System testing of RabbitMQ

Practices + Tooling + Lessons Learned

Jack Vanlightly

@vanlightly
https://jack-vanlightly.com

Now
Principal Software Engineer at Splunk

Apache Pulsar & Apache BookKeeper (massively scalable event streaming/ messaging platform)

Previously
Staff Engineer at VMware
RabbitMQ core team member.

Performance Engineer at CloudAMQP

What do I mean by System Testing?

System Testing

"Determining how the software will behave in the real-world"

System Testing

Assess Performance

Assess Resiliency

Assess Correctness

By running **workloads / operations**

Under specific **conditions**

In various realistic **environments**

And we **measure, we analyze**

System Testing

Assess Performance

Metrics: throughput, latency, utilization, saturation, cost-of-ownership

Assess Resiliency

Handles overload
Copes with adverse conditions

Quality of service, SLIs, SLOs
Throughput, latency, availability
https://landing.google.com/sre/sre-book/chapters/service-level-objectives/

Assess Correctness

Properties

System Testing Objectives

System Testing Objectives

#1 Experiments, quick feedback cycle

- Make it easy for developers/testers to run experiments
- Increase the velocity of development by speeding up the feedback cycle
- Test early and often in the development cycle

System Testing Objectives

#2 Data-driven decision making

- Make important technical decisions on based on data
- Technology changes
  - Compilers change
  - Hardware changes
  - Cloud and kubernetes is changing
- Don’t rely on prior experience alone, base your decisions on data

System Testing Objectives

#2 Data-driven decision making

"Don't be macho. Make decisions while having real data on-hand and limit the damage of hubris."

https://sled.rs/perf.html (embedded database written in Rust)

System Testing Objectives

#3 Gain confidence, less surprises

- Know the lay of the land (the strengths, the weaknesses, where the dragons are)
- How the software copes with adversity
- How the software copes with overload

System Testing Objectives

#4 Faster response to customer issues

- More accurate reproduction of workloads.
- Learn more from customer engagements
- Happier customers

System Testing Objectives

#5 Compare different versions and/or configurations for performance regressions.

System Testing Objectives

#6 Ensure important properties hold under various conditions.

- Stress tests
- Chaos tests
- Upgrades/downgrades
- Migrations

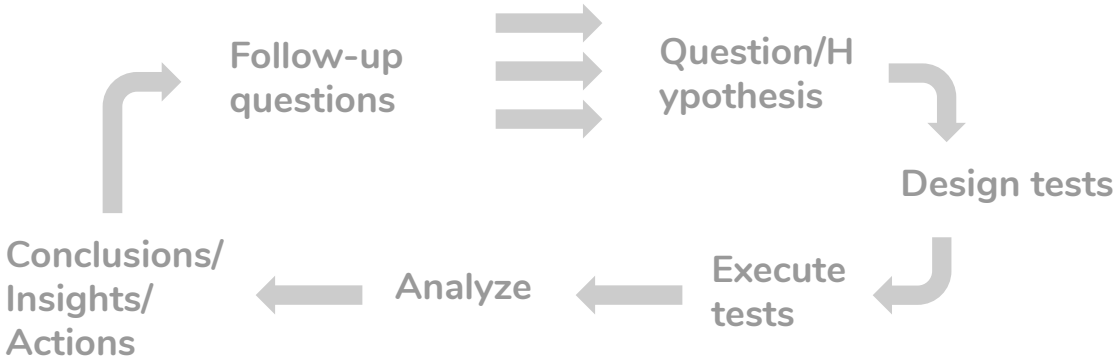
Practices

Question Driven Testing (or the scientific method)

Automated Exploratory Testing

Question Driven Testing (and the scientific method)

- When automation and workload generation is powerful, answering questions is easier.



/system-testing/practices/question-driven

Question Driven Testing (and the scientific method)

Case Study

New replicated queue type released called **Quorum Queues**. Main engineer tells me that he strongly recommends SSDs as performance is bad on HDDs.

Problem:

- Most customers are using HDDs
- We have no customer guidance beyond “don’t use HDDs”, because we don’t know much ourselves.

Question Driven Testing (and the scientific method)

Question 1: Do quorum queues perform as well as mirrored queues on HDDs?

Test: Run a few workloads, at different intensities, using the same hardware/OS/disk for mirrored and quorum queues.

Answer: Mixed. Single queue throughput is lower, but multiple queue throughput can be higher with HDDs. Latency is always a little higher with HDDs.

Question Driven Testing (and the scientific method)

But… quorum queues have a sequential disk IO pattern which suite HDDs well, whereas mirrored queues have a more random IO pattern.

Hypothesis: mixing random and sequential disk access on a single HDD will negatively impact the sequential workload.

Question Driven Testing (and the scientific method)

Question 2: Do quorum queues perform as well as mirrored queues, with a mixed workload that produces a lot of random IO?

Test: Run a suite of tests with a primary workload (either mirrored or quorum queue), with a second workload that is random IO intensive. Run the random IO workload from low, medium to high intensity.

Answer: Even low intensity random IO seriously impacts quorum queue performance. High intensity stops the quorum queue from functioning at all.

Question Driven Testing (and the scientific method)

But… isolating the different disk workloads is possible. RabbitMQ does not currently support this for quorum queues. Let's just get Ansible to mount an extra disk onto the quorum queue data directory to test this. If we get a good result, we will add this support in the next release.

Hypothesis: Isolating the random from sequential workload will allow quorum queues to perform adequately in a mixed workload scenario.

Question Driven Testing (and the scientific method)

Question 3: Can isolating mirrored and quorum queue data onto separate disks make quorum queue performance acceptable in a mixed workload scenario?

Test: Run same tests again, with a two disk configuration.

Answer: Mixed. Quorum queues perform adequately with a second low intensity random IO workload. But performance still degrades too far on high intensity random IO loads.

Question Driven Testing (and the scientific method)

Question 4: If random and sequential workloads are isolated then perhaps the issue is contention in the Erlang IO schedulers?

Let's add monitoring to capture that data and re run all experiments to find out.

...

Question Driven Testing (and the scientific method)

Final result:

1. Created customer guidance on usage of HDDs with quorum queues
2. Added multiple disk support in configuration.
3. Identified future work to make quorum queue performance better on HDDs.

Dimensional Testing

Dimensional Testing

- 1 dimension = 1 variable of the system
- Repeat a test again and again, changing the dimension
- Even the developers cannot always predict how the software will be impacted by a particular configuration, be it software or hardware.

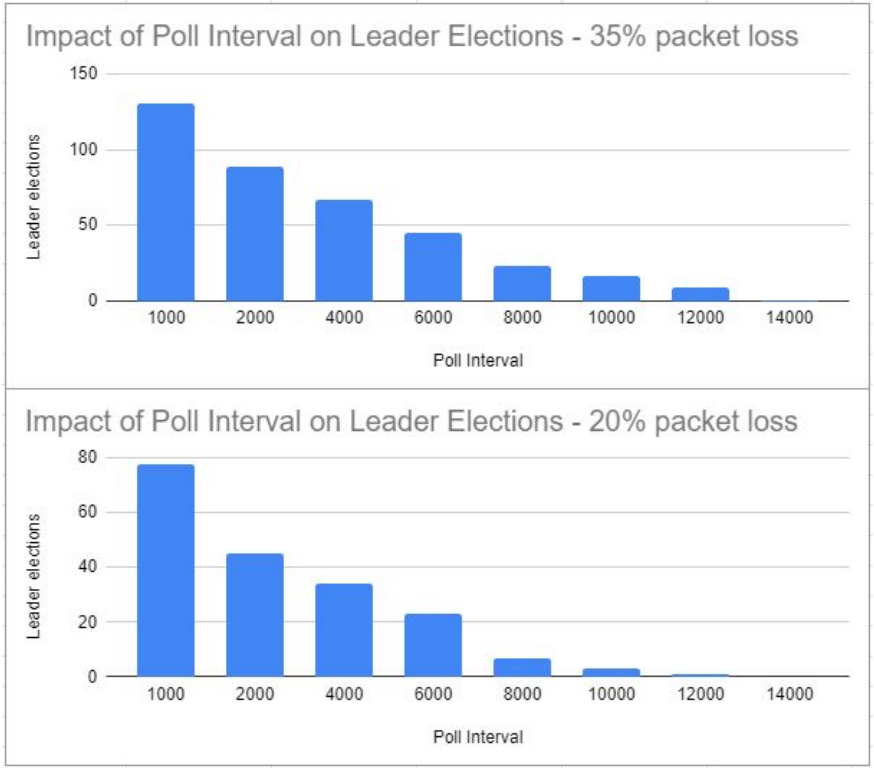
Dimensional Testing

Exploring impact of a server configuration value.

Case study 1

Measure leader election rate as a function of:

- Packet loss
- Failure detector sensitivity
  - Poll interval

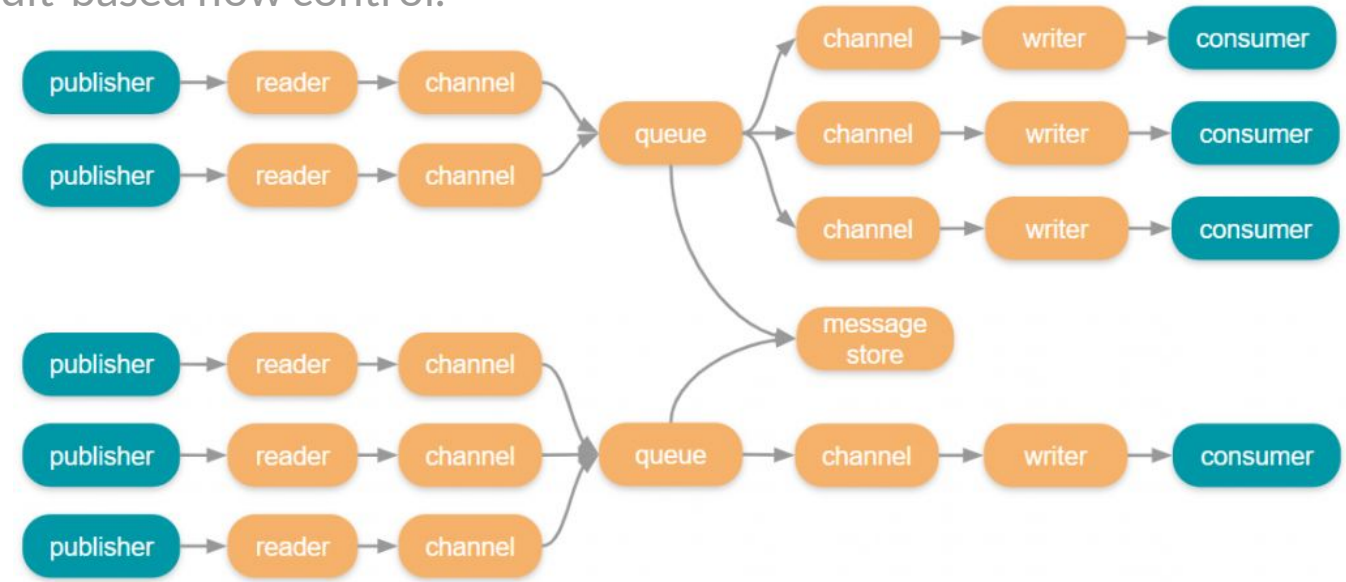


Dimensional Testing

Exploring impact of a server configuration value.

Case study 2

Credit-based flow control.

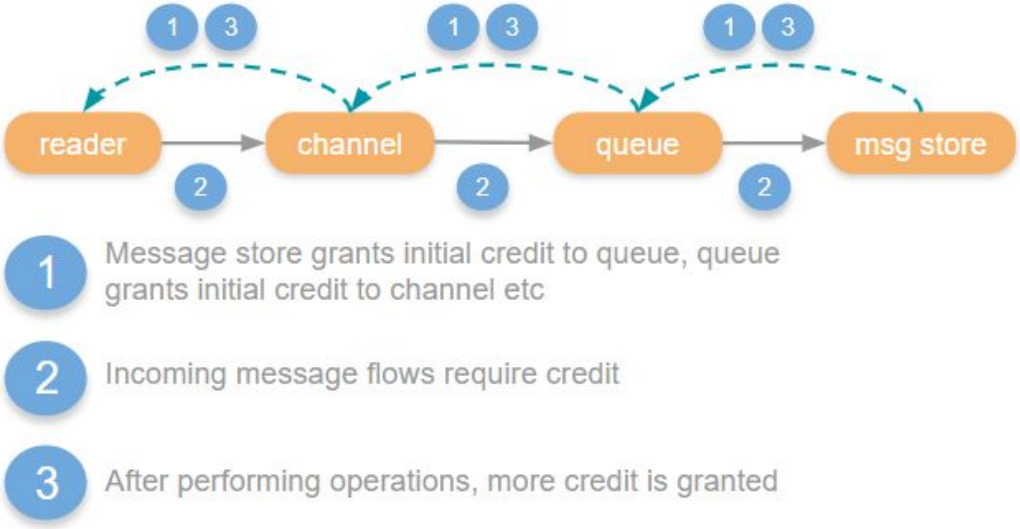


Dimensional Testing

Exploring impact of a server configuration value.

Case study 2

Credit-based flow control.



