DevOps

Martin Kellogg

Reading quiz: DevOps

Reading quiz: DevOps

Q1: **TRUE** or **FALSE**: Google SRE teams operate under the assumption that 100% is the wrong reliability target for basically everything.

Q2: **TRUE** or **FALSE:** SRE tries to retain highly reliable, low overhead backup communication systems that are fully separate from the rest of Google's infrastructure, but often cannot because Google services are so pervasive.

Reading quiz: DevOps

Q1: **TRUE** or **FALSE**: Google SRE teams operate under the assumption that 100% is the wrong reliability target for basically everything.

Q2: **TRUE** or **FALSE:** SRE tries to retain highly reliable, low overhead backup communication systems that are fully separate from the rest of Google's infrastructure, but often cannot because Google services are so pervasive.

Reading quiz: DevOps

Q1: **TRUE** or **FALSE**: Google SRE teams operate under the assumption that 100% is the wrong reliability target for basically everything.

Q2: **TRUE** or **FALSE:** SRE tries to retain highly reliable, low overhead backup communication systems that are fully separate from the rest of Google's infrastructure, but often cannot because Google services are so pervasive.

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

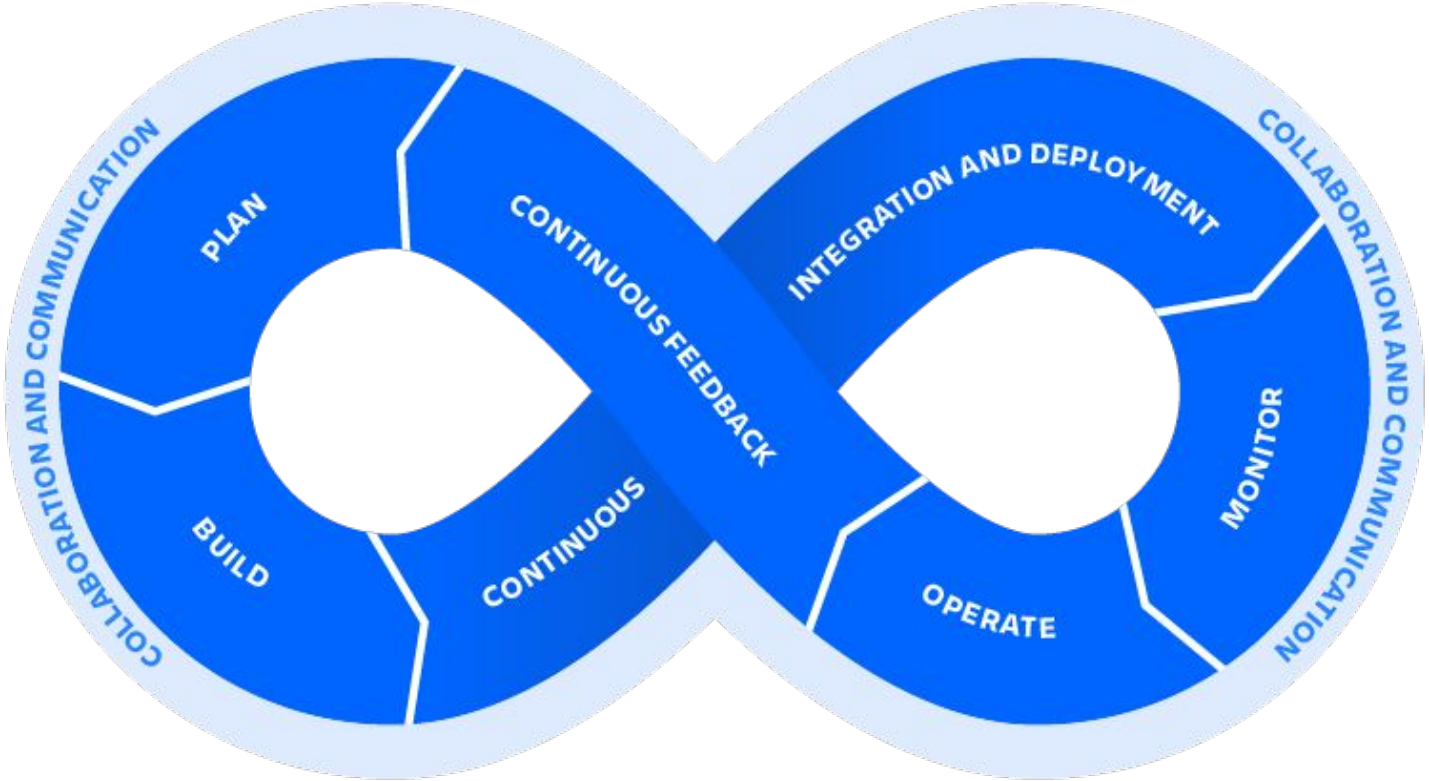


figure credit: Atlassian

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

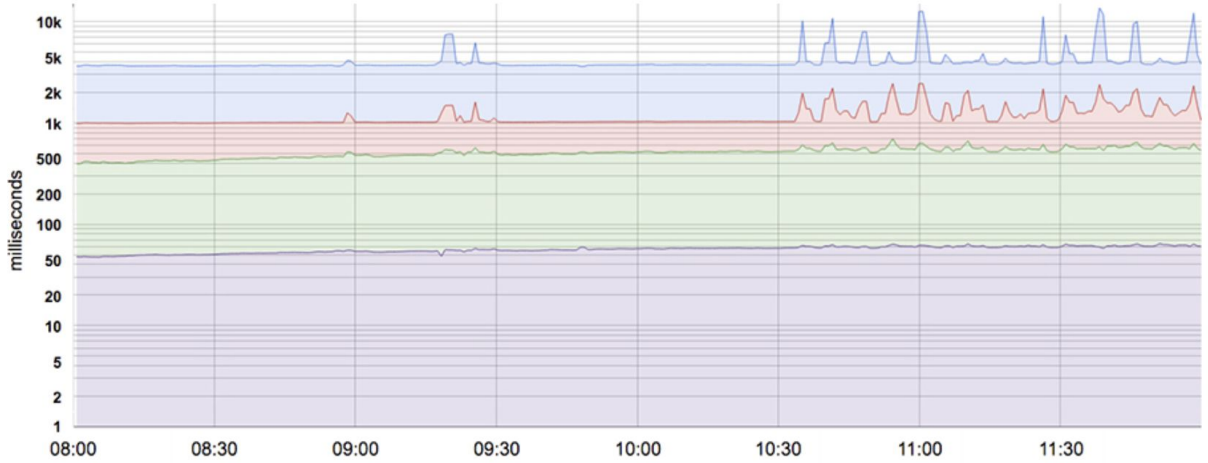
Aside: subtleties in metrics

- 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

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

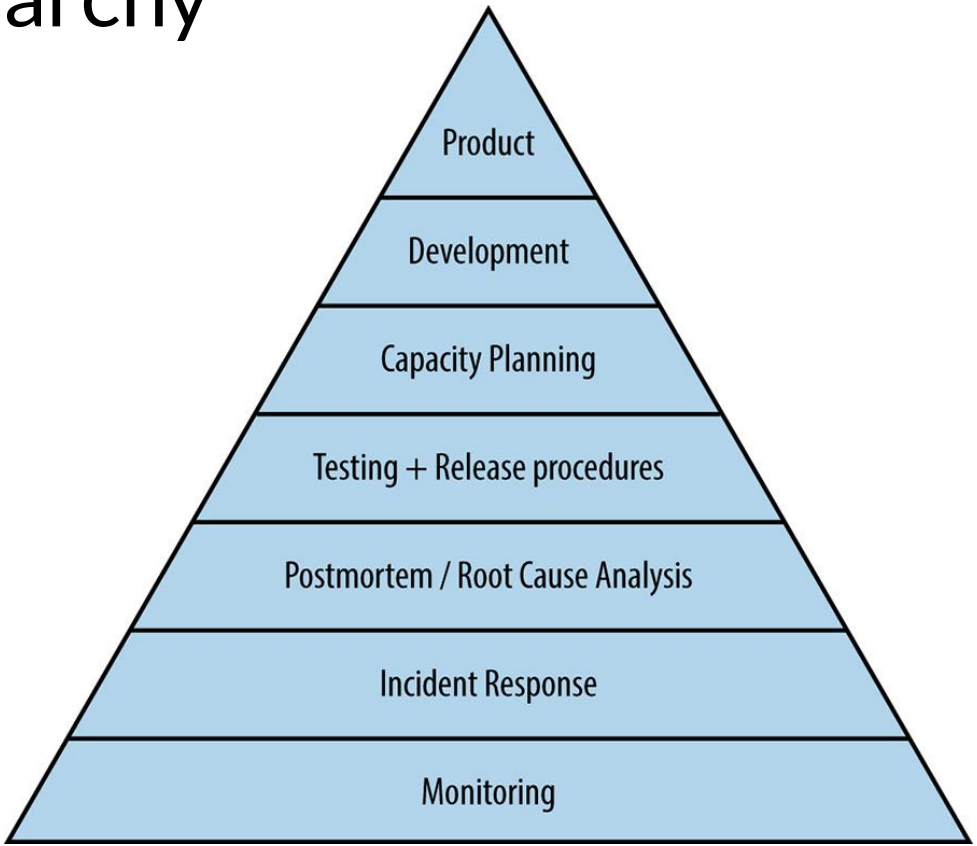
- Once we have defined an SLA (internally or externally), how do we meet it?
  - Easy way to demonstrate that we're meeting an SLA: **collect the metrics** in the SLA!
  - Then, make sure that those metrics actually look good.
- How do we think about how to do this?

