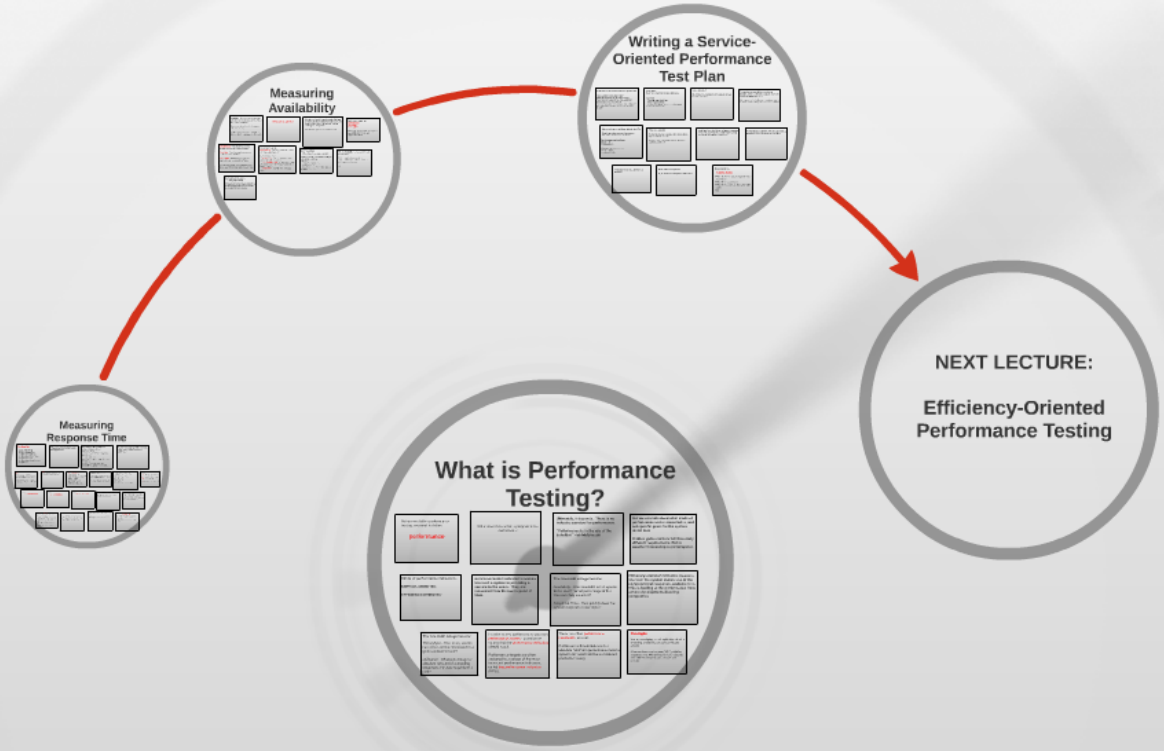
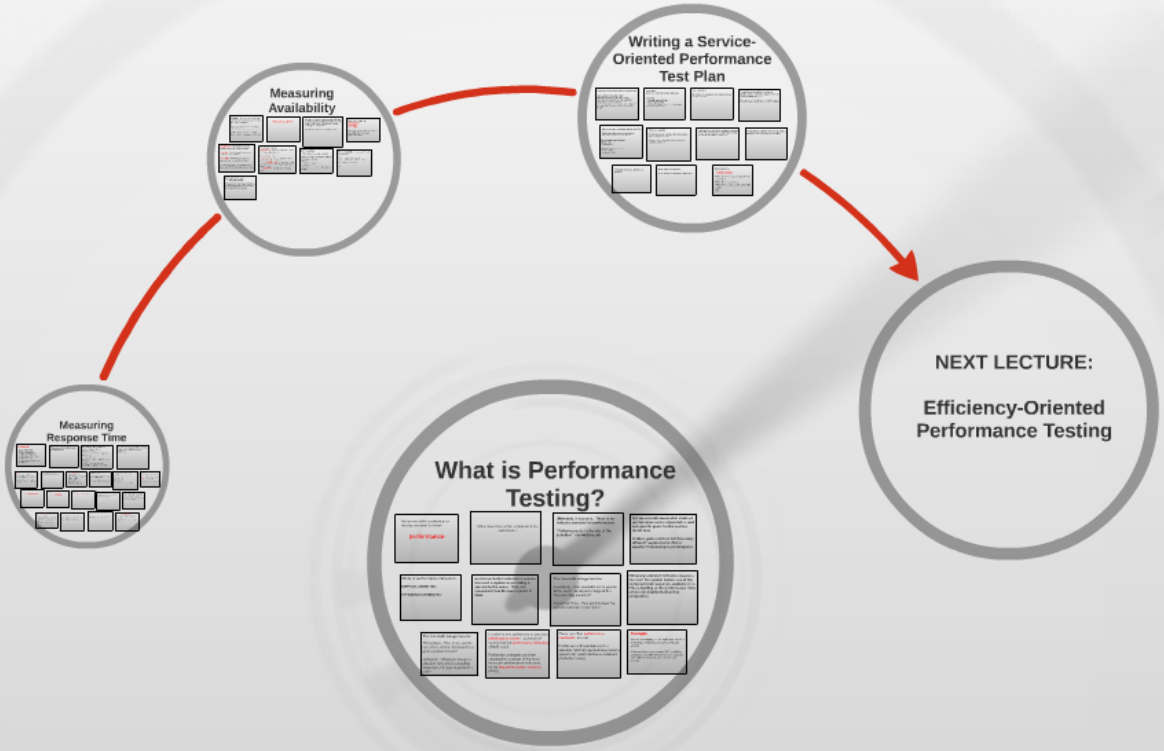
CS1699: Lecture 20 - Performance Testing Part 1



