DevOps (part 2)

Martin Kellogg

Reading quiz: DevOps

Reading quiz: DevOps

Q1: Which of the following does Luu (second article’s author) suggest as the best way to mitigate human error during ops?

- **A.** having multiple people watch or confirm the operation
- **B.** having ops people standing by in case of disaster
- **C.** having a human perform manual error checking
- **D.** none of these

Q2: On Wednesday, 11/27, what room will this class held in?

Reading quiz: DevOps

Q1: Which of the following does Luu (second article's author) suggest as the best way to mitigate human error during ops?

A. having multiple people watch or confirm the operation

B. having ops people standing by in case of disaster

C. having a human perform manual error checking

D. none of these

Q2: On Wednesday, 11/27, what room will this class held in?

Reading quiz: DevOps

Q1: Which of the following does Luu (second article's author) suggest as the best way to mitigate human error during ops?

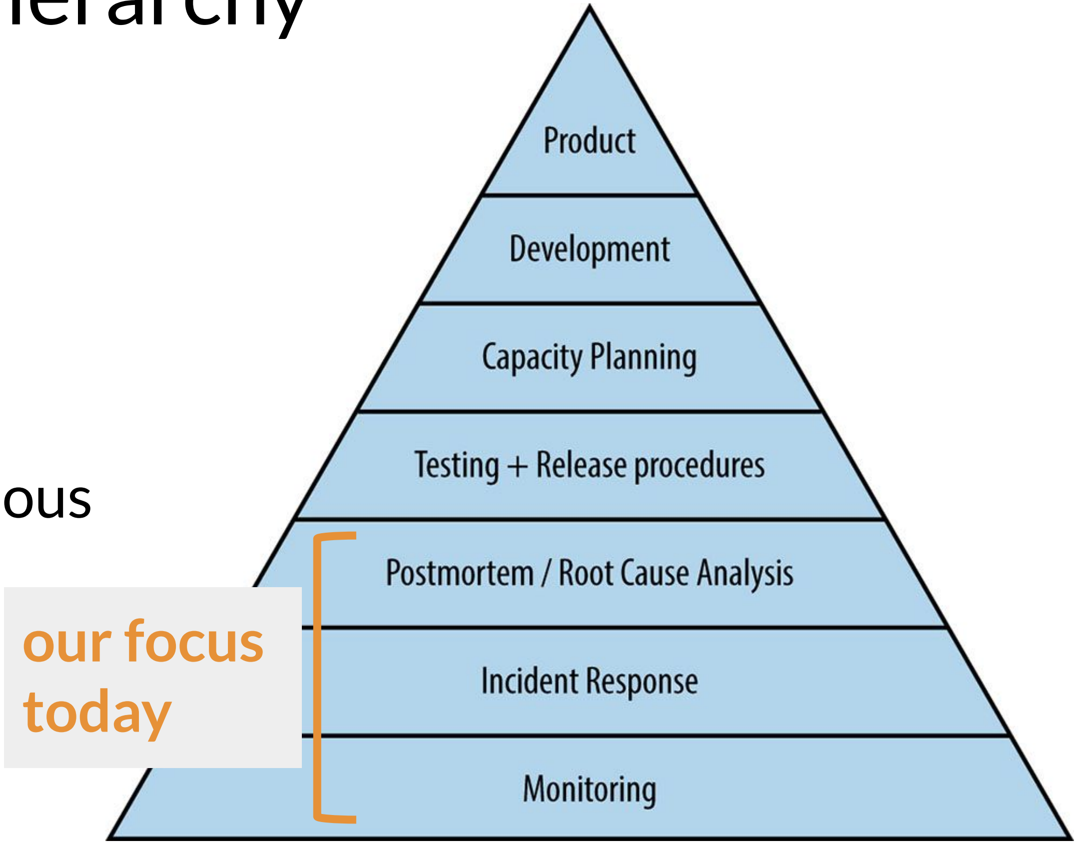
A. having multiple people watch or confirm the operation
B. having ops people standing by in case of disaster
C. having a human perform manual error checking
D. none of these

Q2: On Wednesday, 11/27, what room will this class held in?

GITC 1100 (this room!)

