex aggregations of metrics (which are hard to reason about)
- avoid **absolutes**
  - e.g., don't promise "infinite scaling" or "100% availability"

Advice: choosing metrics

- don't pick target metrics based on **current system performance**
  - this just enshrines the status quo
  - instead, focus on what your users need
- keep it **simple**
  - SLAs, especially, should avoid mentioning complex aggregations of metrics (which are hard to reason about)
- avoid **absolutes**
  - e.g., don't promise "infinite scaling" or "100% availability"
- include as **few metrics** as possible while still covering what matters
  - avoid metrics that aren't useful in arguing for priorities

Advice: choosing metrics

- don't pick target metrics based on **current system performance**
  - this just enshrines
  - instead, focus on
- keep it **simple**
  - SLAs, especially, s
    aggregations of m
- avoid **absolutes**
  - e.g., don't promise
- include as **few metrics** as possible while still covering what matters
  - avoid metrics that aren't useful in arguing for priorities

Finally, watch out for ***Goodhart's Law***:
"When a measure becomes a target, it ceases to be a good measure."

Advice: choosing metrics

- don't pick target metrics based on **current system performance**
  - this just enshrines
  - instead, focus on
- keep it **simple**
  - SLAs, especially, s
    aggregations of m
- avoid **absolutes**
  - e.g., don't promise
- include as **few metrics** as possible while still covering what matters
  - avoid metrics that aren't useful in arguing for priorities

Finally, watch out for ***Goodhart's Law***: "When a measure becomes a target, it ceases to be a good measure."

- regularly re-examine the set of metrics that you're using to make sure they actually correlate with qualitative "good" system health

Reliability: meeting expectations

- Once we have defined an SLA (internally or externally), how do we meet it?

Reliability: meeting expectations

- Once we have defined an SLA (internally or externally), how do we meet it?
  - Easy way to demonstrate that we're meeting an SLA: **collect the metrics** in the SLA!

Reliability: meeting expectations

- Once we have defined an SLA (internally or externally), how do we meet it?
  - Easy way to demonstrate that we're meeting an SLA: **collect the metrics** in the SLA!
  - Then, make sure that those metrics actually look good.

Reliability: meeting expectations

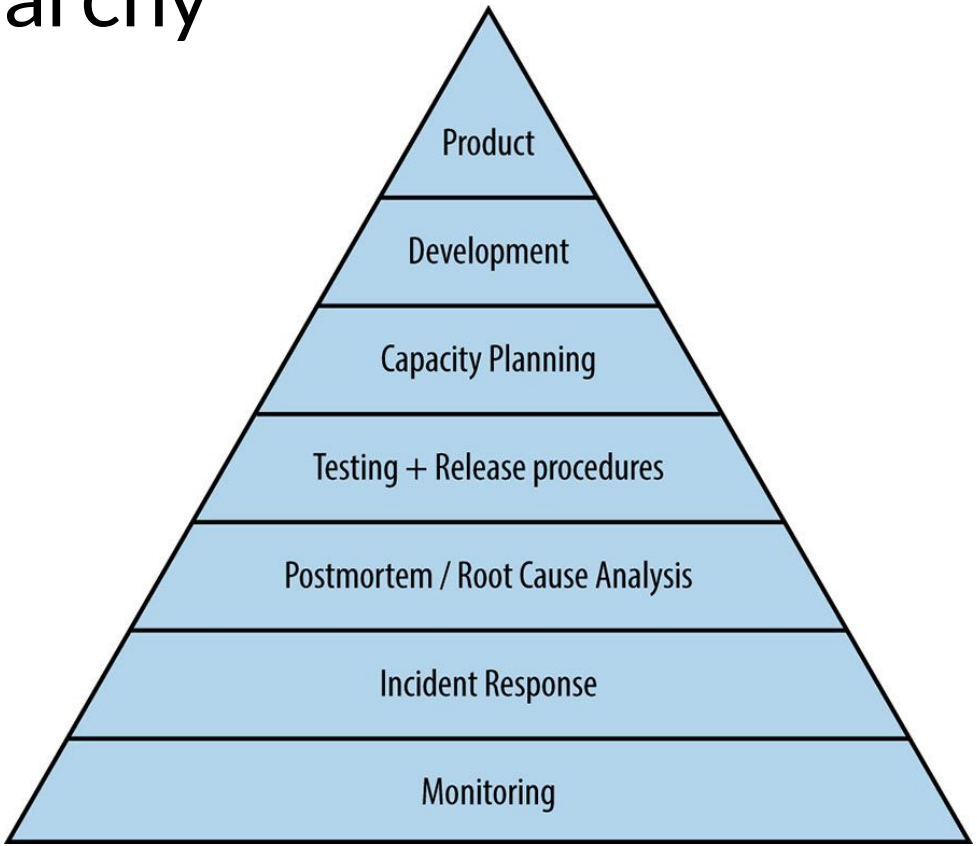
- Once we have defined an SLA (internally or externally), how do we meet it?
  - Easy way to demonstrate that we're meeting an SLA: **collect the metrics** in the SLA!
  - Then, make sure that those metrics actually look good.
- How do we think about how to do this?

Reliability: meeting expectations

- Once we have defined an SLA (internally or externally), how do we meet it?
  - Easy way to demonstrate that we're meeting an SLA: **collect the metrics** in the SLA!
  - Then, make sure that those metrics actually look good.
- How do we think about how to do this?
  - **insight**: there is a **hierarchy** of system components that need to be working well in order to meet an SLA

Service Reliability Hierarchy

- analogy to Maslow's "Hierarchy of Needs" for humans



[Image credit: https://sre.google/sre-book/part-III-practices/]

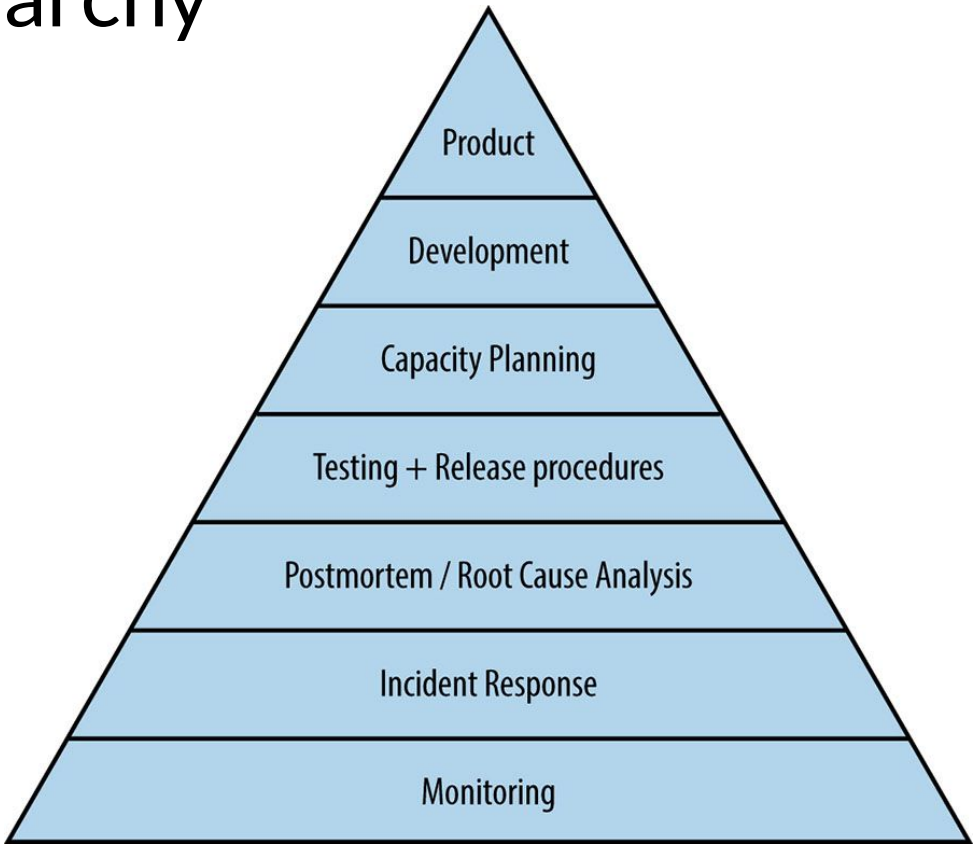
Maslow’s Hierarchy of Needs

Maslow's hierarchy of needs

[Image credit: https://www.thoughtco.com/maslows-hierarchy-of-needs-4582571]

Service Reliability Hierarchy

- analogy to Maslow's "Hierarchy of Needs" for humans
- just like in Maslow's hierarchy, if there is a serious deficiency in a lower level, achieving the higher level becomes a lot harder