CS1699: Lecture 20 - Performance Testing Part 1



What is Performance Testing?

Before we define performance testing, we need to define

performance

What does it mean for a program to be performant?

Ultimately, it depends. There is no industry standard for performance.

"Performance is in the eye of the beholder." -Ian Molyneaux

But we can talk about what kinds of performance we're interested in, and set specific goals for the system under test.

A video game console will have very different requirements than a weather forecasting supercomputer.

Kinds of performance indicators -

SERVICE-ORIENTED

EFFICIENCY-ORIENTED

***Service-oriented indicators* measure how well a system is providing a service to the users. They are measured from the user's point of view.**

The two main categories are:

Availability - How available is the system to the user? What percentage of the time can they access it?

Response Time - How quickly does the system respond to user input?

***Efficiency-oriented indicators* measure how well the system makes use of the computational resources available to it. This is looking at the performance from a more developmental/coding perspective.**

The two main categories are:

Throughput - How many events can occur and be processed in a given amount of time?

Utilization - What percentage or absolute amount of computing resources are used to perform a task?

In order to test performance, you need *performance targets* - quantitative values that the *performance indicators* should reach.

Performance targets are often assigned to a subset of the most important performance indicators, called *key performance indicators* (KPIs).

There are often performance thresholds, as well.

Performance thresholds are the absolute minimum performance level a system can reach and be considered production-ready.

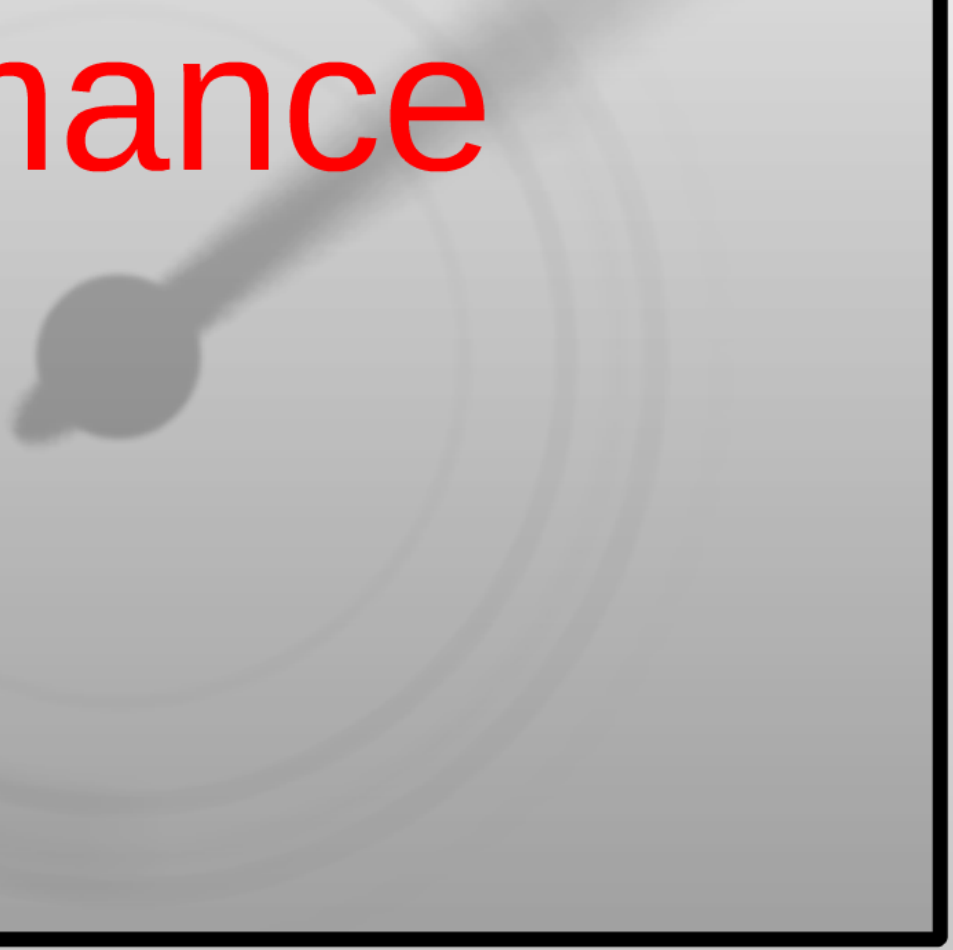
Example

You are developing a web application which is expecting a relatively constant 20 hits per second.