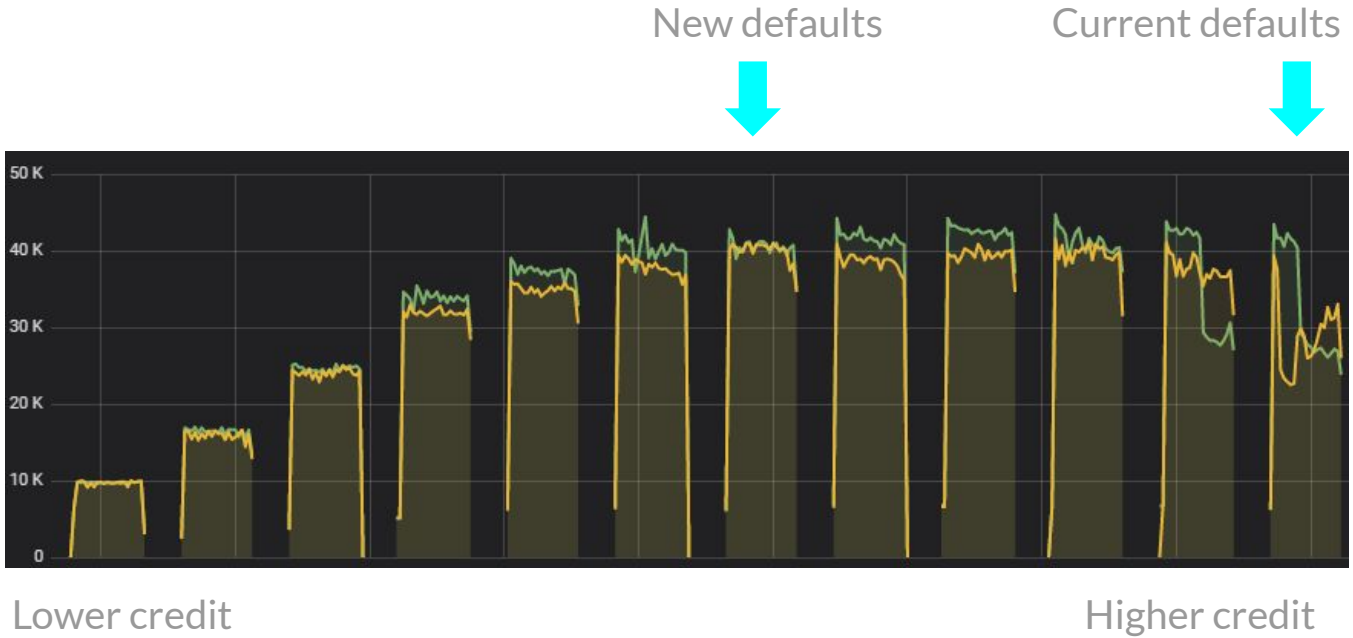
Dimensional Testing

Exploring impact of a server configuration value.

Case study 2

Credit-based flow control.

Run a stress test repeatedly, with different flow control settings.

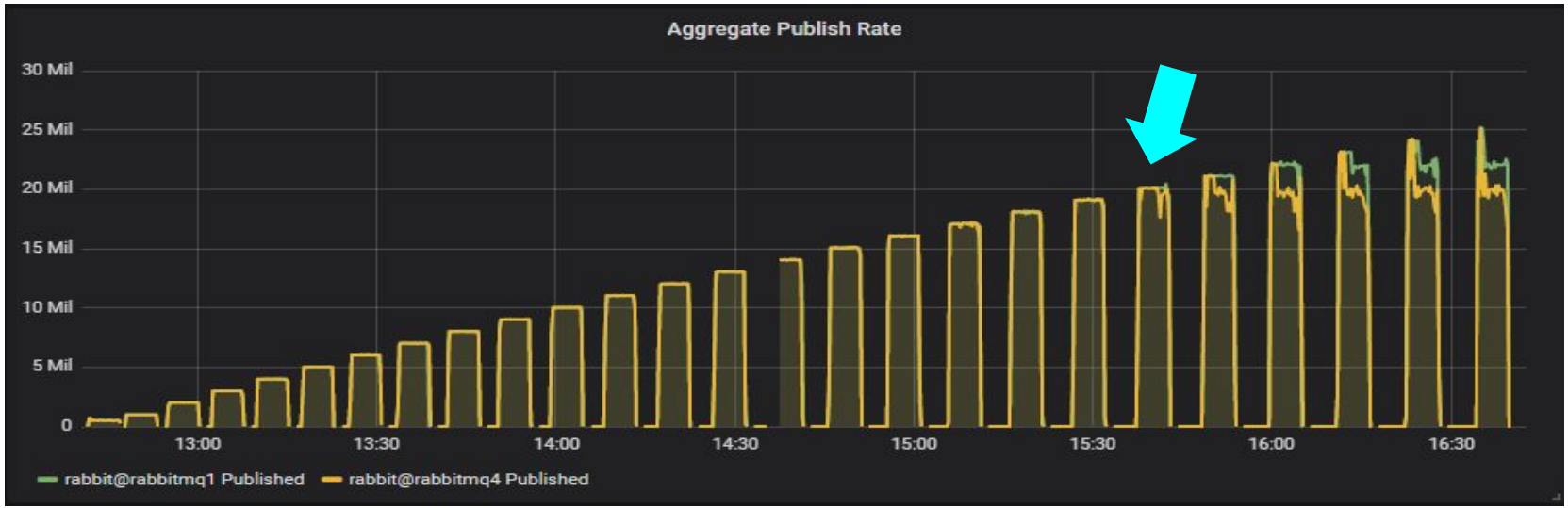


Dimensional Testing

Find the point when bottlenecks hit.

Case study 3

Do the new RabbitMQ streams scale out horizontally? Do we reach contention inside RabbitMQ before reaching hardware limits?

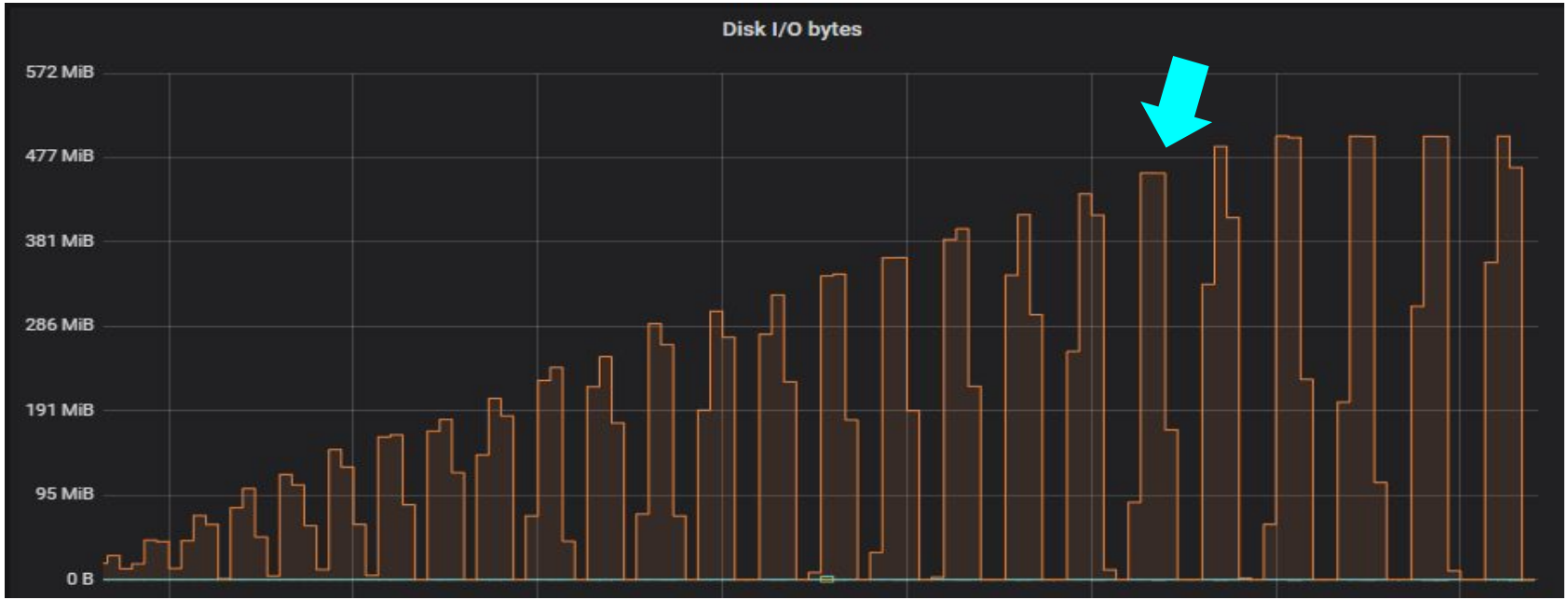


Dimensional Testing

Find the point when bottlenecks hit.

What did all the metrics look like before and after the bottleneck?

Good result, the bottleneck was the disk.



Dimensional Testing

Find the point when bottlenecks hit.

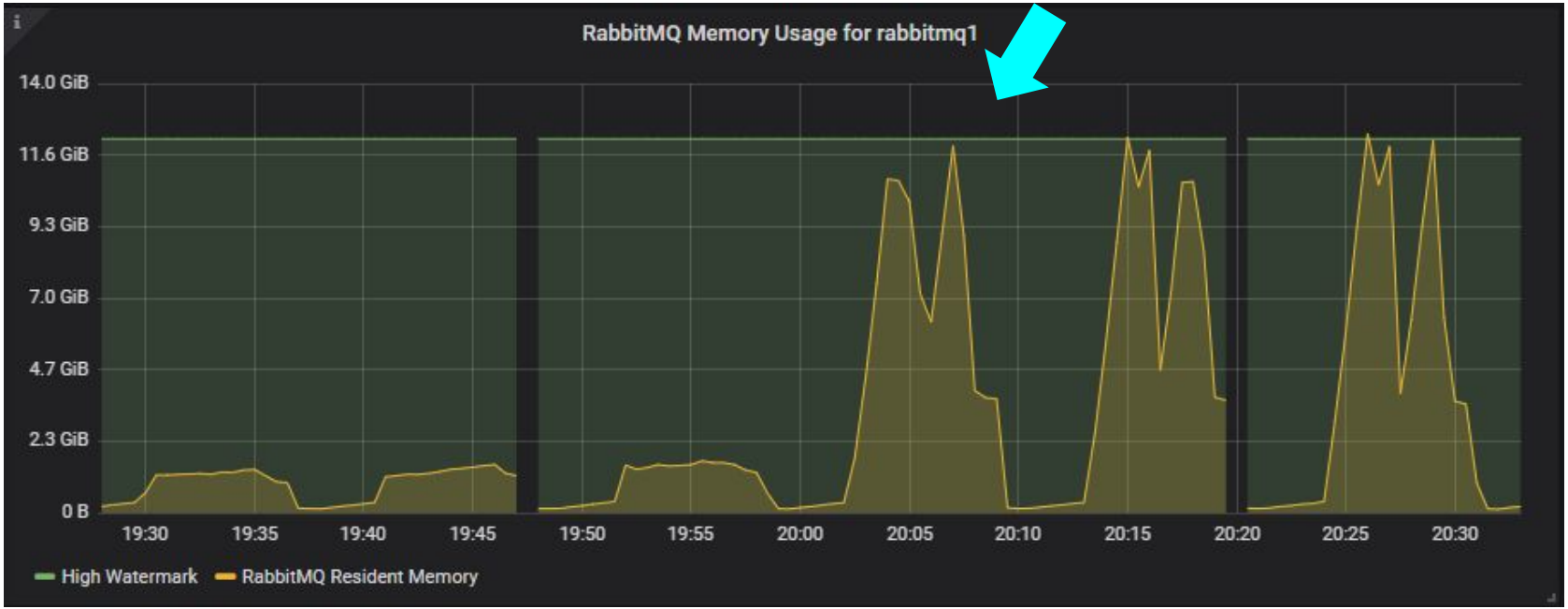
Case study 4

A customer reports scaling issues with their workload. We reproduce.

Dimensional Testing

Find the point when bottlenecks hit.

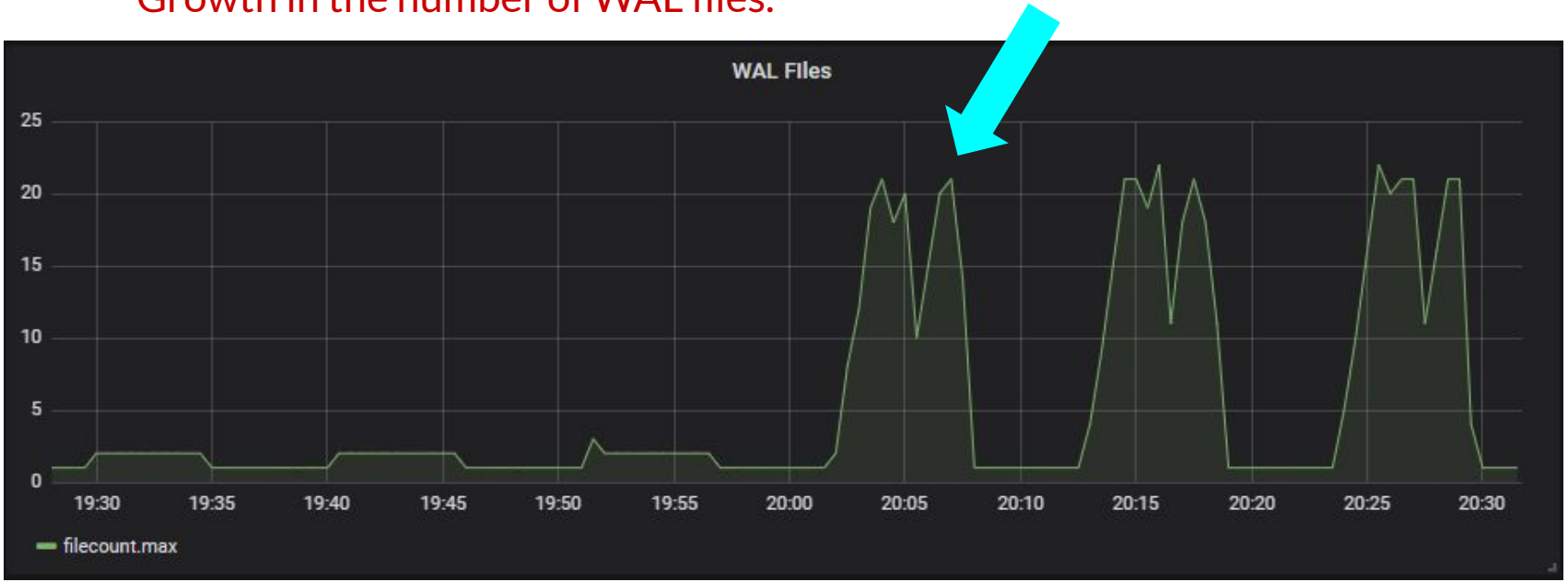
What did all the metrics look like before and after the bottleneck?
Memory pressure causes aggressive throttling of clients.