Service Reliability Hierarchy

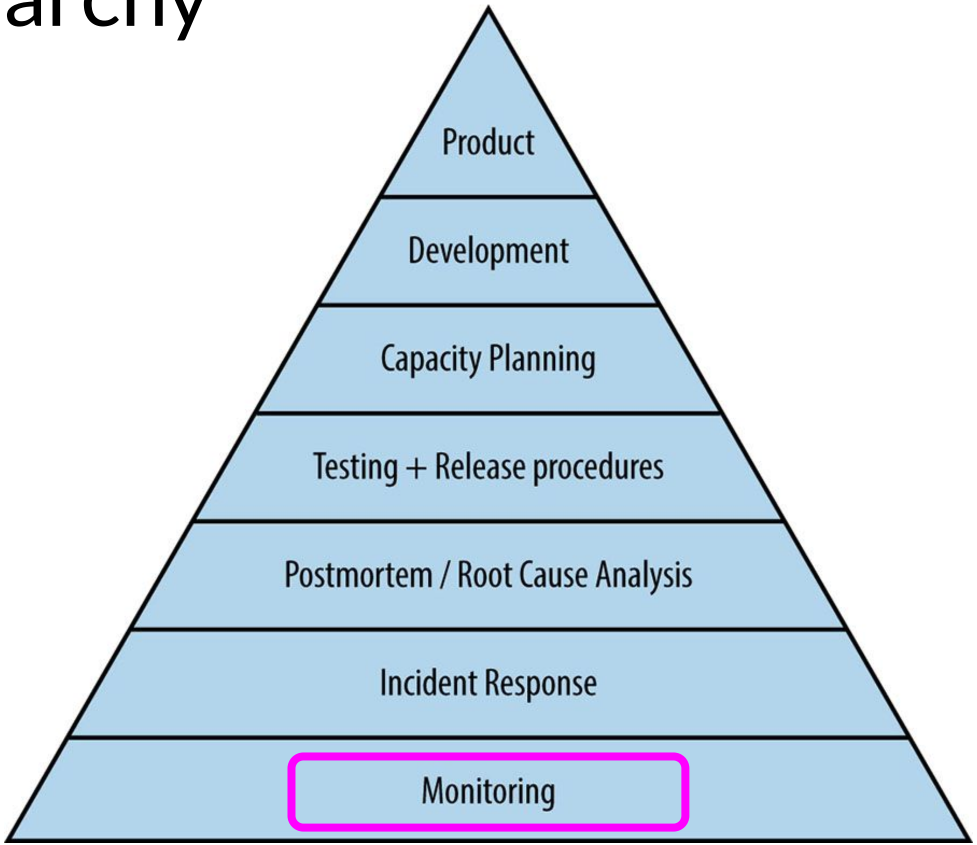
- analogy to Maslow's "Hierarchy of Needs" for humans
- just like in Maslow's hierarchy, if there is a serious deficiency in a lower level achieving the higher level becomes a lot harder



[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/]

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

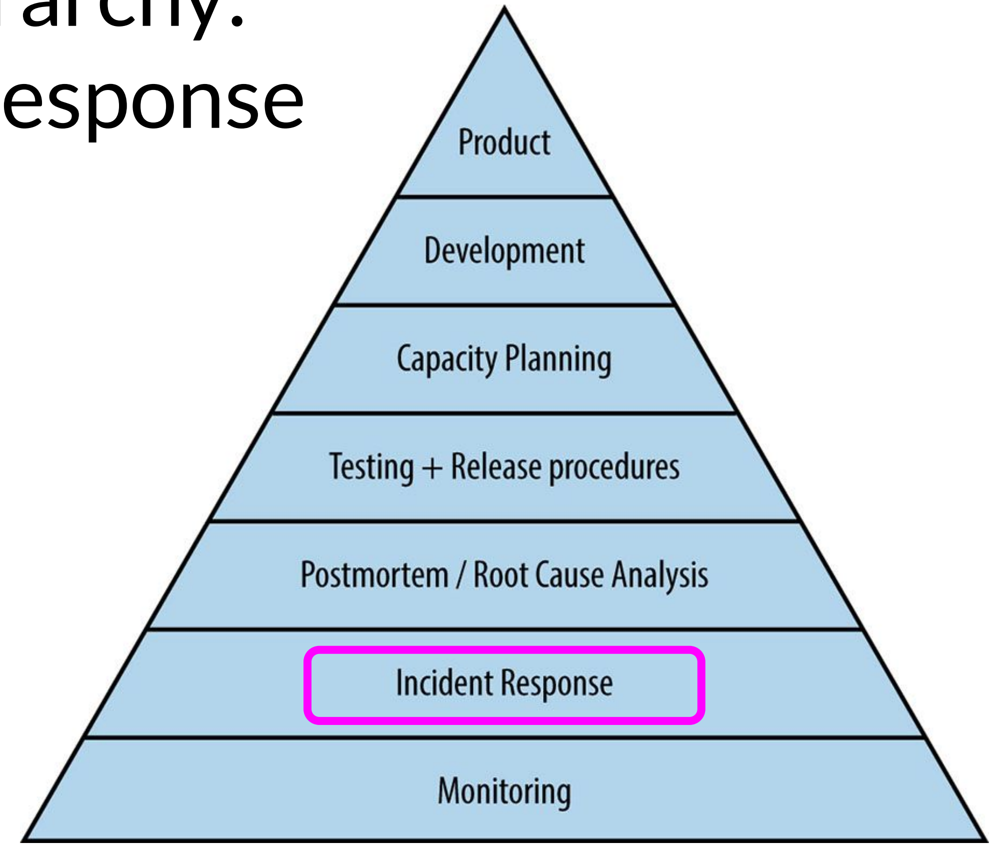
- Being on-call is a **major source of toil** in most services
  - a page about a non-emergency is one of the worst forms of toil, because it **forces** you to react
- For this reason, most teams **rotate** who is on-call
  - e.g., daily, weekly, whatever
  - **everyone** working on the service should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

