Continuous Performance Engineering?
How to move fast without breaking things

Velocity 2017

Thomas Barns, Capacitas
John Pillar, Arcadia Group
Why is this so important?
We live in …
Rapid delivery of change is critical to business survival

Rate of Change
Immediacy is the expectation for performance

- Do they browse as many pages?
- Do they spend as much money?
- Do they come back?
But the increasing rate of change means more risk of breaking performance…

- No time available for traditional performance testing
- Problems released can remain dormant until next peak
- There is a high cost to resolution of issues in production
How have we tried to deliver fast?

(while also maintaining performance…)}
Shift left: do things early

Performance engineering built into each and every step

Design it right  Code it right  Performance test before build  Load test on build  Run a big regression load test…

Local development – pre code commit  CI/CD server
Continuous Performance Engineering

Look for the right stuff
Risk based approach
Shift left: Do things early
Collaborate: Do things together
Automate: get tests in the pipeline
Analyse: automatic sign-off
Manage what’s left

We now do this …

whilst not slowing down this
Our eco-system
Integrating perf tests into the pipeline is key
How do we automate performance analysis?

1. Identify metrics using a Service Metric Framework
2. Compare to thresholds
3. Compare to baselines
4. Match expected patterns
5. Rules engine to give pass/fail

Metrics:
- Cache Hits/Misses
- Queue
- Processor/Threads
- Response Time
- Queuing Time
- Time Outs Count/Rate
- Errors Count/Rate
- Arrival Rate/Count
- Occupancy
- Queue/Buffer Size
- Resource Locking
- Resource Demand
- Resource Evictions
- Resource consumed
- Processed Rate
## Latest Test Results

<table>
<thead>
<tr>
<th>Test Plan</th>
<th>Test Design</th>
<th>Test Run</th>
<th>Start Time</th>
<th>End Time</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sprint</td>
<td>Configuration Test</td>
<td>Config Test_13</td>
<td>10/10/2017 08:57:45</td>
<td>10/10/2017 09:12:40</td>
<td>Test Failed</td>
</tr>
<tr>
<td>Sprint</td>
<td>Configuration Test</td>
<td>Config Test_12</td>
<td>10/10/2017 08:31:42</td>
<td>10/10/2017 08:46:37</td>
<td>Test Passed</td>
</tr>
<tr>
<td>Sprint</td>
<td>Configuration Test</td>
<td>Config Test_11</td>
<td>09/10/2017 21:19:23</td>
<td>09/10/2017 21:34:18</td>
<td>Test Passed</td>
</tr>
<tr>
<td>Sprint</td>
<td>Configuration Test</td>
<td>Config Test_6</td>
<td>09/10/2017 10:33:15</td>
<td>09/10/2017 10:48:10</td>
<td>Test Failed</td>
</tr>
</tbody>
</table>

![JMeter Error Rate Graph](image)

*Patterns?*
What did we learn?
Educate and document
Give people the correct tools
Do things together

Bringing together expertise from software engineers and performance analysts

Smarter Engineering

He knows the NFRs
He understands performance antipatterns
He knows how to design good performance tests
He knows the data to analyse
He knows what to do with it
He puts it into the bigger picture of the business

He knows the functional requirements
He knows the system (he wrote it)
He probably designed it too
He knows the technology behind it inside out
He knows what’s going to change
He can optimise code quickly when he knows its impact
He’s going to deliver software that does the job
Be prepared for automation challenges!

CI build capacity

Docker automation problems

Right sizing the test load

Calibrating to reduce false positives
Live the dream

Don't be too ambitious

Get something working early and build on it.
Was it worth it?

- Cost of defects reduced
- Feedback loops are faster
Next Steps

We still need to:

- Improve collaboration
- Optimise the pipeline
- Improve and calibrate on analysis
- Replicate to other teams
THANK YOU