Dimensional Testing

Did something correlate to that memory jump?

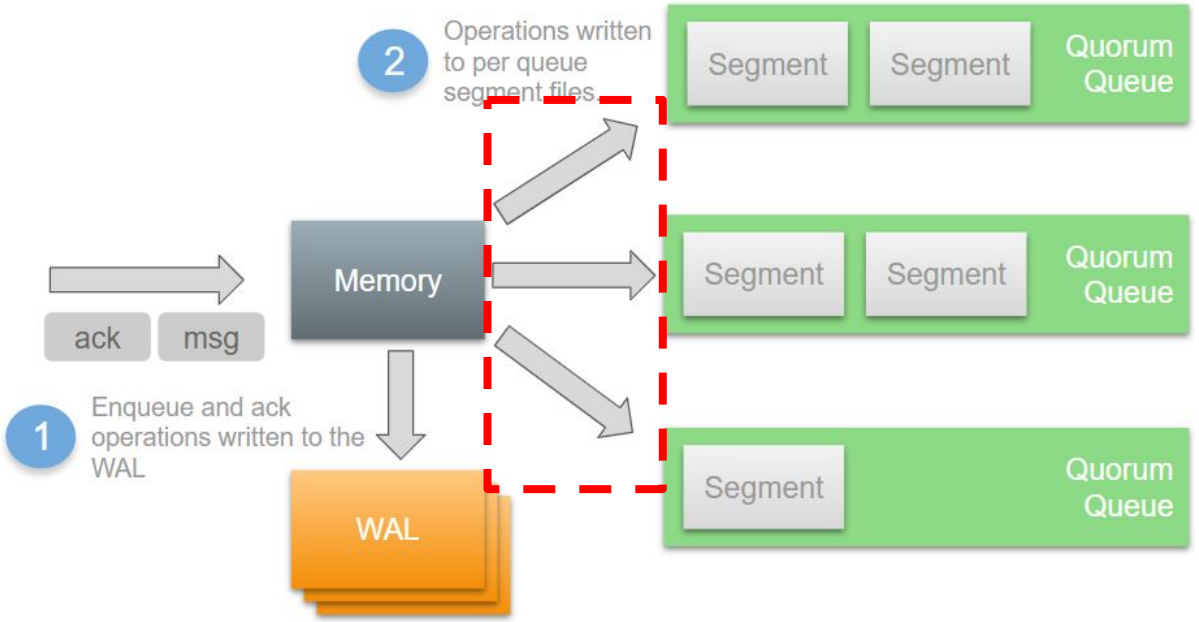
Growth in the number of WAL files.



Dimensional Testing

What do multiple WAL files mean?

Segment writer is the bottleneck.
Causes memory growth.



Dimensional Testing

- Fast Exploration
  - Find better defaults
  - Find problematic configurations
- Understand where breaking points or bottlenecks begin
  - Examine metrics before and after to find clues as to what the bottleneck is
- Find out where the Universal Scalability Law kicks in when scaling out/partitioning.
  https://blog.knoldus.com/understanding-laws-of-scalability-and-the-effects-on-a-distributed-system/

Comparison

Comparison over absolute values

We can handle 10k requests/sec on 3x16 vCPU, 32GB RAM, 250 MiB/s SSD with p99 latency under 100ms.

Good? or Bad? or Meh?

Comparison over absolute values

Compared parallel to single segment writer.

Good optimization

Compared alpha build to current production release

Regression

Comparison over absolute values

Compare a workload on the “system under test” to an “oracle” or “control”

Current production version (oracle)

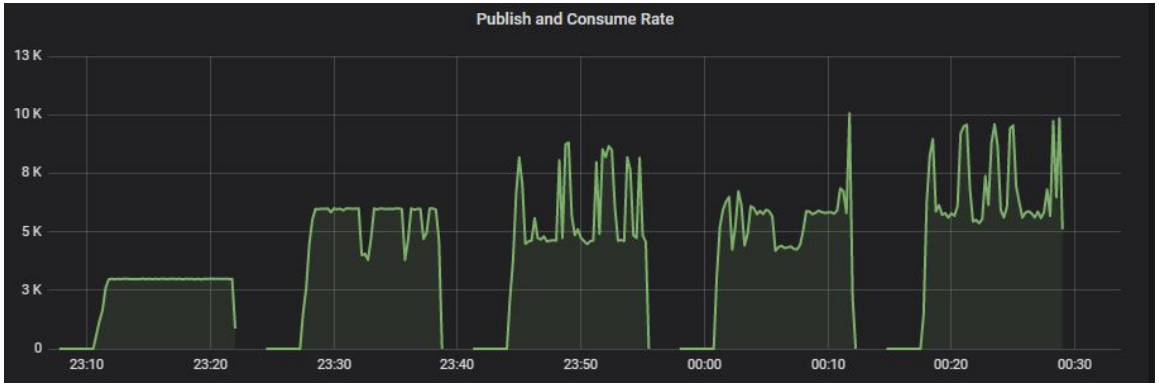
vs

Experimental build

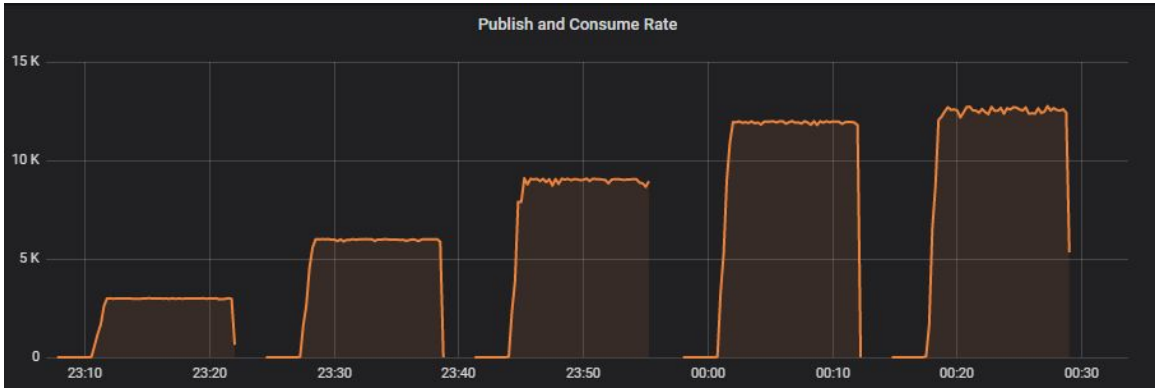
Comparison over absolute values

Comparison over absolute values

The Oracle

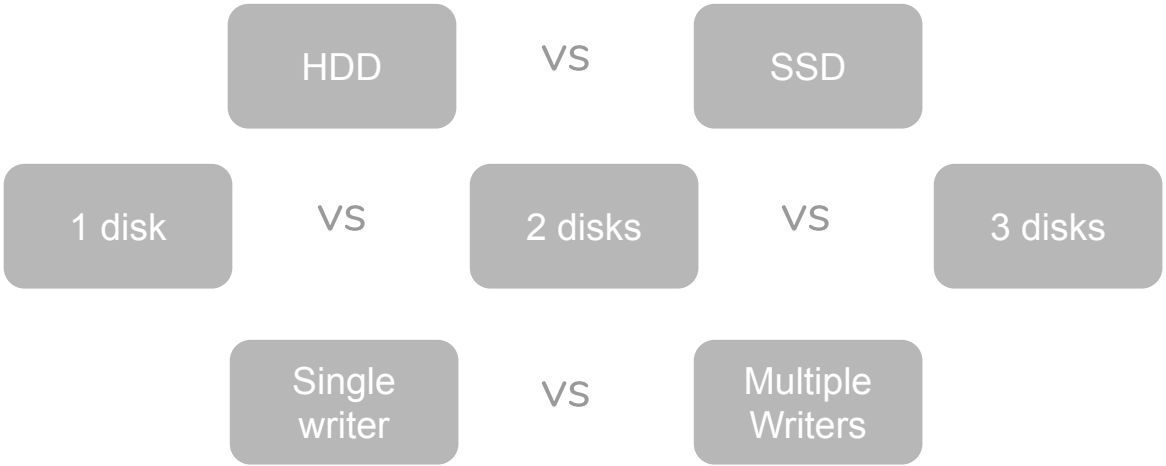


The experimental build

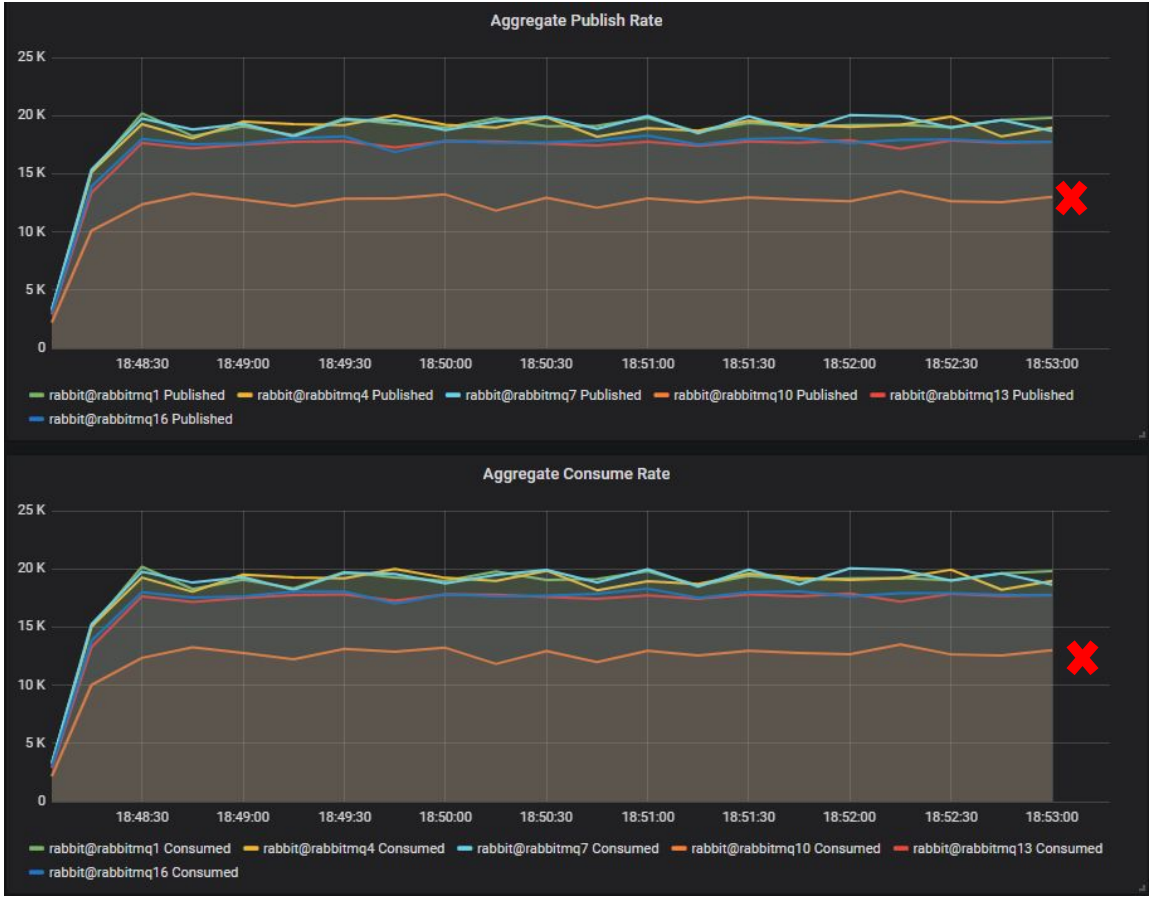


Comparison over absolute values

Compare a workload across different systems with slight differences (storage, OS, you name it…)



- rabbitmq1=ssd 1 disk
- rabbitmq4=ssd 2 disks
- rabbitmq7=ssd 3 disks
- rabbitmq10=hdd 1 disk ✖
- rabbitmq13=hdd 2 disks
- rabbitmq16=hdd 3 disks



Comparison over absolute values

Comparing and contrasting two slightly different workloads over identical systems.