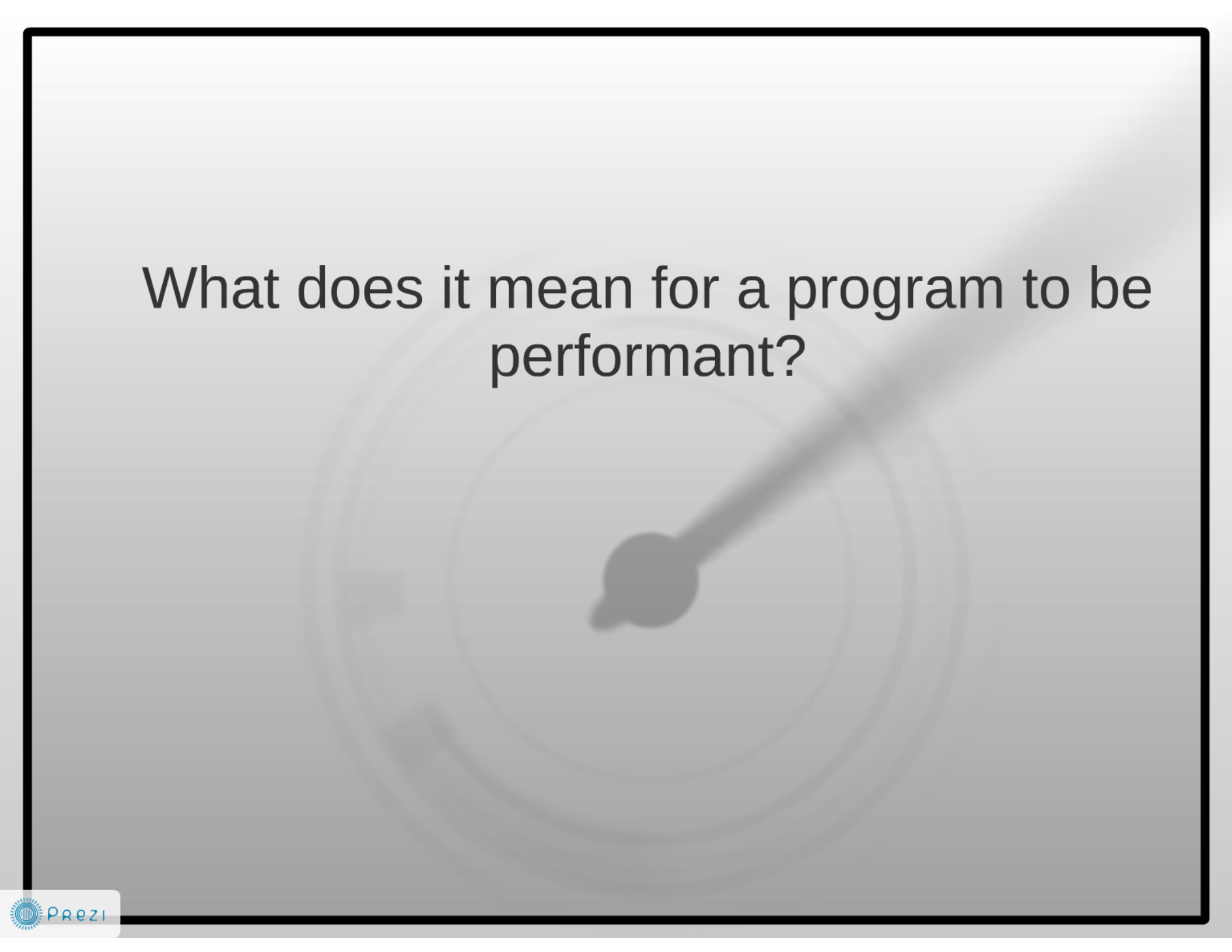
A key performance indicator (KPI) might be response time, with a threshold of 3 seconds mean time to respond and a target of 1 second.

