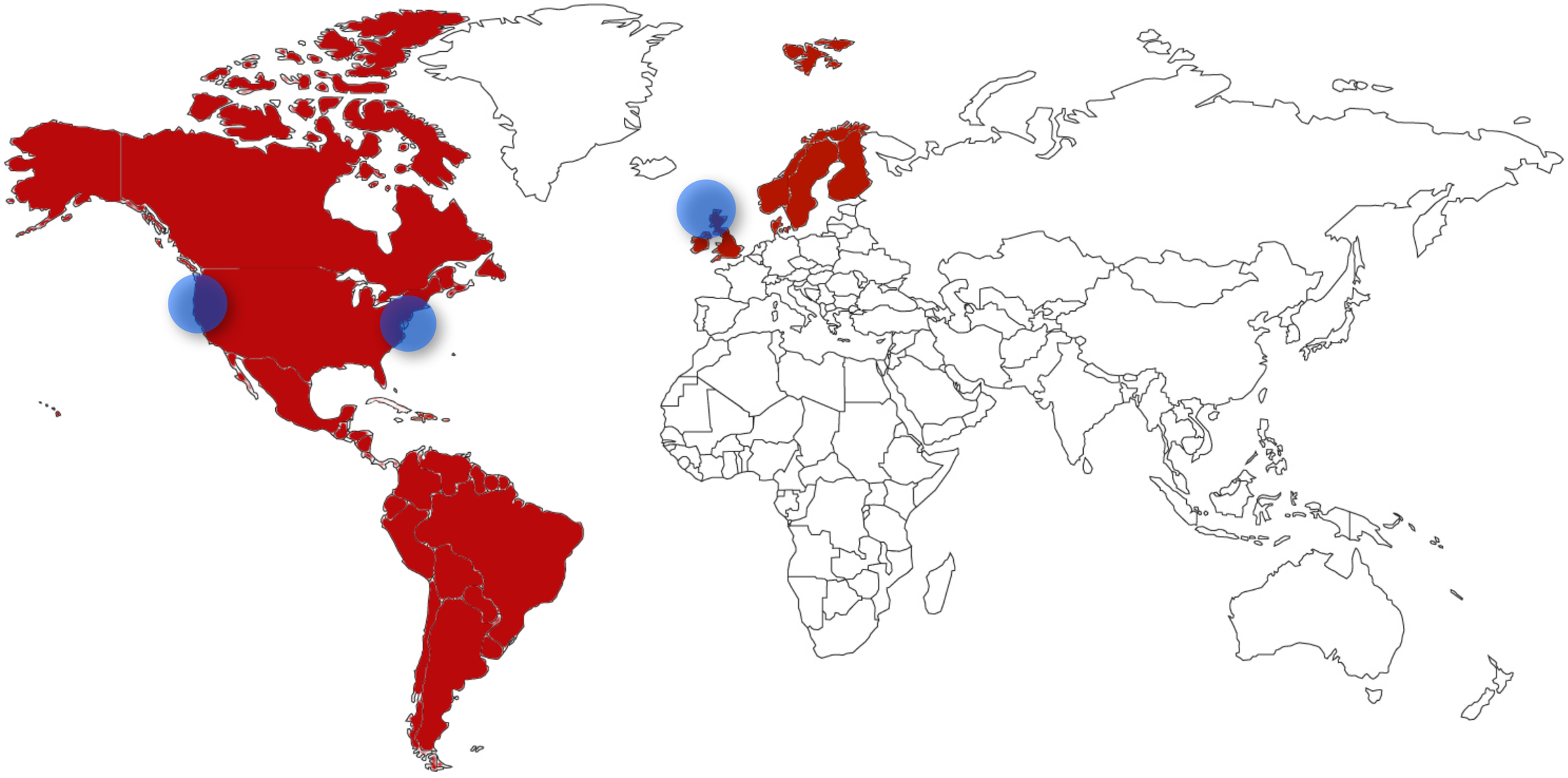
APPLICATION RESILIENCE ENGINEERING AND OPERATIONS AT NETFLIX

HYSTRIX

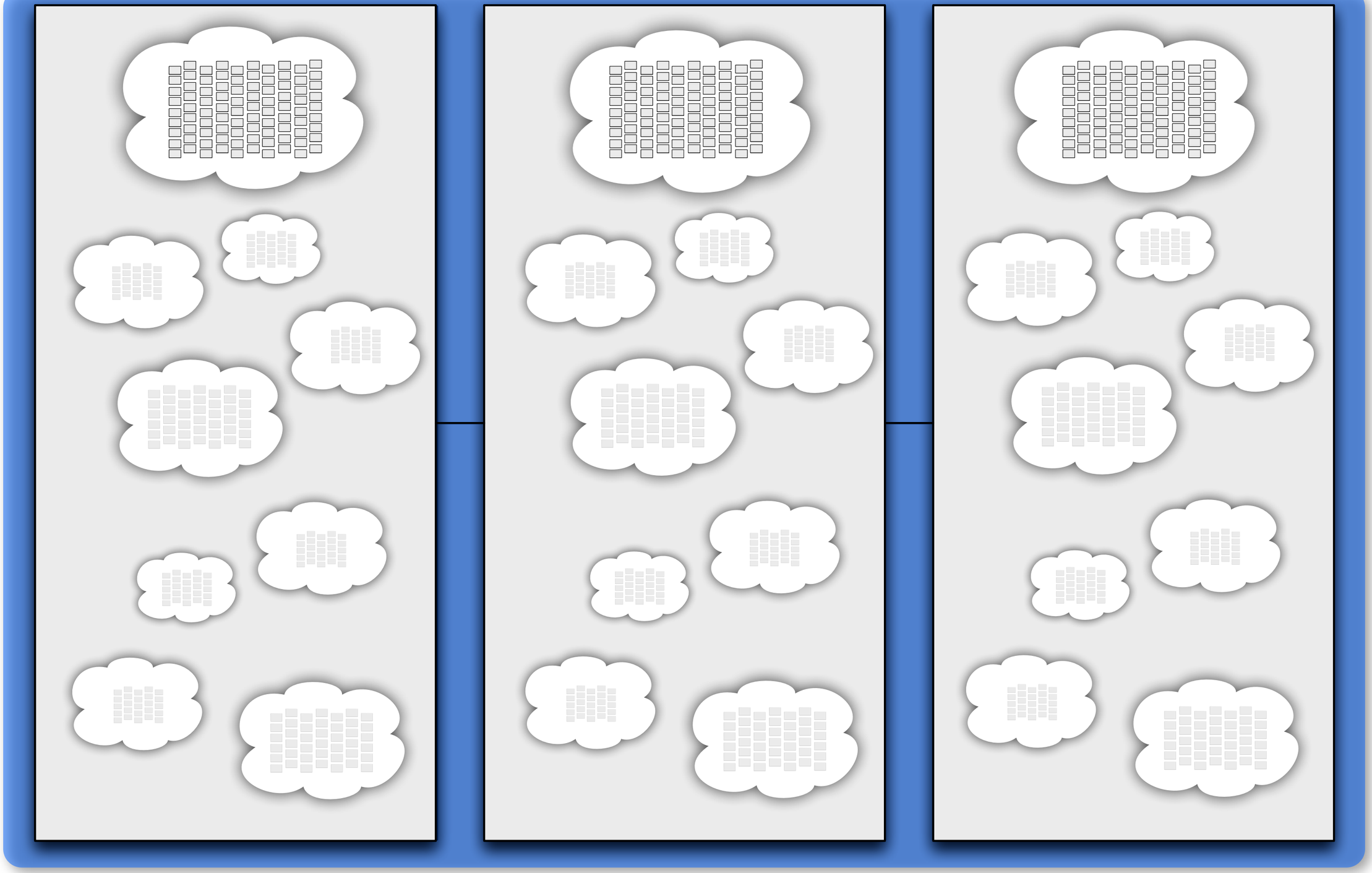
DEFEND YOUR APP

Ben Christensen – @benjchristensen – Software Engineer on API Platform at Netflix

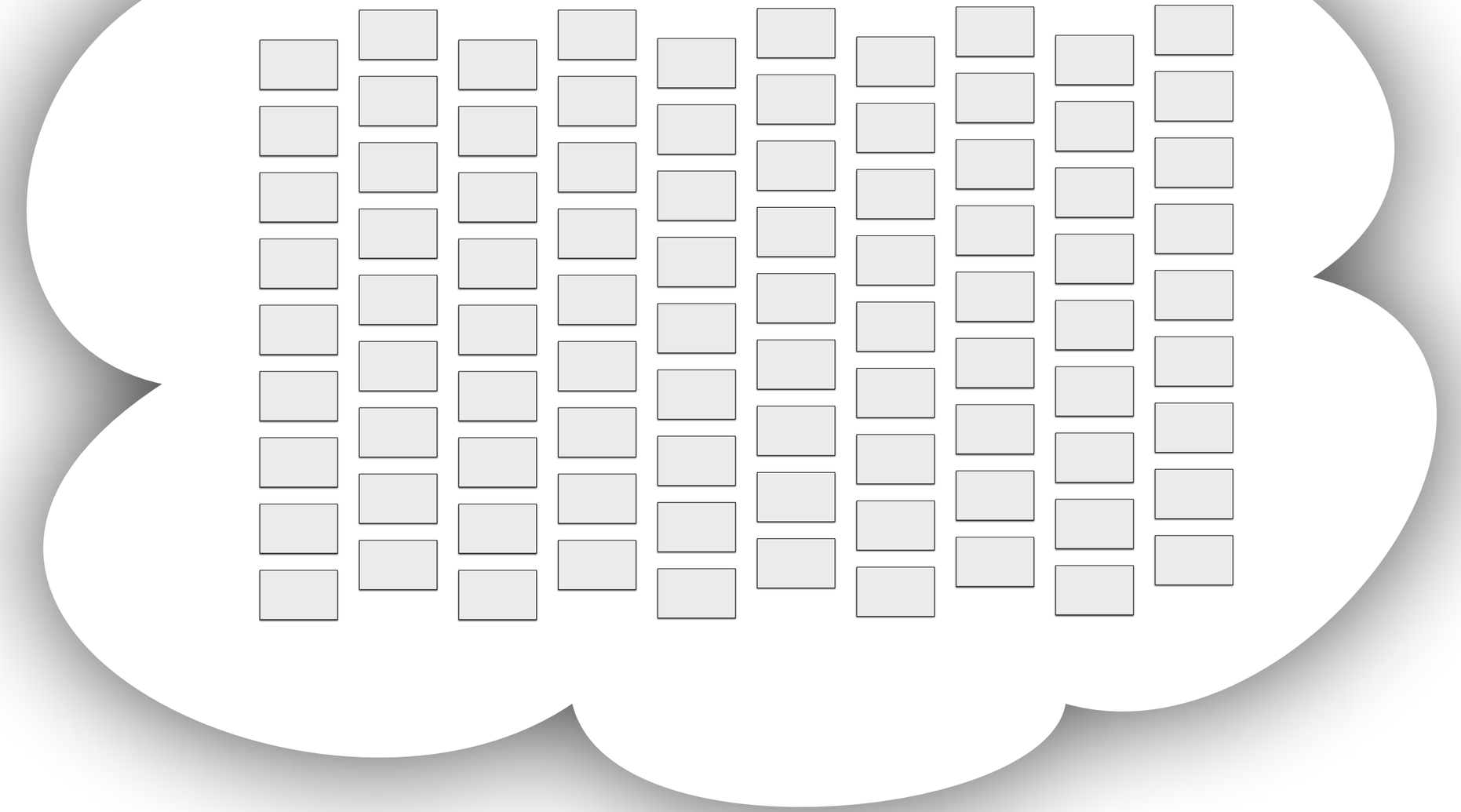


Global deployment spread across data centers in multiple AWS regions.
Geographic isolation, active/active with regional failover coming (http://techblog.netflix.com/2013/05/denominating-multi-region-sites.html)

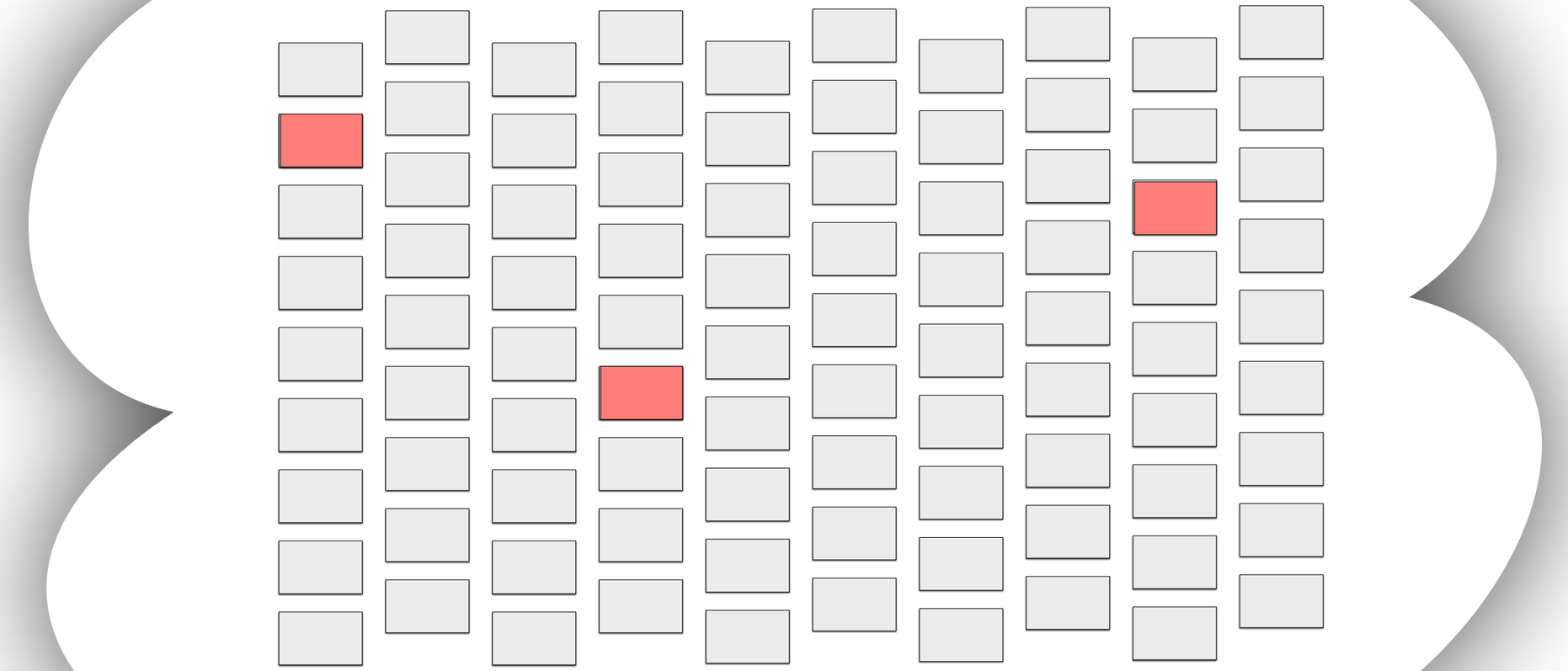
3 data centers (AWS Availability Zones) operate in each region with deployments split across them for redundancy in event of losing an entire zone.



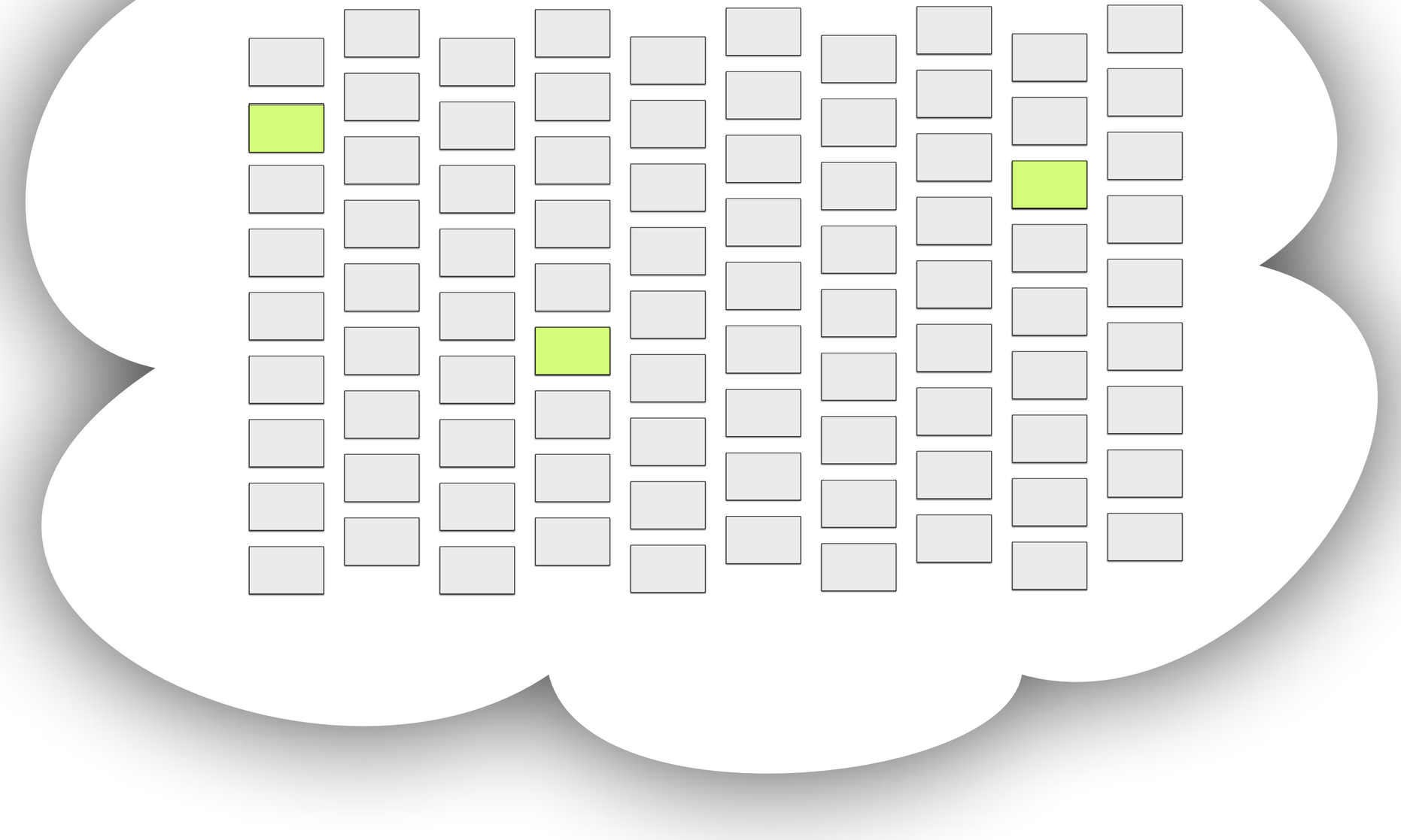
Each zone is populated with application clusters (‘auto-scaling groups’ or ASGs) that make up the service oriented distributed system.
Application clusters operate independently of each other with software and hardware load balancing routing traffic between them.