Case study:

Virtual hosts

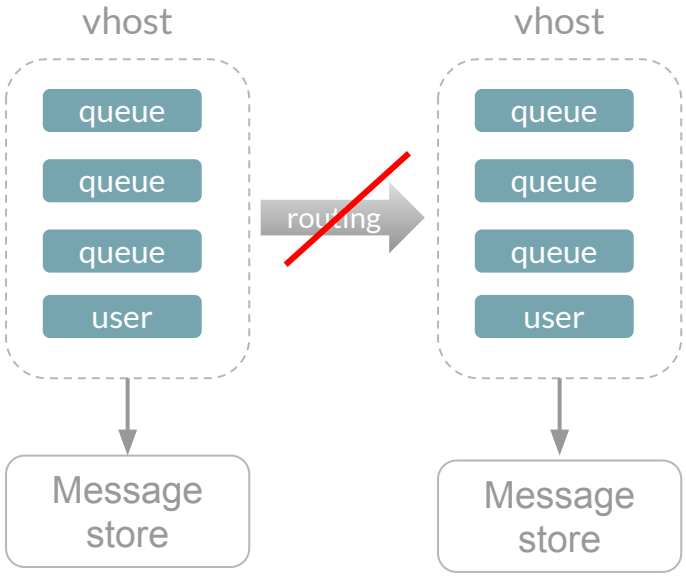
1 vHost,
10 Queues

vs

10 vHosts,
1 Queue per vHost

Comparison over absolute values

Comparing and contrasting two slightly different workloads over identical systems.



Comparison over absolute values

1 vHost,
10 Queues

vs

10 vHosts,
1 Queue per vHost

Q1: Does isolating queues into separate vHosts help throughput and latency? (Each vhost has its own message store)

Q2: Does recovery time improve with a higher vhosts-queue ratio?

Q3: Is there an upper limit on the number of vhosts? What kind of nastty behaviour can I cause by adding lots and lots of vhosts?

Variability

Dealing with variability

Variability is your enemy

- No-one wants a system that has wildly variable performance
- Variability in your results can trick you

Dealing with variability

How do we even know variability exists in the results?

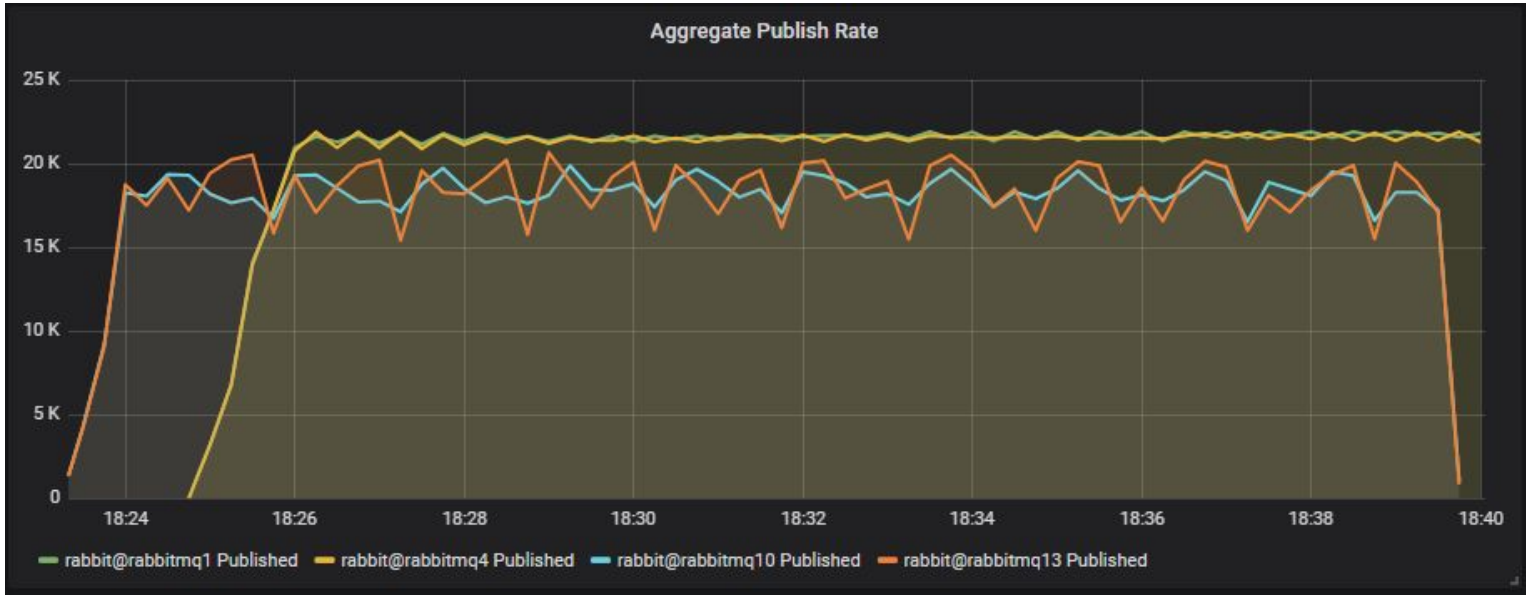
Without knowing the variability, how much confidence can you obtain from your results?

Dealing with variability

Variability can exist within a single workload instance

Dealing with variability

Variability can exist within a single workload instance



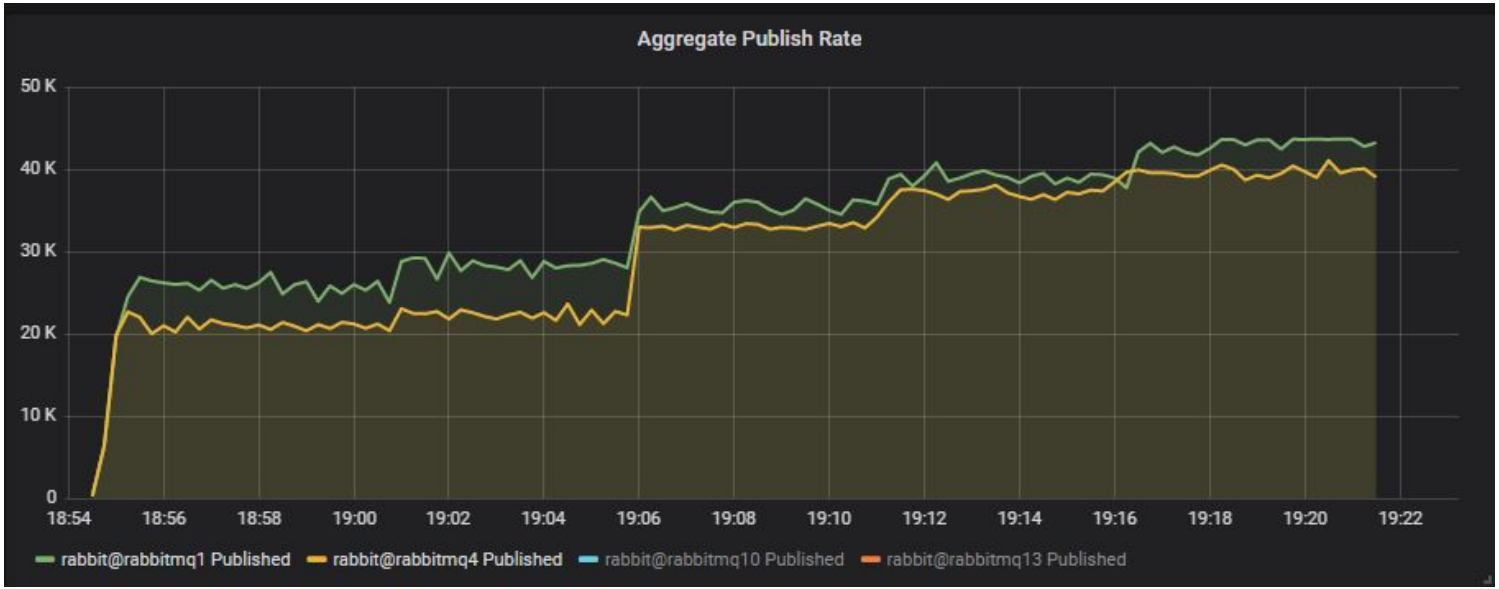
/system-testing/practices/variability

Dealing with variability

Variability can exist across multiple identical executions.

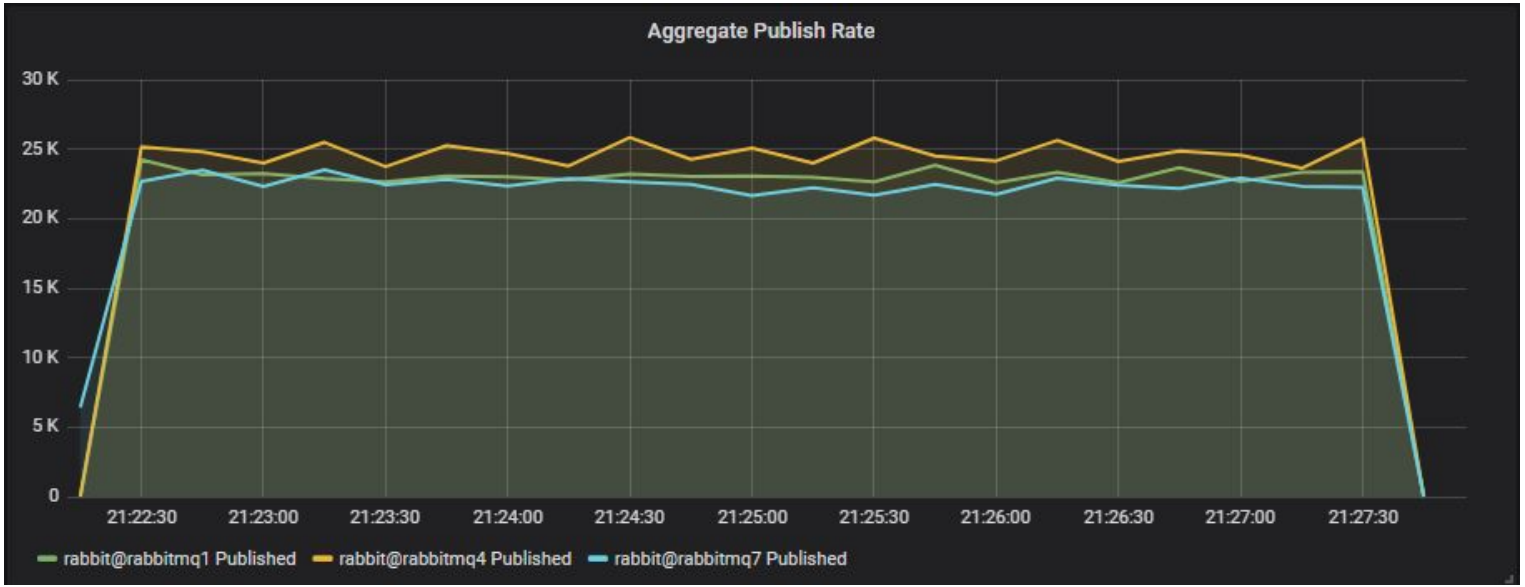
Dealing with variability

Variability can exist across multiple identical executions.



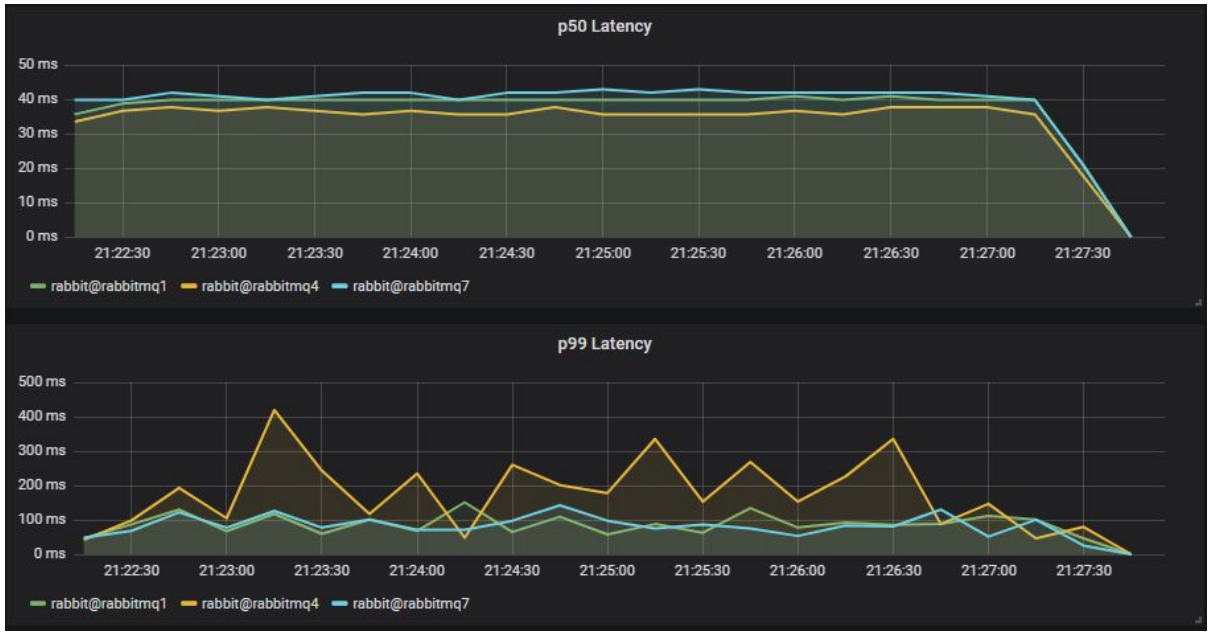
Dealing with variability

Variability can exist across multiple identical executions.



Dealing with variability

Variability can exist across multiple identical executions.



Reducing variability

Detecting variability

- Running tests multiple times
- Measuring variability within each run and across runs.