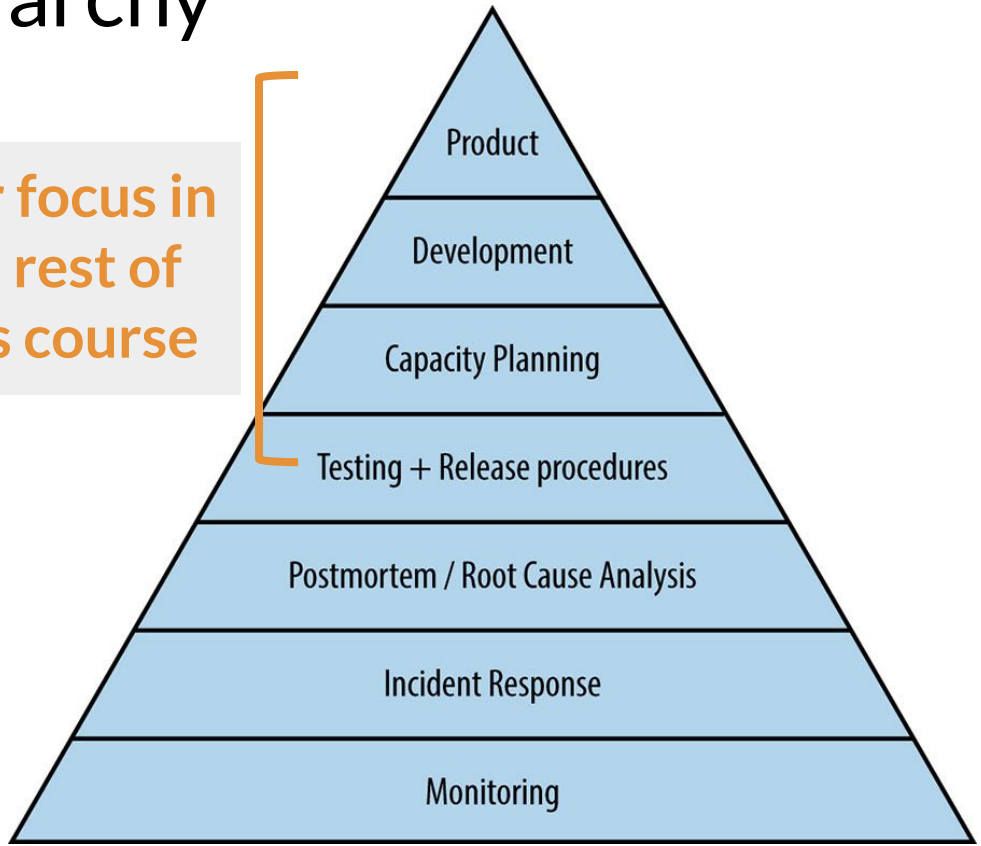


[Image credit: https://sre.google/sre-book/part-III-practices/]

Service Reliability Hierarchy

- analogy to Maslow's "Hierarchy of Needs" for humans
- just like in Maslow's hierarchy, if there is a serious deficiency in a lower level, achieving the higher level becomes a lot harder

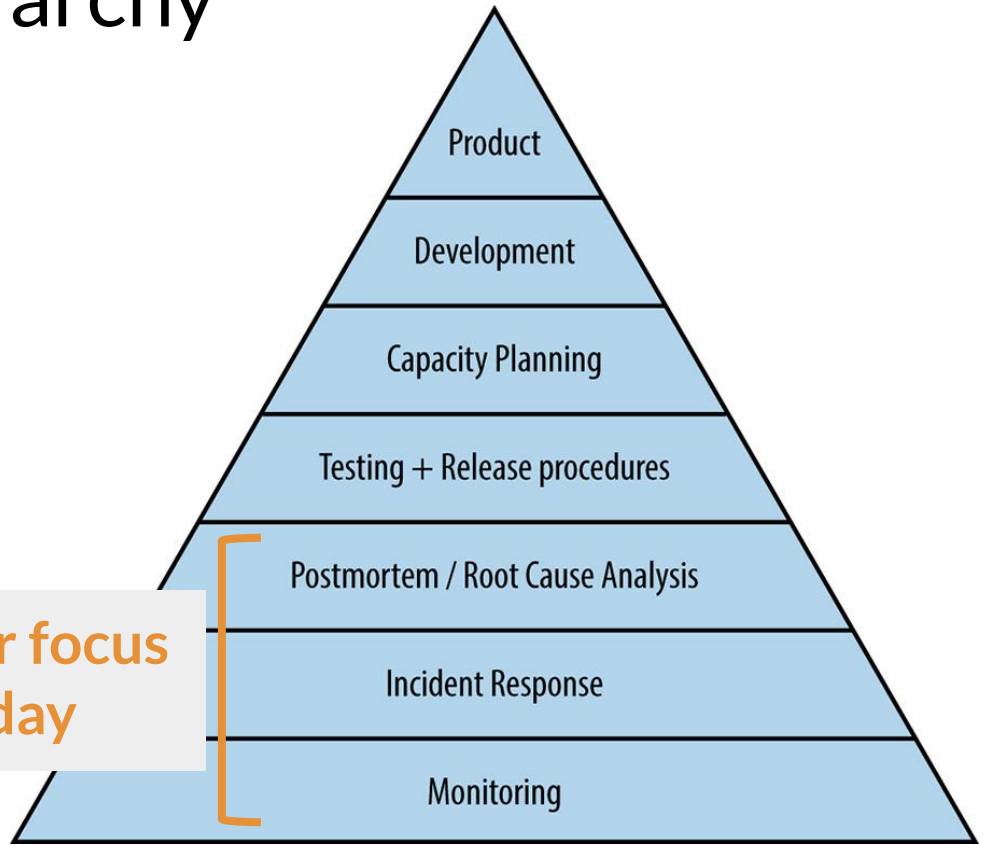
our focus in the rest of this course



[Image credit: https://sre.google/sre-book/part-III-practices/]

Service Reliability Hierarchy

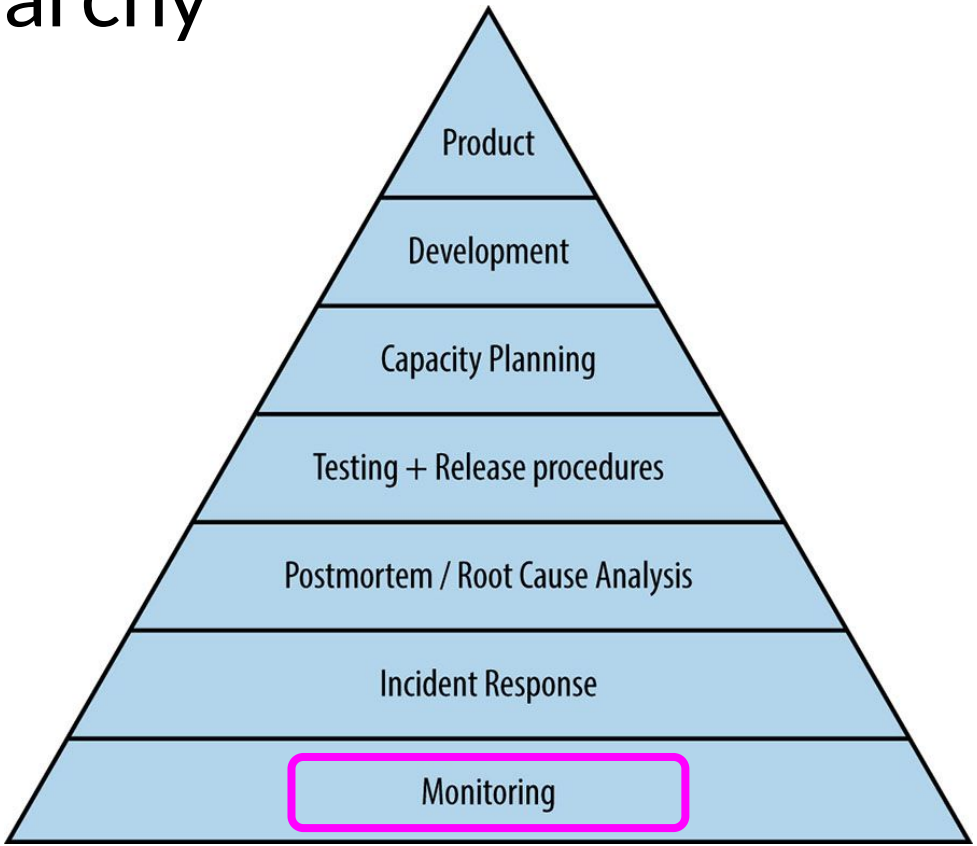
- analogy to Maslow's "Hierarchy of Needs" for humans
- just like in Maslow's hierarchy, if there is a serious deficiency in a lower level achieving the higher level becomes a lot harder