Before we define performance testing, we need to define

performance

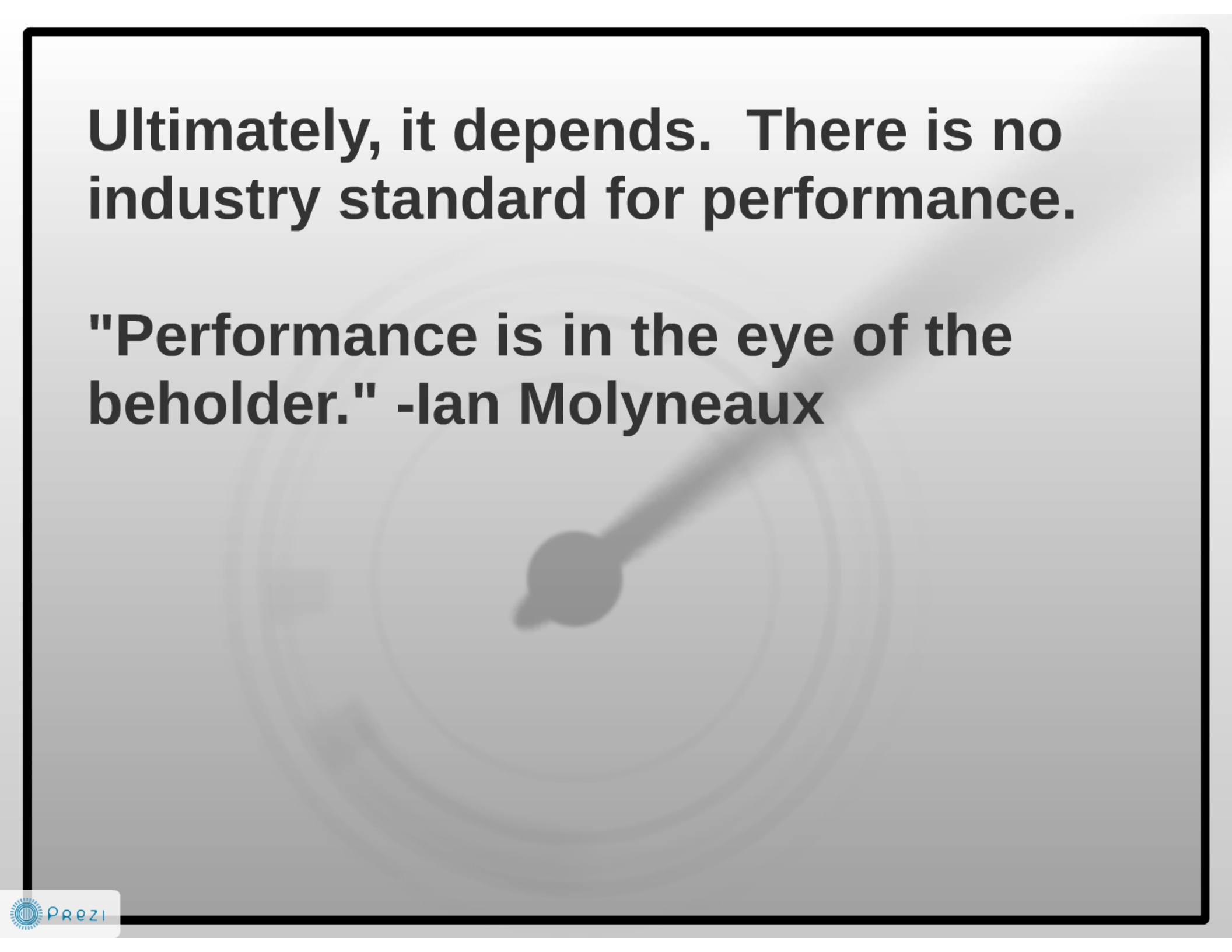


What does it mean for a program to be performant?



Ultimately, it depends. There is no industry standard for performance.

"Performance is in the eye of the beholder." -Ian Molyneaux



But we can talk about what kinds of performance we're interested in, and set specific goals for the system under test.

A video game console will have very different requirements than a weather forecasting supercomputer.

Kinds of performance indicators -

SERVICE-ORIENTED

EFFICIENCY-ORIENTED

Service-oriented indicators **measure how well a system is providing a service to the users. They are measured from the user's point of view.**

The two main categories are:

Availability - How available is the system to the user? What percentage of the time can they access it?

Response Time - How quickly does the system respond to user input?

Efficiency-oriented indicators **measure how well the system makes use of the computational resources available to it. This is looking at the performance from a more developmental/coding perspective.**

The two main categories are:

Throughput - How many events can occur and be processed in a given amount of time?

Utilization - What percentage or absolute amount of computing resources are used to perform a task?

In order to test performance, you need *performance targets* - quantitative values that the *performance indicators* should reach.

Performance targets are often assigned to a subset of the most important performance indicators, called *key performance indicators* (KPIs).

There are often performance thresholds, as well.

Performance thresholds are the absolute minimum performance level a system can reach and be considered production-ready.

Example

You are developing a web application which is expecting a relatively constant 20 hits per second.

A key performance indicator (KPI) might be response time, with a threshold of 3 seconds mean time to respond and a target of 1 second.