Reducing variability

Reduce variability in your environment:

- Avoid sharing infrastructure, including databases
- Isolation between tests
- Choice of hardware, OS configuration

Assess correctness and resilience

Assessing Correctness and Resilience

- How to assess resilience to adverse conditions?
- How to assess correctness?

Assessing Correctness and Resilience

- Metrics
  - Service Level Indicators (SLIs) against Service Level Objectives (SLOs)
    - Throughput
    - Latency
    - Availability

https://landing.google.com/sre/sre-book/chapters/service-level-objectives/

Assessing Correctness and Resilience

- When is it likely you would violate your performance objectives?
  - What are stress loads and what do they look like (learn to recognise a stressed system)
  - What component failures can cause violations?

- How can you mitigate those risks?

Assessing Correctness and Resilience

- Check properties (invariants), RabbitMQ examples:
  - Message loss
  - Message duplication
  - Message ordering

Assessing Resilience and Correctness

- Calculate connection availability time in clients
  - time clients are connected vs disconnected

- Calculate consumption availability time of consumers
  - time consumers are actively consuming messages vs sitting waiting for messages to arrive

Assessing Resilience and Correctness

Use cases:

- Needed by the RabbitMQ on Kubernetes team to verify that all operations are zero-downtime
- Testing upgrades do not cause data loss or unavailability
- Stress tests
- Chaos tests
- Long running tests

Tooling - Our needs

- Easy to run experiments
- Measure, measure, measure
- Easy to compare different versions, configurations, hardware
- Be able to measure and accommodate for variability
- Easy to interpret the results
- Assess resilience and correctness

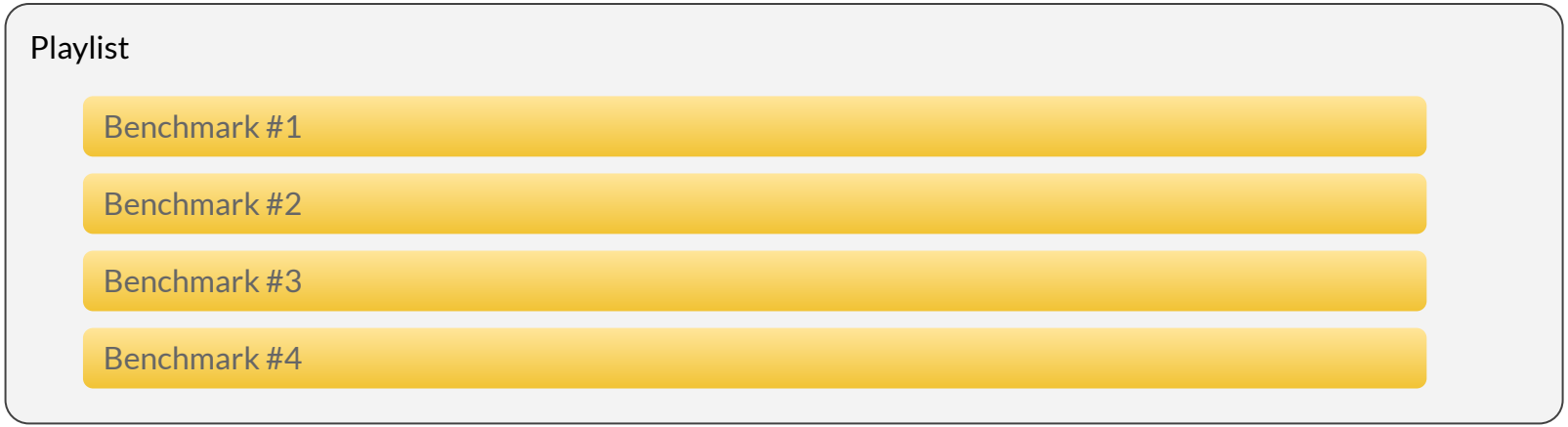
Tooling

Playlists, systems and benchmarks

Playlists, Systems, Benchmarks and Workloads

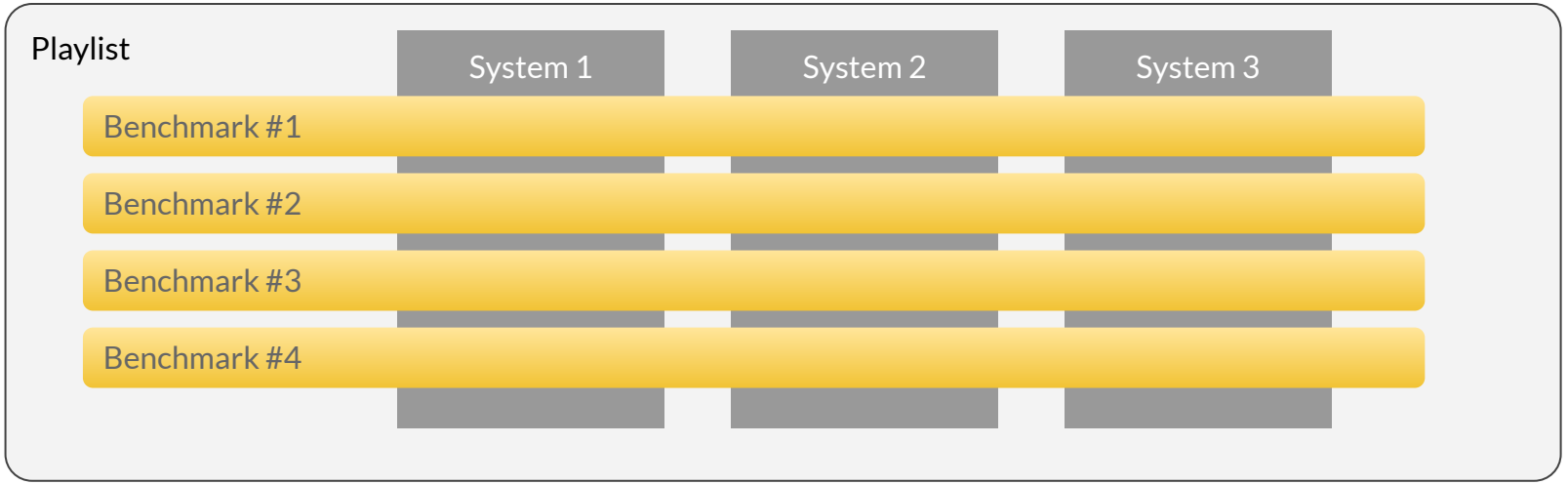
A playlist is a sequence of benchmarks that execute on one or more systems at the same time. Each playlist:

- Acts as a coherent grouping of benchmarks
- Can be run ad hoc or as part of a release process (still manual)



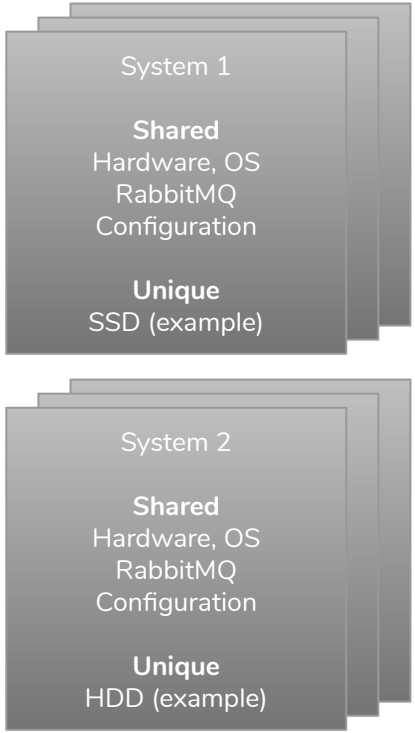
Playlists, Systems, Benchmarks and Workloads

A system is a deployment with a specific configuration. Many can be deployed at the same time.



Systems

- A system is a deployment unit with a specific configuration:
  - Hardware (CPUs, memory, drive size/type, network)
  - Host (EC2, GCP, EKS, GKE)
  - OS (Linux dist, configuration in case of IaaS)
  - RabbitMQ (version, Erlang version)
  - RabbitMQ configuration
  - Cluster size
- Each system can be deployed multiple times in order to get multiple results for the same configuration (for identifying variability and outlier results)



Systems

System 1	System 2
Shared Hardware, OS RabbitMQ Configuration **Unique** SSD	**Shared** Hardware, OS RabbitMQ Configuration **Unique** HDD

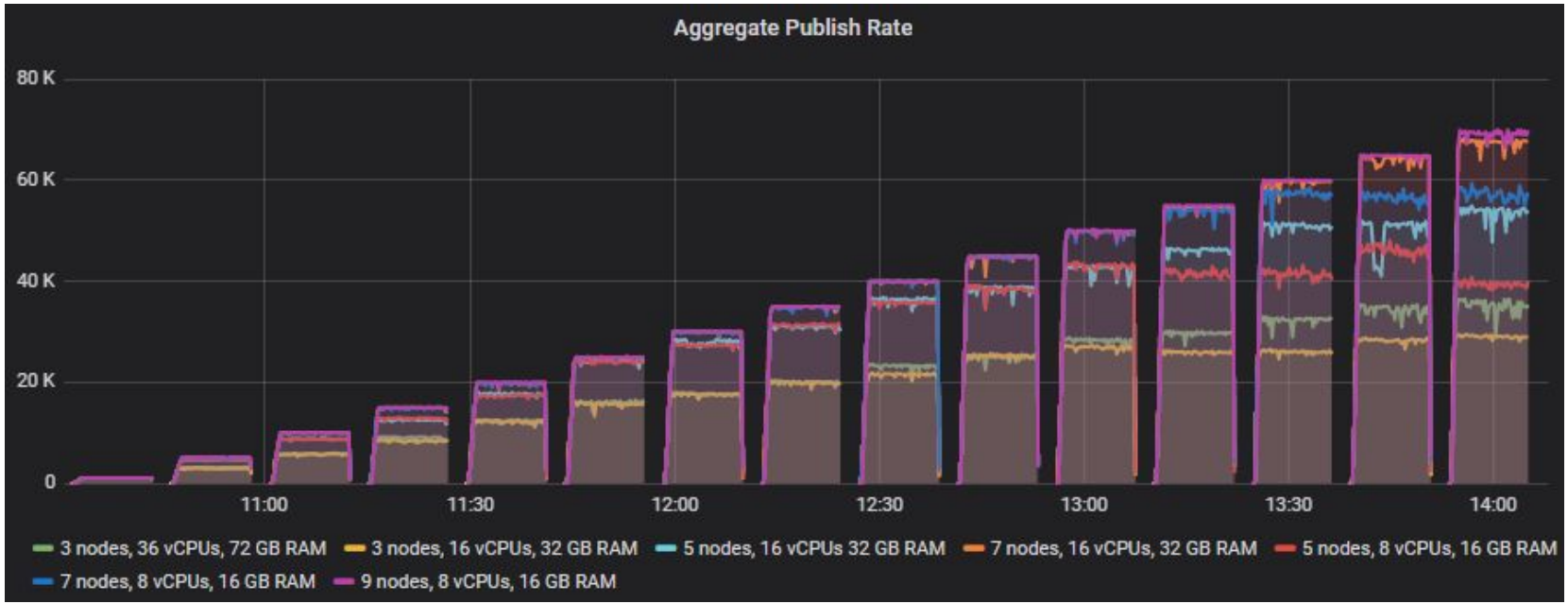
System 1	System 2
Shared Hardware, OS RabbitMQ Configuration **Unique** Config set to X	**Shared** Hardware, OS RabbitMQ Configuration **Unique** Config set to Y

System 1	System 2
Shared Hardware, OS RabbitMQ Configuration **Unique** 3.8.9	**Shared** Hardware, OS RabbitMQ Configuration **Unique** 3.8.10 alpha 1

System 1	System 2
Shared Hardware, OS RabbitMQ Configuration **Unique** 9 nodes	**Shared** Hardware, OS RabbitMQ Configuration **Unique** 11 nodes

Systems

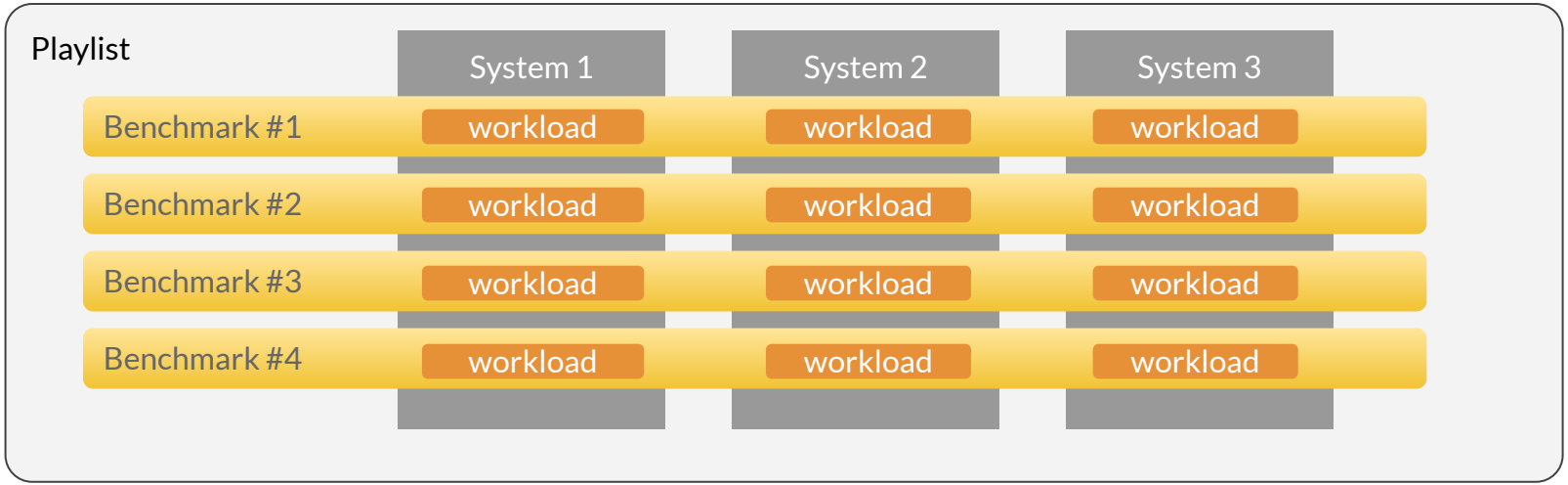
7 system cluster sizing test. Increasing throughput.