[Image credit: https://sre.google/sre-book/part-III-practices/]

Service Reliability Hierarchy

- analogy to Maslow's "Hierarchy of Needs" for humans
- just like in Maslow's hierarchy, if there is a serious deficiency in a lower level, achieving the higher level becomes a lot harder



[Image credit: https://sre.google/sre-book/part-III-practices/]

DevOps

Today's agenda:

- Operations, Toil, and the DevOps philosophy
- Ops challenge example: deployment
- Achieving reliability
  - the service reliability hierarchy + SLAs/targets
  - **monitoring**
  - incident/emergency response
  - post-mortems + learning from failure

Monitoring

Definition: ***monitoring*** is collecting, processing, aggregating, and displaying real-time quantitative data about a system, such as query counts and types, error counts and types, processing times, and server lifetimes

Monitoring

Definition: ***monitoring*** is collecting, processing, aggregating, and displaying real-time quantitative data about a system, such as query counts and types, error counts and types, processing times, and server lifetimes

- essentially, monitoring is responsible for collecting your metrics

Monitoring

Definition: ***monitoring*** is collecting, processing, aggregating, and displaying real-time quantitative data about a system, such as query counts and types, error counts and types, processing times, and server lifetimes

- essentially, monitoring is responsible for collecting your metrics
- without monitoring, you have no way to tell whether the service is **even working**

Monitoring

Definition: ***monitoring*** is collecting, processing, aggregating, and displaying real-time quantitative data about a system, such as query counts and types, error counts and types, processing times, and server lifetimes

- essentially, monitoring is responsible for collecting your metrics
- without monitoring, you have no way to tell whether the service is **even working**
- you want to be aware of problems **before** your users notice them

Monitoring

Definition: ***monitoring*** is colle
displaying real-time quantitat
counts and types, error count
lifetimes

Monitoring is why **logging** is so important in practice: if your monitoring depends on your logging framework, it is a very important component of your service!

- essentially, monitoring is responsible for collecting your metrics
- without monitoring, you have no way to tell whether the service is **even working**
- you want to be aware of problems **before** your users notice them

Monitoring: alerting

Monitoring: alerting

Definition: an ***alert*** is a notification intended to be read by a human and that is pushed to a system such as a bug or ticket queue, an email alias, or a pager

Monitoring: alerting

Definition: an ***alert*** is a notification intended to be read by a human and that is pushed to a system such as a bug or ticket queue, an email alias, or a pager

- **tickets** = alert to a bug or ticket queue, which a human will hopefully get to eventually

Monitoring: alerting

Definition: an ***alert*** is a notification intended to be read by a human and that is pushed to a system such as a bug or ticket queue, an email alias, or a pager

- **tickets** = alert to a bug or ticket queue, which a human will hopefully get to eventually
- **email alert** = alert sent to an email alias for a human to respond to during their next work day

Monitoring: alerting

Definition: an ***alert*** is a notification intended to be read by a human and that is pushed to a system such as a bug or ticket queue, an email alias, or a pager

- **tickets** = alert to a bug or ticket queue, which a human will hopefully get to eventually
- **email alert** = alert sent to an email alias for a human to respond to during their next work day
- **page** = alert send directly to a human (via a pager)

Monitoring: being on-call

- A major part of modern DevOps is being “**on-call**”

Monitoring: being on-call

- A major part of modern DevOps is being “**on-call**”
- When you are the on-call for a service, any pages about that service go to you

Monitoring: being on-call

- A major part of modern DevOps is being “**on-call**”
- When you are the on-call for a service, any pages about that service go to you
  - even in the middle of the night!

Monitoring: being on-call

- A major part of modern DevOps is being “**on-call**”
- When you are the on-call for a service, any pages about that service go to you
  - even in the middle of the night!
- Getting paged should be an event

Monitoring: being on-call

- A major part of modern DevOps is being “**on-call**”
- When you are the on-call for a service, any pages about that service go to you
  - even in the middle of the night!
- Getting paged should be an event
  - ideally, pages correspond 1:1 with **emergencies**

Monitoring: being on-call

- A major part of modern DevOps is being "**on-call**"
- When you are the on-call for a service, any pages about that service go to you
  - even in the middle of the night!
- Getting paged should be an event
  - ideally, pages correspond 1:1 with **emergencies**
    - (less ideal but still good: you get paged if and only if there is an emergency)

Monitoring: being on-call

- A major part of modern DevOps is being "**on-call**"
- When you are the on-call for a service, any pages about that service go to you
  - even in the middle of the night!
- Getting paged should be an event
  - ideally, pages correspond 1:1 with **emergencies**
    - (less ideal but still good: you get paged if and only if there is an emergency)
- Example from earlier: "cleaning up a service's alerting config" = fixing **what corresponds** to pages vs email alerts vs tickets

Monitoring: being on-call

- Being on-call is a **major source of toil** in most services

Monitoring: being on-call

- Being on-call is a **major source of toil** in most services
  - a page about a non-emergency is one of the worst forms of toil, because it **forces** you to react

Monitoring: being on-call

- Being on-call is a **major source of toil** in most services
  - a page about a non-emergency is one of the worst forms of toil, because it **forces** you to react
- For this reason, most teams **rotate** who is on-call

Monitoring: being on-call

- Being on-call is a **major source of toil** in most services
  - a page about a non-emergency is one of the worst forms of toil, because it **forces** you to react
- For this reason, most teams **rotate** who is on-call
  - e.g., daily, weekly, whatever
  - **everyone** working on the service should be in this rotation!

Monitoring: being on-call

- Being on-call is a **major source of toil** in most services
  - a page about a non-emergency is one of the worst forms of toil, because it **forces** you to react
- For this reason, most teams **rotate** who is on-call
  - e.g., daily, weekly, whatever
  - **everyone** working on the service should be in this rotation!
- The person on-call typically assumes all **operational burden** (i.e., toil) for the service for the duration of their on-call shift

Monitoring: being on-call

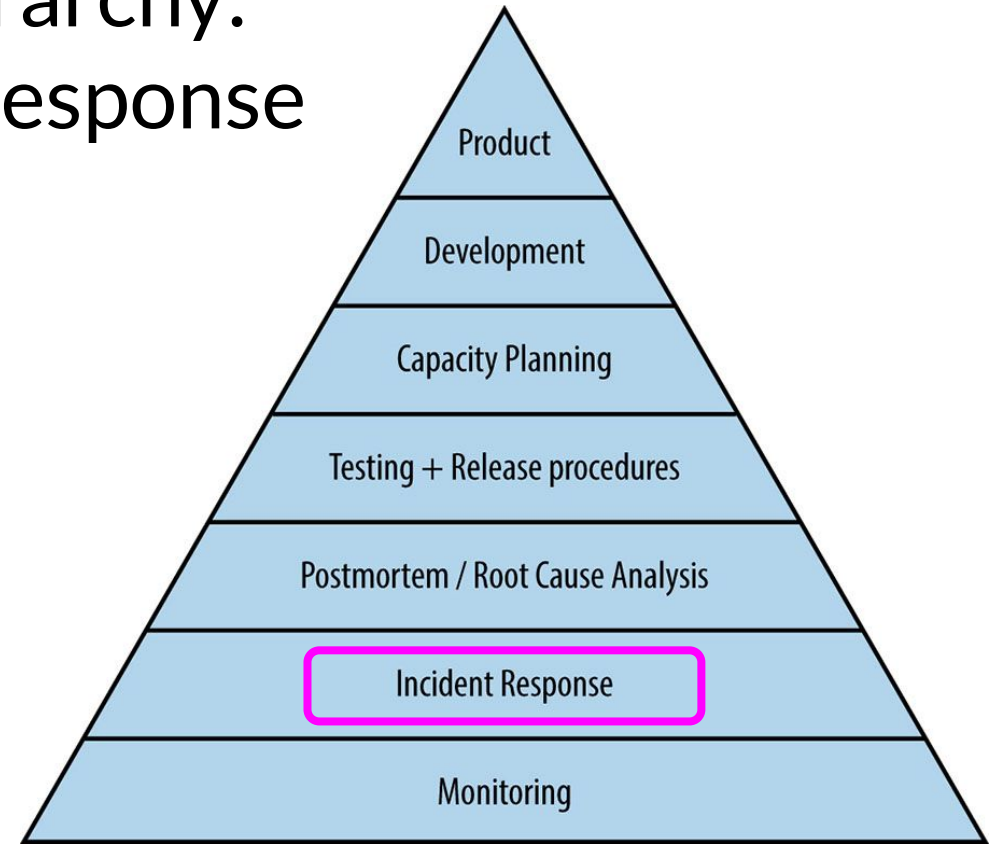
- Being on-call is a **major source of toil** in most services
  - a page about a non-emergency is one of the worst forms of toil, because it **forces** you to react
- For this reason, most teams **rotate** who is on-call
  - e.g., daily, weekly, whatever
  - **everyone** working on the service should be in this rotation!
- The person on-call typically assumes all **operational burden** (i.e., toil) for the service for the duration of their on-call shift
  - but can (**and should**) page other team members in an emergency

DevOps

Today's agenda:

- Operations, Toil, and the DevOps philosophy
- Achieving reliability
  - the service reliability hierarchy + SLAs/targets
  - monitoring and reliability testing
  - **incident/emergency response**
  - preventing problems before they occur
  - post-mortems + learning from failure

Service Reliability Hierarchy: Incident/Emergency Response



[Image credit: https://sre.google/sre-book/part-III-practices/]

Emergency Response

- So you’re the on-call, and you get a page. What happens next?