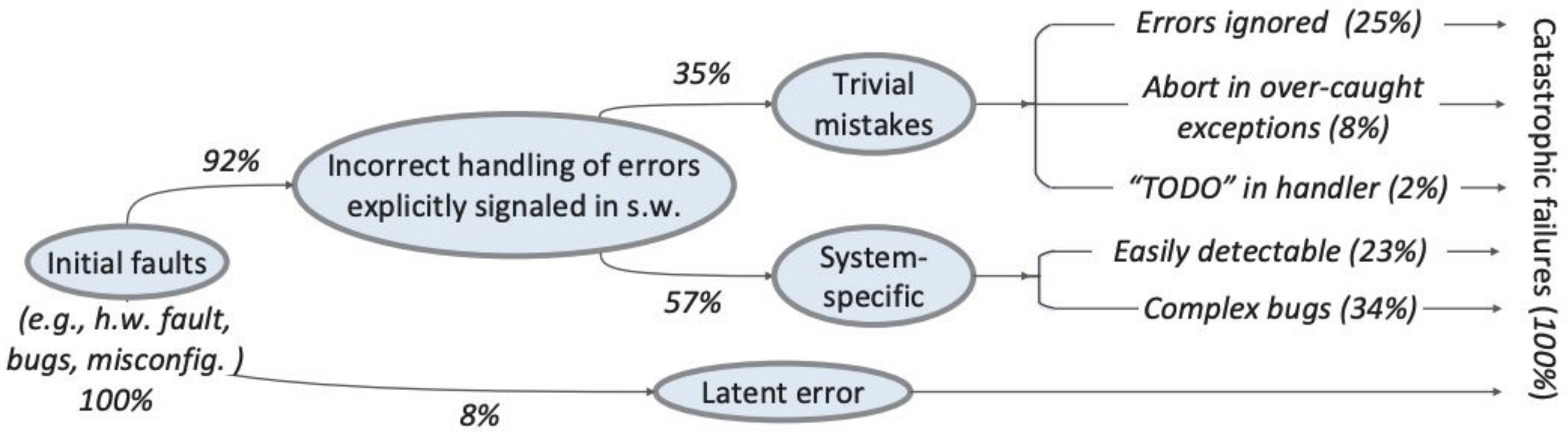
- **error handling**: code that is only called when something is wrong
  - why is this likely to cause an emergency?

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!

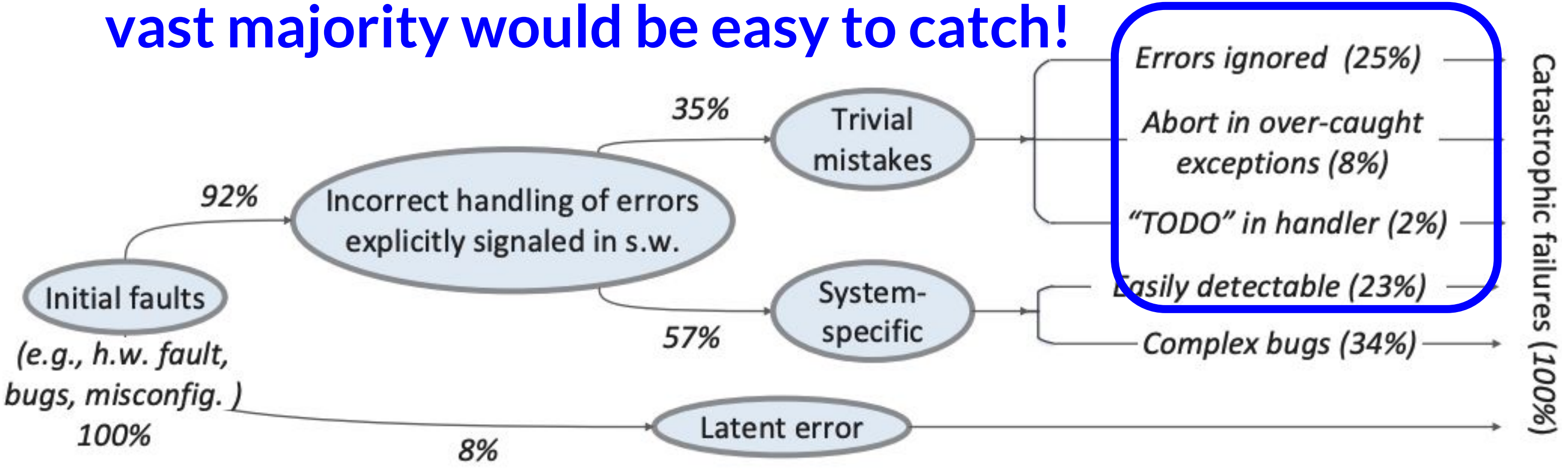


[*Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems*. Yuan et al. OSDI 2014.]