/system-testing/tooling/playlists

Playlists, Systems, Benchmarks and Workloads

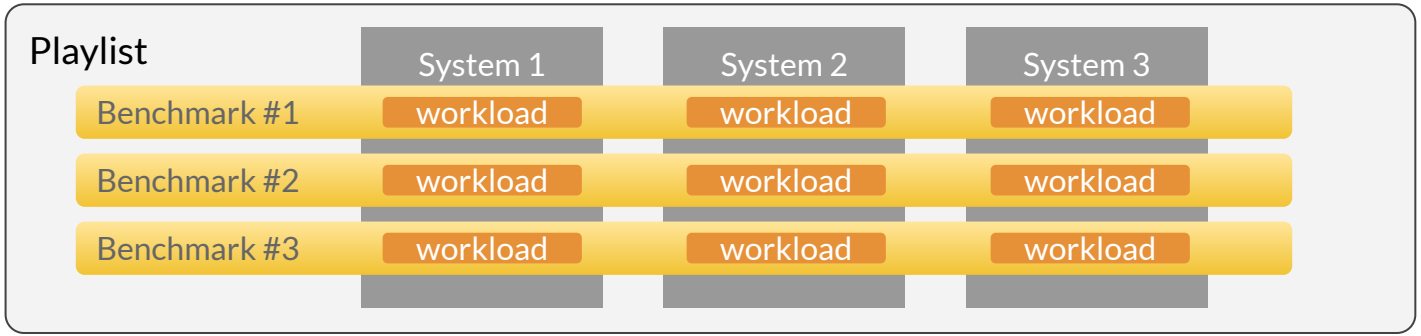
A benchmark is a monitored/recorded workload that is applied at the same time to each system independently.



Common/unique workloads and systems

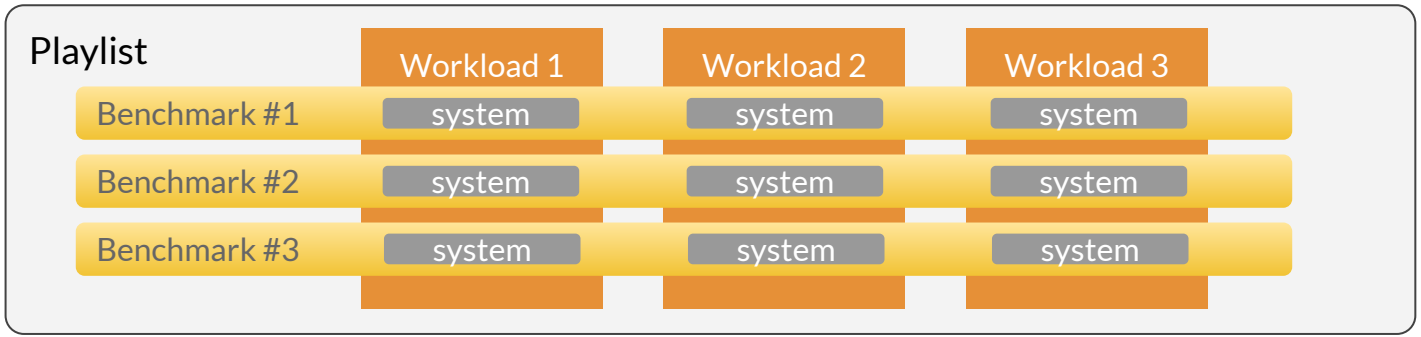
Unique systems

Common workload



Common systems

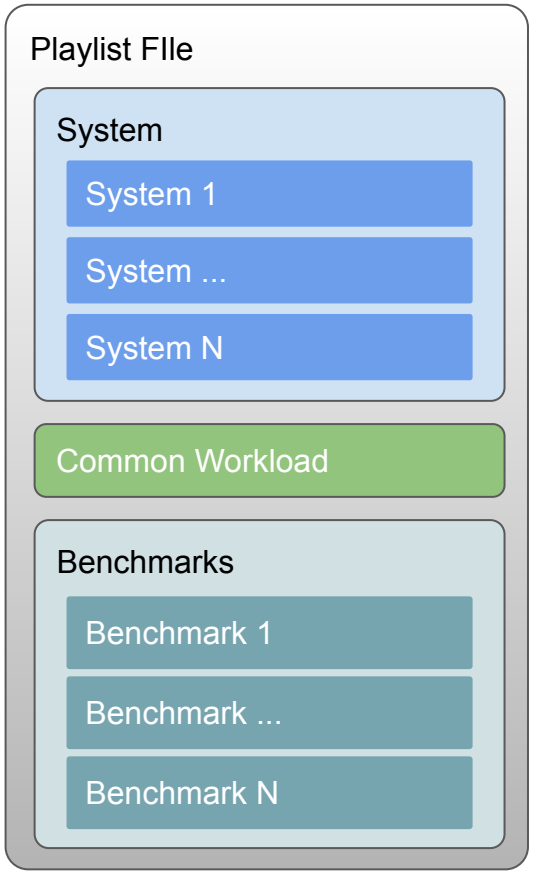
Unique workloads



/system-testing/tooling/playlists

Playlist Files

1. YAML based API
2. Host specific deployers



/system-testing/tooling/playlists

Playlists

Playlist file

```yaml
name: test-ec2-playlist
run: 1
systems:
  - name: s1
    host: ec2
    file: standard-8-vcpu-latest.yml
    overrides:
      hardware:
        rabbitmq:
          volume-config: 1-gp2-1000
          count: 3
  - name: s2
    host: ec2
    file: standard-8-vcpu-latest.yml
    overrides:
      hardware:
        rabbitmq:
          volume-config: 1-io1-200
          count: 3
```

```yaml
common-workload:
  main:
    topology:
      file: point-to-point/point-to-point-safe.json
    policies:
      file: mirrored-queue.json
      variables:
        ha-mode: exactly
        ha-params: 2
    step-seconds: 600
  loadgen-config:
    mode: benchmark
    warm-up-seconds: 60
benchmarks:
  - benchmark:
      - workload:
          main:
            topology:
              variables:
                queues: 1
                consumers: 1
                publishers: 1
  - benchmark:
      - workload:
          main:
            topology:
              variables:
                queues: 2
                consumers: 2
                publishers: 2
```

Playlists

System file

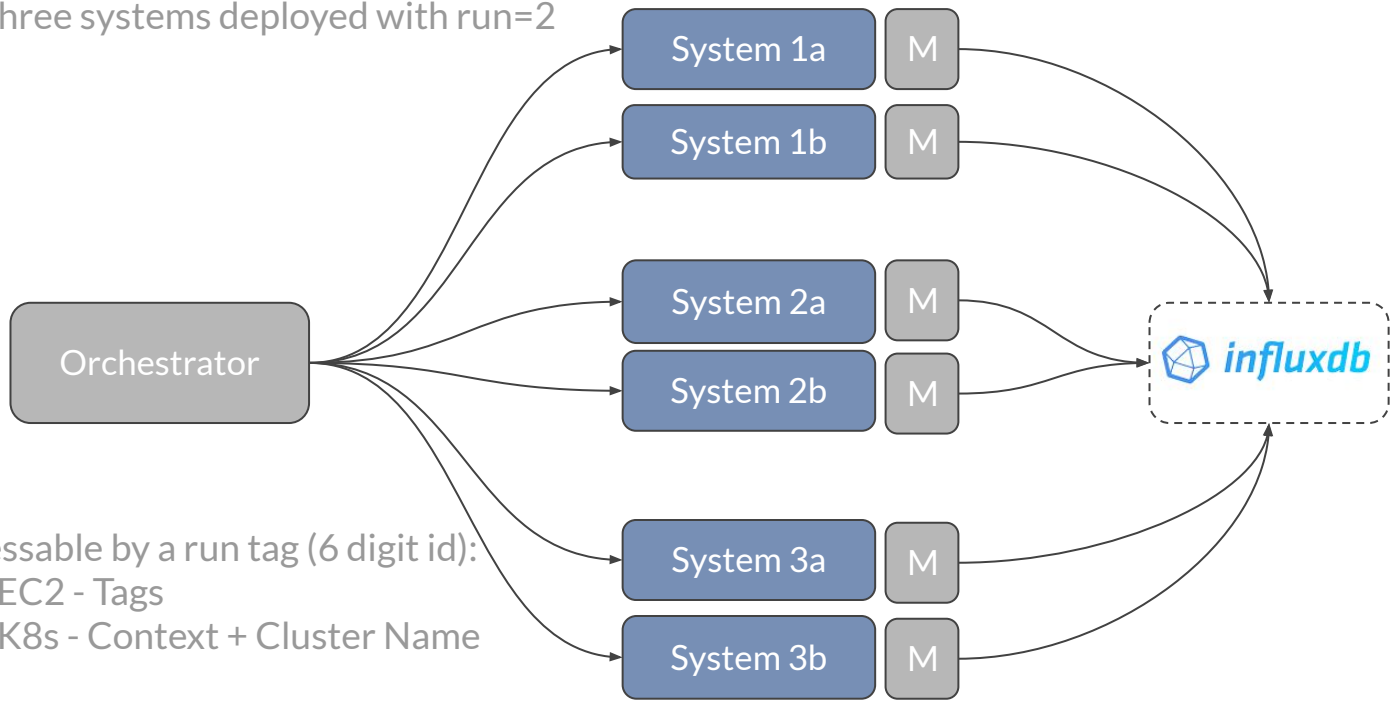
```
name: standard-8-vcpu-latest
host: ec2
hardware:
  loadgen:
    instance: 8-core-intel
  rabbitmq:
    instance: 8-core-intel
    volume-config: 1-gp2-200
    count: 3
rabbitmq:
  broker:
    version: 3.8.8
    generic-unix-url: https://github.com/rabbitmq/rabbitmq-server/releases/download/v3.8.8/rabbitmq-server-generic-u
  erlang:
    version: 23.0.3
    deb-url: https://packages.erlang-solutions.com/erlang/debian/pool/esl-erlang_23.0.3-1~ubuntu~bionic_amd64.deb
  config:
    file: pause-minority-debug.yml
```

Tooling

Orchestration and Observability

Step 1 - Deploy

Three systems deployed with run=2



Addressable by a run tag (6 digit id):

- EC2 - Tags
- K8s - Context + Cluster Name

Architecture - Step 1 - Deploy

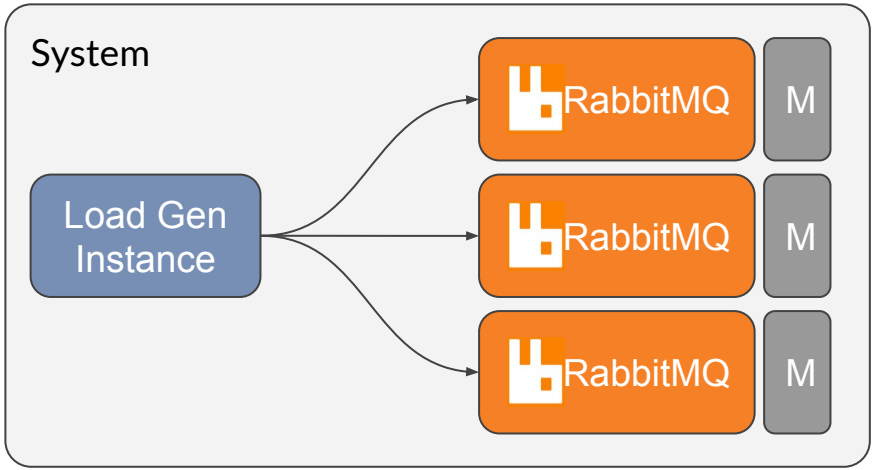
EKS > Clusters

Clusters (4) Info

Find clusters by name

Cluster name	Kubernetes version
benchmarking-eks-s1a-428437	1.17
benchmarking-eks-s1b-428437	1.17
benchmarking-eks-s2a-428437	1.17
benchmarking-eks-s2b-428437	1.17

Step 1 - Deploy



Monitoring:

