

# DevOps (2/2)

Martin Kellogg

# DevOps (2/2)

Today's agenda:

- Reading Quiz
- The service reliability hierarchy + SLAs/targets
- Monitoring
- Incident/emergency response
- Post-mortems + learning from failure

# DevOps (2/2)

Today's agenda:

- Reading Quiz
- The service reliability hierarchy
- Monitoring
- Incident/emergency response
- Post-mortems + learning

## Announcements:

- if you haven't yet had your sprint 3 retro meeting, do so in the next ~24 hours
- reading for Thursday posted (it's short, sorry I forgot until today)
- next Tuesday: panel of engineers
  - Canvas assignment to submit questions (at least 1 required for participation points)
- exam next Thursday
  - practice exam "soon"

# DevOps (2/2)

Today's agenda:

- **Reading Quiz**
- The service reliability hierarchy + SLAs/targets
- Monitoring
- Incident/emergency response
- Post-mortems + learning from failure

# Reading quiz: DevOps 2

Q1: **TRUE** or **FALSE**: if an on-call engineer fails to properly deal with an incident, the post-mortem about the incident should name and shame them for messing up

Q2: Which of the following does Dan Luu advocate for when making a high-risk change?

- A. having multiple people watch or confirm the operation
- B. having ops people standing by in case of disaster
- C. automating the change instead of letting a human do it

# Reading quiz: DevOps 2

Q1: **TRUE** or **FALSE**: if an on-call engineer fails to properly deal with an incident, the post-mortem about the incident should name and shame them for messing up

Q2: Which of the following does Dan Luu advocate for when making a high-risk change?

- A. having multiple people watch or confirm the operation
- B. having ops people standing by in case of disaster
- C. automating the change instead of letting a human do it

# Reading quiz: DevOps 2

Q1: **TRUE** or **FALSE**: if an on-call engineer fails to properly deal with an incident, the post-mortem about the incident should name and shame them for messing up

Q2: Which of the following does Dan Luu advocate for when making a high-risk change?

- A. having multiple people watch or confirm the operation
- B. having ops people standing by in case of disaster
- C. automating the change instead of letting a human do it

# DevOps (2/2)

Today's agenda:

- Reading Quiz
- **The service reliability hierarchy + SLAs/targets**
- Monitoring
- Incident/emergency response
- 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 an 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 an 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 an 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 an easy-to-collect metrics.  
**approximate** the objective
3. define the levels of those metrics 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, you often use aggregated measurements. This needs to be done carefully, e.g., consider “the number of requests per second”
  - even this apparently straightforward metric **aggregates** data over the second
- We need to consider questions like “what is the average number of requests once a second, or by averaging over a longer period?”
  - 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