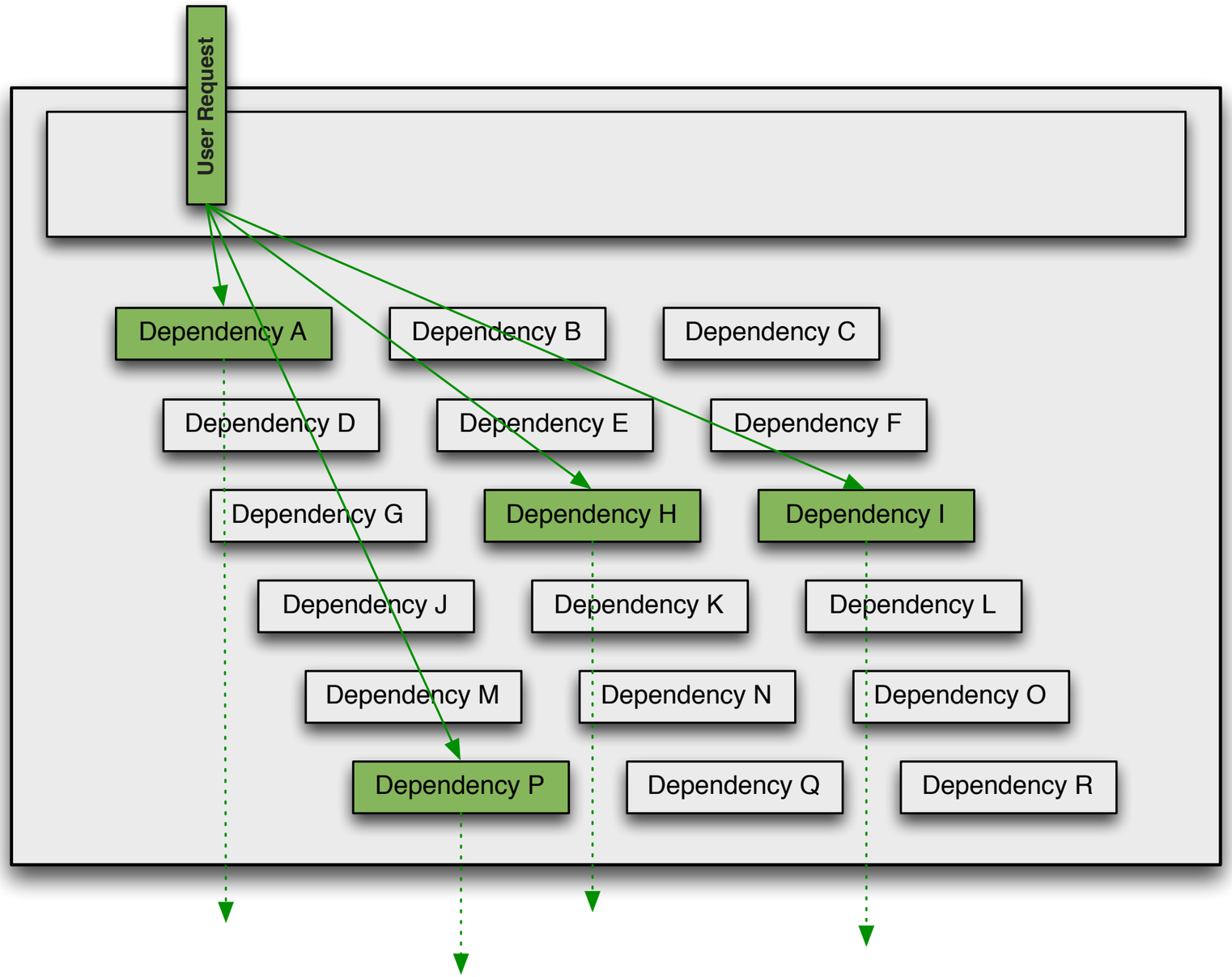
Application clusters are made up of 1 to 100s of machine instances per zone. Service registry and discovery work with software load balancing to allow machines to launch and disappear (for planned or unplanned reasons) at any time and become part of the distributed system and serve requests. Auto-scaling enables system-wide adaptation to demand as it launches instances to meet increasing traffic and load or handle instance failure.



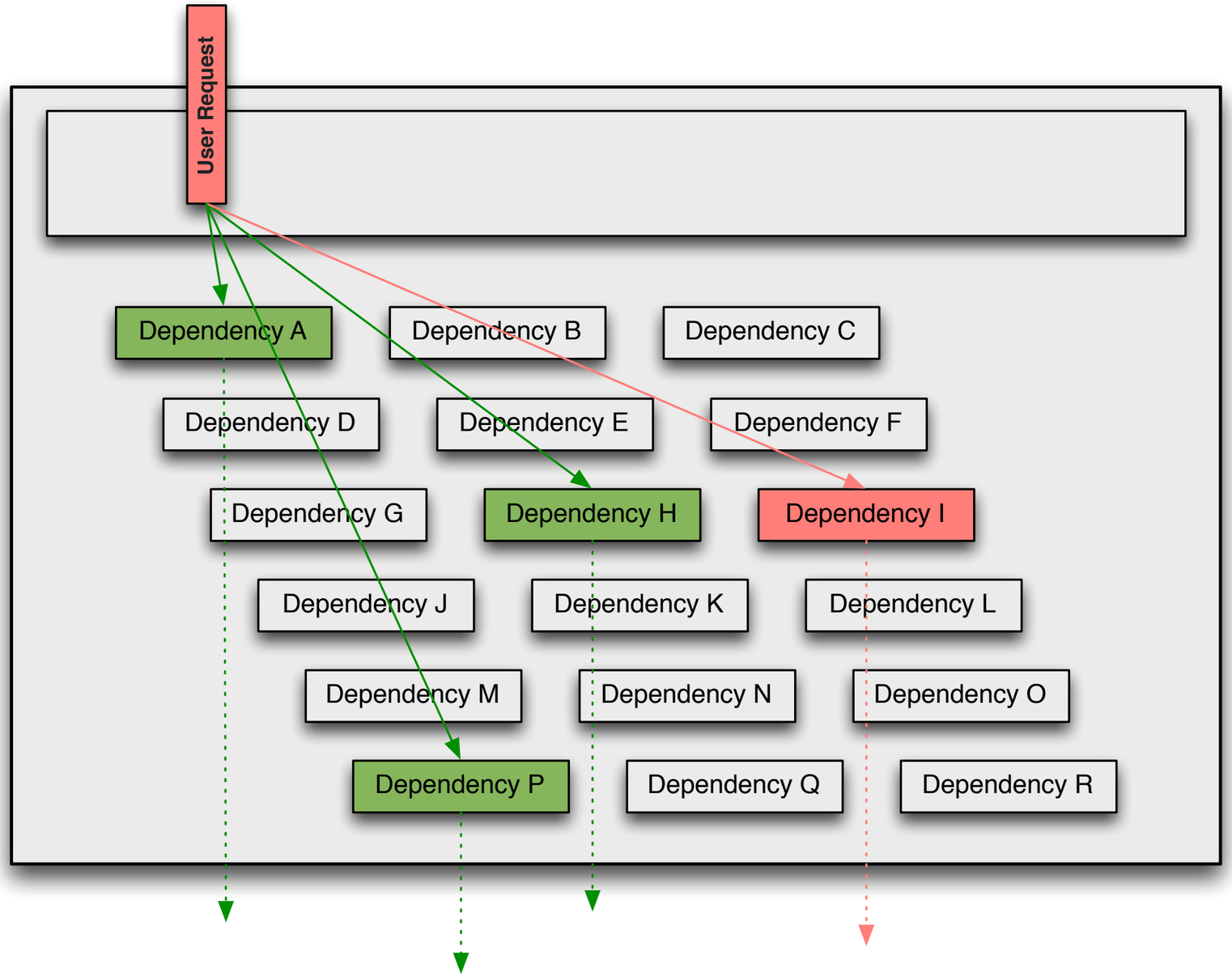
Failed instances are dropped from discovery so traffic stops routing to them. Software load balancers on client applications detect and skip them until discovery removes them.



Auto-scale policies brings on new instances to replace failed ones or to adapt to increasing demand.

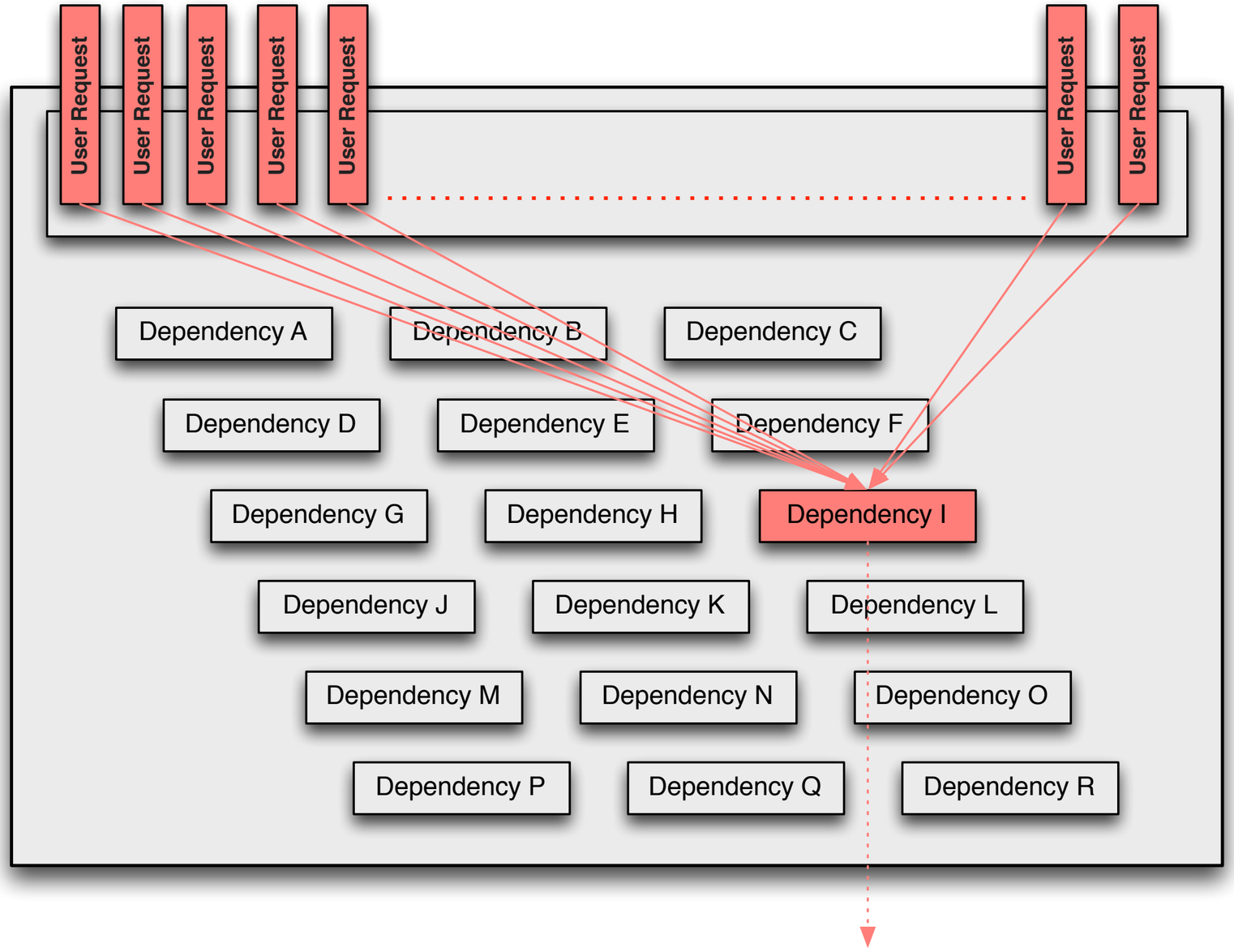


Applications communicate with dozens of other applications in the service-oriented architecture. Each of these client/server dependencies represents a relationship within the complex distributed system.



USER REQUEST BLOCKED BY LATENCY IN SINGLE NETWORK CALL

Any one of these relationships can fail at any time. They can be intermittent or cluster-wide, immediate with thrown exceptions or returned error codes or latency from various causes. Latency is particularly challenging for applications to deal with as it causes resource utilization in queues and pools and blocks user requests (even with async IO).



AT HIGH VOLUME ALL REQUEST THREADS CAN BLOCK IN SECONDS

Latency at high volume can quickly saturate all application resources (queues, pools, sockets, etc) causing total application failure and the inability to serve user requests even if all other dependencies are healthy.

DOZENS OF DEPENDENCIES.

ONE GOING BAD TAKES EVERYTHING DOWN.

$99.99\%^{30} = 99.7\%$ UPTIME

0.3% OF 1 BILLION = 3,000,000 FAILURES

2+ HOURS DOWNTIME/MONTH

REALITY IS GENERALLY WORSE.

Large distributed systems are complex and failure will occur. If failure from every component is allowed to cascade across the system they will all affect the user.

CONSTRAINTS

SPEED OF ITERATION

CLIENT LIBRARIES

MIXED ENVIRONMENT

Solution design was done with constraints, context and priorities of the Netflix environment.

CONSTRAINTS

SPEED OF ITERATION

CLIENT LIBRARIES

MIXED ENVIRONMENT

Speed of iteration is optimized for and this leads to client/server relationships where client libraries are provided rather than each team writing their own client code against a server protocol. This means “3rd party” code from many developers and teams is constantly being deployed into applications across the system. Large applications such as the Netflix API have dozens of client libraries.

CONSTRAINTS

SPEED OF ITERATION

CLIENT LIBRARIES

MIXED ENVIRONMENT

Speed of iteration is optimized for and this leads to client/server relationships where client libraries are provided rather than each team writing their own client code against a server protocol. This means “3rd party” code from many developers and teams is constantly being deployed into applications across the system. Large applications such as the Netflix API have dozens of client libraries.