Emergency Response

- So you’re the on-call, and you get a page. What happens next?
  - “**emergency response**”

Emergency Response

- So you're the on-call, and you get a page. What happens next?
  - "**emergency response**"
  - as the on-call, **you are in charge** in an emergency by default

Emergency Response

- So you're the on-call, and you get a page. What happens next?
  - "**emergency response**"
  - as the on-call, **you are in charge** in an emergency by default
- What constitutes an emergency?

Emergency Response

- So you're the on-call, and you get a page. What happens next?
  - "**emergency response**"
  - as the on-call, **you are in charge** in an emergency by default
- What constitutes an emergency?
  - depends on your service, but typically these qualify:
    - big % of user requests aren't getting responses
    - big % of user requests have really high latency
    - lots of your servers are unavailable/down (even if users aren't yet impacted)

Emergency Response: causes of emergencies

Emergency Response: causes of emergencies

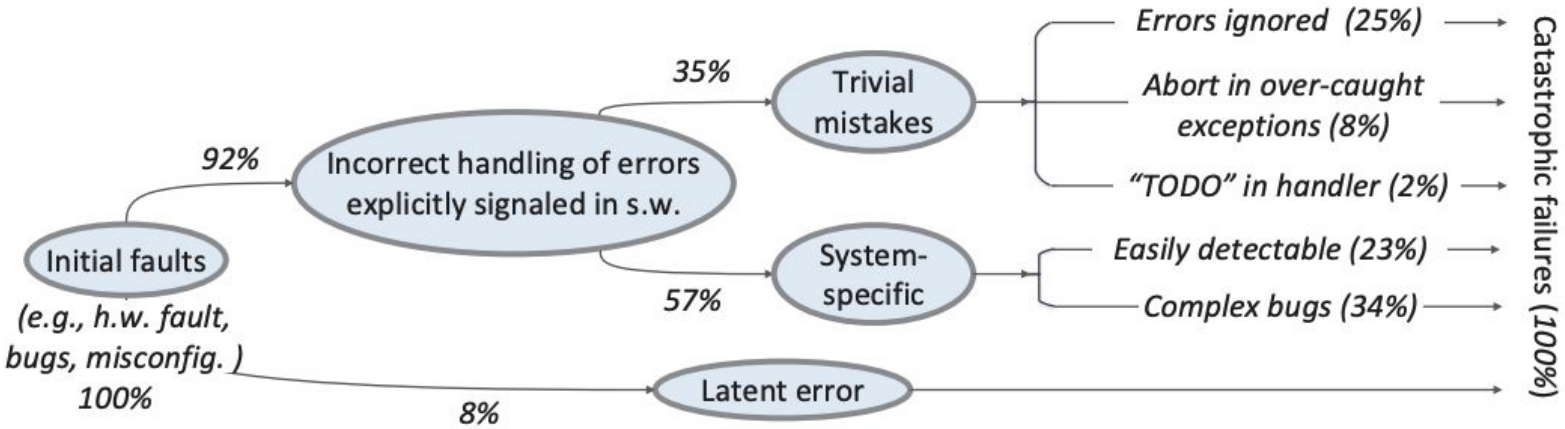
- **error handling**: code that is only called when something is wrong
  - why is this likely to cause an emergency?

Emergency Response: causes of emergencies

- **error handling**: code that is only called when something is wrong
  - why is this likely to cause an emergency?
    - less likely to have tests for failure cases!

Emergency Response: causes of emergencies

- **error handling**: code that is only called when something is wrong
  - why is this likely to cause an emergency?
    - less likely to have tests for failure cases!

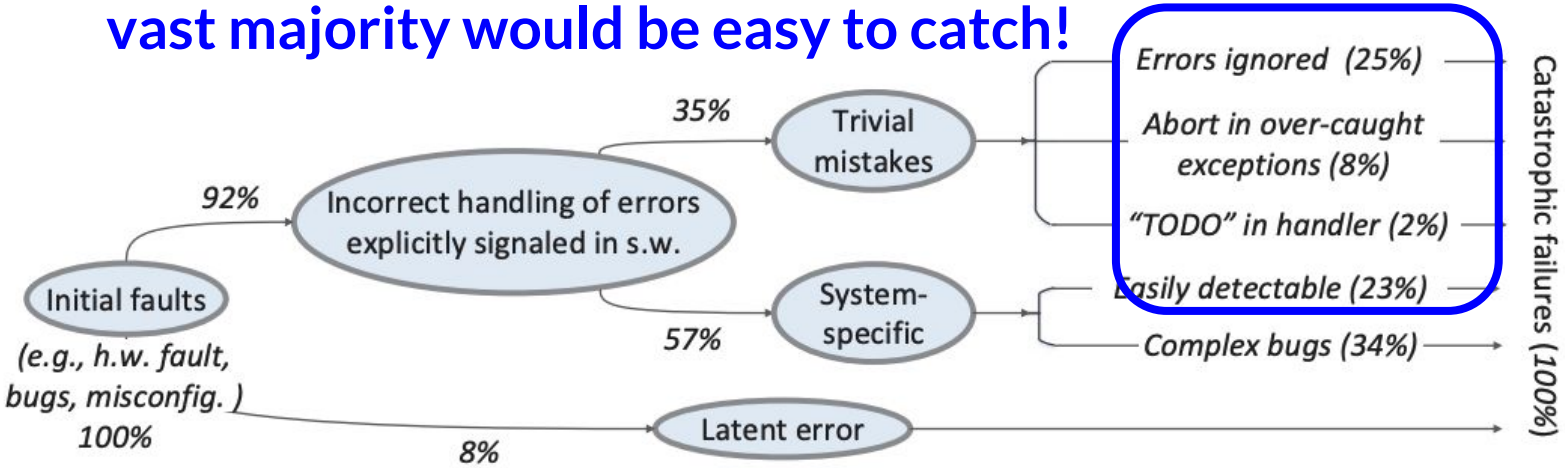


[*Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems.* Yuan et al. OSDI 2014.]

Emergency Response: causes of emergencies

- **error handling**: code that is only called when something is wrong
  - why is this likely to cause an emergency?
    - less likely to have tests for failure cases!

vast majority would be easy to catch!



[*Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems.* Yuan et al. OSDI 2014.]

Emergency Response: causes of emergencies

- **configuration changes**:
  - especially for services, how the servers that run the system are configured is often as important as the code itself

Emergency Response: causes of emergencies

- **configuration changes**:
  - especially for services, how the servers that run the system are configured is often as important as the code itself
  - changes to the infrastructure (e.g., adding or removing servers) are just as risky as changes to the code
    - but testing them is harder!

Emergency Response: causes of emergencies

- **hardware**:
  - pop quiz: how long does an average hard disk last?

Emergency Response: causes of emergencies

- **hardware**:
  - pop quiz: how long does an average hard disk last?
    - answer: 3-5 years

Emergency Response: causes of emergencies

- **hardware**:
  - pop quiz: how long does an average hard disk last?
    - answer: 3-5 years
  - **law of large numbers**: suppose you have 10,000 hard disks. What are the odds that one of them fails today (assuming each has a 5 year average lifespan?)
    - get out a piece of paper and do the math

Emergency Response: causes of emergencies

- **hardware**:
  - pop quiz: how long does an average hard disk last?
    - answer: 3-5 years
  - **law of large numbers**: suppose you have 10,000 hard disks. What are the odds that one of them fails today (assuming each has a 5 year average lifespan?)
    - get out a piece of paper and do the math
  - **almost 100%!**
    - each disk lasts 365*5 = 1825 days. 10k disks = **~5 fail/day**

Emergency Response: causes of emergencies

- **hardware**:
  - pop quiz: how long does an average hard disk last?
    - answer: 3-5 years
  - **law of large numbers**: suppose you have 10,000 hard disks. What are the odds that[illegible]ch has a 5 year average lif[illegible]
    - get out a piece of p[illegible]
  - **almost 100%!**
    - each disk lasts 365*5 = 1825 days. 10k disks = **~5 fail/day**

Implication: in large systems, you **must plan for hardware failures**, because they **will occur**

Emergency Response: causes of emergencies

- **human/process error**:
  - pop quiz: as a human, have you ever made a mistake at something you're usually good at?