Measuring Response Time

The Easy Way

1. Do something
2. Click Stopwatch
3. Wait for response
4. When you see it, click stopwatch again
5. Write down number on stopwatch

What are some problems with this approach?

1. Impossible to measure sub-second response times
2. Human error
3. Probably the most boring thing a person can do
4. Time-consuming
5. Impossible to measure "hidden" responses
6. Startup or transient costs
7. Inability to get large data sets

Performance testing, more than most other testing, relies heavily on automation.

Reason 1

You should never trust a single result in performance testing, especially for response times. You should almost always be discussing a mean value, maximum value, etc.

There are so many variables in a single test run (other processes taking up CPU, pipelining issues, memory swaps, VM startup times, etc.) that a single test run is almost worthless.

Reason 2

Minor issues with human error can cause massive changes in performance results.

When performance testing...

Always use a tool.
Always get multiple values.
It's often a good idea to cut off the first n% of results (to eliminate startup costs).
Be mindful of caching.
Be mindful of utilization.
Don't do it on production systems.
Have control over the system.
Provide good input data.
Always compare apples to apples (hardware, OS, other processes, etc.)

Performance testing is more like a science than other kinds of testing.

You want to run multiple experiments and eliminate all the other variables OTHER THAN THE CODE UNDER TEST.

So don't run version 1 on a ChromeBook and version 2 on a Cray supercomputer and talk about the massive speed improvement.

What kinds of events should you time?

It all depends on the system you're testing!

Time for calculation to take place
Time for character to appear on screen
Time for image to appear
Time to download
Time for server response
Time for page to load
Time for code to execute (more of a unit-level test)

What is time?

user : Amount of time user code executes

system : Amount of time kernel code executes

total : user time + system time

real : Actual amount of time taken (wall clock time)

Example

time curl http://www.example.com

Example

Ruby Benchmarking

Example

Web Response Time with Chrome

Users almost always care about wall clock time.

Use real and system time to help developers, not as KPIs in and of themselves.

Response time is easy to test in a variety of ways. Determining what is a good TARGET is difficult, but there are some guidelines.

This ventures more into the realm of usability testing, but as rough guidelines:

The Three Key Time Limits

< 0.1 S : Response time required to feel that system is instantaneous
< 1 S : Response time required for flow of thought not to be interrupted
< 10 S : Response time required for user to stay focused on the application (and not go re-load Reddit)
-taken from *Usability Engineering* by Jakob Nielsen, 1993

Things haven't changed much since then!

Response Time Is Important!

According to Google, any amount of time over 400 ms for a page to load causes a drop-off in visitors.

People will choose websites (often subconsciously) based on a 250 ms difference in load times.

Optimizing response time is challenging, but can be very fun.

Algorithmic analysis, caching hacks, and usability tricks all come into play.

A Final Note

It is possible for responses to be too fast! Think of a scroll box that scrolls so fast the user can't see what's happening.

O brave new world, that hath such computational power in it!