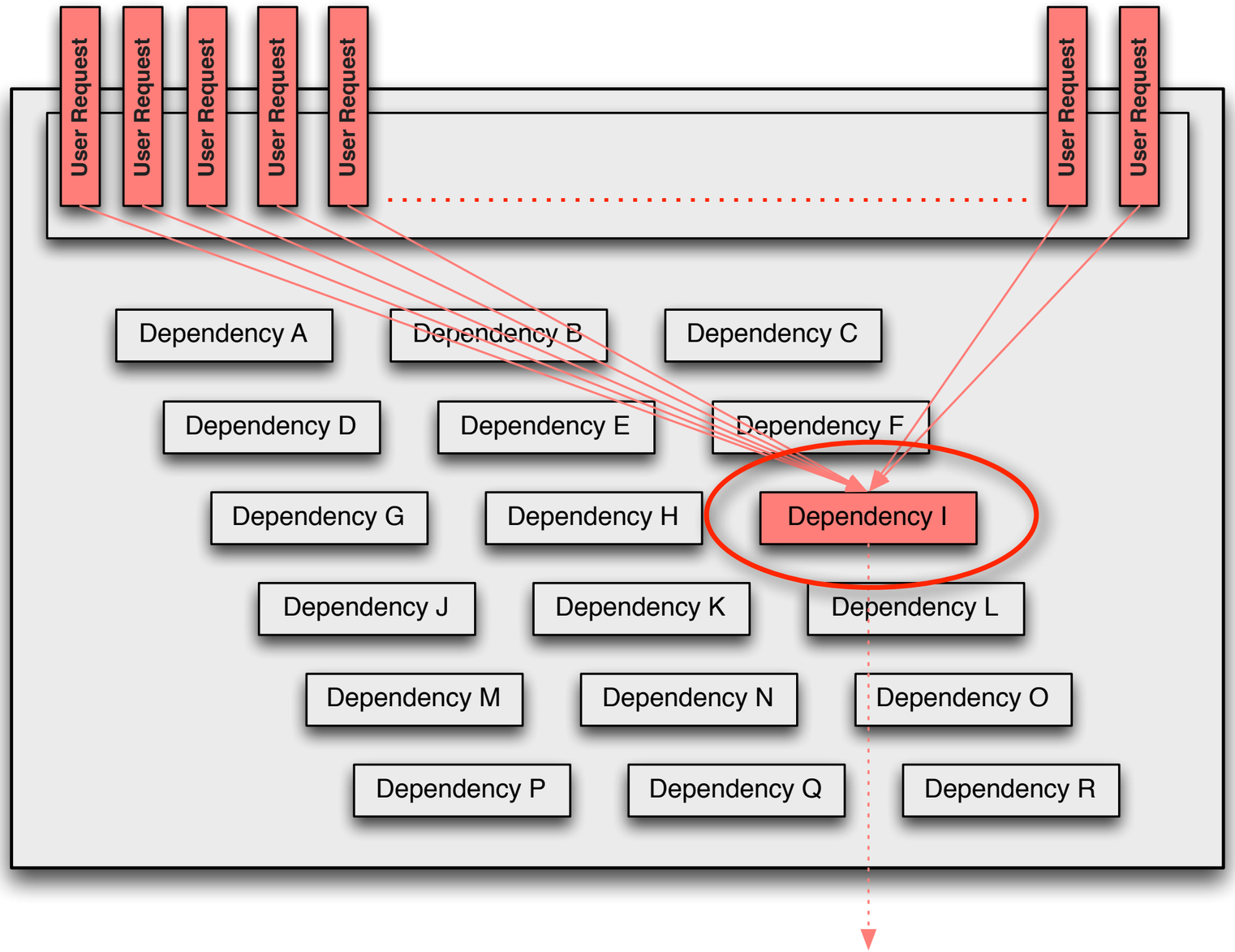
CONSTRAINTS

SPEED OF ITERATION

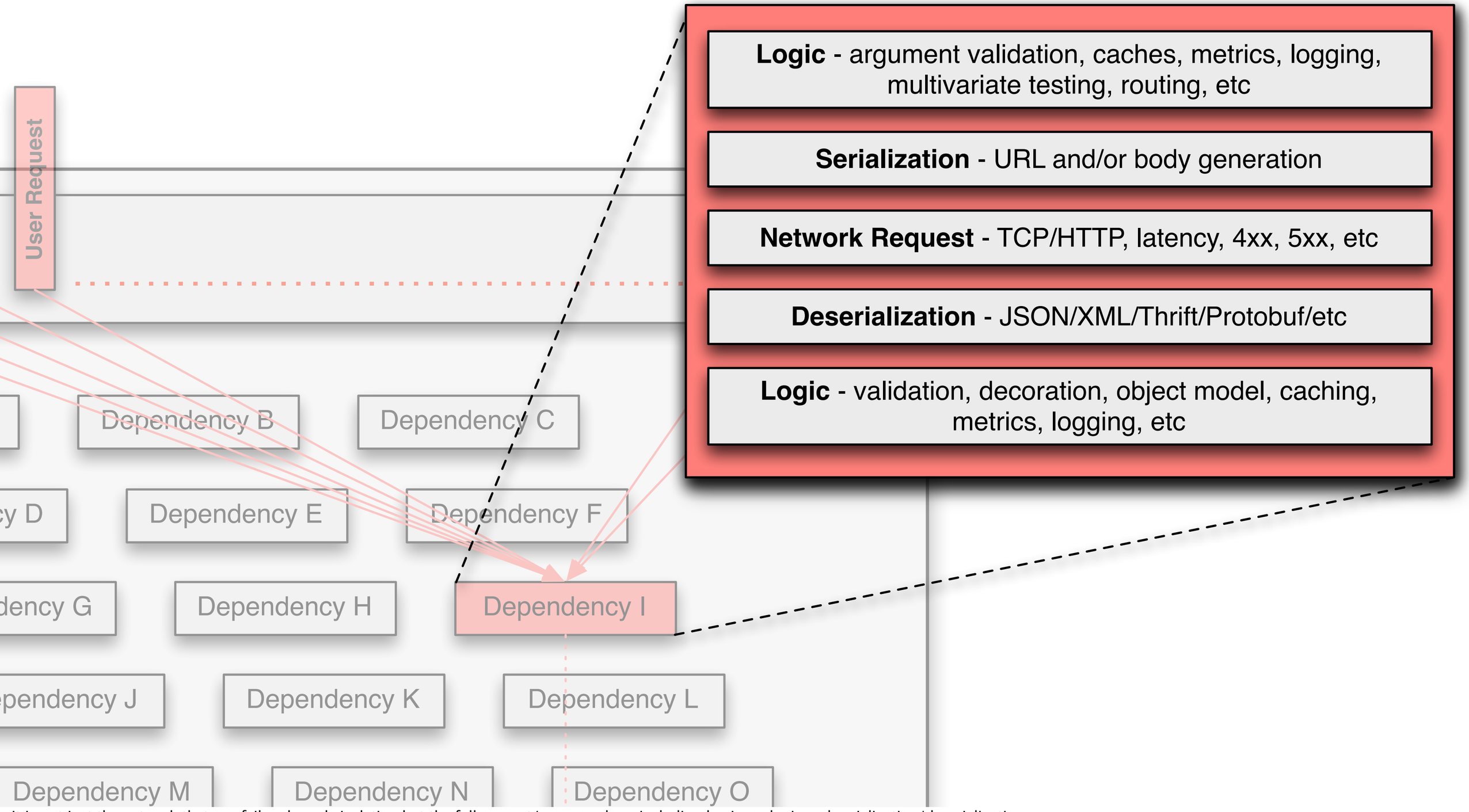
CLIENT LIBRARIES

MIXED ENVIRONMENT

The environment is also diverse with different types of client/server communications and protocols. This heterogenous and always changing environment affects the approach for resilience engineering and is potentially very different than approaches taken for a tightly controlled codebase or homogenous architecture.

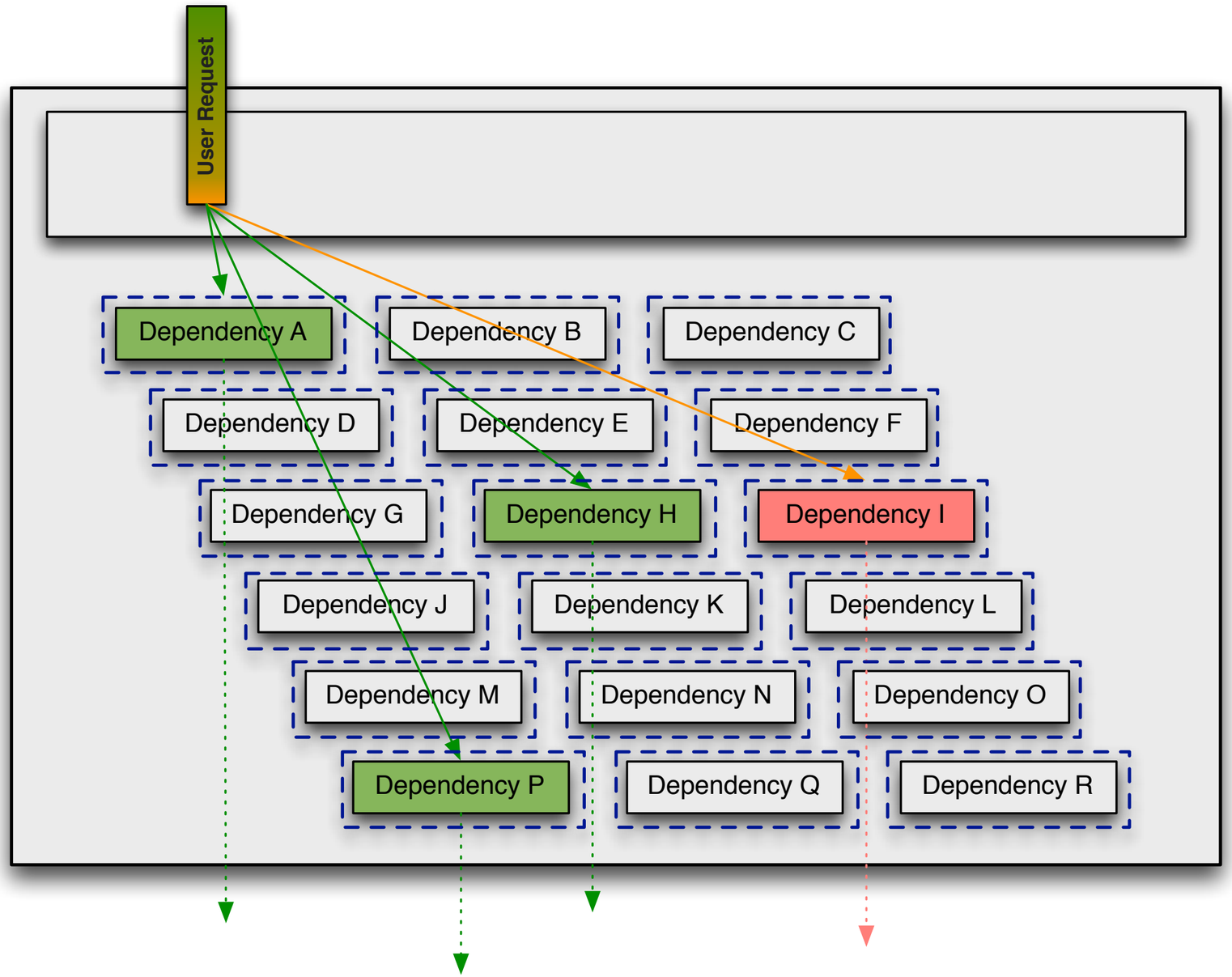


Each dependency - or distributed system relationship - must be isolated so its failure does not cascade or saturate all resources.

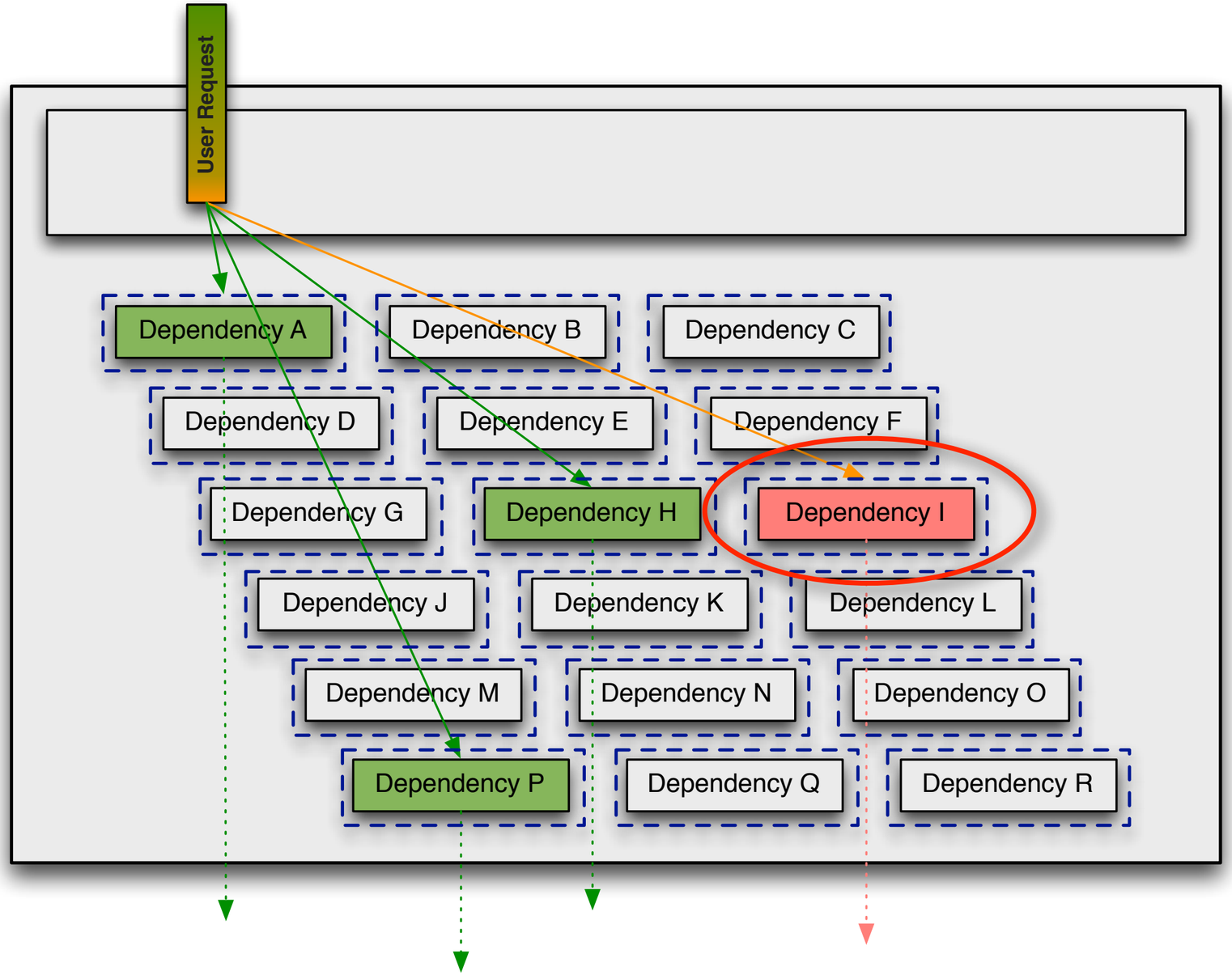


It is not just the network that can fail and needs isolation but the full request/response loop including business logic and serialization/deserialization.

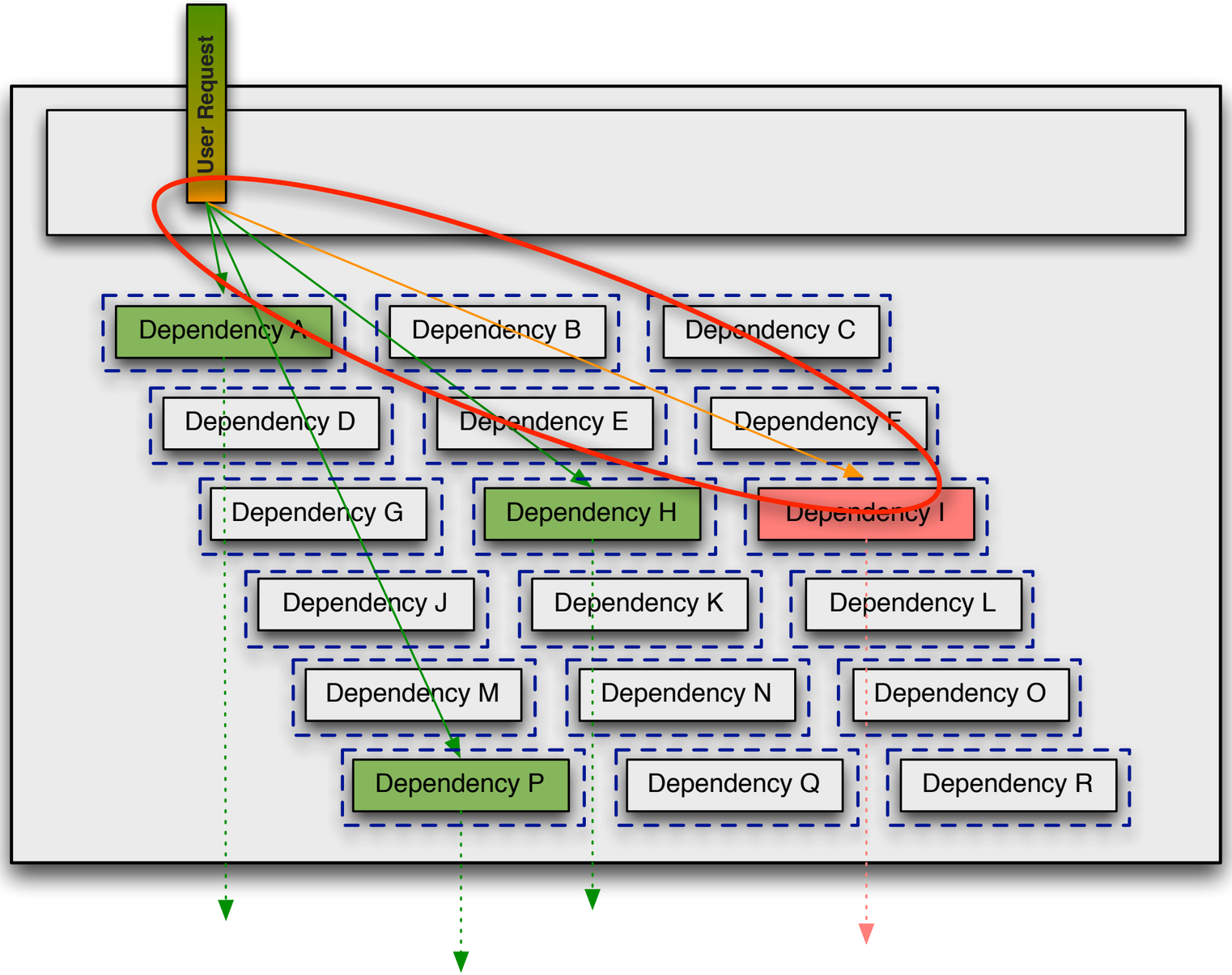
Protecting against a network failure only to return a response that causes application logic to fail elsewhere in the application only moves the problem.



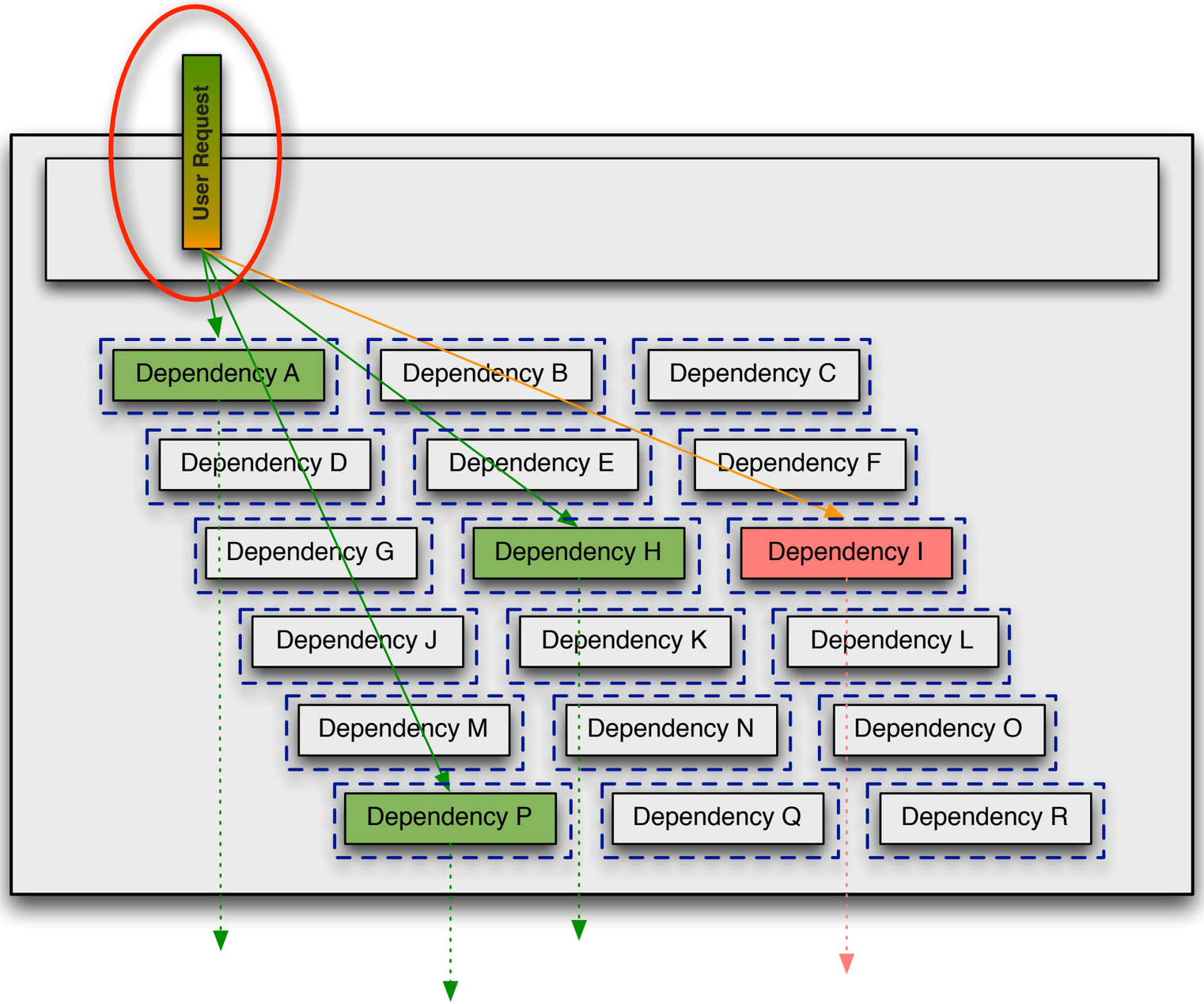
Bulkheading is an approach to isolating failure and latency. It can be used to compartmentalize each system relationship so their failure impact is limited and controllable.