Emergency Response: causes of emergencies

- **human/process error**:
  - pop quiz: as a human, have you ever made a mistake at something you're usually good at?
    - of course you have! we all make mistakes sometimes!

Emergency Response: causes of emergencies

- **human/process error**:
  - pop quiz: as a human, have you ever made a mistake at something you're usually good at?
    - of course you have! we all make mistakes sometimes!
  - it is a mistake for a human to repeatedly perform a task that could lead to catastrophic failure if it is not done perfectly

Emergency Response: causes of emergencies

- **human/process error**:
  - pop quiz: as a human, have you ever made a mistake at something you're usually good at?
    - of course you have! we all make mistakes sometimes!
  - it is a mistake for a human to repeatedly perform a task that could lead to catastrophic failure if it is not done perfectly
    - computers are good at this!
    - analogy: just like hardware components sometimes fail, any step carried out by humans should be assumed to have a non-zero failure rate

Emergency Response: have a plan

Emergency Response: have a plan

- An **unmanaged** emergency occurs when the team hasn't put a plan in place beforehand about what to do in that situation

Emergency Response: have a plan

- An **unmanaged** emergency occurs when the team hasn't put a plan in place beforehand about what to do in that situation
  - unmanaged emergencies are typically hard to recover from

Emergency Response: have a plan

- An **unmanaged** emergency occurs when the team hasn't put a plan in place beforehand about what to do in that situation
  - unmanaged emergencies are typically hard to recover from
  - "plans are useless, but planning is indispensable"

Emergency Response: have a plan

- An **unmanaged** emergency occurs when the team hasn't put a plan in place beforehand about what to do in that situation
  - unmanaged emergencies are typically hard to recover from
  - "plans are useless, but planning is indispensable"
- **Best practice**: teams should have ***playbooks*** (or ***runbooks***) that list the steps to take in an emergency

Emergency Response: have a plan

- An **unmanaged** emergency occurs when the team hasn't put a plan in place beforehand about what to do in that situation
  - unmanaged emergencies are typically hard to recover from
  - "plans are useless, but planning is indispensable"
- **Best practice**: teams should have ***playbooks*** (or ***runbooks***) that list the steps to take in an emergency
  - playbooks are built up over a service's lifetime (i.e., they record how previous incidents might have been avoided or mitigated)

Emergency Response: have a plan

- An **unmanaged** emergency occurs when the team hasn't put a plan in place beforehand about what to do in that situation
  - unmanaged emergencies are typically hard to recover from
  - "plans are useless, but planning is indispensable"
- **Best practice**: teams should have ***playbooks*** (or ***runbooks***) that list the steps to take in an emergency
  - playbooks are built up over a service's lifetime (i.e., they record how previous incidents might have been avoided or mitigated)
  - often, playbooks have specific guidance for particular alerts

Emergency Response: have a plan

- An **unmanaged** emergency occurs when the team hasn't put a plan in place beforehand about what to do in that situation
  - unmanaged emergencies are typically hard to recover from
  - "plans are useless, but planning is indispensable"
- **Best practice**: teams should have ***playbooks*** (or ***runbooks***) that list the steps to take in an emergency
  - playbooks are built up over a service's lifetime (i.e., they record how previous incidents might have been avoided or mitigated)
  - often, playbooks have specific guidance for particular alerts
  - playbooks also have a psychological function: **prevent panic**

Emergency Response: best practices

Emergency Response: best practices

- Know your priorities:

Emergency Response: best practices

- Know your priorities:
  - **damage control**: take proactive steps to prevent the incident from becoming worse (e.g., remove unnecessary traffic)

Emergency Response: best practices

- Know your priorities:
  - **damage control**: take proactive steps to prevent the incident from becoming worse (e.g., remove unnecessary traffic)
  - **restore service**: get the service back to a healthy state, even if you aren't sure about the cause (e.g., by rolling back recent changes)

Emergency Response: best practices

- Know your priorities:
  - **damage control**: take proactive steps to prevent the incident from becoming worse (e.g., remove unnecessary traffic)
  - **restore service**: get the service back to a healthy state, even if you aren't sure about the cause (e.g., by rolling back recent changes)
  - **preserve evidence**: save logs, etc., for post-mortem analysis

Emergency Response: best practices

- Know your priorities:
  - **damage control**: take proactive steps to prevent the incident from becoming worse (e.g., remove unnecessary traffic)
  - **restore service**: get the service back to a healthy state, even if you aren't sure about the cause (e.g., by rolling back recent changes)
  - **preserve evidence**: save logs, etc., for post-mortem analysis
- **Practice** makes perfect
  - don't wait for an actual emergency to find out if your playbook works: simulate one instead!

Emergency Response: rolling back

- One of the most important techniques in emergency response is **rolling back** to the last known working state

Emergency Response: rolling back

- One of the most important techniques in emergency response is **rolling back** to the last known working state
  - key idea: most emergencies are caused by some **change**

Emergency Response: rolling back

- One of the most important techniques in emergency response is **rolling back** to the last known working state
  - key idea: most emergencies are caused by some **change**
  - so, to fix the incident, we should **undo** the change

Emergency Response: rolling back

- One of the most important techniques in emergency response is **rolling back** to the last known working state
  - key idea: most emergencies are caused by some **change**
  - so, to fix the incident, we should **undo** the change
- The need to roll back has important implications:

Emergency Response: rolling back

- One of the most important techniques in emergency response is **rolling back** to the last known working state
  - key idea: most emergencies are caused by some **change**
  - so, to fix the incident, we should **undo** the change
- The need to roll back has important implications:
  - avoid changes that **cannot be undone** (“two-way doors”)

Emergency Response: rolling back

- One of the most important techniques in emergency response is **rolling back** to the last known working state
  - key idea: most emergencies are caused by some **change**
  - so, to fix the incident, we should **undo** the change
- The need to roll back has important implications:
  - avoid changes that **cannot be undone** (“two-way doors”)
  - your version control system is your friend here!

Emergency Response: rolling back

- One of the most important techniques in emergency response is **rolling back** to the last known working state
  - key idea: most emergencies are caused by some **change**
  - so, to fix the incident, we should **undo** the change
- The need to roll back has important implications:
  - avoid changes that **cannot be undone** (“two-way doors”)
  - your version control system is your friend here!
    - make sure to commit things that might cause incidents if they change to version control, e.g., your **config files**

Emergency Response: rolling back

- One of the most importan
  rolling back to the last kn
  - key idea: most emerge
  - so, to fix the incident,
- The need to roll back has important implications:
  - avoid changes that **cannot be undone** (“two-way doors”)
  - your version control system is your friend here!
    - make sure to commit things that might cause incidents if they change to version control, e.g., your **config files**

Easy rollbacks are one motivation for “**infrastructure-as-code**”: if your infrastructure configuration is in version control, it’s easy to go back to the last working one!

DevOps

Today's agenda:

- Operations, Toil, and the DevOps philosophy
- Achieving reliability
  - the service reliability hierarchy + SLAs/targets
  - monitoring and reliability testing
  - incident/emergency response
  - **preventing problems before they occur**
  - post-mortems + learning from failure

Preventing Problems

- while it’s important to have a plan for responding to emergencies, it’s better if they **never happen at all**

Preventing Problems

- while it's important to have a plan for responding to emergencies, it's better if they **never happen at all**
  - we can use many of the techniques that we discussed in this class to help prevent emergencies!

Preventing Problems

- while it's important to have a plan for responding to emergencies, it's better if they **never happen at all**
  - we can use many of the techniques that we discussed in this class to help prevent emergencies!
- however, there are some **DevOps-specific** testing and deployment strategies that can help:

Preventing Problems

- while it's important to have a plan for responding to emergencies, it's better if they **never happen at all**
  - we can use many of the techniques that we discussed in this class to help prevent emergencies!
- however, there are some **DevOps-specific** testing and deployment strategies that can help:
  - integrating testing and monitoring
  - stress testing services
  - canaries and "baking the binary"

Integrating Testing and Monitoring

- We can view monitoring as a form of **black-box testing**

Integrating Testing and Monitoring

- We can view monitoring as a form of **black-box testing**
  - that is, our monitoring systems are constantly “testing” the real, production system!