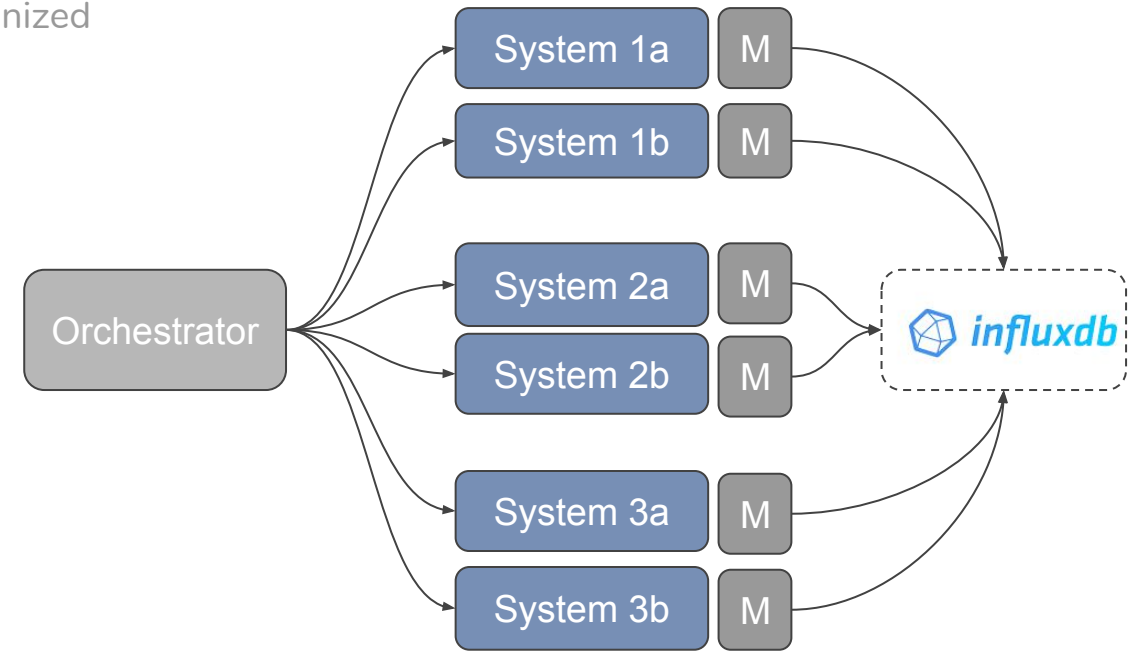
- LoadGen - Micrometer (Java)
- RabbitMQ has the Prometheus plugin (exposes /metrics)
- IaaS - Telegraf service
- K8s - Telegraf side-car

Step 2 - Run

Execute workload on all systems, synchronized

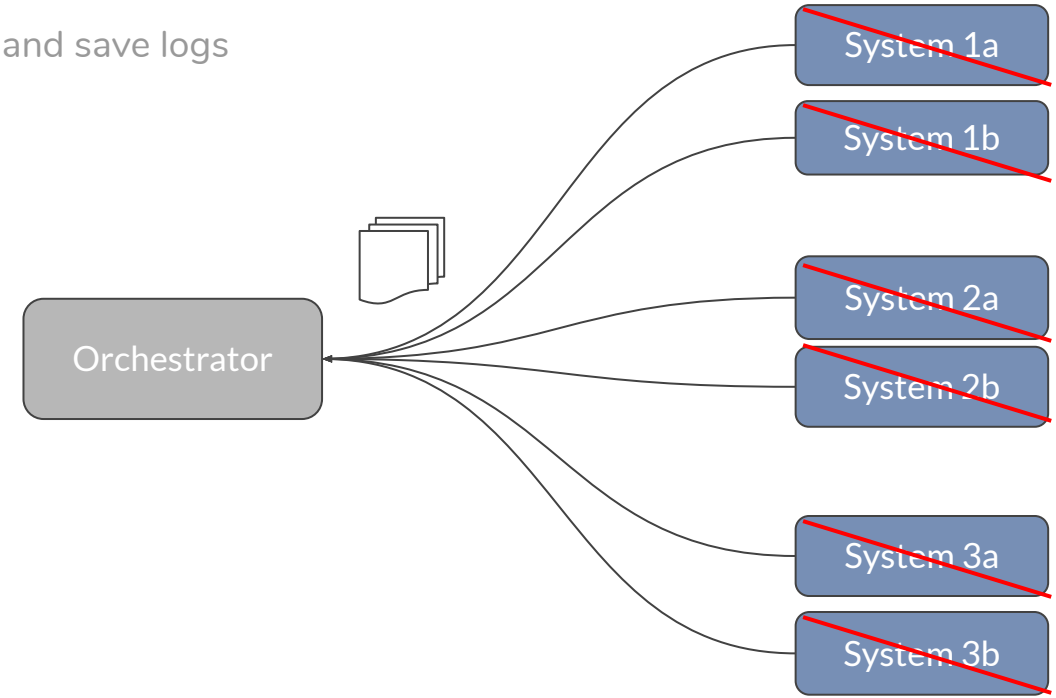
For each benchmark in the playlist:

- Deploy workload generation artefacts
- Apply any network conditions
- Apply any configuration changes
- Kick-off Java loadgen tool in each system
- Wait for all to complete

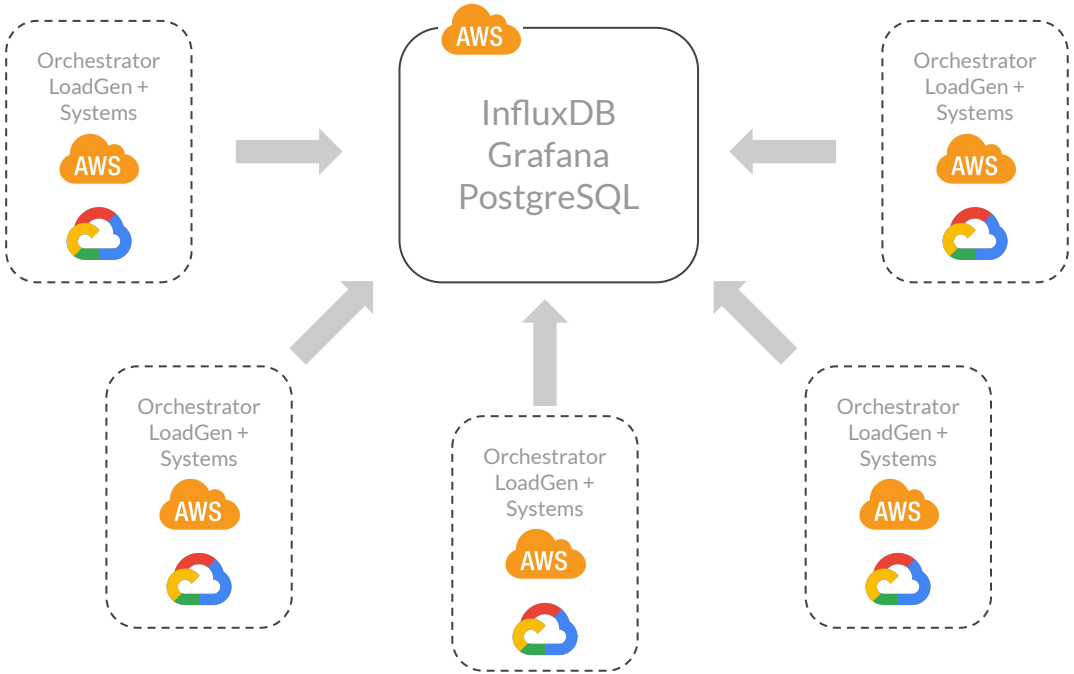


Step 3 - Teardown

1. Gather, compress and save logs
2. Destroy instances



Permanent and Ephemeral



Tooling

History and Analysis

Rabbit Test Tool

History and repeatability

- Every test configuration recorded in PostgreSQL
- All binaries sourced from Github, Bintray or S3.
- Any test can be rerun with identical configuration and versions
  - Same configuration
  - Same hardware
  - Same binaries (even experimental builds)

Rabbit Test Tool

Analysis

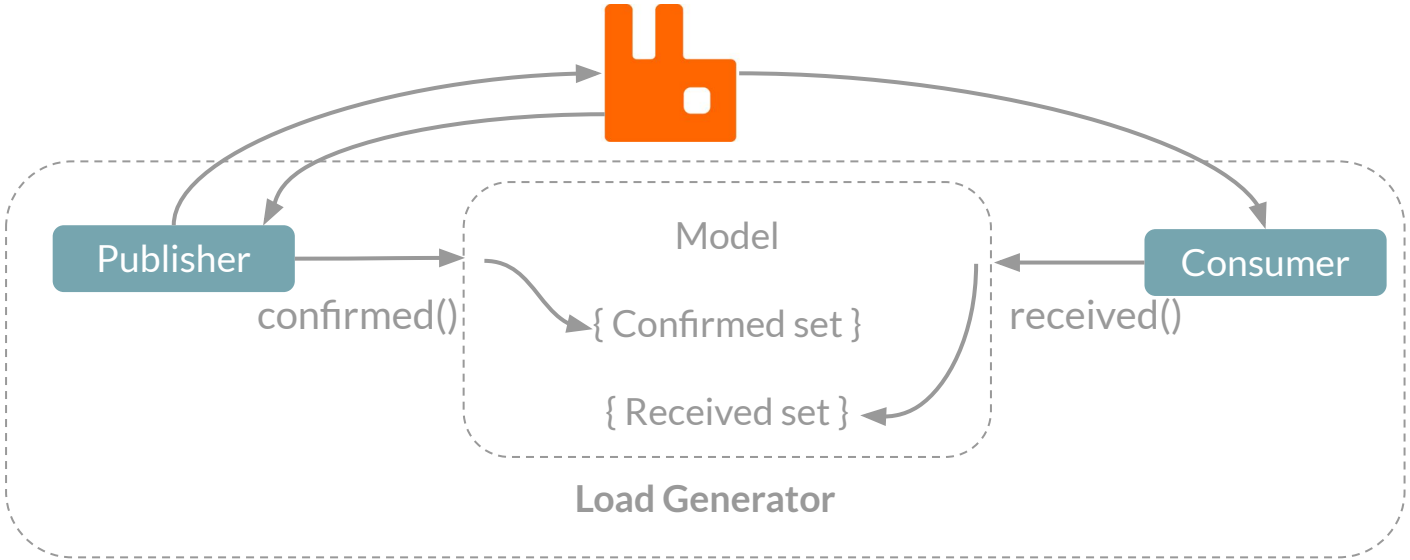
- InfluxDB, Grafana
  - Visualization for humans
  - Data mining

- Statistical analysis
  - Measure variance
  - Perform comparisons (incl regression detection)

- Correctness/Resilience analysis:
  - Model driven mode

Model-driven Property Based Test Mode

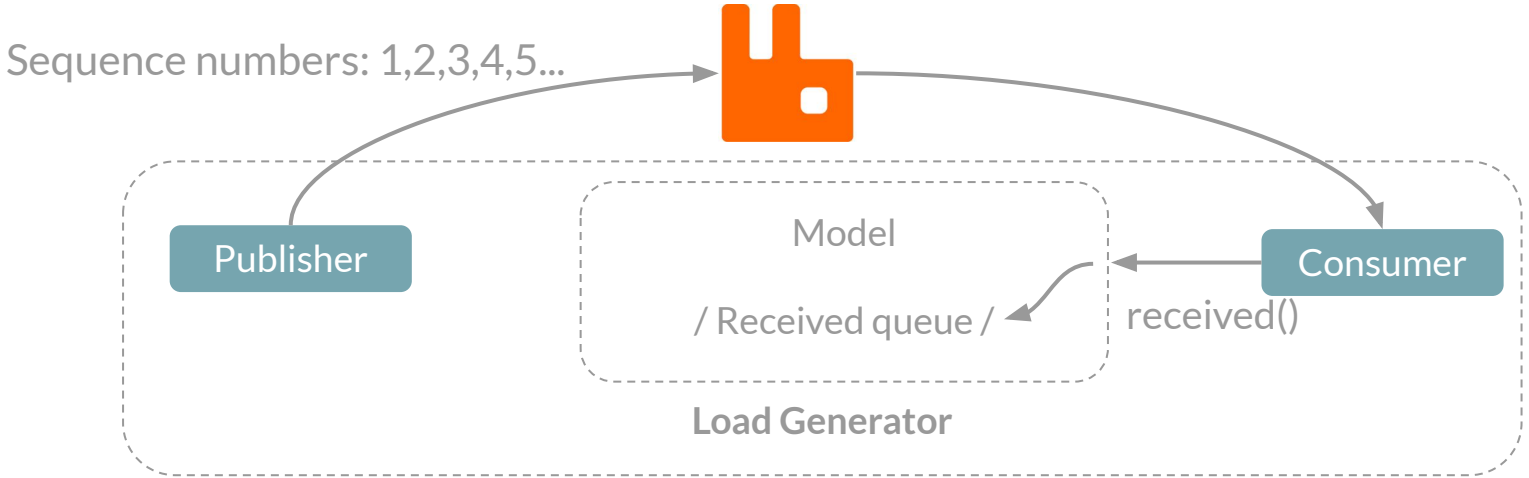
Detecting message loss



{ Confirmed } /difference { received } = { lost }

Model-driven Property Based Test Mode

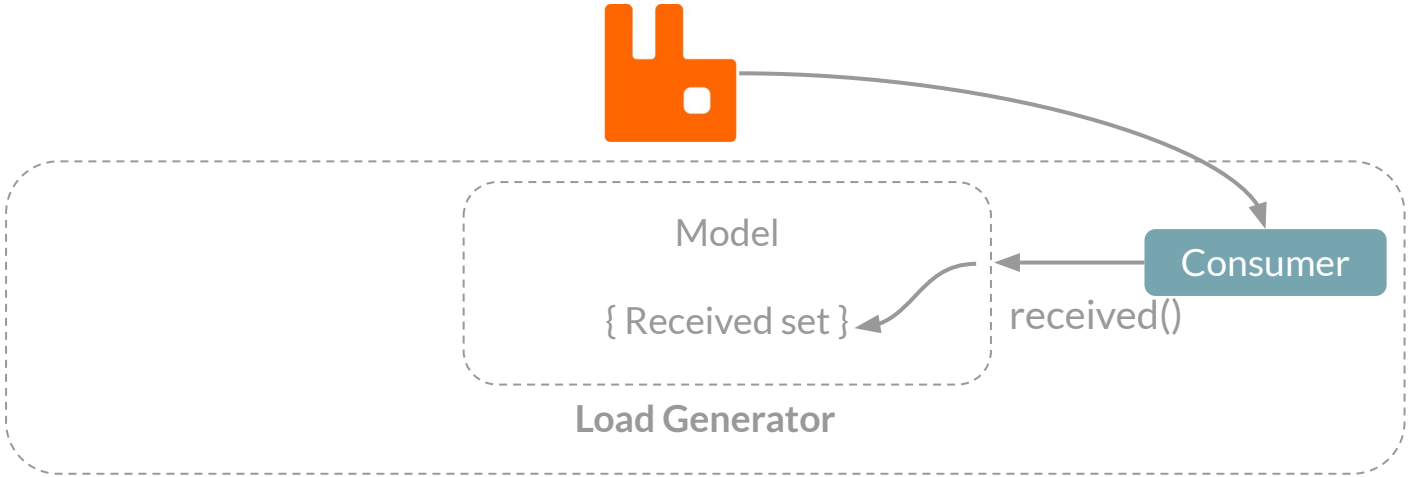
Detecting message ordering violations



/ 1, 2, 4, 3, 5 / = ordering violation!