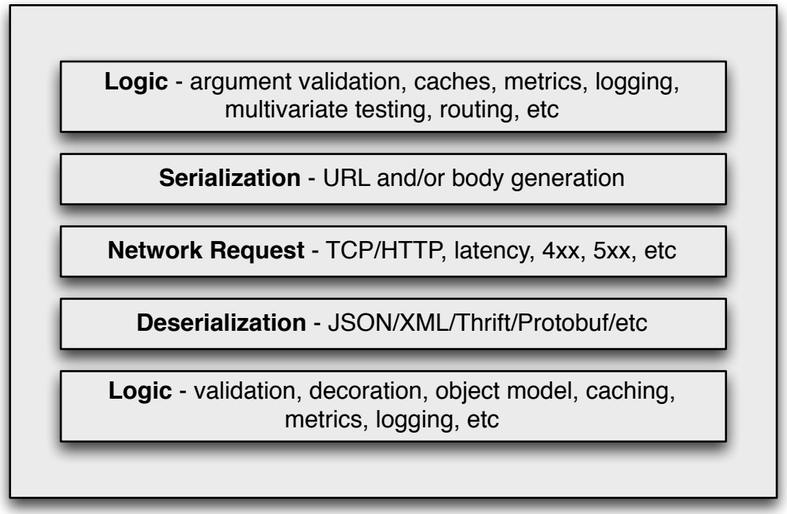
Bulkheading is an approach to isolating failure and latency. It can be used to compartmentalize each system relationship so their failure impact is limited and controllable.



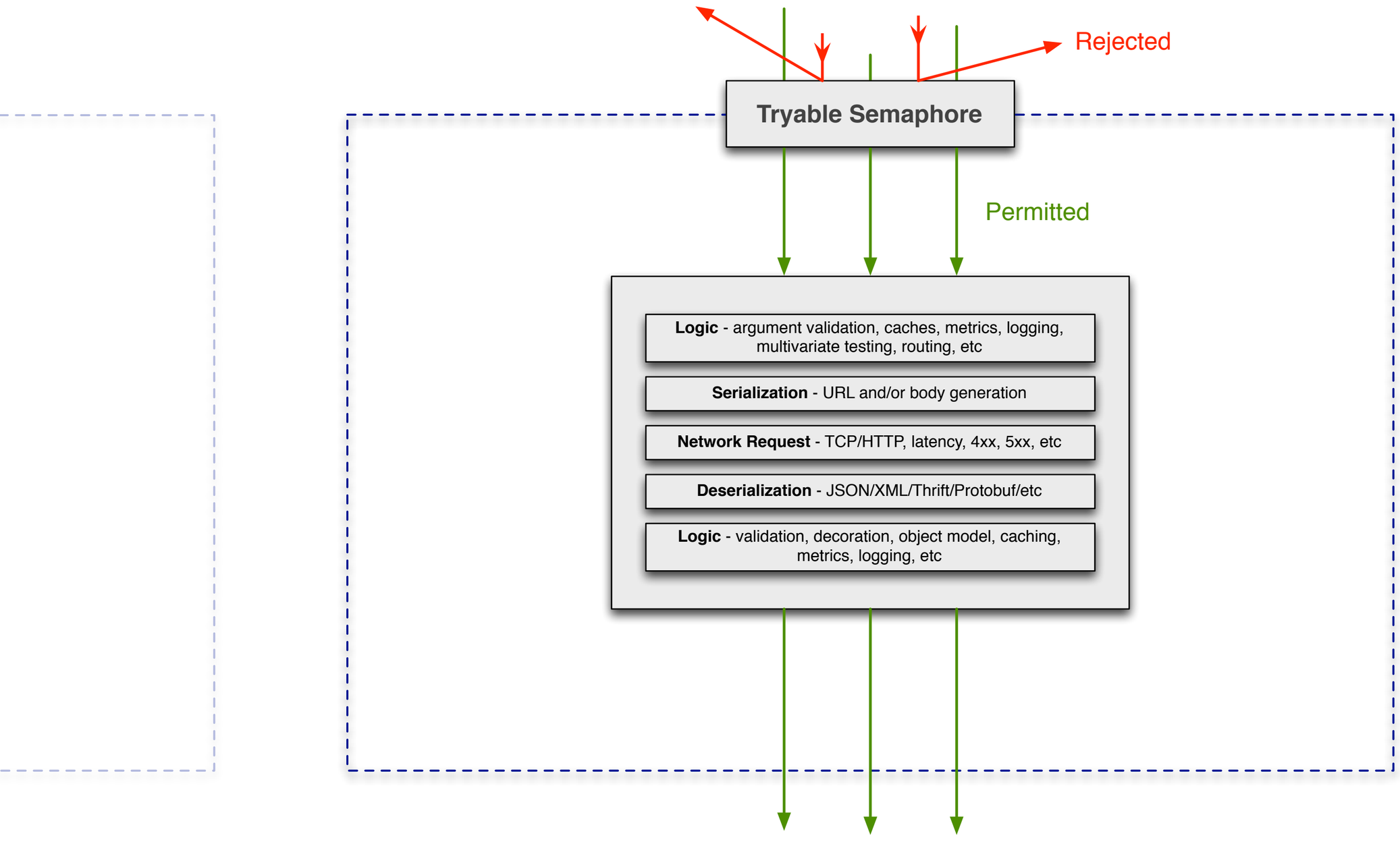
Responses can be intercepted and replaced with fallbacks.



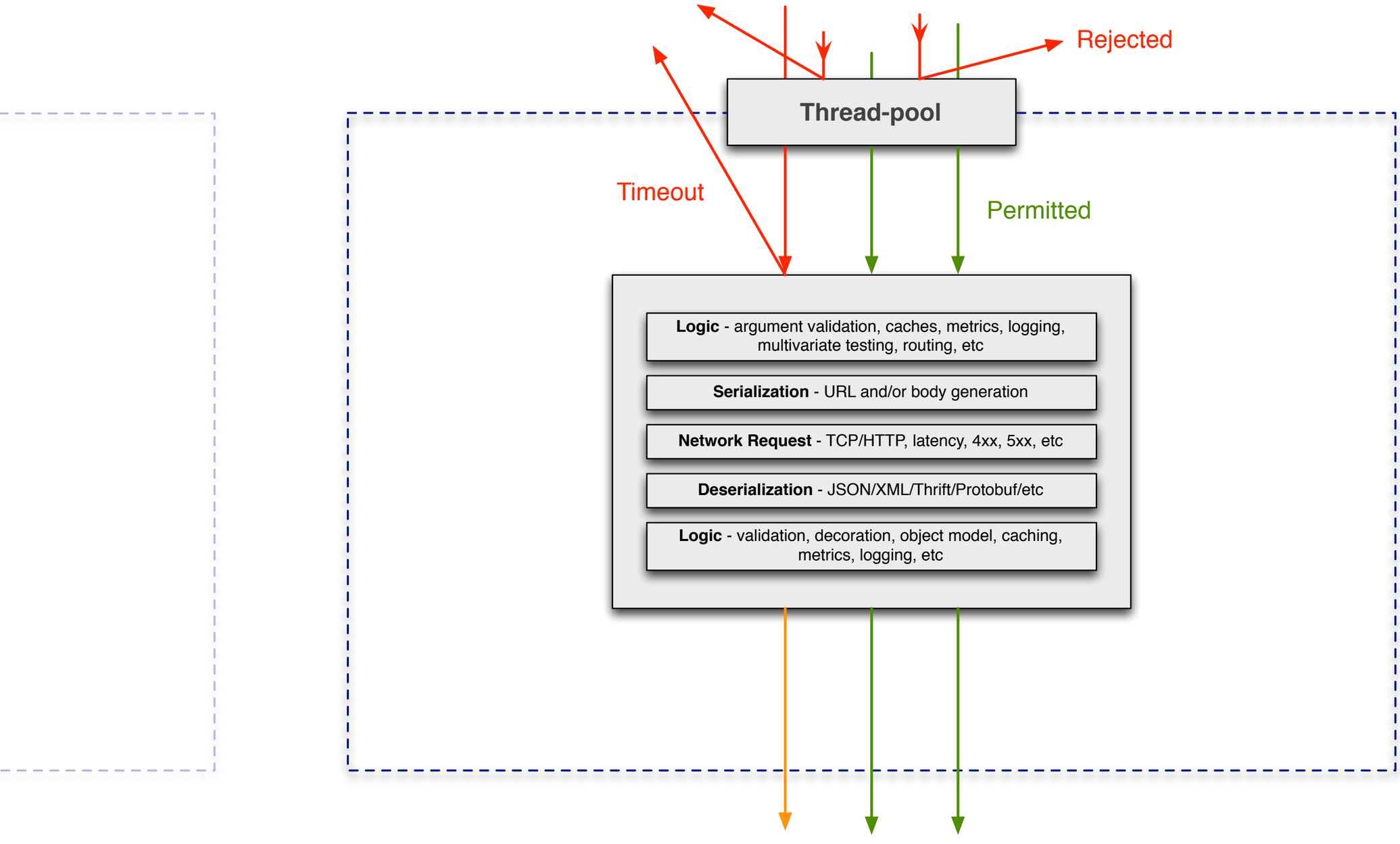
A user request can continue in a degraded state with a fallback response from the failing dependency.



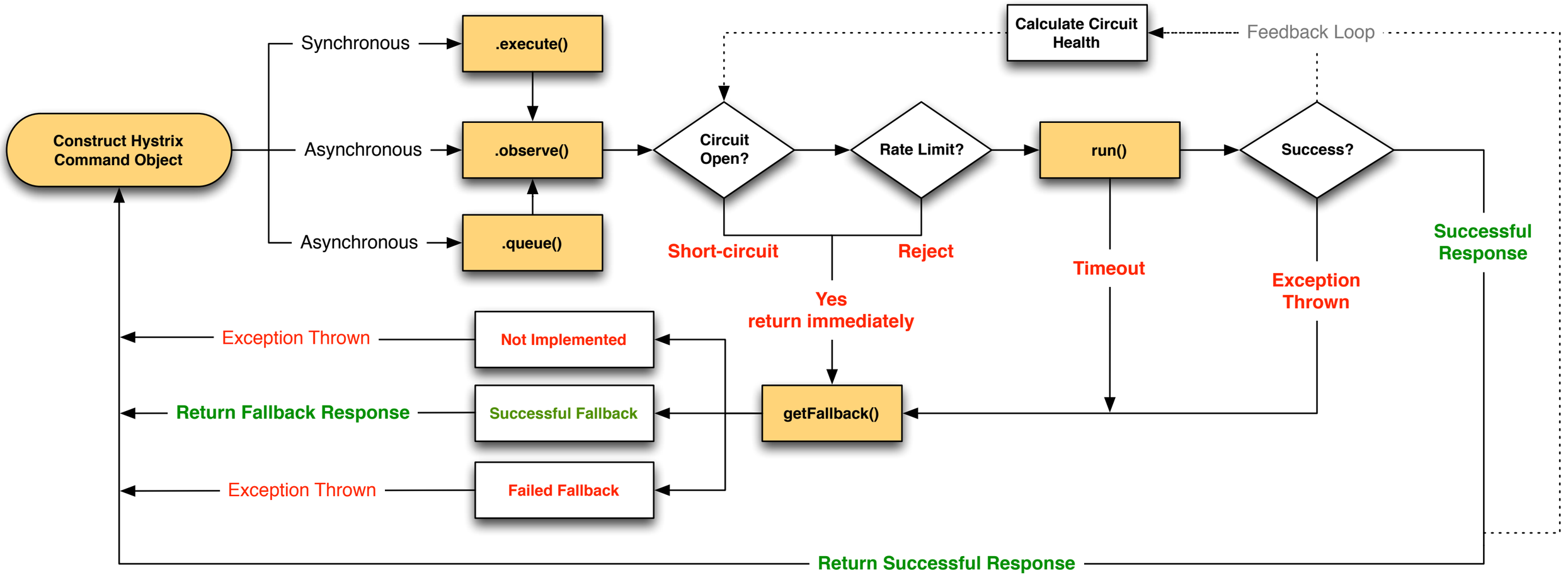
A bulkhead wraps around the entire client behavior not just the network portion.



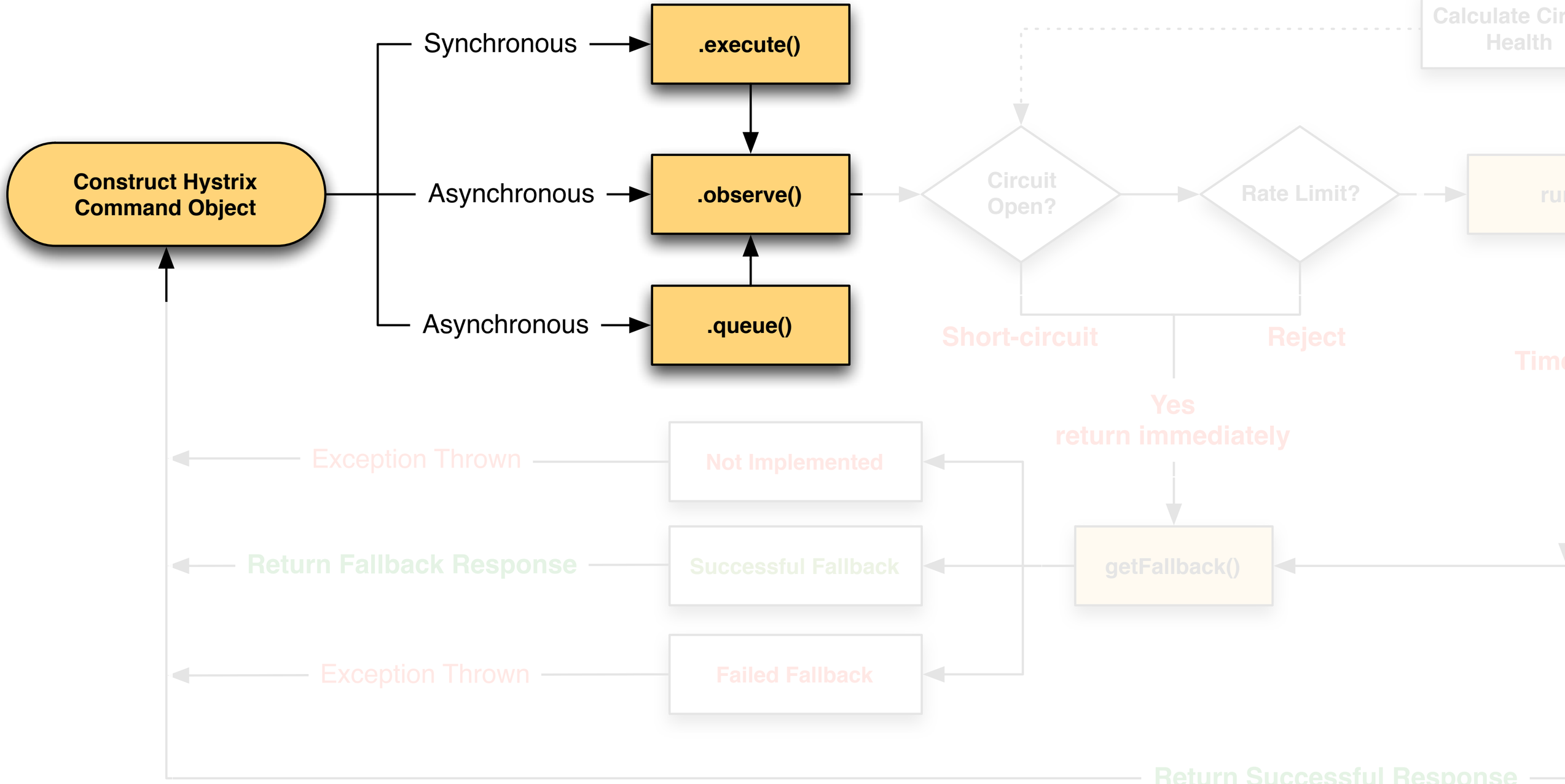
An effective form of bulkheading is a tryable semaphore that restricts concurrent execution. Read more at https://github.com/Netflix/Hystrix/wiki/How-it-Works#semaphores



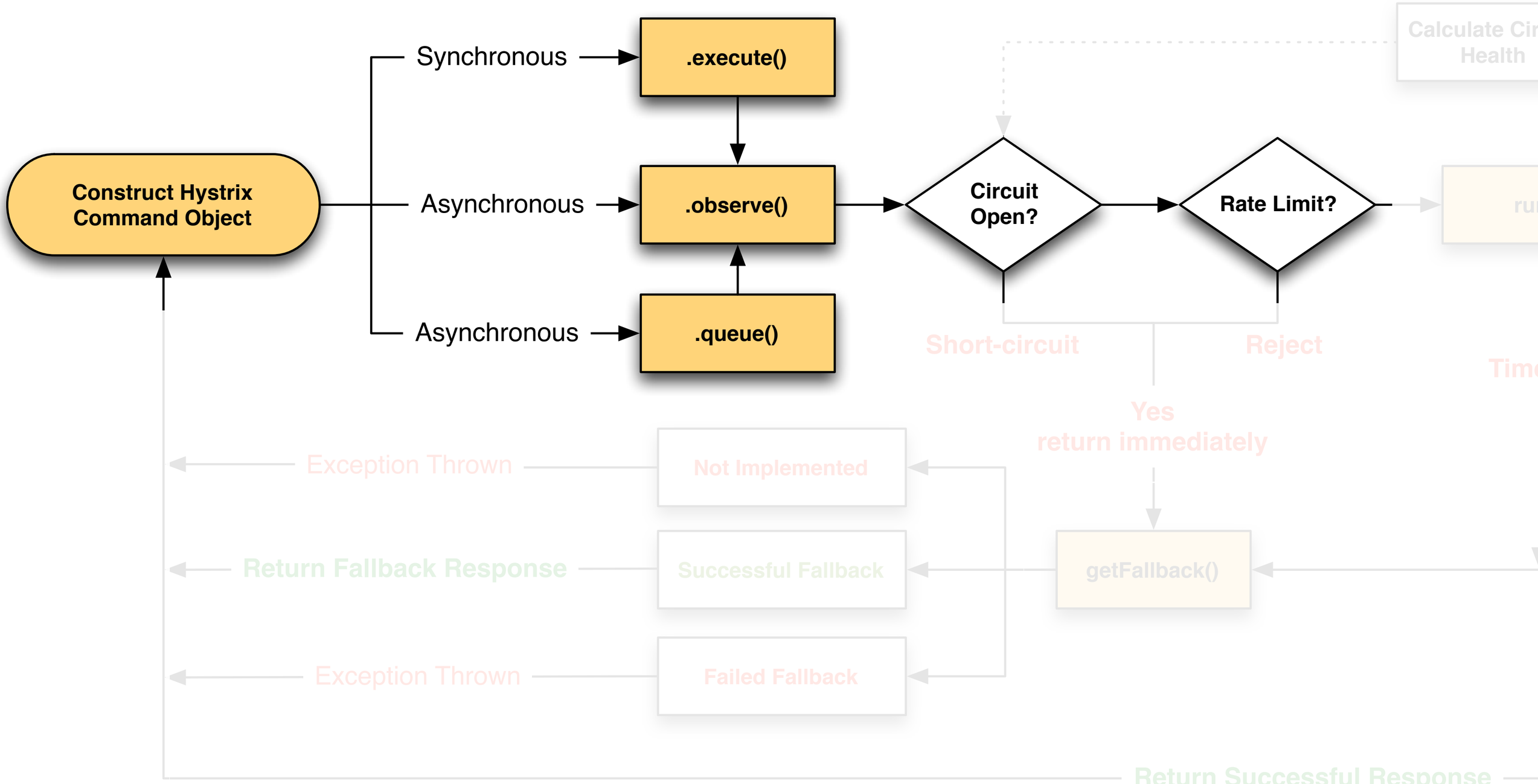
A thread-pool also limits concurrent execution while also offering the ability to timeout and walk away from a latent thread. Read more at https://github.com/Netflix/Hystrix/wiki/How-it-Works#threads--thread-pools