Integrating Testing and Monitoring

- We can view monitoring as a form of **black-box testing**
  - that is, our monitoring systems are constantly “testing” the real, production system!
- If we view our monitoring system this way, we can **apply many testing techniques** to monitoring

Integrating Testing and Monitoring

- We can view monitoring as a form of **black-box testing**
  - that is, our monitoring systems are constantly “testing” the real, production system!
- If we view our monitoring system this way, we can **apply many testing techniques** to monitoring
  - for example, should there be a **relationship** between a pair of metrics that we’re collecting? (= ***metamorphic testing***)

Integrating Testing and Monitoring

- We can view monitoring as a form of **black-box testing**
  - that is, our monitoring systems are constantly “testing” the real, production system!
- If we view our monitoring system this way, we can **apply many testing techniques** to monitoring
  - for example, should there be a **relationship** between a pair of metrics that we’re collecting? (= ***metamorphic testing***)
    - if so, we can define an alert that goes off if that relationship is ever violated

Stress Testing

Stress Testing

Definition: a ***stress test*** is any test designed to find the limits of the external conditions under which a service can safely operate

Stress Testing

Definition: a ***stress test*** is any test designed to find the limits of the external conditions under which a service can safely operate

- Stress tests answer questions like:
  - “How full can a database get before writes start to fail?”

Stress Testing

Definition: a ***stress test*** is any test designed to find the limits of the external conditions under which a service can safely operate

- Stress tests answer questions like:
  - “How full can a database get before writes start to fail?”
  - “How many queries a second can be sent to an application server before it becomes overloaded, causing requests to fail?”

Stress Testing

Definition: a ***stress test*** is any test designed to find the limits of the external conditions under which a service can safely operate

- Stress tests answer questions like:
  - “How full can a database get before writes start to fail?”
  - “How many queries a second can be sent to an application server before it becomes overloaded, causing requests to fail?”
- **Chaos Monkey** is one example of a stress testing technique
  - Others include intentionally **scaling up** another service
    - i.e., simulate a spike in demand with artificial traffic

Stress Testing: Chaos Monkey

- ***Chaos Monkey*** was invented in 2011 by Netflix to test the resilience of its IT infrastructure

Stress Testing: Chaos Monkey

- ***Chaos Monkey*** was invented in 2011 by Netflix to test the resilience of its IT infrastructure
- *“Imagine a monkey entering a "data center", these "farms" of servers that host all the critical functions of our online activities. The monkey **randomly** rips cables, destroys devices and returns everything that passes by the hand.*

– Antonio Martinez, Chaos Monkey

Stress Testing: Chaos Monkey

- ***Chaos Monkey*** was invented in 2011 by Netflix to test the resilience of its IT infrastructure
- *“Imagine a monkey entering a "data center", these "farms" of servers that host all the critical functions of our online activities. The monkey **randomly** rips cables, destroys devices and returns everything that passes by the hand. The challenge for IT managers is to **design the** information **system** they are responsible for **so that it can work despite these monkeys**, which no one ever knows when they arrive and what they will destroy.”*

  – Antonio Martinez, Chaos Monkey

Stress Testing: Chaos Monkey

- *“We have created Chaos Monkey, a program that randomly chooses a server and disables it during its usual hours of activity. Some will find that crazy, but we could not depend on the random occurrence of an event to test our behavior in the face of the very consequences of this event.*

  - Greg Orzell, Netflix Chaos Monkey Upgraded

Stress Testing: Chaos Monkey

- *“We have created Chaos Monkey, a program that randomly chooses a server and disables it during its usual hours of activity. Some will find that crazy, but we could not depend on the random occurrence of an event to test our behavior in the face of the very consequences of this event.* ***Knowing that this would happen frequently has created a strong alignment among engineers to build redundancy and process automation to survive such incidents****, without impacting the millions of Netflix users.*

  - Greg Orzell, Netflix Chaos Monkey Upgraded

Stress Testing: Chaos Monkey

- *“We have created Chaos Monkey, a program that randomly chooses a server and disables it during its usual hours of activity. Some will find that crazy, but we could not depend on the random occurrence of an event to test our behavior in the face of the very consequences of this event.* ***Knowing that this would happen frequently has created a strong alignment among engineers to build redundancy and process automation to survive such incidents****, without impacting the millions of Netflix users.* ***Chaos Monkey is one of our most effective tools to improve the quality of our services.****”*
  - Greg Orzell, Netflix Chaos Monkey Upgraded

Aside: cascading failures

- A common cause of failures in a microservice-based system is ***cascading failures***: one service fails (for any reason), which causes other services that depend on it to fail, which causes other services to fail, etc.

Aside: cascading failures

- A common cause of failures in a microservice-based system is ***cascading failures***: one service fails (for any reason), which causes other services that depend on it to fail, which causes other services to fail, etc.
  - cascading failures are typically much harder to recover from
    - many parts of the system have failed, not just one!

Aside: cascading failures

- A common cause of failures in a microservice-based system is ***cascading failures***: one service fails (for any reason), which causes other services that depend on it to fail, which causes other services to fail, etc.
  - cascading failures are typically much harder to recover from
    - many parts of the system have failed, not just one!
  - one of the goals of Chaos Monkey is to detect such cascading failures before they actually happen in production

Canaries and Staged Deployments

- Another important consideration is limiting the ***blast radius*** of a failure, if one does occur

Canaries and Staged Deployments

- Another important consideration is limiting the ***blast radius*** of a failure, if one does occur
  - the blast radius is how many users/requests are impacted

Canaries and Staged Deployments

- Another important consideration is limiting the ***blast radius*** of a failure, if one does occur
  - the blast radius is how many users/requests are impacted
- An important technique for limiting blast radius is ***staged deployment***, which is also sometimes called ***canary testing***

Canaries and Staged Deployments

- Another important consideration is limiting the ***blast radius*** of a failure, if one does occur
  - the blast radius is how many users/requests are impacted
- An important technique for limiting blast radius is ***staged deployment***, which is also sometimes called ***canary testing***
  - in a staged deployment of a change, at first **only a small percentage** of the active fleet is modified

Canaries and Staged Deployments

- Another important consideration is limiting the ***blast radius*** of a failure, if one does occur
  - the blast radius is how many users/requests are impacted
- An important technique for limiting blast radius is ***staged deployment***, which is also sometimes called ***canary testing***
  - in a staged deployment of a change, at first **only a small percentage** of the active fleet is modified
    - this part of the fleet is monitored for failures, and if none occur then **more and more** of the fleet is updated

Canaries and Staged Deployments

- Another important consideration is limiting the ***blast radius*** of a failure, if one does occur
  - the blast radius is how many users/requests are impacted
- An important technique for limiting blast radius is ***staged deployment***, which is als
  - in a staged deploym
    percentage of the ac
    - this part of the fleet is monitored for failures, and if none occur then **more and more** of the fleet is updated

This incubation period while the fleet is partially upgraded is sometimes called "***baking the binary***".

Staged Deployment: concrete example

Staged Deployment: concrete example

- Consider a given underlying fault that:

Staged Deployment: concrete example

- Consider a given underlying fault that:
  - relatively rarely impacts user traffic

Staged Deployment: concrete example

- Consider a given underlying fault that:
  - relatively rarely impacts user traffic
  - is deployed via a staged upgrade rollout that is **exponential**

Staged Deployment: concrete example

- Consider a given underlying fault that:
  - relatively rarely impacts user traffic
  - is deployed via a staged upgrade rollout that is **exponential**
- We would expect a growing cumulative number of reported variances, governed by the equation $CU = RK$, where:

Staged Deployment: concrete example

- Consider a given underlying fault that:
  - relatively rarely impacts user traffic
  - is deployed via a staged upgrade rollout that is **exponential**
- We would expect a growing cumulative number of reported variances, governed by the equation ***CU*** **=** ***RK***, where:
  - ***C*** = cumulative number of reports

Staged Deployment: concrete example

- Consider a given underlying fault that:
  - relatively rarely impacts user traffic
  - is deployed via a staged upgrade rollout that is **exponential**
- We would expect a growing cumulative number of reported variances, governed by the equation $CU = RK$, where:
  - C = cumulative number of reports
  - U = order of the fault (see next slide)

Staged Deployment: concrete example

- Consider a given underlying fault that:
  - relatively rarely impacts user traffic
  - is deployed via a staged upgrade rollout that is **exponential**
- We would expect a growing cumulative number of reported variances, governed by the equation $CU = RK$, where:
  - C = cumulative number of reports
  - U = order of the fault (see next slide)
  - R = the rate of reports

Staged Deployment: concrete example

- Consider a given underlying fault that:
  - relatively rarely impacts user traffic
  - is deployed via a staged upgrade rollout that is **exponential**
- We would expect a growing cumulative number of reported variances, governed by the equation $CU = RK$, where:
  - C = cumulative number of reports
  - U = order of the fault (see next slide)
  - R = the rate of reports
  - K = the period over which the traffic grows by a factor of e

Staged Deployment: concrete example

- Consider a given underlying fault that:

Note that *C*, *R*, and *K* should all be **measurable** by your monitoring system.

but that is **exponential**

- We would expect a growing cumulative number of reported variances, governed by the equation ***CU* = *RK***, where:
  - *C* = cumulative number of reports
  - *U* = order of the fault (see next slide)
  - *R* = the rate of reports
  - *K* = the period over which the traffic grows by a factor of *e*

Staged Deployment: concrete example

- Ideally, we have automated monitoring and rollback that will contain this bug. Let’s assume that’s the case.