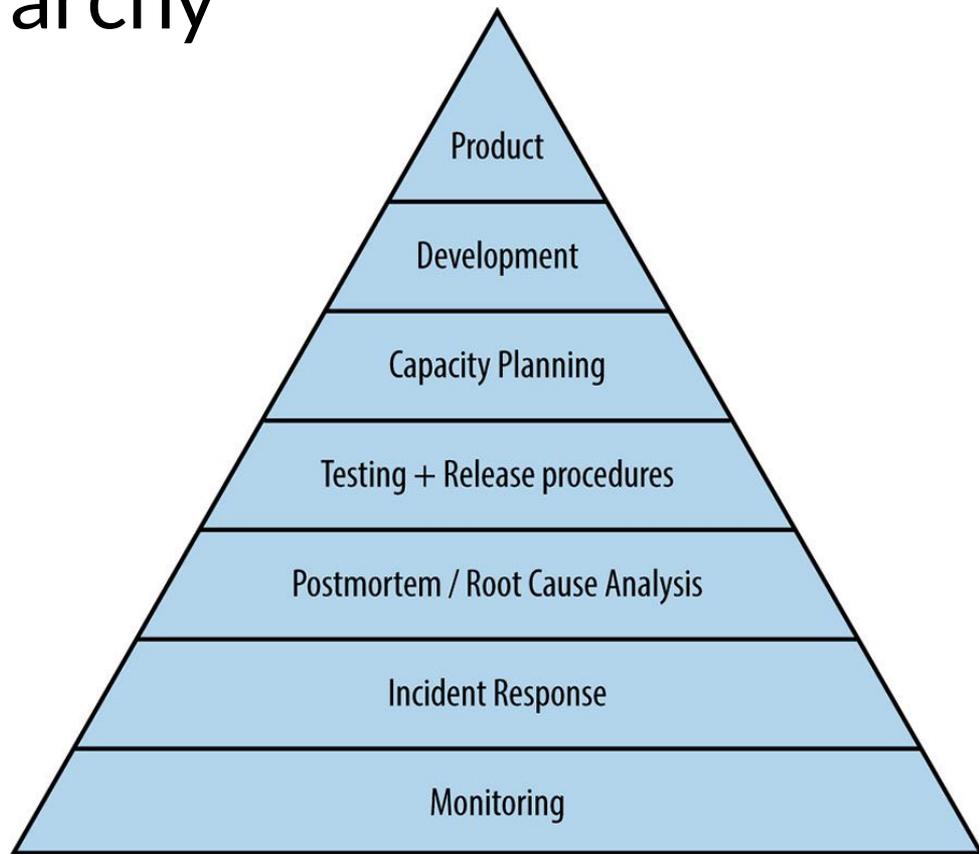
- Once we have defined an SLA (internally or externally), how do we meet it?
  - Easy way to demonstrate that we're meeting an SLA: **collect the metrics** in the SLA!
  - Then, make sure that those metrics actually look good.
- How do we think about how to do this?