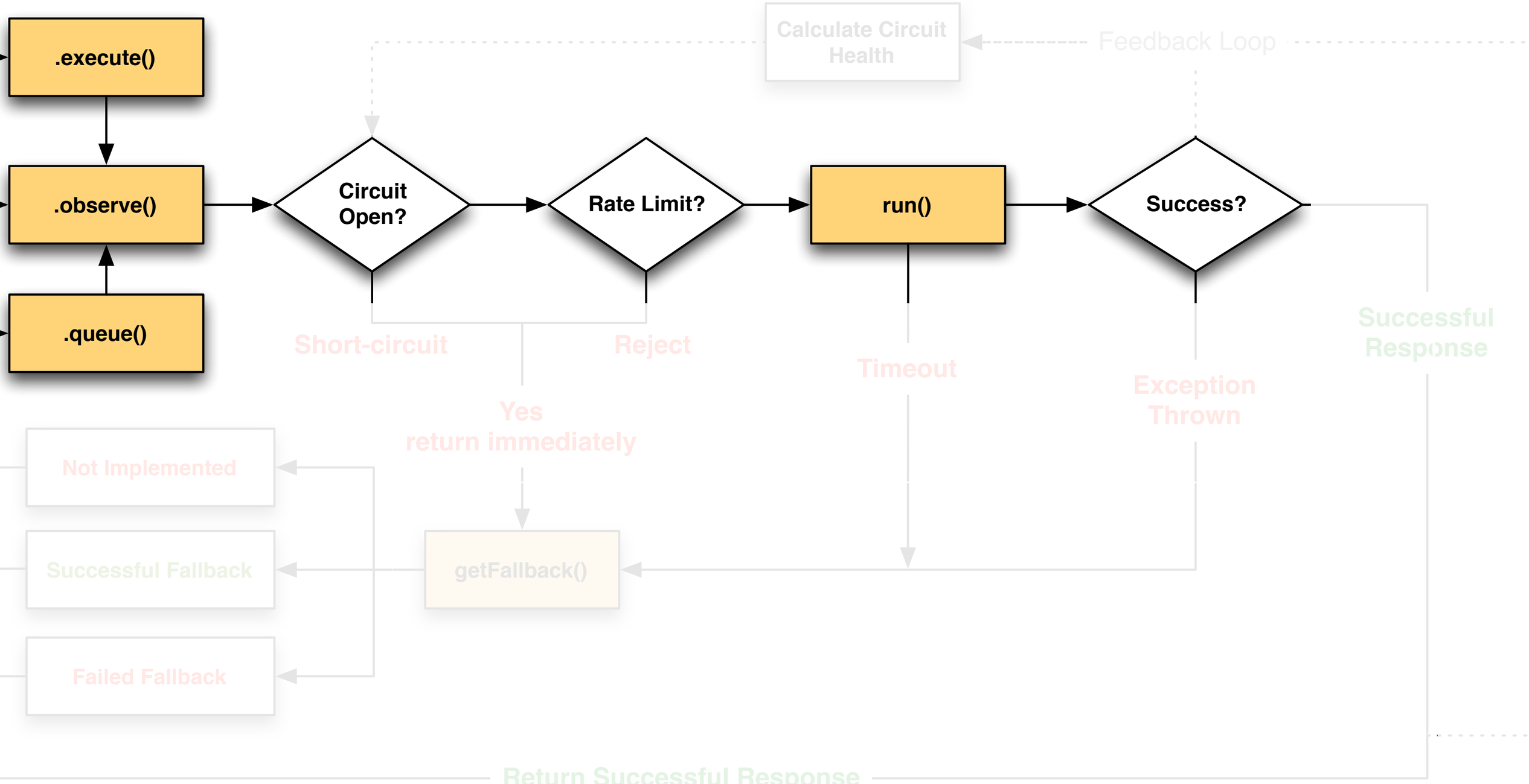
Hystrix execution flow chart. Read more at https://github.com/Netflix/Hystrix/wiki/How-it-Works#flow-chart



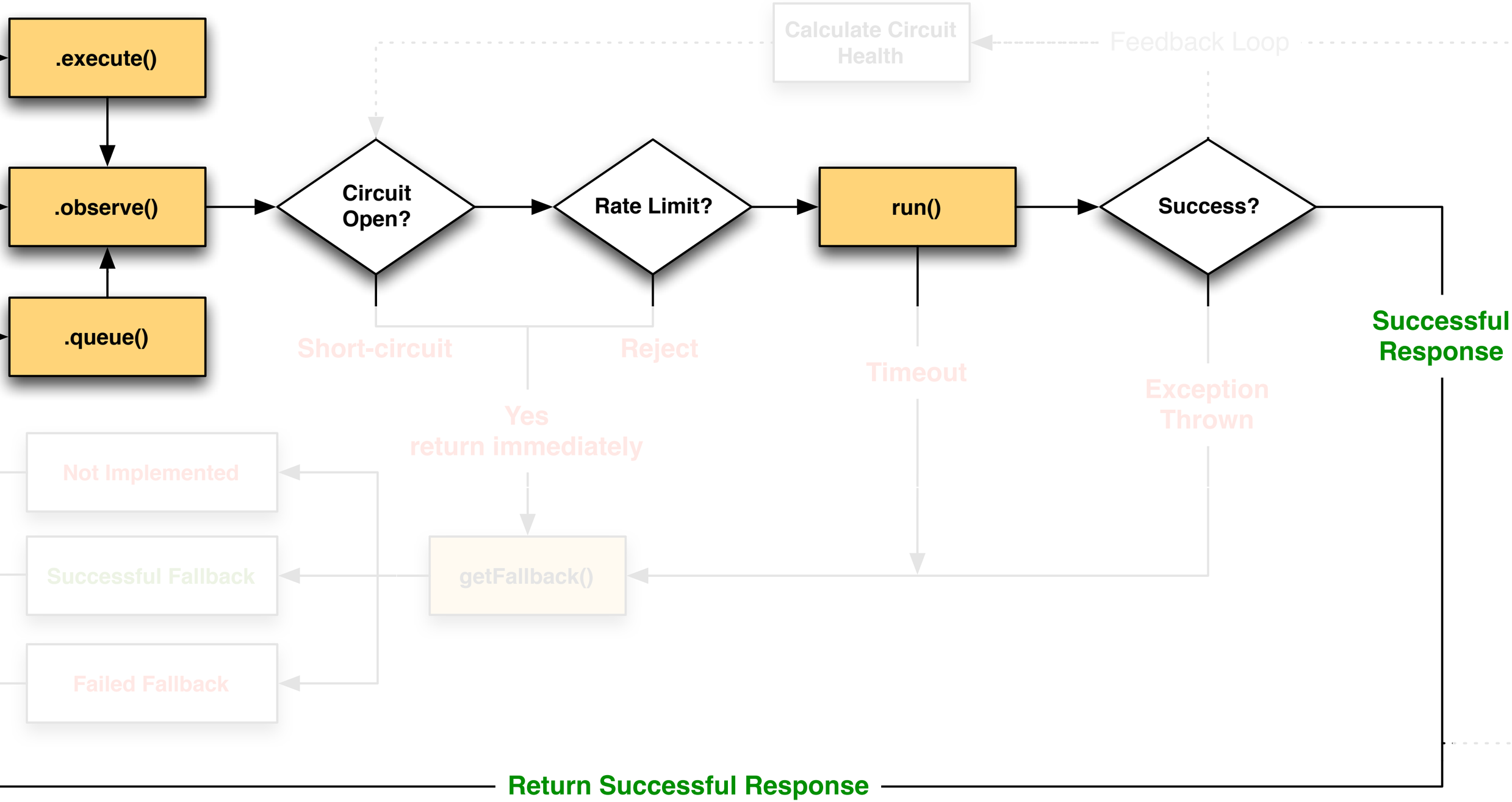
Execution can be synchronous or asynchronous (via a Future or Observable).



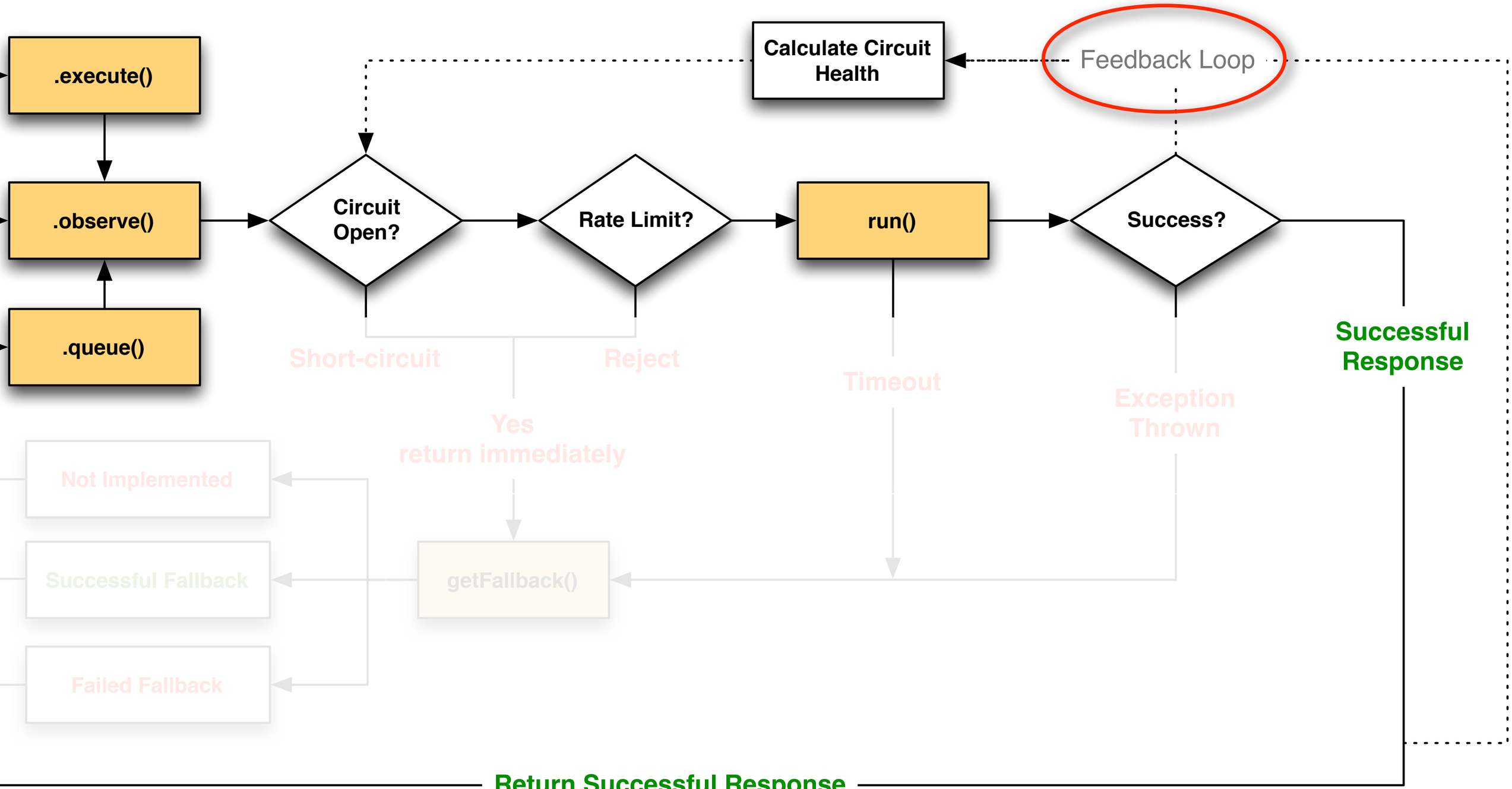
Current state is queried before allowing execution to determine if it is short-circuited or throttled and should reject.



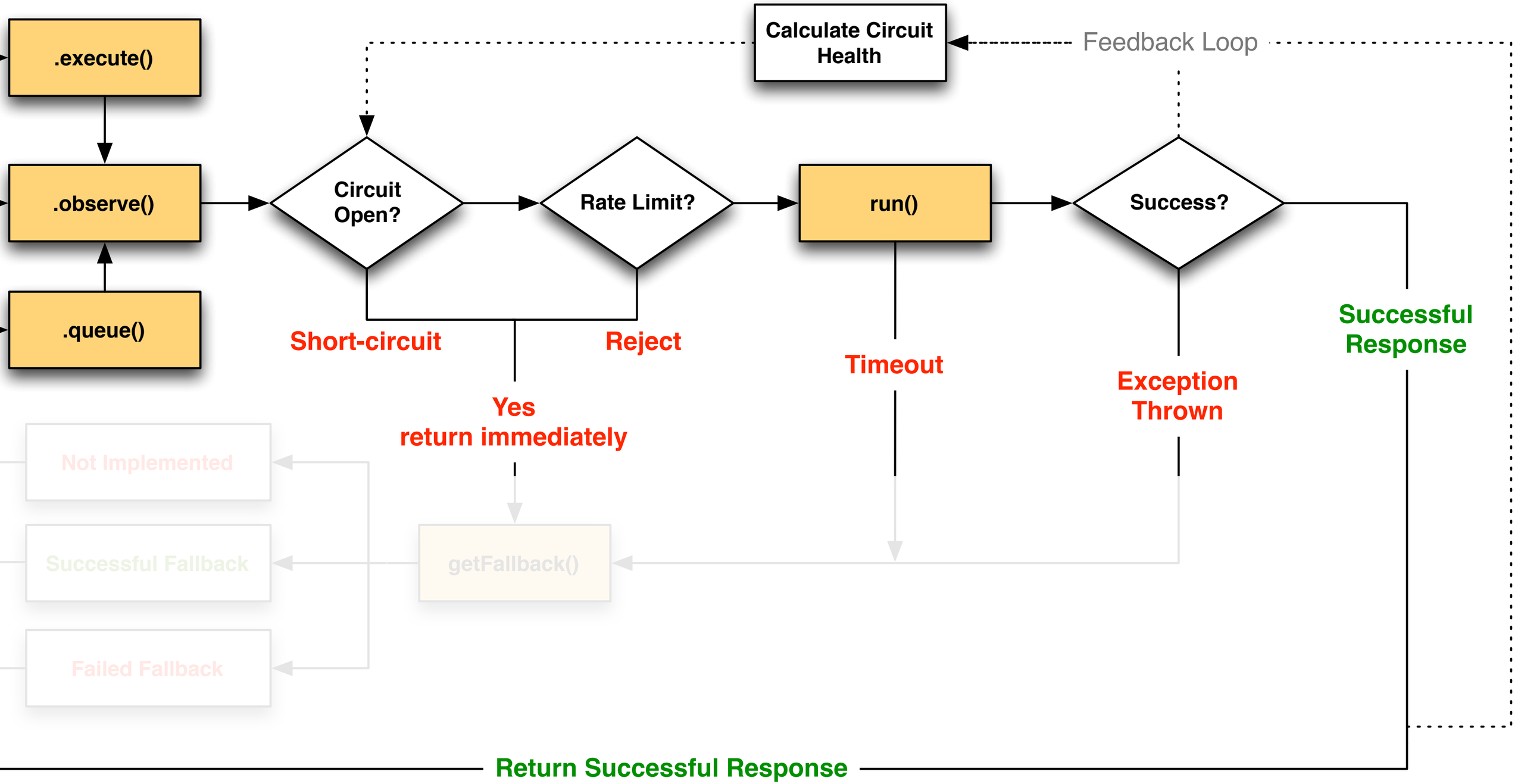
If not rejected execution proceeds to the run() method which performs underlying work.



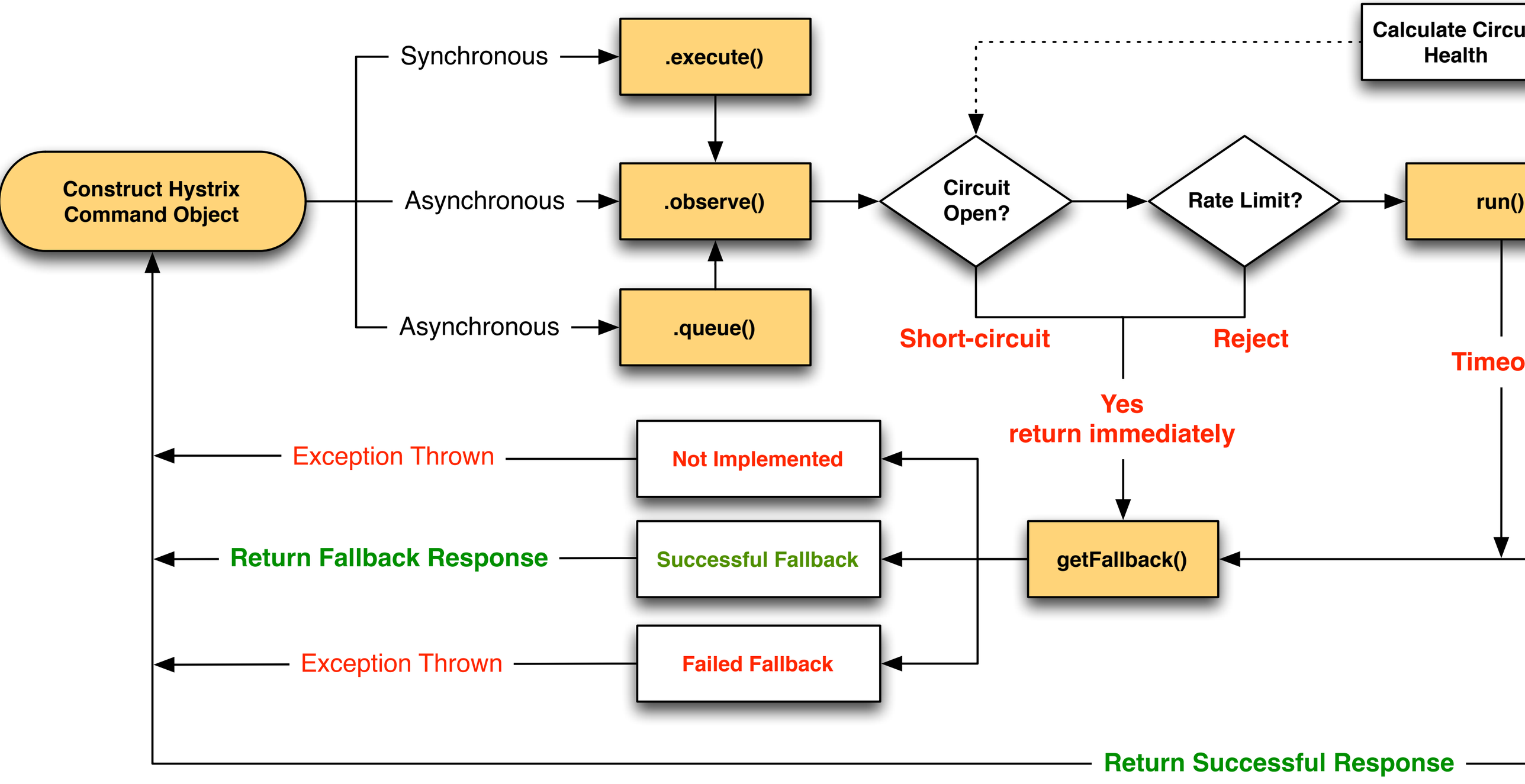
Successful responses return.