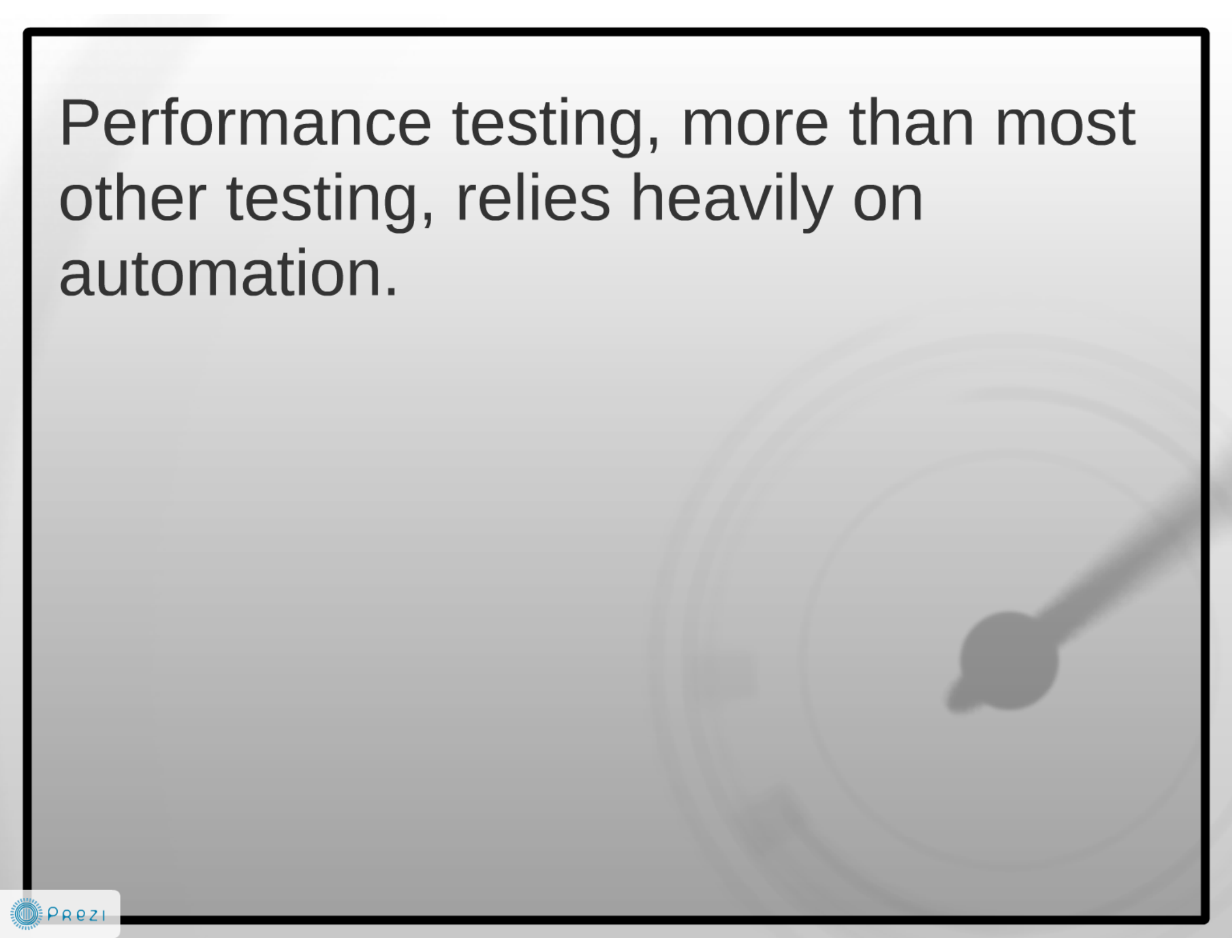
The Easy Way

1. Do something
2. Click Stopwatch
3. Wait for response
4. When you see it, click stopwatch again
5. Write down number on stopwatch

What are some problems with this approach?

1. Impossible to measure sub-second response times
2. Human error
3. Probably the most boring thing a person can do
4. Time-consuming
5. Impossible to measure "hidden" responses
6. Startup or transient costs
7. Inability to get large data sets

Performance testing, more than most other testing, relies heavily on automation.



Reason 1

You should never trust a single result in performance testing, especially for response times. You should almost always be discussing a mean value, maximum value, etc.

There are so many variables in a single test run (other processes taking up CPU, pipelining issues, memory swaps, VM startup times, etc.) that a single test run is almost worthless.

Reason 2

Minor issues with human error can cause massive changes in performance results.

When performance testing...

Always use a tool.
Always get multiple values.
It's often a good idea to cut off the first n% of results (to eliminate startup costs).
Be mindful of caching.
Be mindful of utilization.
Don't do it on production systems.
Have control over the system.
Provide good input data.
Always compare apples to apples (hardware, OS, other processes, etc.)

Performance testing is more like a science than other kinds of testing.

You want to run multiple experiments and eliminate all the other variables OTHER THAN THE CODE UNDER TEST.

So don't run version 1 on a ChromeBook and version 2 on a Cray supercomputer and talk about the massive speed improvement.

What kinds of events should you time?

It all depends on the system you're testing!

Time for calculation to take place
Time for character to appear on screen
Time for image to appear
Time to download
Time for server response
Time for page to load
Time for code to execute (more of a unit-level test)

What is time?

user : Amount of time user code executes

system : Amount of time kernel code executes

total : user time + system time

real : Actual amount of time taken (wall clock time)