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/]

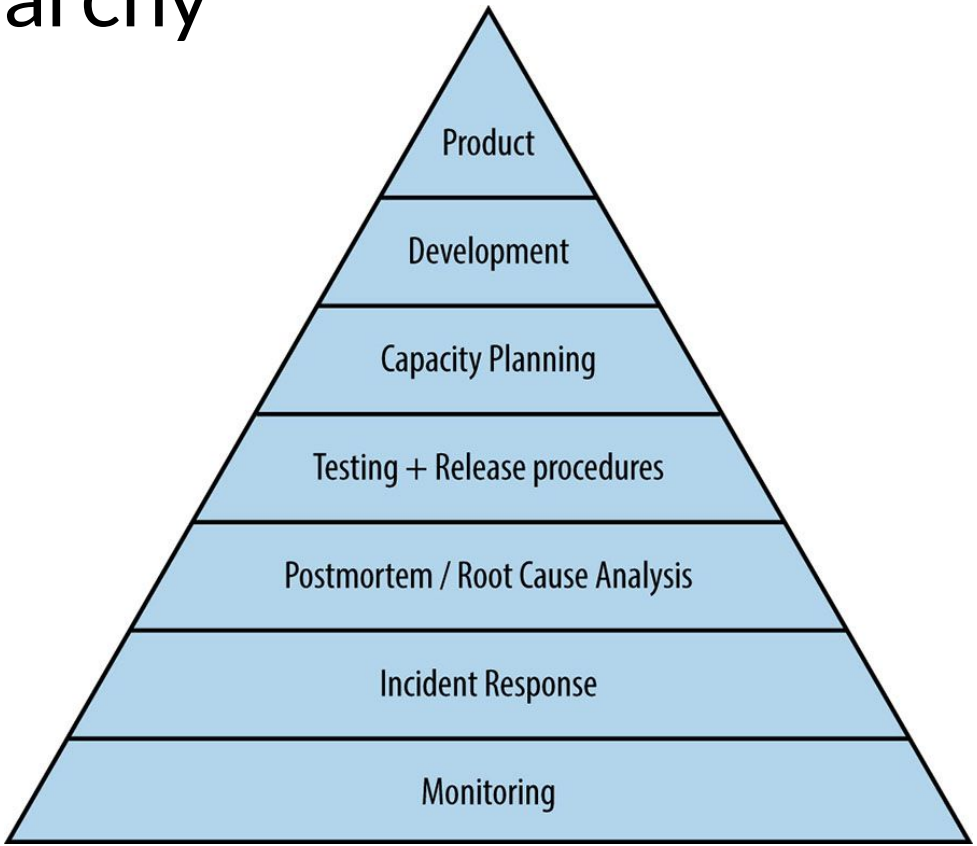
Maslow’s Hierarchy of Needs

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

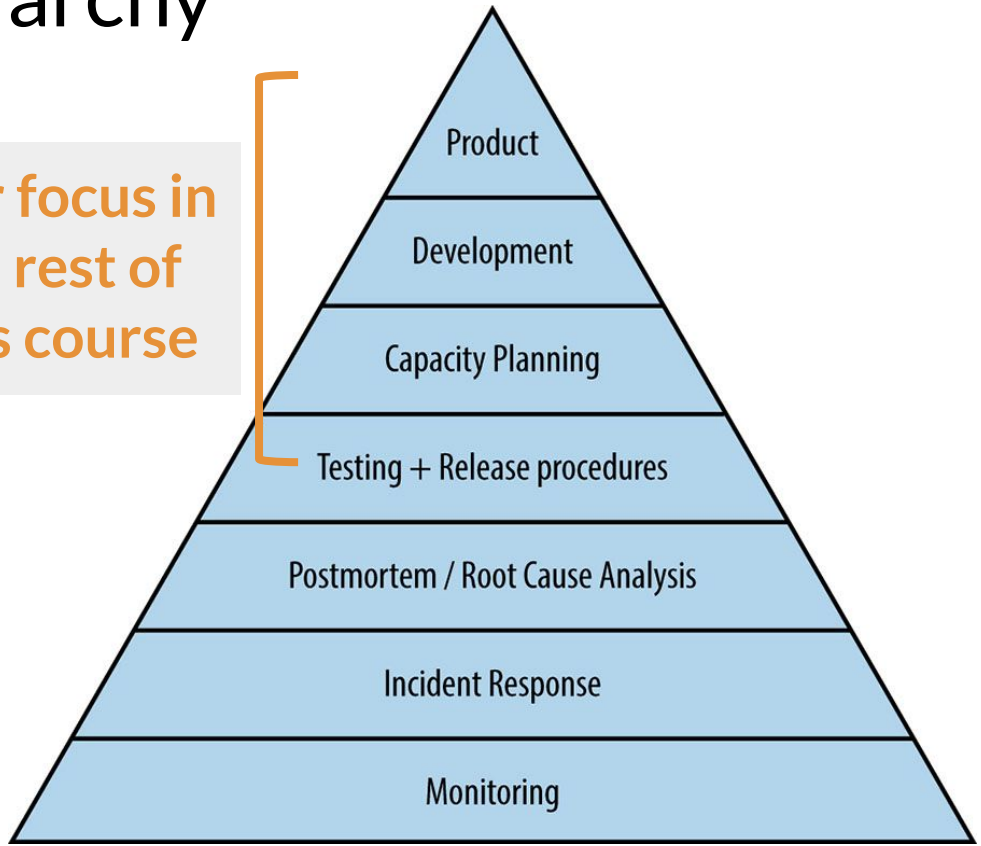


[Image credit: https://sre.google/sre-book/part-III-practices/]