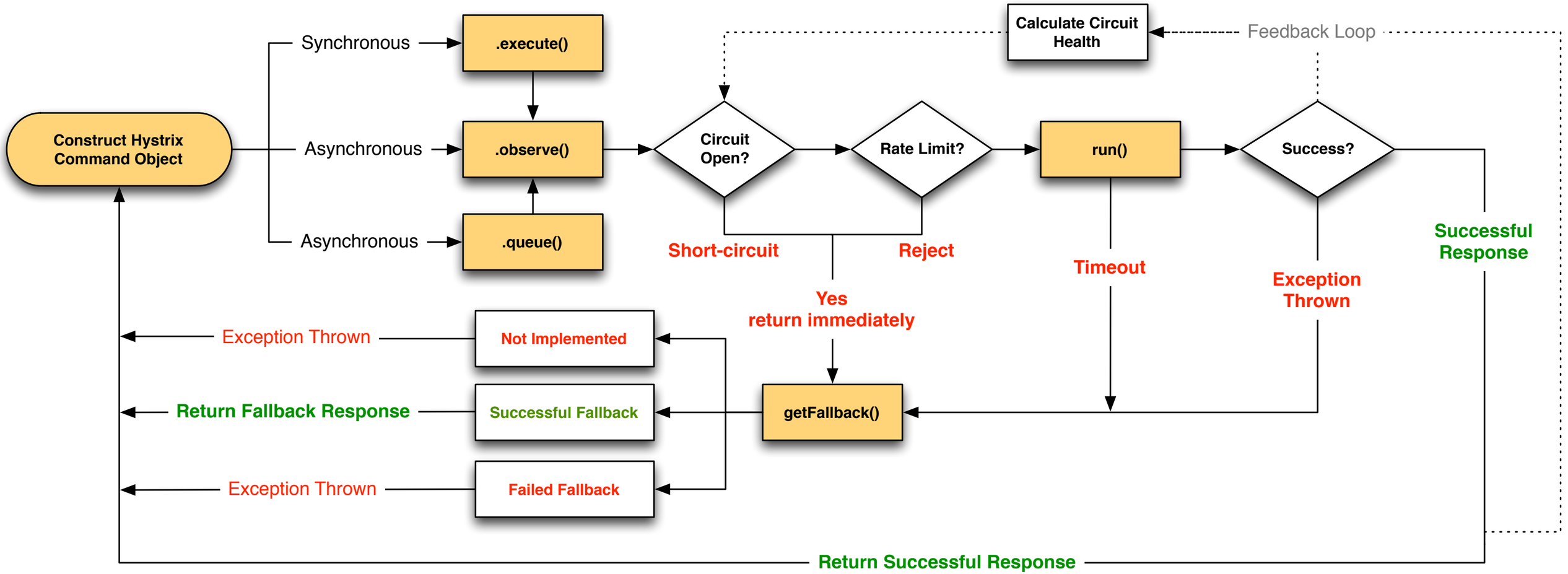
All requests, successful and failed, contribute to a feedback loop used to make decisions and publish metrics.



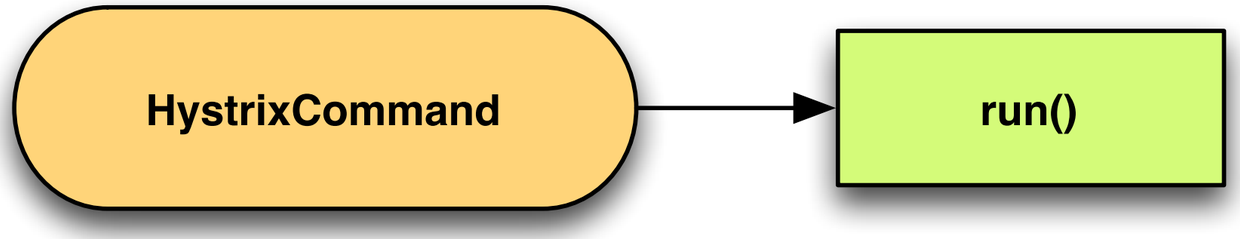
All failure states are routed through the same path.



Every failure is given the opportunity to retrieve a fallback which can result in one of three results.

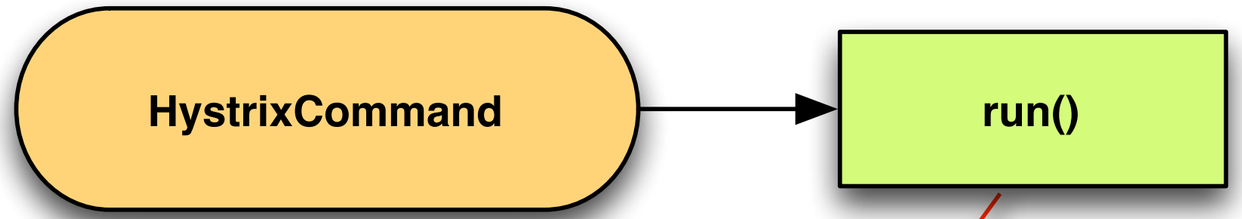


Hystrix execution flow chart. Read more at https://github.com/Netflix/Hystrix/wiki/How-it-Works#flow-chart




```
public class CommandHelloWorld extends HystrixCommand<String> {

    ...

    protected String run() {
        return "Hello " + name + "!";
    }
}
```

Basic successful execution pattern and sample code. Read more at https://github.com/Netflix/Hystrix/wiki/How-To-Use#wiki-Hello-World

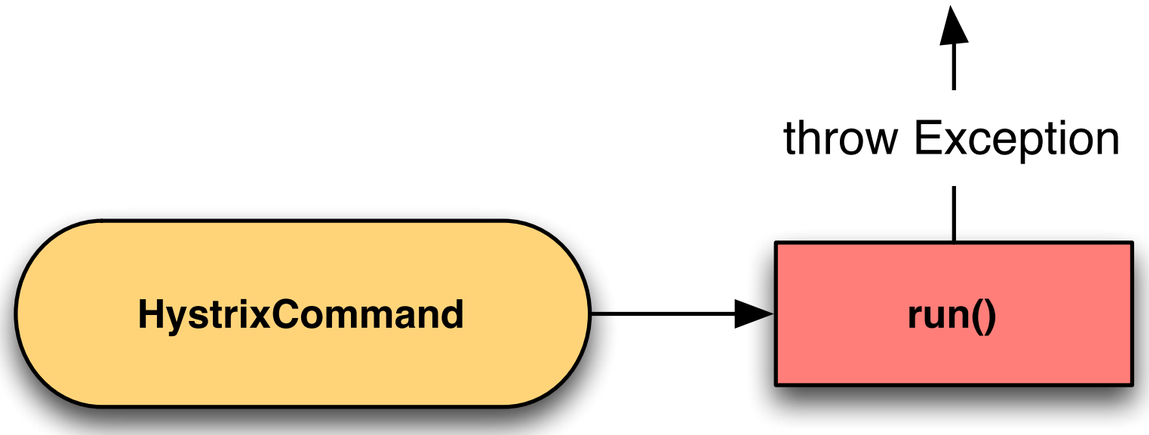



```
public class CommandHelloWorld extends HystrixCommand<String> {

    ...

    protected String run() {
        return "Hello " + name + "!";
    }
}
```

`run()` INVOKES "CLIENT" LOGIC

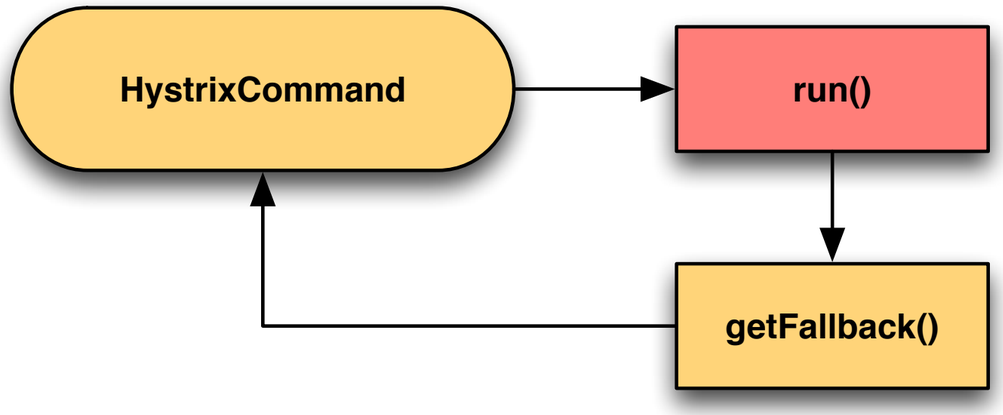
The run() method is where the wrapped logic goes.

FAIL FAST



Failing fast is the default behavior if no fallback is implemented. Even without a fallback this is useful as it prevents resource saturation beyond the bulkhead so the rest of the application can continue functioning and enables rapid recovery once the underlying problem is resolved. Read more at https://github.com/Netflix/Hystrix/wiki/How-To-Use#fail-fast

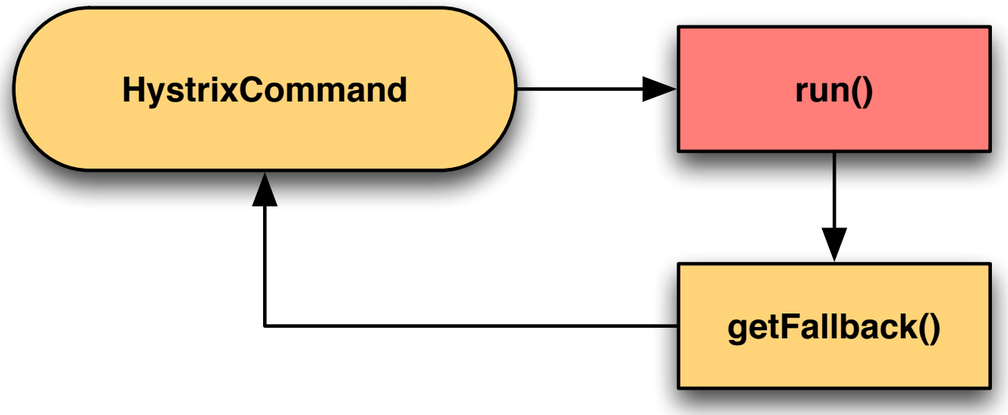
FAIL SILENT




```
return null;
return new Option<T>();
return Collections.emptyList();
return Collections.emptyMap();
```

Silent failure is an approach for removing non-essential functionality from the user experience by returning a value that equates to “no data”, “not available” or “don’t display”. Read more at https://github.com/Netflix/Hystrix/wiki/How-To-Use#fail-silent

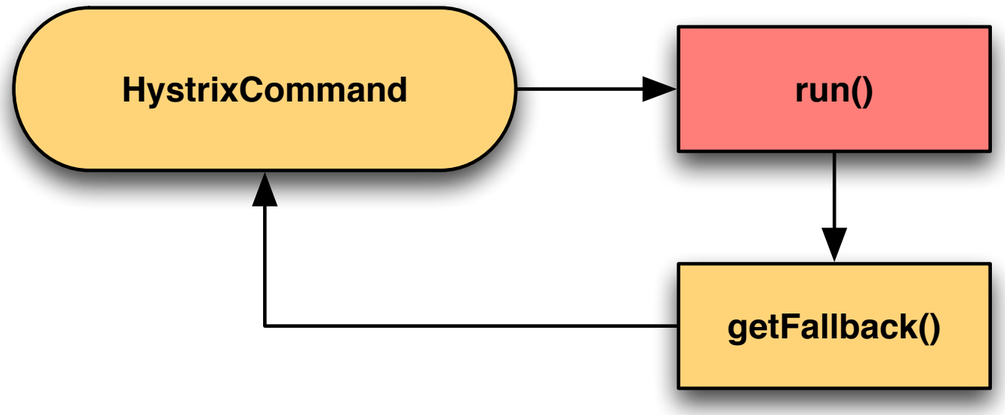
STATIC FALLBACK




```
return true;
return DEFAULT_OBJECT;
```

Static fallbacks can be used when default data or behavior can be returned to the user. Read more at https://github.com/Netflix/Hystrix/wiki/How-To-Use#fallback-static

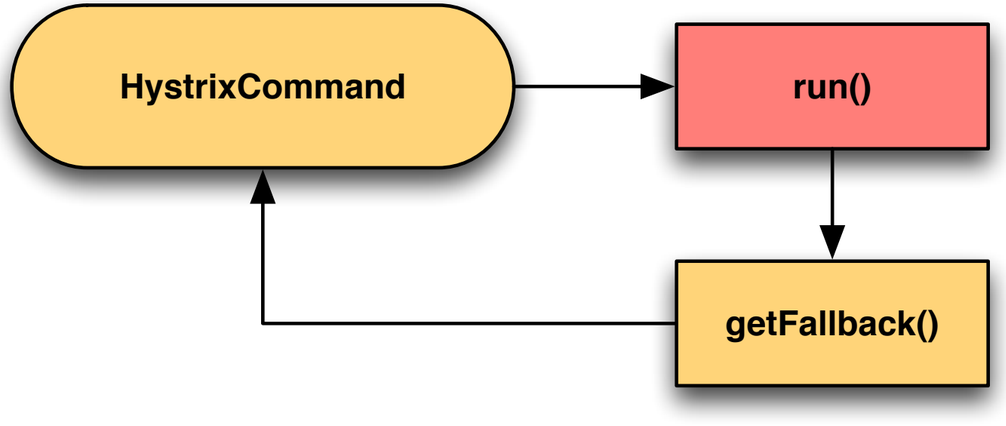
STUBBED FALLBACK




```
return new UserAccount(customerId, "Unknown Name",
                       countryCodeFromGeoLookup, true, true, false);
return new VideoBookmark(movieId, 0);
```

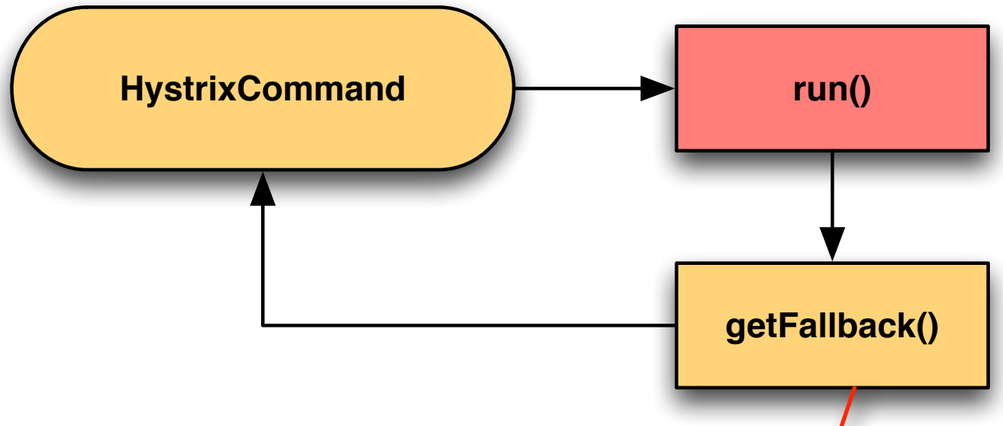
Stubbed fallbacks are an extension of static fallbacks when some data is available (such as from request arguments, authentication tokens or other functioning system calls) and combined with default values for data that can not be retrieved. Read more at https://github.com/Netflix/Hystrix/wiki/How-To-Use#fallback-stubbed

STUBBED FALLBACK




```
public class CommandHelloWorld extends HystrixCommand<String> {
    ...
    protected String run() {
        return "Hello " + name + "!";
    }
    protected String getFallback() {
        return "Hello Failure " + name + "!";
    }
}
```

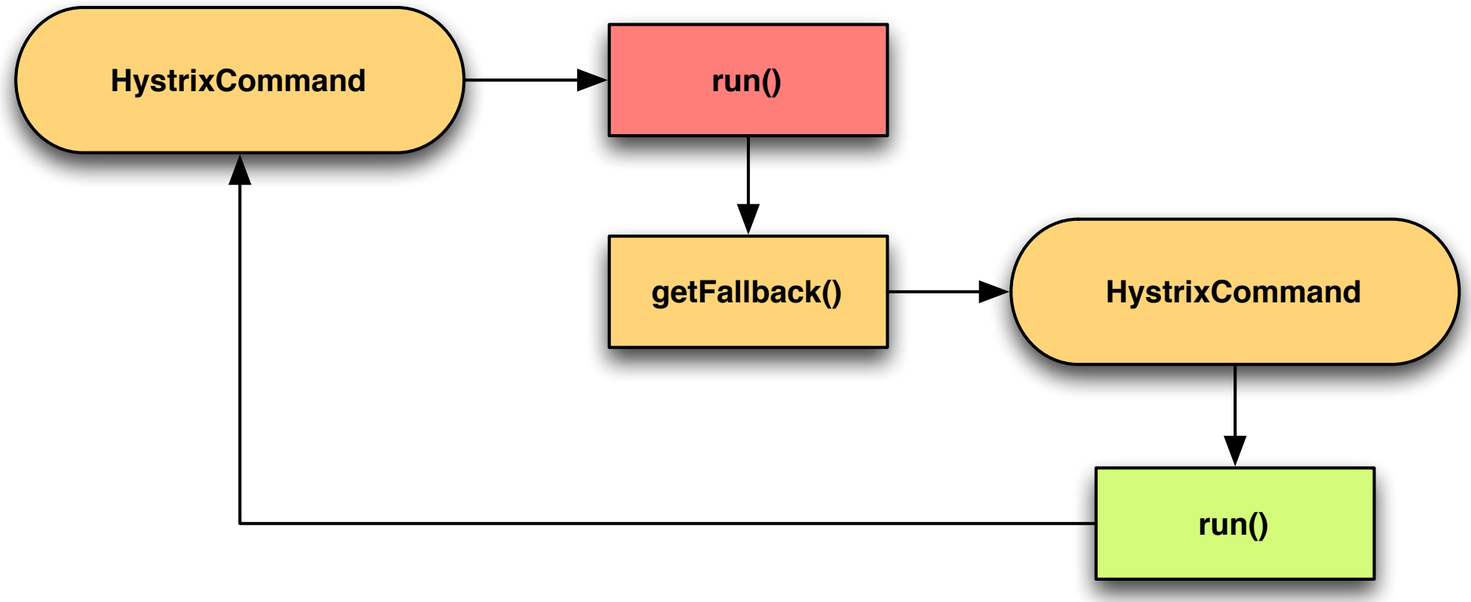
STUBBED FALLBACK




```
public class CommandHelloWorld extends HystrixCommand<String> {

    ...

    protected String run() {
        return "Hello " + name + "!";
    }

    protected String getFallback() {
        return "Hello Failure " + name + "!";
    }
}
```