# 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



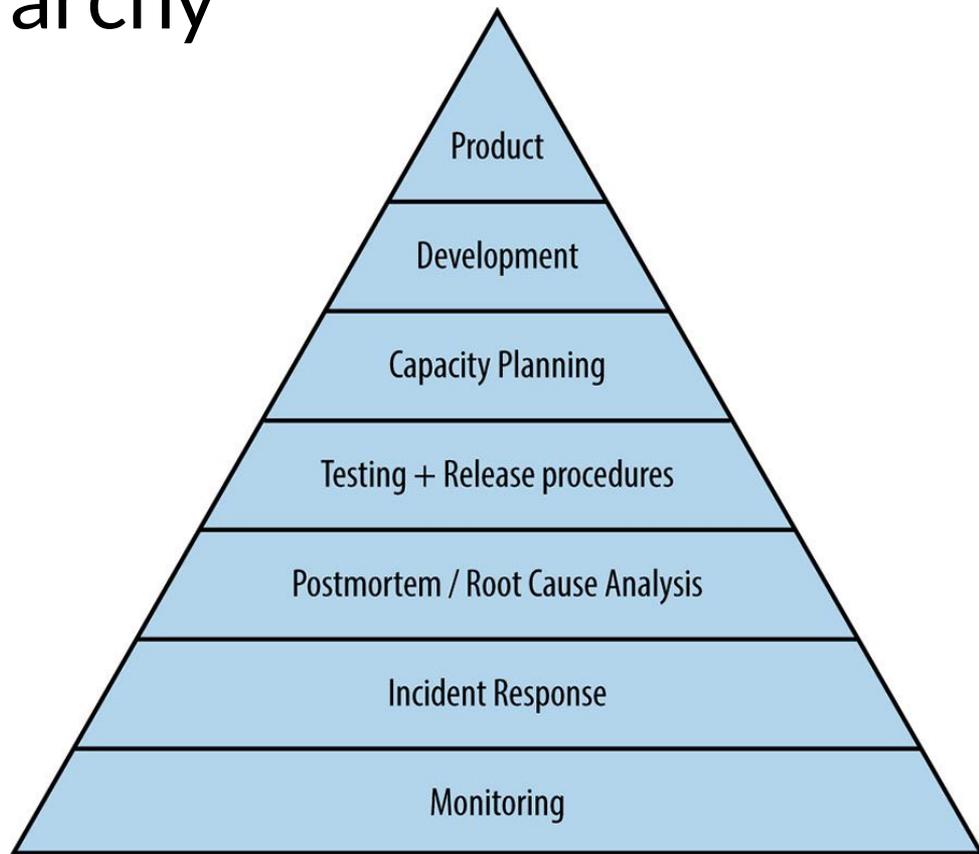
# Maslow's Hierarchy of Needs



Maslow's hierarchy of needs

# Service Reliability Hierarchy

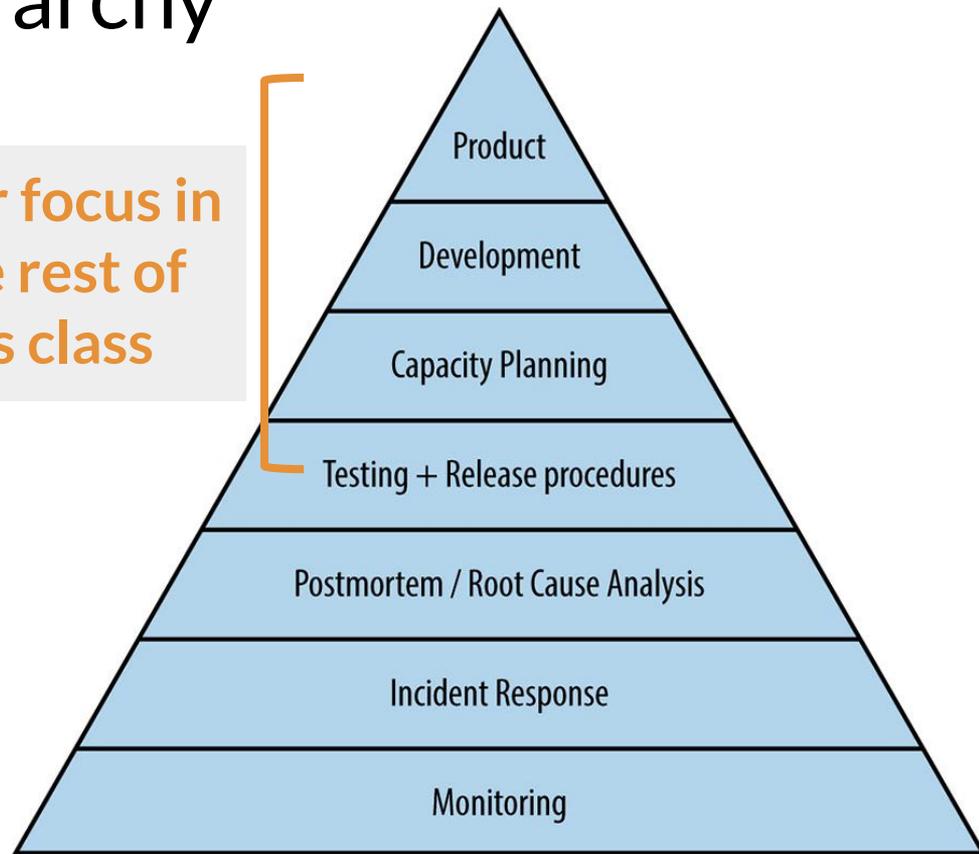
- analogy to Maslow's "Hierarchy of Needs" for humans
- just like in Maslow's hierarchy, if there is a serious deficiency in a lower level, achieving the higher level becomes a lot harder