Example

time curl http://www.example.com

Example

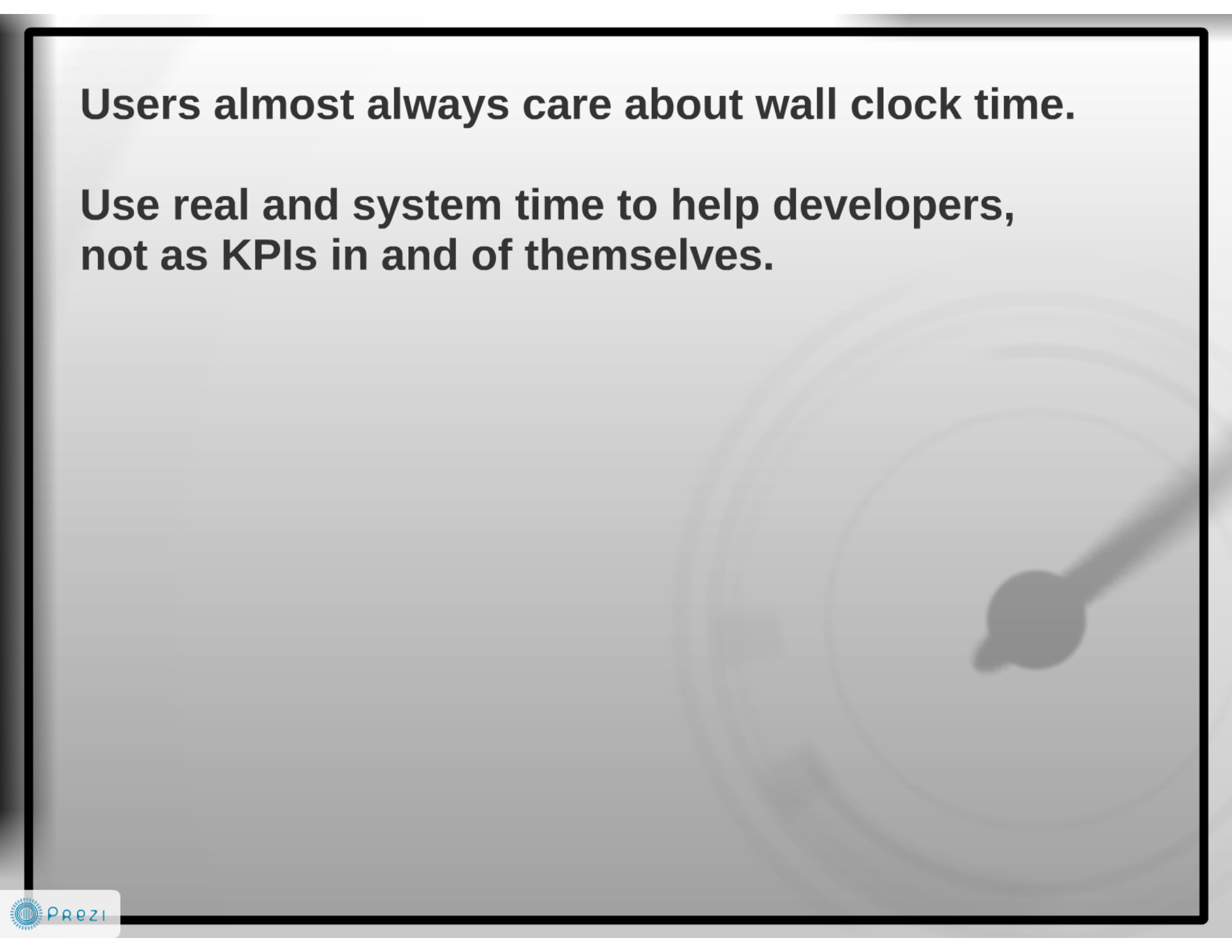
Ruby Benchmarking

Example

Web Response Time with Chrome

Users almost always care about wall clock time.

Use real and system time to help developers, not as KPIs in and of themselves.



Response time is easy to test in a variety of ways. Determining what is a good TARGET is difficult, but there are some guidelines.

This ventures more into the realm of usability testing, but as rough guidelines:

The Three Key Time Limits

< 0.1 S : Response time required to feel that system is instantaneous
< 1 S : Response time required for flow of thought not to be interrupted
< 10 S : Response time required for user to stay focused on the application (and not go re-load Reddit)

-taken from *Usability Engineering* by Jakob Nielsen, 1993

Things haven't changed much since then!

Response Time Is Important!

According to Google, any amount of time over 400 ms for a page to load causes a drop-off in visitors.

People will choose websites (often subconsciously) based on a 250 ms difference in load times.

Optimizing response time is challenging, but can be very fun.

Algorithmic analysis, coding hacks, and usability tricks all come into play.

Measuring Availability

Availability - often referred to as uptime - how often can a user access the system when he/she expects?

Often talked about as a SLA (service-level agreement).

Number of 9's -> Five 9's = 99.999% uptime, Six 9's = 99.9999% uptime, etc.

How would you test this?

It's rather difficult to test directly. Waiting until you go into production and seeing how long it stays up is a suboptimal testing strategy on many levels.

You have to approach it from the side.

Determine values for:
Concurrency
Scalability
Throughput

Model the system with appropriate possibility for failure and appropriate load

Concurrency - How many users can access the system at any one time?

Scalability - Can this number increase, easily or automatically?

Throughput - How many events can happen in a given amount of time?

Some of these will be more important than others, depending on the system under test.

Determine these by:
Load testing - how many concurrent users can the system handle?

Kinds of load testing:
Baseline Test - A bare minimum amount of use, to provide a baseline
Soak / Stability Test - Leave it running for an extended period of time, usually at low levels of usage
Stress Test - I'm givin' 'er all she's got, Cap'n!

Building a Model

What does the average user do?