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

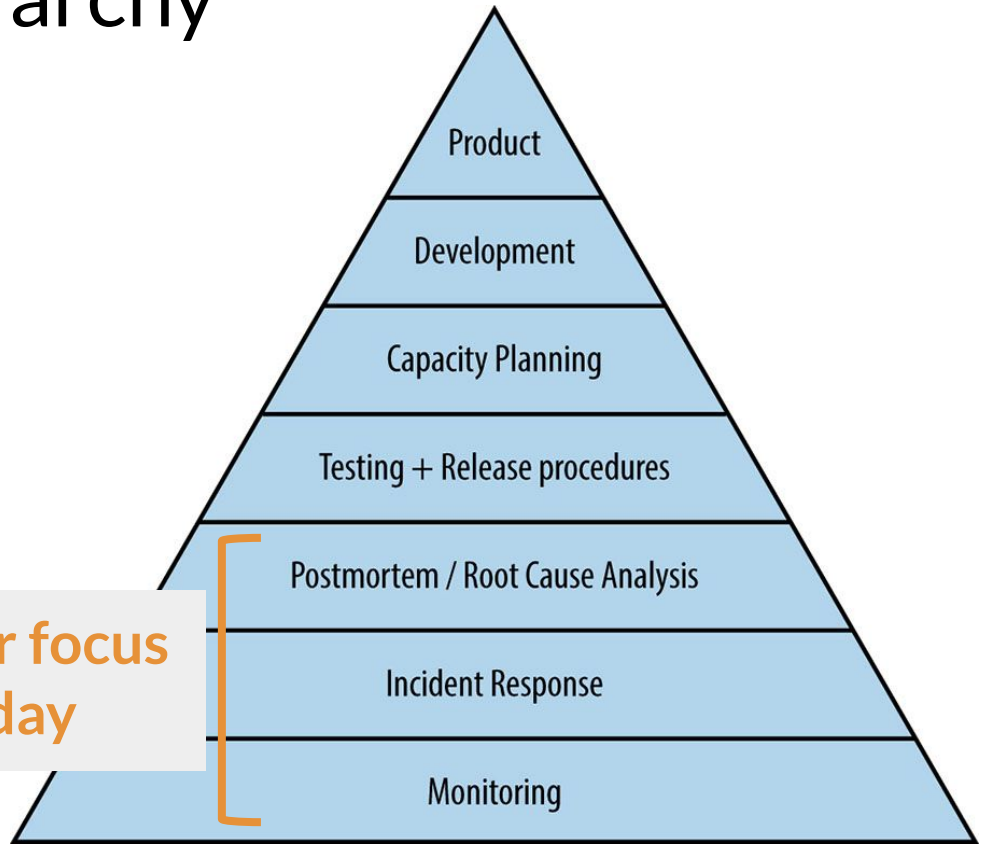
our focus in the rest of this course



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

Service Reliability Hierarchy

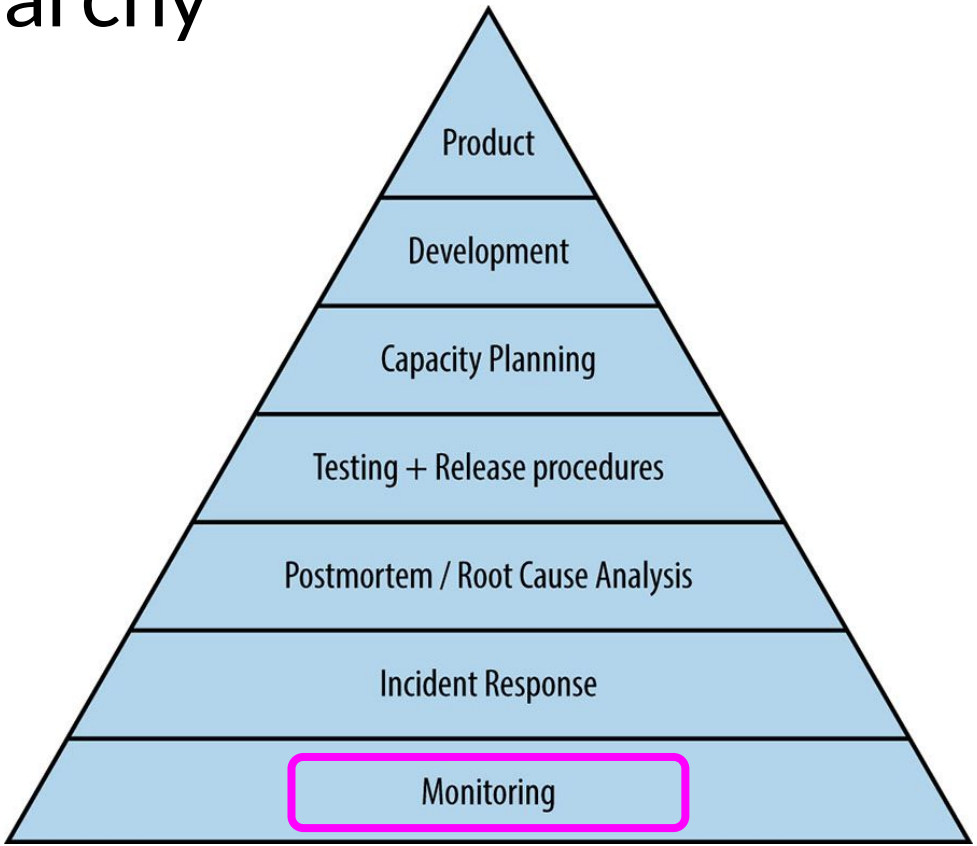
- analogy to Maslow's "Hierarchy of Needs" for humans
- just like in Maslow's hierarchy, if there is a serious deficiency in a lower level achieving the higher level becomes a lot harder



[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/]