# 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

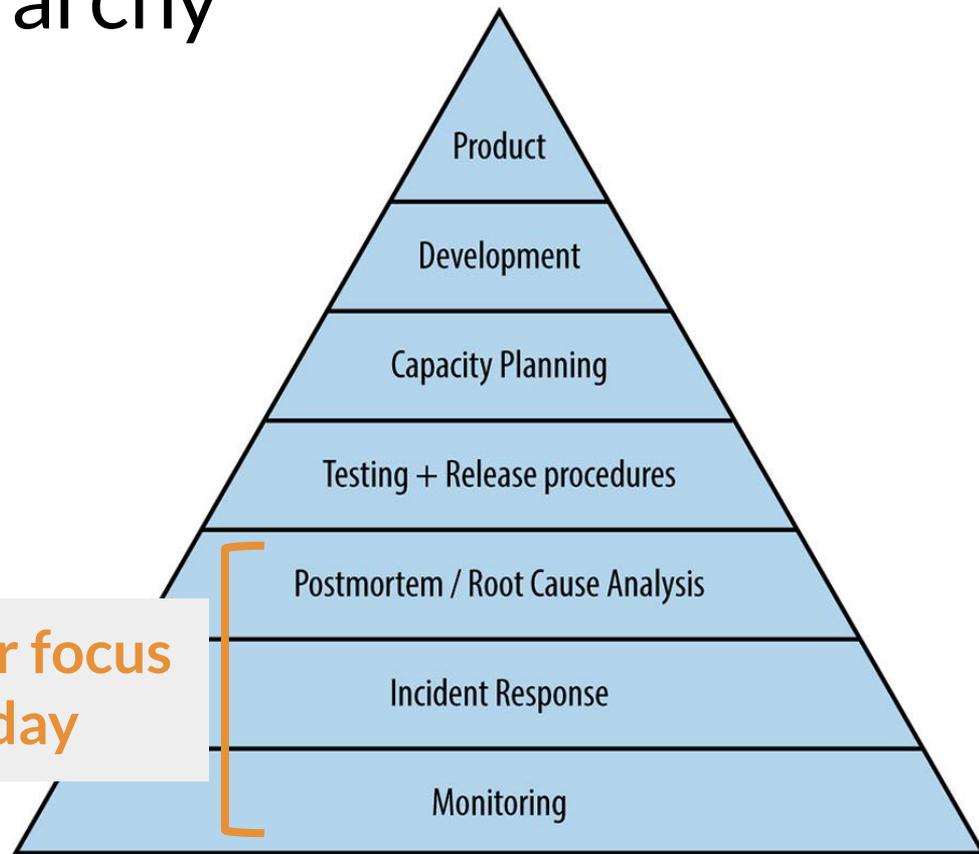
our focus in  
the rest of  
this class



# Service Reliability Hierarchy

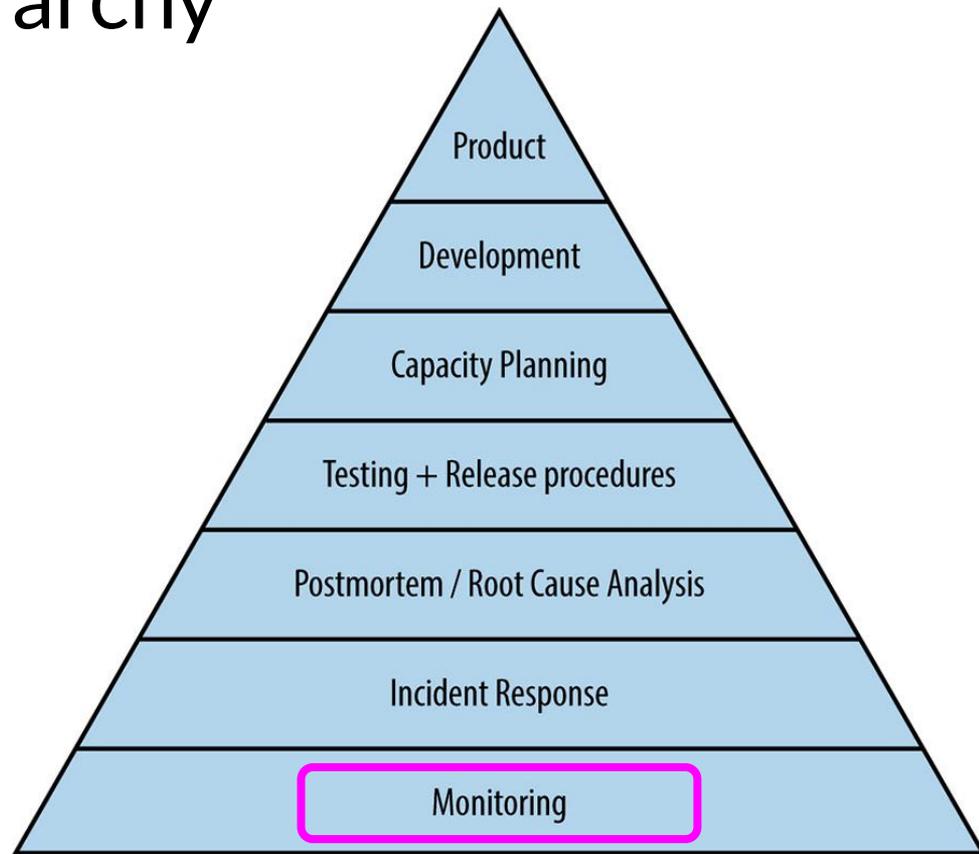
- analogy to Maslow's "Hierarchy of Needs" for humans
- just like in Maslow's hierarchy, if there is a serious deficiency in a lower level achieving the higher level becomes a lot harder