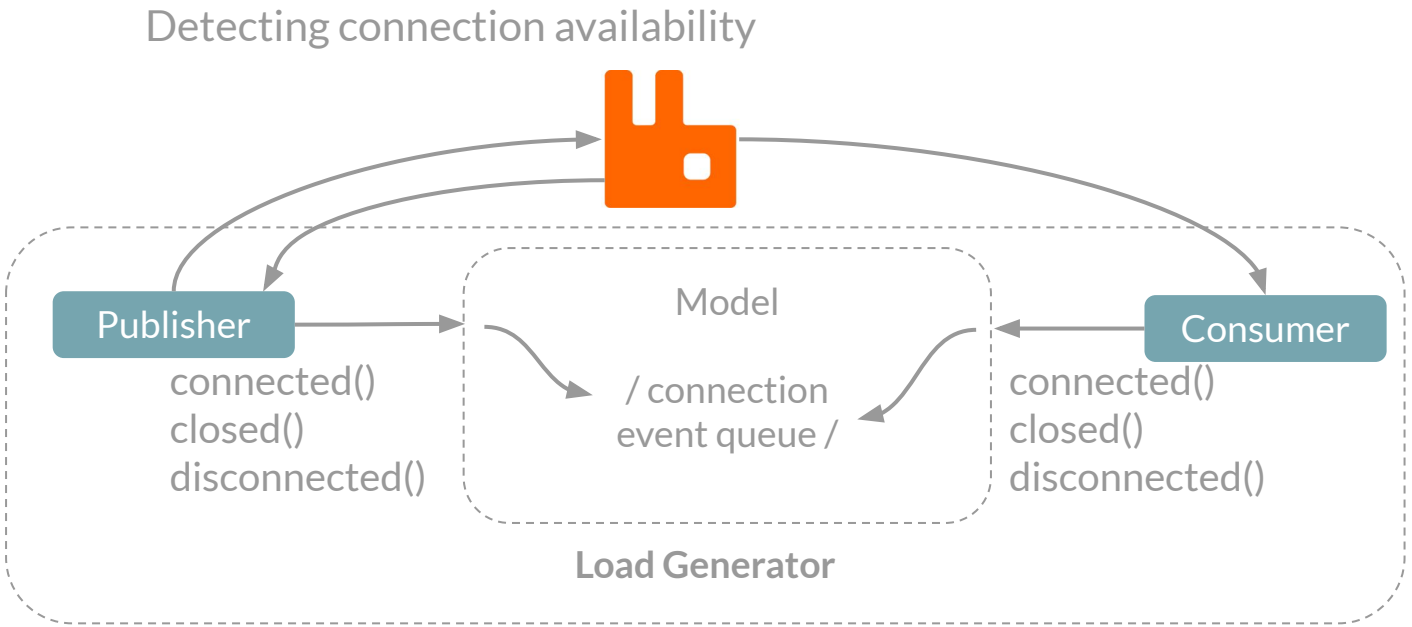
Model-driven Property Based Test Mode

Detecting message duplication



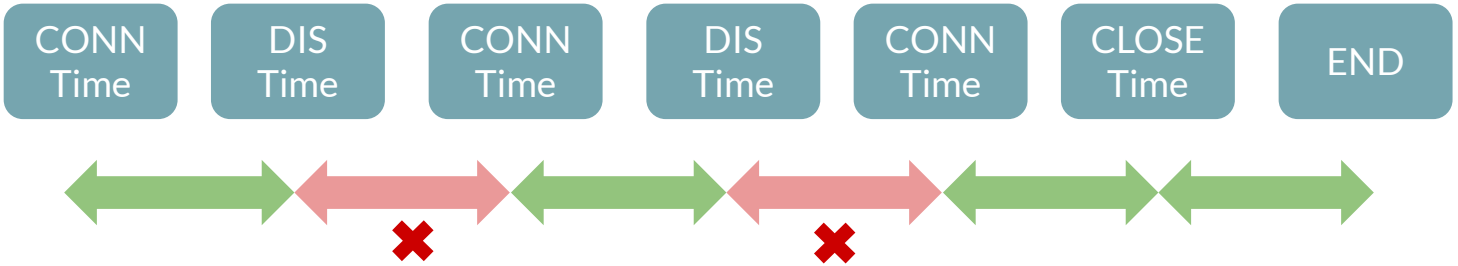
5 -> { 1 , 2, 3, 4, 5 } = Message duplication violation

Model-driven Property Based Test Mode



Model-driven Property Based Test Mode

Detecting connection availability



% Connection Availability

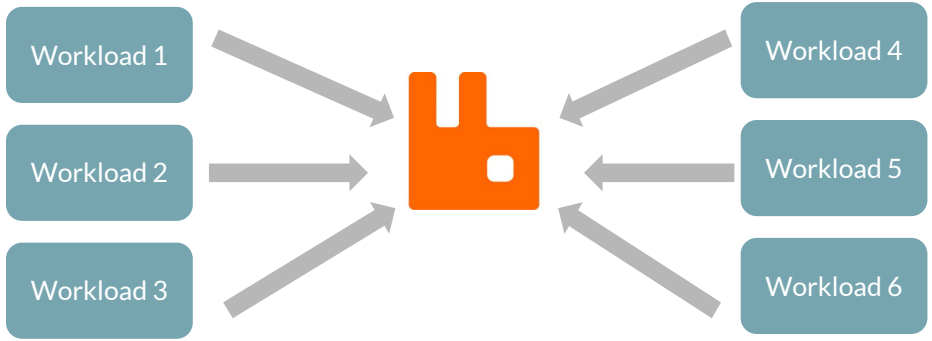
Model-driven Property Based Test Mode

- All results logged periodically to PostgreSQL with a final summary report.
- Low impact so could be turned on by default
- Can be too expensive for some extreme tests with tens of millions of messages a second

Part 3 - Mistakes/ Things learned

#1 Micro not monolithic load generation

For complex, mixed workloads run multiple load generators each with a simpler workload.



#1 Micro not monolithic load generation

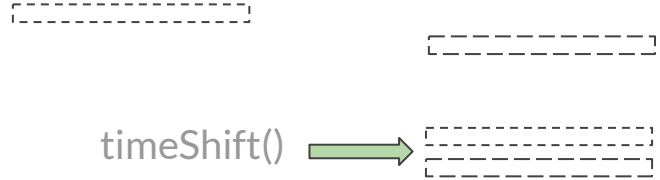
- Compose load-generators to form the complex workloads you need.

- Make sure you can isolate the metrics of each load-generator. This allows you to analyse the impact each workload has on the other.

#2 Use time shifting for overlaying metrics

Use timeShift() / offset modifiers for alignment

- Reuse old results
- Greater scheduling freedom



https://docs.influxdata.com/influxdb/v2.0/reference/flux/stdlib/built-in/transformations/timeshift/
https://prometheus.io/docs/prometheus/latest/querying/basics/#offset-modifier

#3 Have a UI, create dashboard links

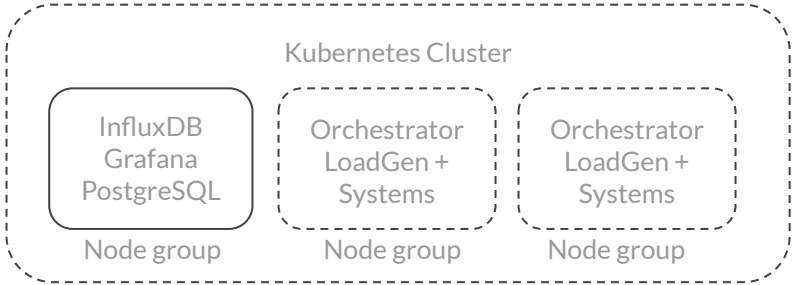
- Generate dashboard links by selecting benchmarks to compare
- Use time shifting feature to overlay results from two or more tests selected in the UI

#4 Validate Early?

- High cost of bad input that affects a multi-hour test midway
- Try to ensure that all input is valid before deployment.
- MongoDB sees it differently, validation in the testing framework is too onerous. https://dl.acm.org/doi/pdf/10.1145/3395032.3395323

#6 Consider ways of reducing deployment times

- Provisioning tools like Ansible can be slow
  - For IaaS systems, think about creating machine images to avoid unnecessary config management steps. https://www.packer.io/
- EKS is slow to deploy a K8s cluster
  - Consider a single long-lived cluster with ephemeral node groups



https://eksctl.io/usage/eks-managed-nodes/

#7 Use log aggregation/search tools

- Allows you to easily watch logs in realtime

- Allows you to data mine from all previous tests

#8 Use defaults files

- Hard-coding defaults into the testing framework makes it hard to find out what the defaults are, or that defaults even exist.

Some important conclusions...

- System testing is complimentary to your other testing, good for:
  - Performance
  - Resilience
  - Correctness
- Don’t wait till the end of the dev cycle to run system tests
- Powerful automation and workload generation make answering questions easy.
- Model driven, property based checking is a powerful but simple concept.

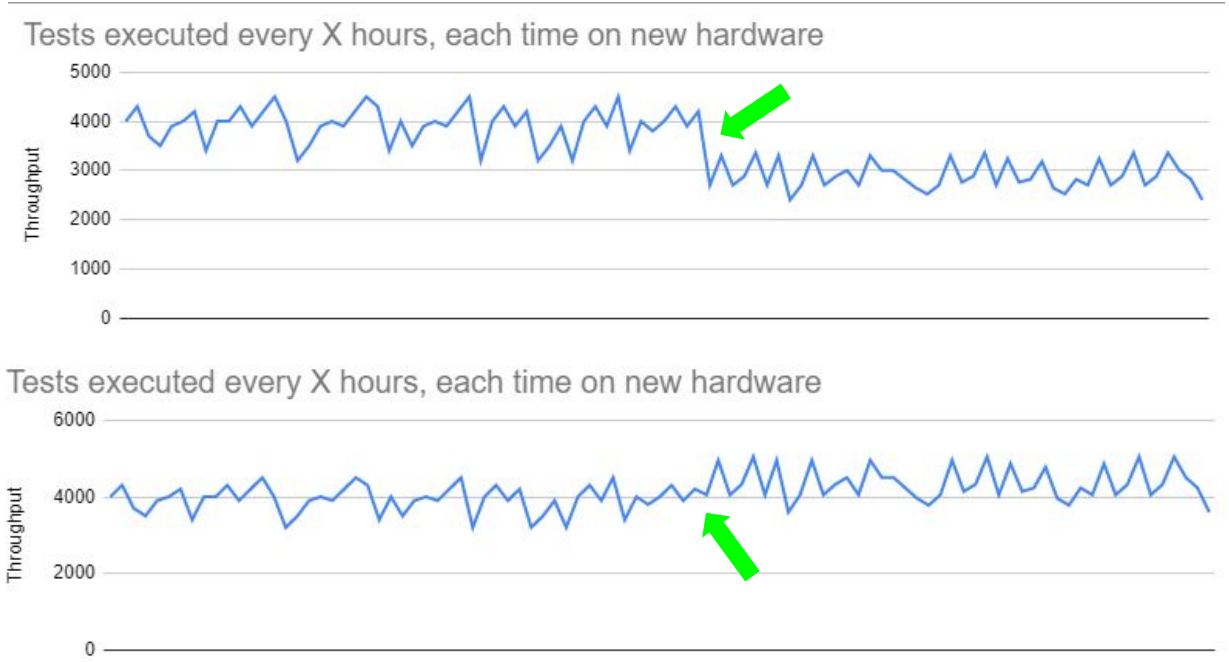
Some reading materials

- Funny but insightful: https://sled.rs/perf.html
- MongoDB paper on their system testing framework and practices: https://dl.acm.org/doi/pdf/10.1145/3395032.3395323
- MongoDB paper on automated regression detection (using signal processing approach): https://dl.acm.org/doi/10.1145/3358960.3375791
- Statistical rigor in benchmarking (using confidence interval approach): https://dri.es/files/oopla07-georges.pdf
- Paper on causes of tail latency: https://syslab.cs.washington.edu/papers/latency-socc14.pdf
- Paper on reducing variability: https://www.usenix.org/system/files/osdi18-maricq.pdf

Thank you

Dealing with variability

Signal processing strategy (used by MongoDB):



/system-testing/practices/variability/measurement/signal-processing