$CU = RK$

Staged Deployment: concrete example

- Ideally, we have automated monitoring and rollback that will contain this bug. Let's assume that's the case.
  - our monitoring can tell us C and R, and we should already know K (because we chose the deployment rate)

$CU = RK$

Staged Deployment: concrete example

- Ideally, we have automated monitoring and rollback that will contain this bug. Let's assume that's the case.
  - our monitoring can tell us C and R, and we should already know K (because we chose the deployment rate)
- from these, we can compute U, the ***order of the fault***:

$CU = RK$

Staged Deployment: concrete example

- Ideally, we have automated monitoring and rollback that will contain this bug. Let's assume that's the case.
  - our monitoring can tell us *C* and *R*, and we should already know *K* (because we chose the deployment rate)
- from these, we can compute *U*, the *order of the fault*:
  - U=1: each request encountered code that is simply broken

$CU = RK$

Staged Deployment: concrete example

- Ideally, we have automated monitoring and rollback that will contain this bug. Let's assume that's the case.
  - our monitoring can tell us C and R, and we should already know K (because we chose the deployment rate)
- from these, we can compute U, the *order of the fault*:
  - U=1: each request encountered code that is simply broken
  - U=2: each request randomly damages data that a future request may see.

$CU = RK$

Staged Deployment: concrete example

- Ideally, we have automated monitoring and rollback that will contain this bug. Let's assume that's the case.
  - our monitoring can tell us *C* and *R*, and we should already know *K* (because we chose the deployment rate)
- from these, we can compute *U*, the *order of the fault*:
  - U=1: each request encountered code that is simply broken
  - U=2: each request randomly damages data that a future request may see.
  - U=3: the randomly damaged data is also a valid identifier to a previous request.

CU = RK

Observe that ***order*** here is like big-O notation:

- U=1 means that only the request itself is impacted
- U=2 means that a linear-ish number of other requests will be impacted
- U=3 means exponentially more requests will be impacted
- etc.

know

- from these, we can compute ***U***, the ***order of the fault***:
  - U=1: each request encountered code that is simply broken
  - U=2: each request randomly damages data that a future request may see.
  - U=3: the randomly damaged data is also a valid identifier to a previous request.

CU = *RK*

Staged Deployment: concrete example

- Once we have an estimate for ***U***, we have a better idea of how much work we'll need to do to fully restore service

Staged Deployment: concrete example

- Once we have an estimate for ***U***, we have a better idea of how much work we'll need to do to fully restore service
  - if U=1, then we're already okay: the rollback is sufficient, because each failure only impacts the incoming request

Staged Deployment: concrete example

- Once we have an estimate for ***U***, we have a better idea of how much work we'll need to do to fully restore service
  - if U=1, then we're already okay: the rollback is sufficient, because each failure only impacts the incoming request
  - if U > 1, we'll need to do some operations work to rollback **the state of the system**, in addition to rolling back the code
    - this might involve writing automation to trace all requests that hit the bug, restoring from a backup, etc.

CU = RK

Staged Deployment: concrete example

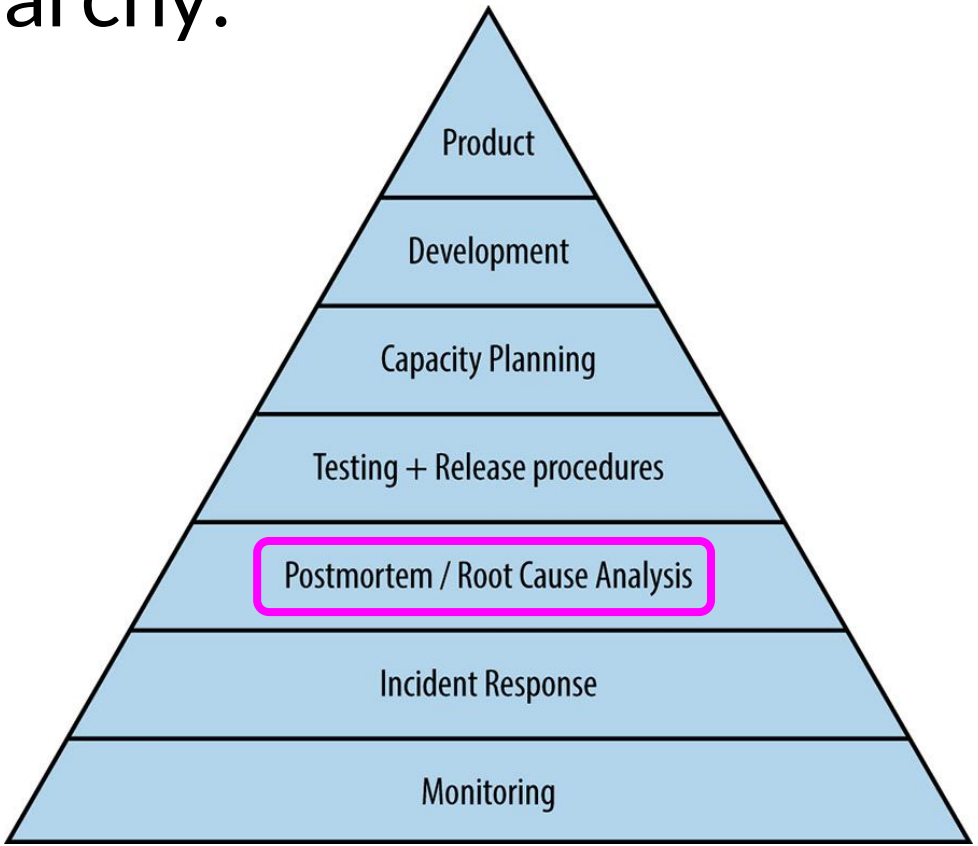
- Once we have an estimate for ***U***, we have a better idea of how much work we'll need to do to fully restore service
  - if U=1, then we're already okay: the rollback is sufficient, because each failure only impacts the incoming request
  - if U > 1, we'll need to do some operations work to rollback **the state of the system**, in addition to rolling back the code
    - this might involve writing automation to trace all requests that hit the bug, restoring from a backup, etc.
- As we do all of this, it's important to **keep records**
  - they'll be useful later for **writing the post-mortem** (next topic!)

DevOps

Today's agenda:

- Operations, Toil, and the DevOps philosophy
- Achieving reliability
  - the service reliability hierarchy + SLAs/targets
  - monitoring and reliability testing
  - incident/emergency response
  - preventing problems before they occur
  - **post-mortems + learning from failure**

Service Reliability Hierarchy: Post-mortems



[Image credit: https://sre.google/sre-book/part-III-practices/]

Post-mortems

Definition: a ***postmortem*** or ***post-mortem*** (from Latin for “after death”) is a written record of an incident, its impact, the actions taken to mitigate or resolve it, the root cause(s), and the follow-up actions to prevent the incident from recurring

Post-mortems

Definition: a ***postmortem*** or ***post-mortem*** (from Latin for “after death”) is a written record of an incident, its impact, the actions taken to mitigate or resolve it, the root cause(s), and the follow-up actions to prevent the incident from recurring

- **writing** the postmortem is a good way to **fully understand** what caused an emergency (cf., “writing clarifies your thinking”)

Post-mortems

Definition: a ***postmortem*** or ***post-mortem*** (from Latin for “after death”) is a written record of an incident, its impact, the actions taken to mitigate or resolve it, the root cause(s), and the follow-up actions to prevent the incident from recurring

- **writing** the postmortem is a good way to **fully understand** what caused an emergency (cf., “writing clarifies your thinking”)
- good postmortems are **blameless** and **actionable**:

Post-mortems

Definition: a ***postmortem*** or ***post-mortem*** (from Latin for “after death”) is a written record of an incident, its impact, the actions taken to mitigate or resolve it, the root cause(s), and the follow-up actions to prevent the incident from recurring

- **writing** the postmortem is a good way to **fully understand** what caused an emergency (cf., “writing clarifies your thinking”)
- good postmortems are **blameless** and **actionable**:
  - “**blameless**” = find the faults in the process, not the people

Post-mortems

Definition: a ***postmortem*** or ***post-mortem*** (from Latin for “after death”) is a written record of an incident, its impact, the actions taken to mitigate or resolve it, the root cause(s), and the follow-up actions to prevent the incident from recurring

- **writing** the postmortem is a good way to **fully understand** what caused an emergency (cf., “writing clarifies your thinking”)
- good postmortems are **blameless** and **actionable**:
  - “**blameless**” = find the faults in the process, not the people
  - “**actionable**” = give specific guidance for how to avoid the problem in the future (these become tickets)

Post-mortems: blameless

- Why not assign blame after an incident?
  - After all, **someone** should be responsible, right?