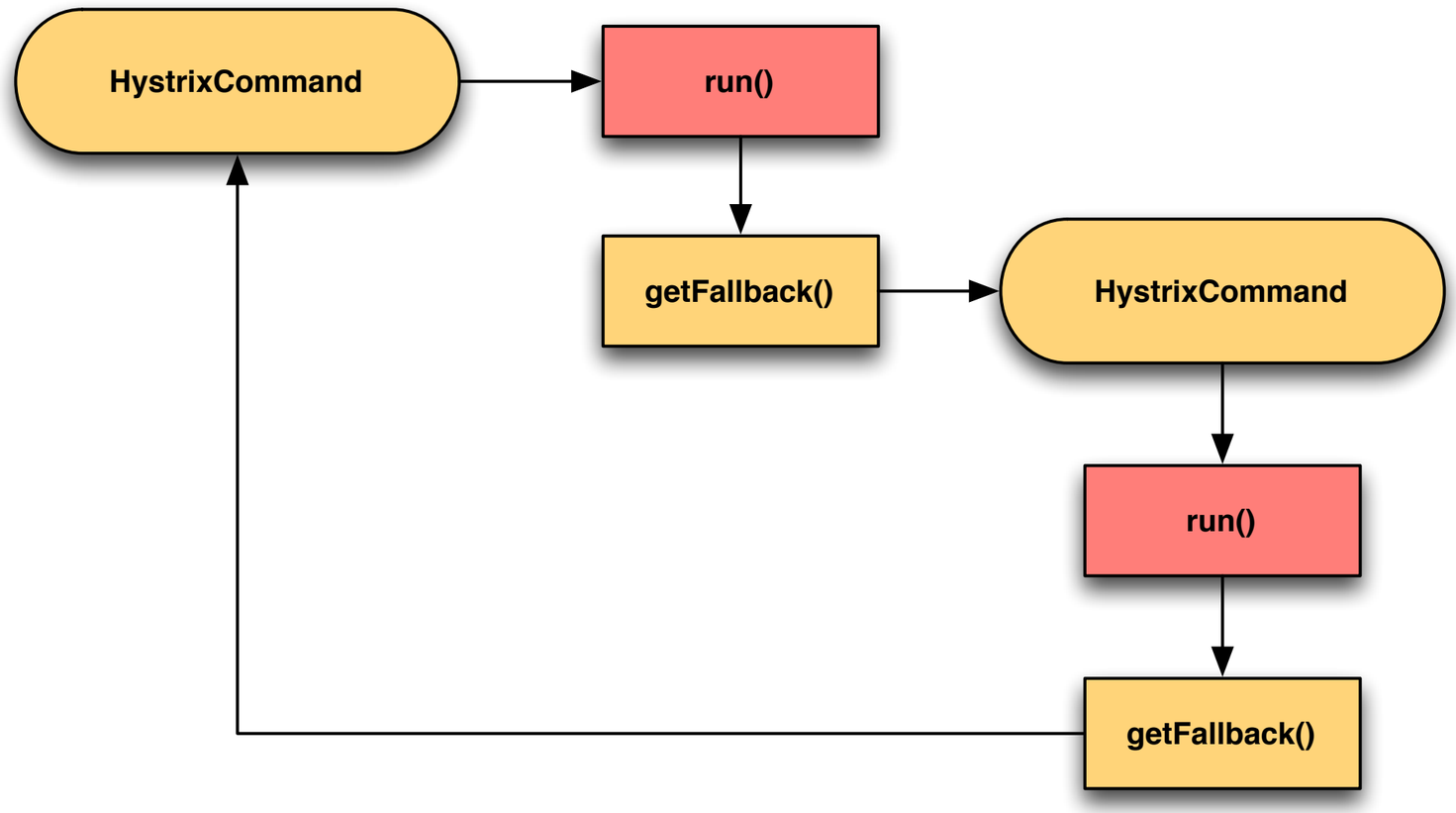
The getFallback() method is executed whenever failure occurs (after run() invocation or on rejection without run() ever being invoked) to provide opportunity to do fallback.

FALLBACK VIA NETWORK



Fallback via network is a common approach for falling back to a stale cache (such as a memcache server) or less personalized value when not able to fetch from the primary source. Read more at https://github.com/Netflix/Hystrix/wiki/How-To-Use#fallback-cache-via-network

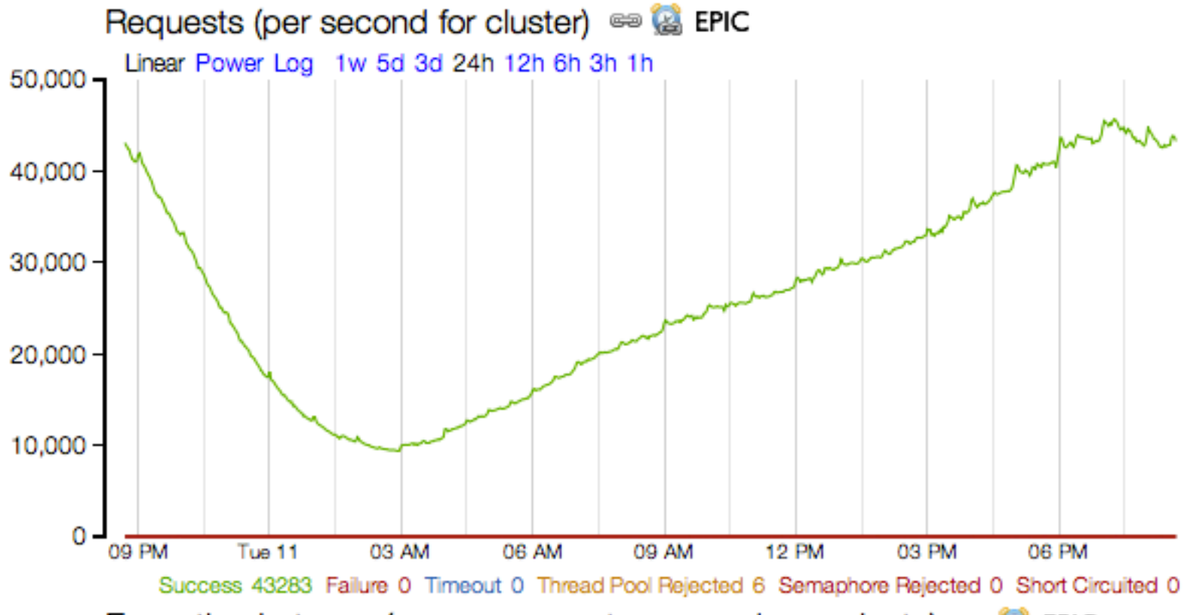
FALLBACK VIA NETWORK THEN LOCAL

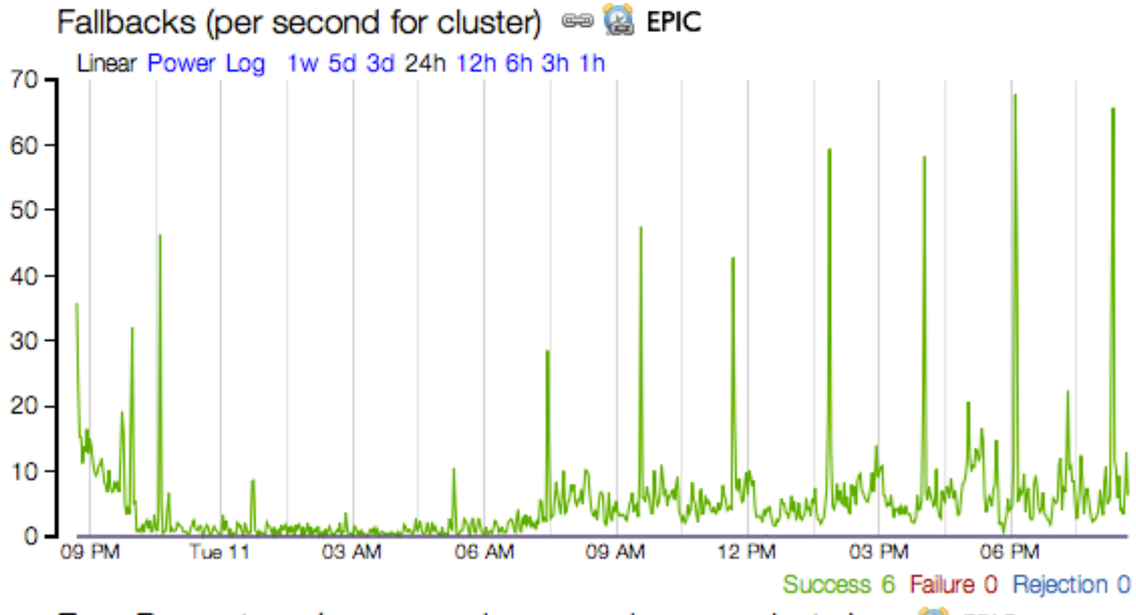


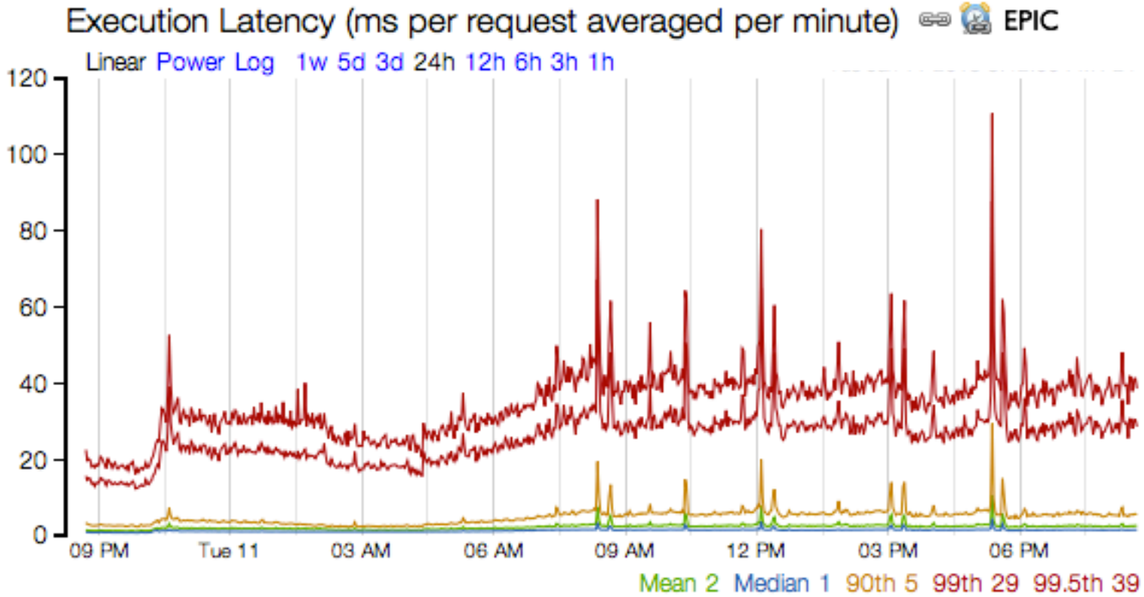
When the fallback performs a network call it's preferable for it to also have a fallback that does not go over the network otherwise if both primary and secondary systems fail it will fail by throwing an exception (similar to fail fast except after two fallback attempts).

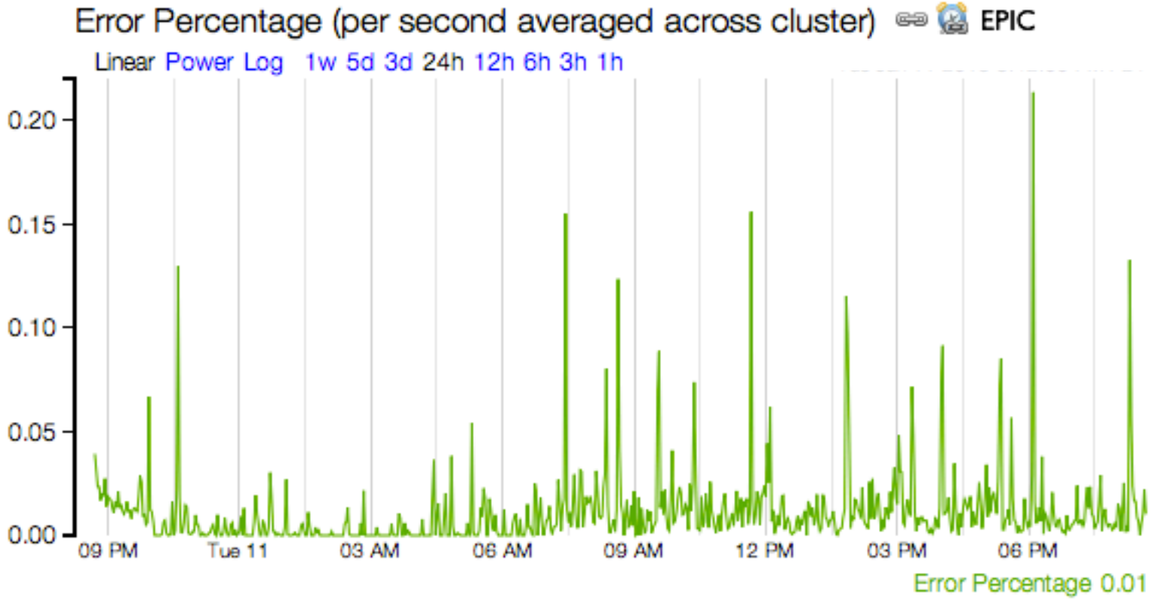
SO NOW WHAT?

Code is only part of the solution. Operations is the other critical half.

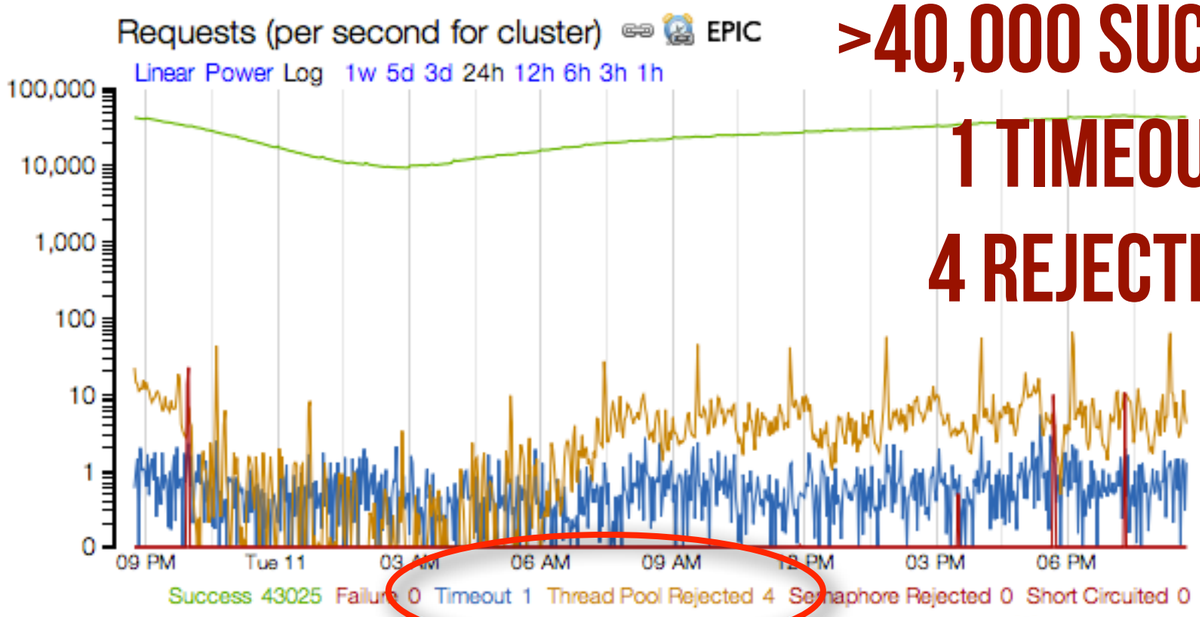








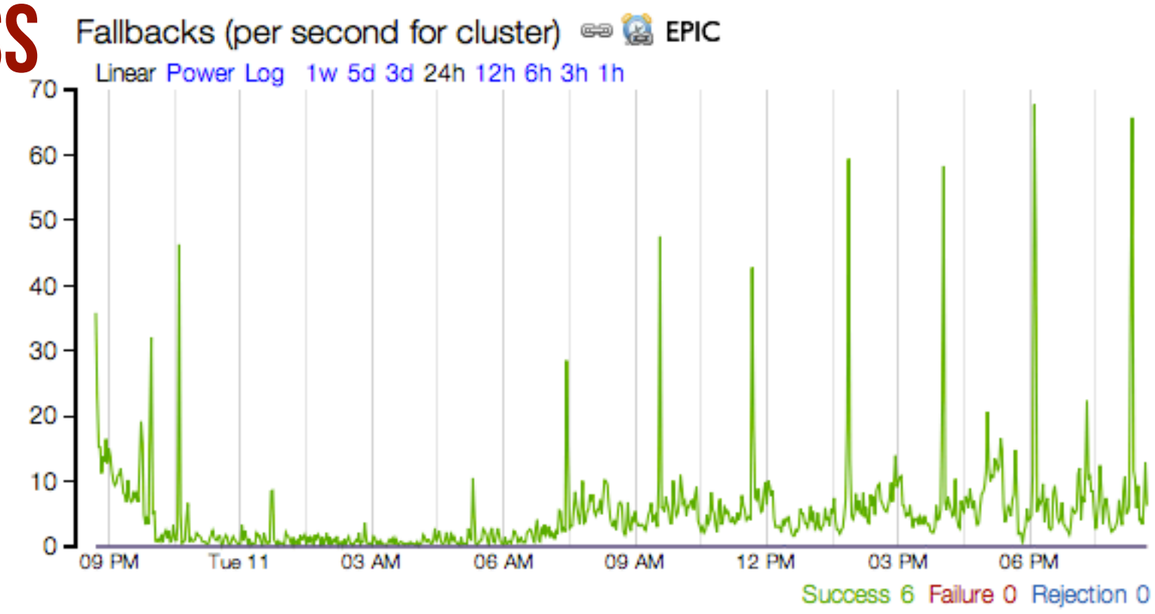
Historical metrics representing all possible states of success, failure, decision making and performance related to each bulk head.