our focus today



# 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



# DevOps (2/2)

Today's agenda:

- Reading Quiz
- 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 collecting and displaying real-time quantitative data such as request counts and types, error counts, and response 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** 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** for the service for the duration of their on-call shift
  - but can (**and should**) page other team members in an emergency

# DevOps (2/2)

Today's agenda:

- Reading Quiz
- The service reliability hierarchy + SLAs/targets
- Monitoring
- **Incident/emergency response**
- Post-mortems + learning from failure

# 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: 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 important things is **rolling back** to the last known good state.
  - key idea: most emergency incidents are caused by changes that were not properly tested.
  - so, to fix the incident, you need to revert to the last known good state.

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

- 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**

# DevOps (2/2)

Today's agenda:

- Reading Quiz
- The service reliability hierarchy + SLAs/targets
- Monitoring
- Incident/emergency response
- **Post-mortems + learning from failure**

# 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 blame?
  - After all, **some**
- Some reasons:
  - Gives people **co**
  - Avoids creating **swept under the**
  - **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:**<sup>163</sup> Estimated 1.21B queries lost, no revenue impact.

**Root Causes:**<sup>164</sup> 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:** Completed

**Summary:** Shakespeare Sonnet++  
a new sonnet.

**Impact:**<sup>163</sup> Estimated 100% query time loss, no revenue impact.

**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.

**Root Causes:**<sup>164</sup> 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 <b>DONE</b>
Use flux capacitor to balance load between clusters	prevent	martym	Bug 5554823 <b>TODO</b>
Schedule cascading failure test during next DiRT	process	docbrown	n/a <b>TODO</b>
Investigate running index MR/fusion continuously	prevent	jennifer	Bug 5554824 <b>TODO</b>

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

Plug file descriptor leak in search ranking prevent

agoogle

Bug 5554825 **DONE**

# Post-mortems: example

Action Item	Type	Owner	Bug
Update playbook with instructions for responding to cascading failure	mitigate	jennifer	n/a <b>DONE</b>
Use flux capacitor to balance load between clusters	prevent	martym	Bug 5554823 <b>TODO</b>
Schedule cascading failure test during next DiRT	process	docbrown	n/a <b>TODO</b>
Investigate running index MR/fusion continuously	prevent	jennifer	Bug 5554824 <b>TODO</b>

**and 5 more...**

Plug file descriptor leak in search ranking prevent

agoogler

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

Bug 5554825 **DONE**

# 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<sup>166</sup>

- 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<sup>167</sup>

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

# Post-mortems: example

## Timeline<sup>167</sup>

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**

opens 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!