Post-mortems: blameless

- Why not assign blame after an incident?
  - After all, **someone** should be responsible, right?
- Some reasons:
  - Gives people **confidence to escalate** issues without fear
  - Avoids creating a culture in which incidents and issues are **swept under the rug** (which is worse long-term!)
  - **Learning experience**: engineers who have experienced an incident won't make the same mistakes again
  - You can't "fix" people, but you can fix **systems and processes**

Post-mortems: blameless

- Why not assign bla
  - After all, **some**
- Some reasons:
  - Gives people **c**
  - Avoids creating
    swept under th
  - **Learning experience**: engineers who have experienced an incident won't make the same mistakes again
  - You can't "fix" people, but you can fix **systems and processes**

Historically, software engineering adopted a lot of "blameless culture" from **aviation and medicine**, where mistakes can be fatal! We might not have the same stakes, but **all complex systems are similar** in a lot of ways.

Post-mortems: peer review

- Post-mortems are most effective when they are **peer-reviewed**

Post-mortems: peer review

- Post-mortems are most effective when they are **peer-reviewed**
  - My peers might be more senior professors, but yours will be **more senior engineers**

Post-mortems: peer review

- Post-mortems are most effective when they are **peer-reviewed**
  - My peers might be more senior professors, but yours will be **more senior engineers**
- Peer review **raises the bar**: senior engineers on other teams will expect you to **explain and justify** the changes you are proposing in response to an incident

Post-mortems: peer review

- Post-mortems are most effective when they are **peer-reviewed**
  - My peers might be more senior professors, but yours will be **more senior engineers**
- Peer review **raises the bar**: senior engineers on other teams will expect you to **explain and justify** the changes you are proposing in response to an incident
  - leads to more actionable takeaways and better understanding of what went wrong

Post-mortems: peer review

- Post-mortems are most effective when they are **peer-reviewed**
  - My peers might be more senior professors, but yours will be **more senior engineers**
- Peer review **raises the bar**: senior engineers on other teams will expect you to **explain and justify** the changes you are proposing in response to an incident
  - leads to more actionable takeaways and better understanding of what went wrong
  - also enables engineers on different teams to learn from each others' mistakes

Post-mortems: example

Shakespeare Sonnet++ Postmortem (incident #465)

Date: 2015-10-21

Authors: jennifer, martym, agoogler

Status: Complete, action items in progress

Summary: Shakespeare Search down for 66 minutes during period of very high interest in Shakespeare due to discovery of a new sonnet.

Impact:[163] Estimated 1.21B queries lost, no revenue impact.

Root Causes:[164] Cascading failure due to combination of exceptionally high load and a resource leak when searches failed due to terms not being in the Shakespeare corpus. The newly discovered sonnet used a word that had never before appeared in one of Shakespeare's works, which happened to be the term users searched for. Under normal circumstances, the rate of task failures due to resource leaks is low enough to be unnoticed.

Trigger: Latent bug triggered by sudden increase in traffic.

[source: https://sre.google/sre-book/example-postmortem/]

Post-mortems: example

Shakespeare Sonnet++ Postmortem (incident #465)

Date: 2015-10-21

Authors: jennifer, martym, agoogler

Status: Compl

Summary: Sha
a new sonnet.

Resolution: Directed traffic to sacrificial cluster and added 10x capacity to mitigate cascading failure. Updated index deployed, resolving interaction with latent bug. Maintaining extra capacity until surge in public interest in new sonnet passes. Resource leak identified and fix deployed.

Detection: Borgmon detected high level of HTTP 500s and paged on-call.

Impact:[163] Esti

Root Causes:[164] Cascading failure due to combination of exceptionally high load and a resource leak when searches failed due to terms not being in the Shakespeare corpus. The newly discovered sonnet used a word that had never before appeared in one of Shakespeare's works, which happened to be the term users searched for. Under normal circumstances, the rate of task failures due to resource leaks is low enough to be unnoticed.

Trigger: Latent bug triggered by sudden increase in traffic.

[source: https://sre.google/sre-book/example-postmortem/]

Post-mortems: example

Action Item	Type	Owner	Bug
Update playbook with instructions for responding to cascading failure	mitigate	jennifer	n/a DONE
Use flux capacitor to balance load between clusters	prevent	martym	Bug 5554823 TODO
Schedule cascading failure test during next DiRT	process	docbrown	n/a TODO
Investigate running index MR/fusion continuously	prevent	jennifer	Bug 5554824 TODO
Plug file descriptor leak in search ranking	prevent	agoogler	Bug 5554825 DONE

[source: https://sre.google/sre-book/example-postmortem/]

Post-mortems: example

Action Item	Type	Owner	Bug
Update playbook with instructions for responding to cascading failure	mitigate	jennifer	n/a DONE
Use flux capacitor to balance load between clusters	prevent	martym	Bug 5554823 TODO
Schedule cascading failure test during next DiRT	process	docbrown	n/a TODO
Investigate running index MR/fusion continuously	prevent	jennifer	Bug 5554824 TODO
Plug file descriptor leak in search ranking	prevent	agoogler	Bug 5554825 DONE

and 5 more…

[source: https://sre.google/sre-book/example-postmortem/]

Post-mortems: example

Lessons Learned

What went well

- Monitoring quickly alerted us to high rate (reaching ~100%) of HTTP 500s
- Rapidly distributed updated Shakespeare corpus to all clusters

What went wrong

- We're out of practice in responding to cascading failure
- We exceeded our availability error budget (by several orders of magnitude) due to the exceptional surge of traffic that essentially all resulted in failures

Where we got lucky[166]

- Mailing list of Shakespeare aficionados had a copy of new sonnet available
- Server logs had stack traces pointing to file descriptor exhaustion as cause for crash
- Query-of-death was resolved by pushing new index containing popular search term

[source: https://sre.google/sre-book/example-postmortem/]

Post-mortems: example

Timeline[167]

2015-10-21 **(all times UTC)**

- 14:51 News reports that a new Shakespearean sonnet has been discovered in a Delorean's glove compartment
- 14:53 Traffic to Shakespeare search increases by 88x after post to **/r/shakespeare** points to Shakespeare search engine as place to find new sonnet (except we don't have the sonnet yet)
- 14:54 **OUTAGE BEGINS** — Search backends start melting down under load
- 14:55 docbrown receives pager storm, `ManyHttp500s` from all clusters
- 14:57 All traffic to Shakespeare search is failing: see **https://monitor**
- 14:58 docbrown starts investigating, finds backend crash rate very high
- 15:01 **INCIDENT BEGINS** docbrown declares incident #465 due to cascading failure, coordination on `#shakespeare`, names jennifer incident commander
- 15:02 someone coincidentally sends email to **shakespeare-discuss@** re sonnet discovery, which happens to be at top of martym's inbox

[source: https://sre.google/sre-book/example-postmortem/]

Post-mortems: example

Timeline[167]

2015-10-21 **(all times UTC)**

- 14:51 News reports that a new Shakespearean sonnet has been discovered in a Delorean's glove compartment
- 14:53 Traffic to Shakespeare search increases by 88x after post to **/r/shakespeare** points to Shakespeare search engine as place to find new sonnet (except we don't have the sonnet yet)
- 14:54 **OUTAGE BEGINS** — Search backends start melting down under load
- 14:55 docbrown receives pager storm, `ManyHttp500s` from all clusters
- 14:57 All traffic to Shakespeare search is failing: see **https://monitor**
- 14:58 docbrown starts investigating, finds backend crash rate very high
- 15:01 **INCIDENT BEGINS** docbrown declares incident #465 due to cascading failure, coordination on

this goes on for several pages!

- **shows importance of keeping records**

ppens to be at

[source: https://sre.google/sre-book/example-postmortem/]

DevOps: takeaways

- Many modern engineering organizations prefer to combine, rather than separate, development and operations
  - this works best when most systems are services
- Major benefit of DevOps approach is elimination of toil
  - developers are best at building automation
- Planning for incidents/emergencies is critical
  - Monitoring allows on-call to quickly identify problems
  - Have a plan (ideally, in a playbook) for incidents
  - Use post-mortems to learn from prior emergencies
    - not to blame people for causing them!