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 ... ch has a 5 year average lif...
    - get out a piece of p...
  - **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 of the techniques that we have learned in this class 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 of the techniques that we have learned in this class to **monitoring**
  - for example, should there be a **relationship** between a pair of metrics that we’re collecting?

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 of the techniques that we have learned in this class to **monitoring**
  - for example, should there be a **relationship** between a pair of metrics that we’re collecting?
    - 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

Aside: Chaos Monkey

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

Aside: 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

Aside: 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

Aside: 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

Aside: 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

Aside: 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: Chaos Monkey: why?

Aside: Chaos Monkey: why?

- 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: Chaos Monkey: why?

- 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: Chaos Monkey: why?

- 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 important goal of Chaos Monkey is to detect such cascading failures before they actually happen in production

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

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 deployme
    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.

ut 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.

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)

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***:

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

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.

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.

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.

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.

Staged Deployment: concrete example

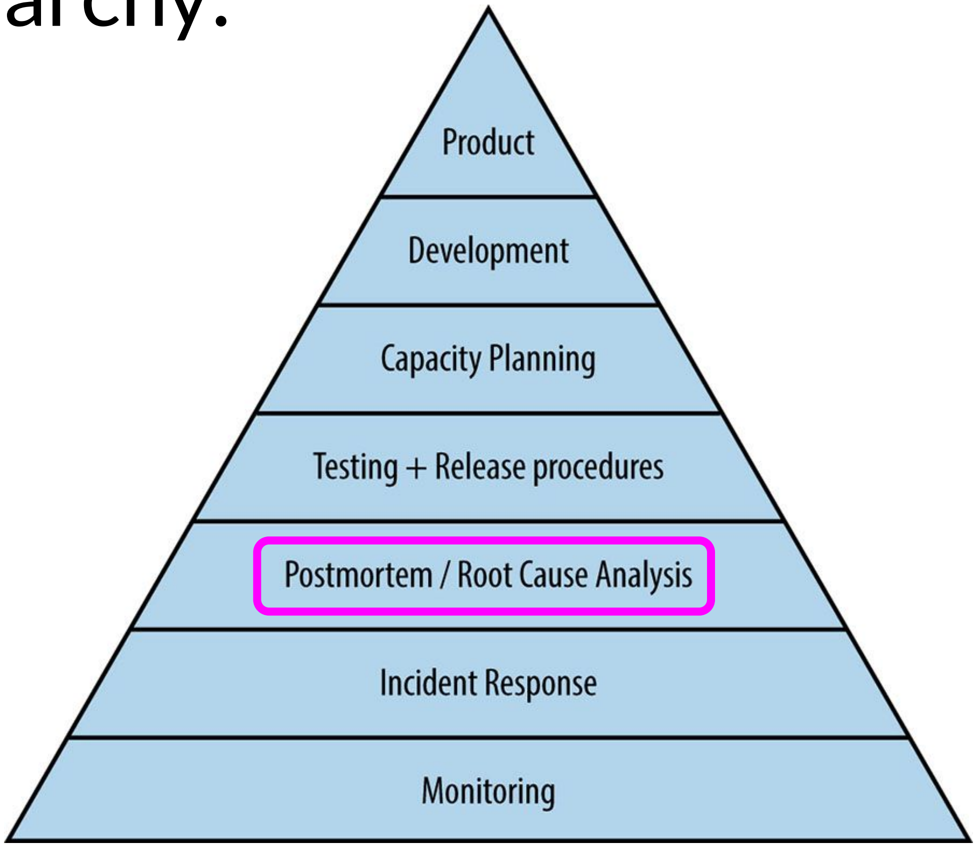
- Once we have an estimate for ***U***, we have a better idea of how much work we'll need to do to fully restore service
  - if U=1, then we're already okay: the rollback is sufficient, because each failure only impacts the incoming request
  - if U > 1, we'll need to do some operations work to rollback **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 blar
  - After all, **somе**
- Some reasons:
  - Gives people **c**
  - Avoids creatinց e
    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

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.

Summary: Sha

a new sonnet.

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

ppens to be at

this goes on for several pages!

- **shows importance of keeping records**

[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!