Oftentimes, this is the most difficult part of availability testing!
You can -
1. Make assumptions
2. Make a model
3. Get real data from similar apps or previous versions

For true availability numbers, also need to determine:

1. Likelihood of hardware failure
2. Likelihood of program-ending bugs
3. Planned maintenance

etc.

Even with all this work...
... things go wrong.

Amazon and other major providers often breach their SLAs and have to refund money to users.

Availability - often referred to as uptime - how often can a user access the system when he/she expects?

Often talked about as a SLA (service-level agreement).

Number of 9's -> Five 9's = 99.999% uptime, Six 9's = 99.9999% uptime, etc.

How would you test this?

It's rather difficult to test directly. Waiting until you go into production and seeing how long it stays up is a suboptimal testing strategy on many levels.

You have to approach it from the side.

Determine values for:

Concurrency

Scalability

Throughput

Model the system with appropriate possibility for failure and appropriate load

Concurrency - How many users can access the system at any one time?

Scalability - Can this number increase, easily or automatically?

Throughput - How many events can happen in a given amount of time?

Some of these will be more important than others, depending on the system under test.

Determine these by:

Load testing - how many concurrent users can the system handle?

Kinds of load testing:

Baseline Test - A bare minimum amount of use, to provide a baseline

Soak / Stability Test - Leave it running for an extended period of time, usually at low levels of usage

Stress Test - I'm givin' 'er all she's got, Cap'n!

Building a Model

What does the average user do?

Oftentimes, this is the most difficult part of availability testing!

You can -

1. Make assumptions
2. Make a model
3. Get real data from similar apps or previous versions

For true availability numbers, also need to determine:

1. Likelihood of hardware failure
2. Likelihood of program-ending bugs
3. Planned maintenance

etc.

Even with all this work...

... things go wrong.

Amazon and other major providers often breach their SLAs and have to refund money to users.

Writing a Service-Oriented Performance Test Plan

Think from a user's and an admin's perspective!

How fast do I expect this to be?
What things matter to me, speedwise?
How often do I expect this to be available?
How many users do I expect?
Are large variances in response time allowed?
Do I expect the number of users to fluctuate greatly?

From there -
Determine Key Performance Indicators

Example:
- Average page load time
- Max page load time
- Number of concurrent users where page load time < 5 seconds

Use a test server!

Do not do this on production hardware except for minor checks!

This test server should be as similar as possible across tests (don't upgrade OS, don't install new programs, etc.)

Better yet, use VirtualBox or something similar to make a virtual image you can always reload.

Determine targets and thresholds for the KPIs.

Targets are what you are shooting for; thresholds are the bare minimum.

E.g. Average page load time:
Target: 1 S
Threshold: 5 S

Maximum concurrent users:
Target: 10,000
Threshold: 1,000

Write up a test plan.

Remember that all response time tests should happen numerous times.

Save this data! You may want to do further analysis on it later!

Load time tests don't have to happen numerous times. The fact that they're running for a long time tends to smooth out wrinkles.

Remember that systems should be as stable as possible! Define these in your test plan.

Keep track of all data, as much as possible.

Be as realistic as you can.

If you do make assumptions, note them!

Most importantly..

HAVE A PLAN

What if performance requirements aren't met?
What if they can't be?
What if they can be, but at a high cost?
etc.

Think from a user's and an admin's perspective!

How fast do I expect this to be?
What things matter to me, speedwise?
How often do I expect this to be available?
How many users do I expect?
Are large variances in response time allowed?
Do I expect the number of users to fluctuate greatly?

From there -
Determine Key Performance Indicators

Example:
- Average page load time
- Max page load time
- Number of concurrent users where page load time < 5 seconds

Use a test server!

Do not do this on production hardware except for minor checks!

This test server should be as similar as possible across tests (don't upgrade OS, don't install new programs, etc.)

Better yet, use VirtualBox or something similar to make a virtual image you can always reload.

Determine targets and thresholds for the KPIs.

Targets are what you are shooting for; thresholds are the bare minimum.

E.g. Average page load time:
Target: 1 S
Threshold: 5 S

Maximum concurrent users:
Target: 10,000
Threshold: 1,000

Write up a test plan.

Remember that all response time tests should happen numerous times.

Save this data! You may want to do further analysis on it later!

Load time tests don't have to happen numerous times. The fact that they're running for a long time tends to smooth out wrinkles.

Remember that systems should be as stable as possible! Define these in your test plan.

Keep track of all data, as much as possible.

Be as realistic as you can.

If you do make assumptions, note them!

NEXT LECTURE:

Efficiency-Oriented Performance Testing

A Final Note

It is possible for responses to be too fast! Think of a scroll box that scrolls so fast the user can't see what's happening.

O brave new world, that hath such computational power in it!

Most importantly..

HAVE A PLAN

What if performance requirements aren't met?
What if they can't be?
What if they can be, but at a high cost?
etc.

CS1699: Lecture 20 - Performance Testing Part 1