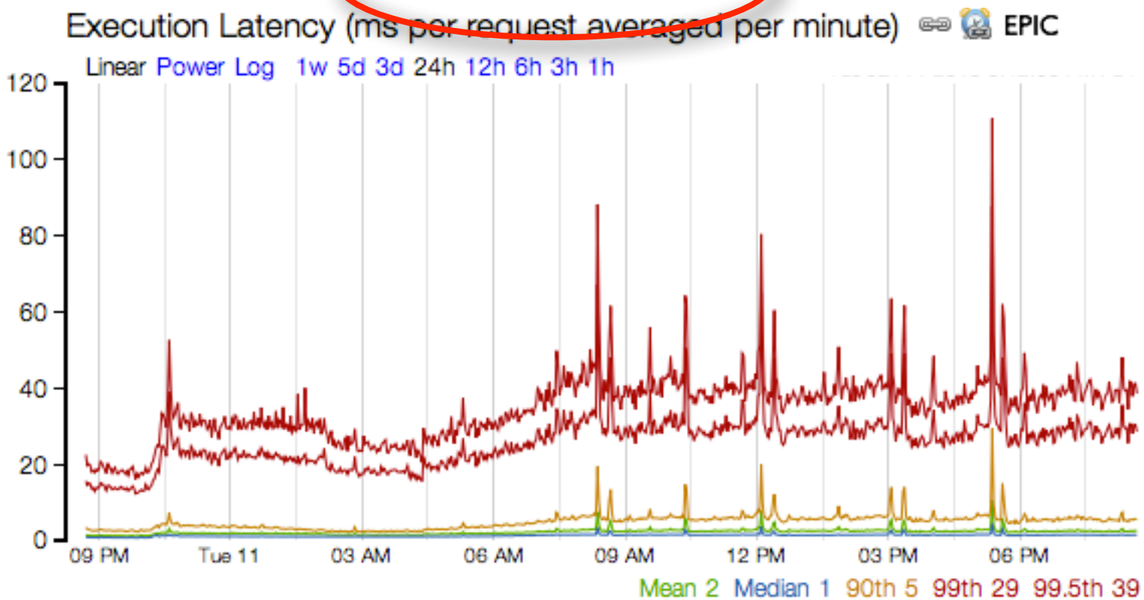


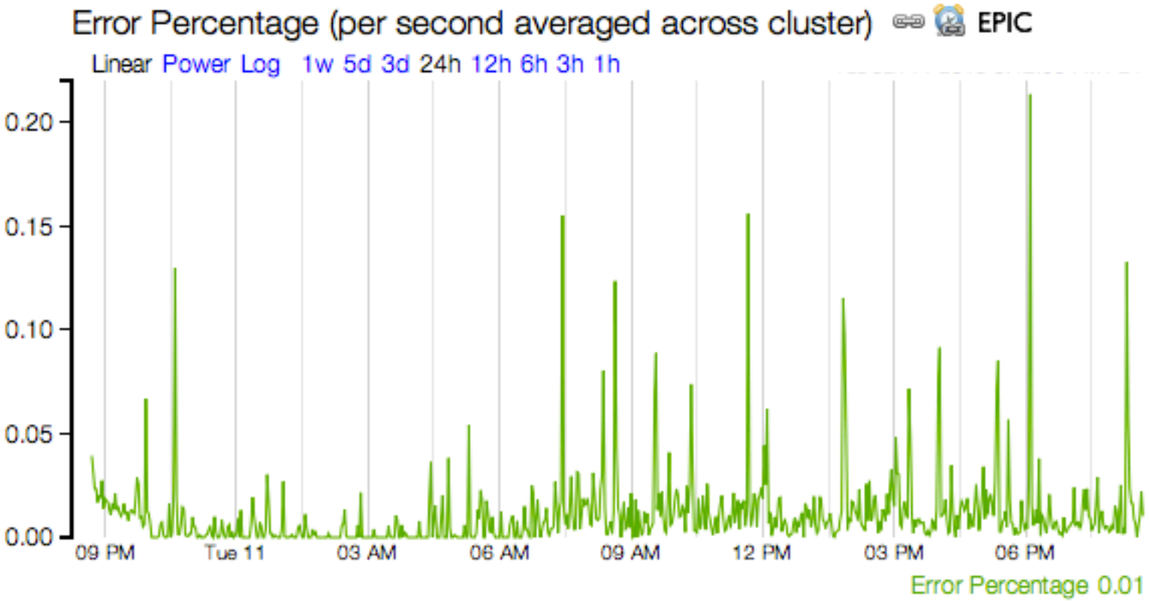
>40,000 SUCCESS

1 TIMEOUT

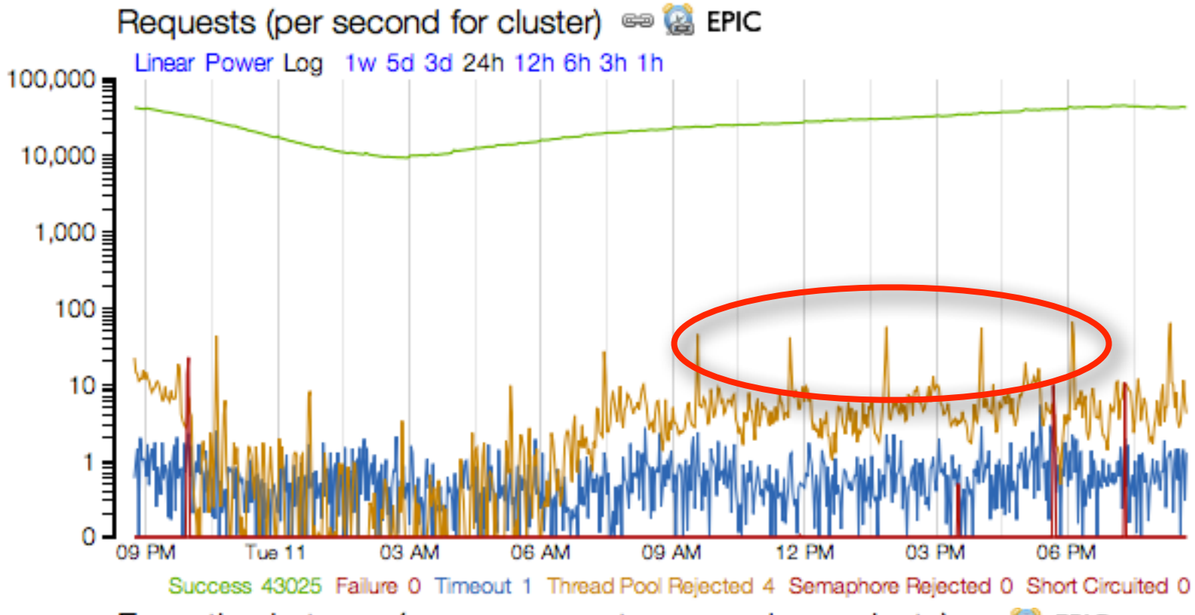
4 REJECTED

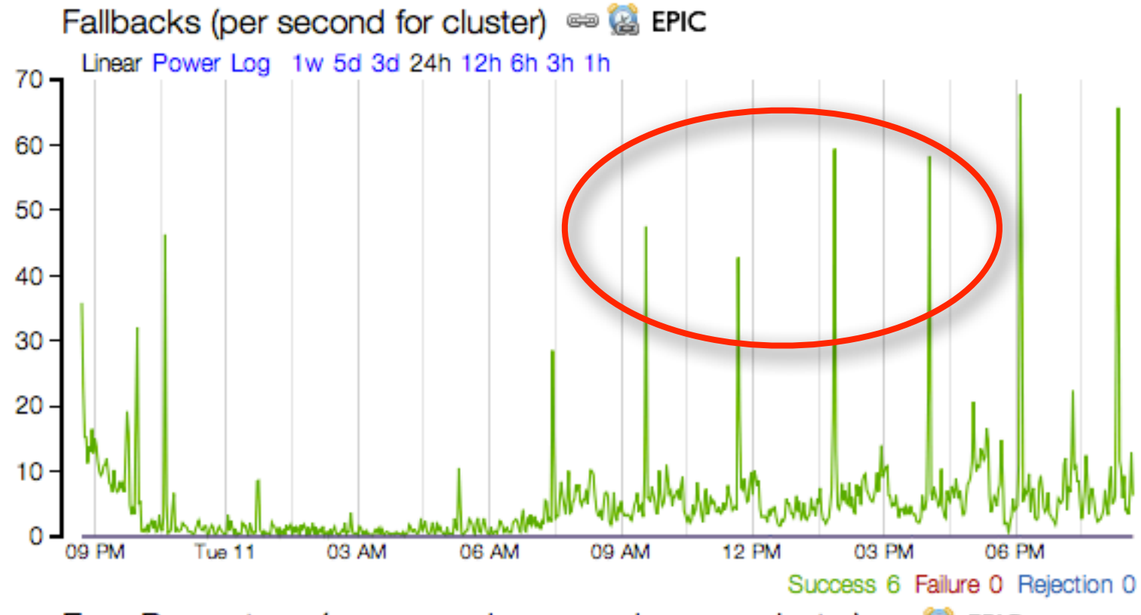


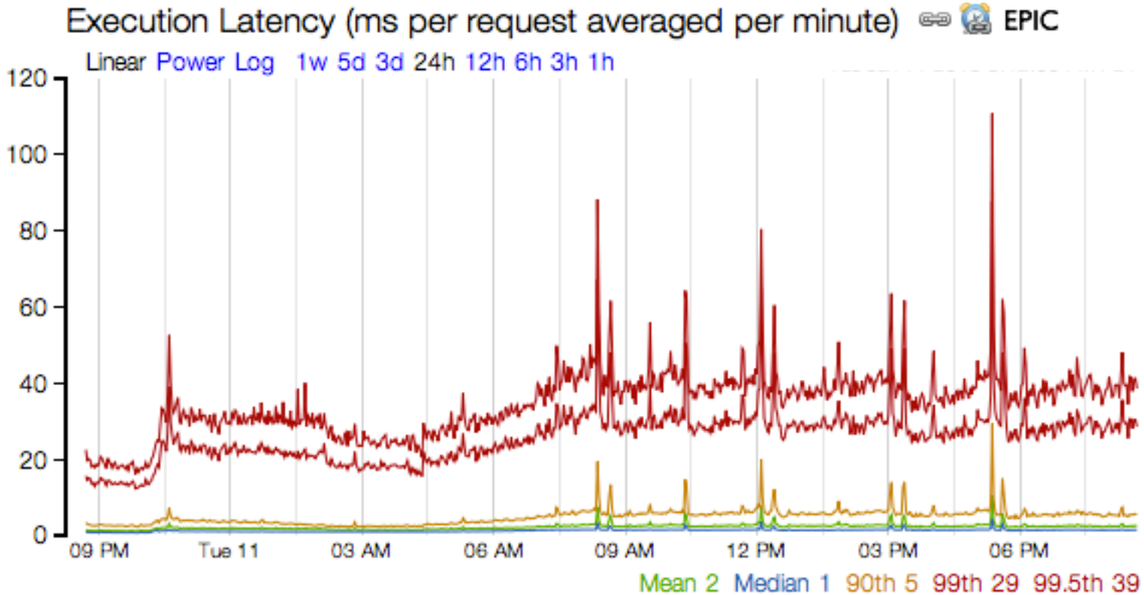


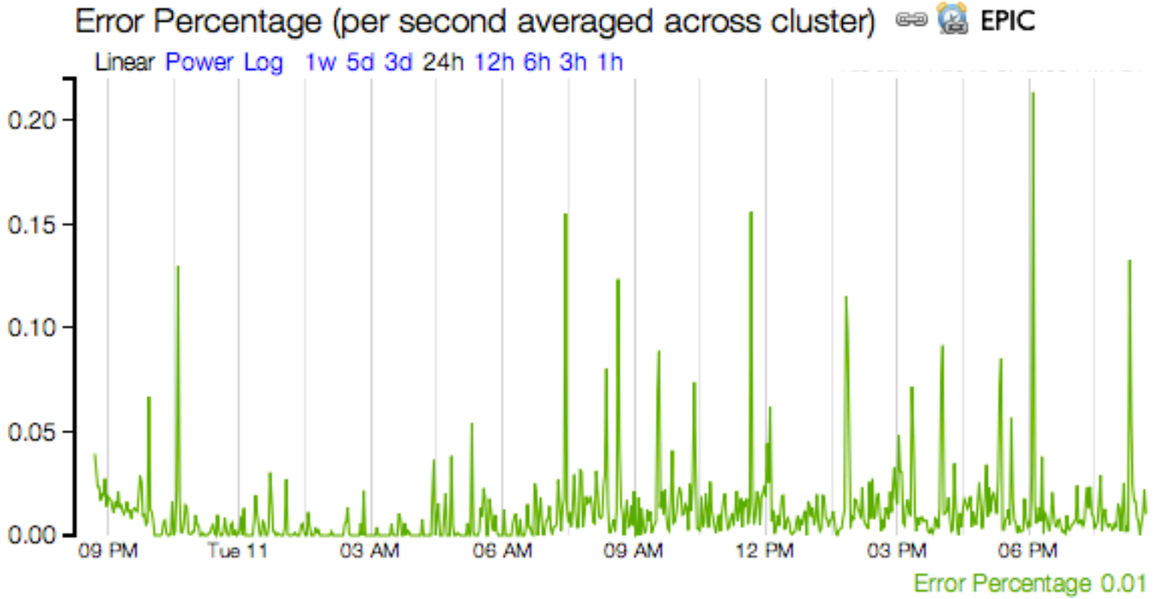


Looking closely at high volume systems it is common to find constant failure.

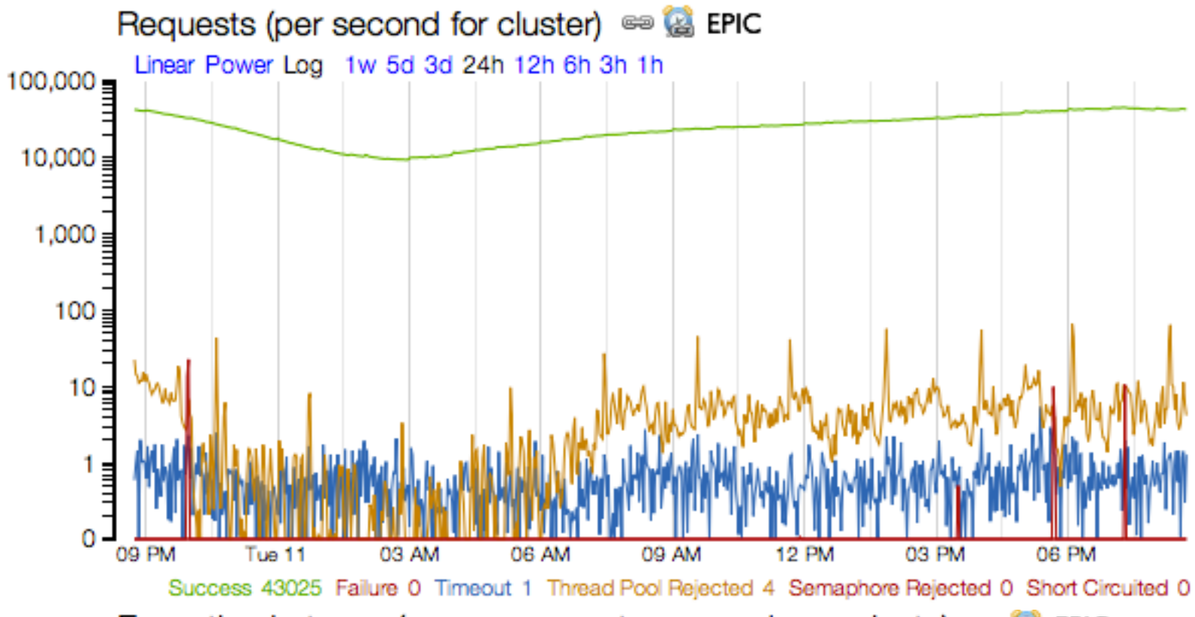


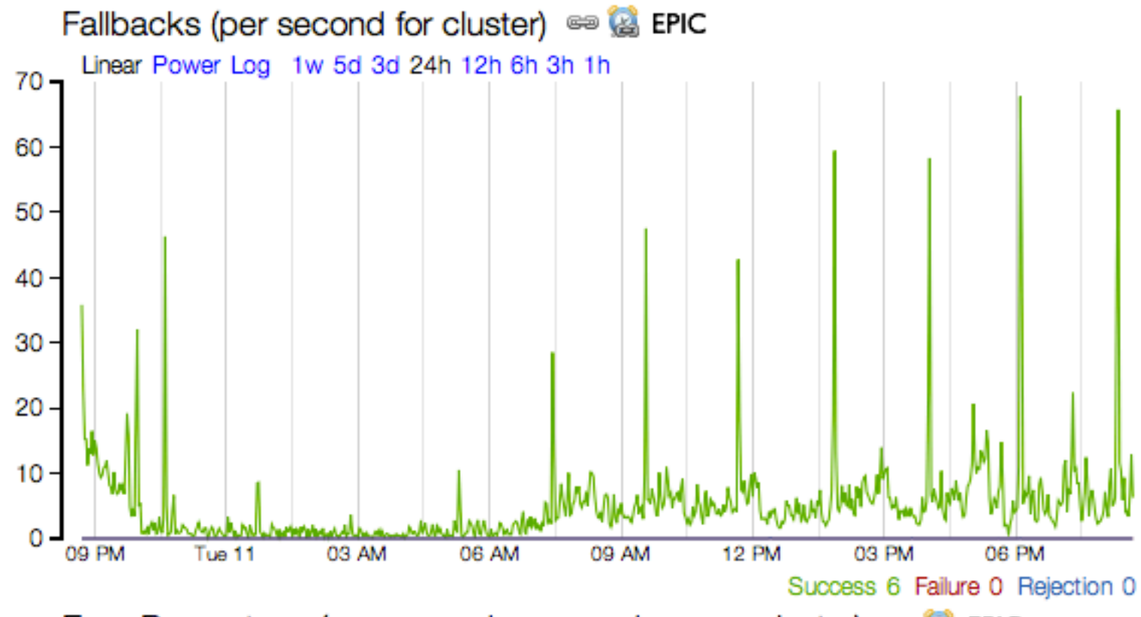


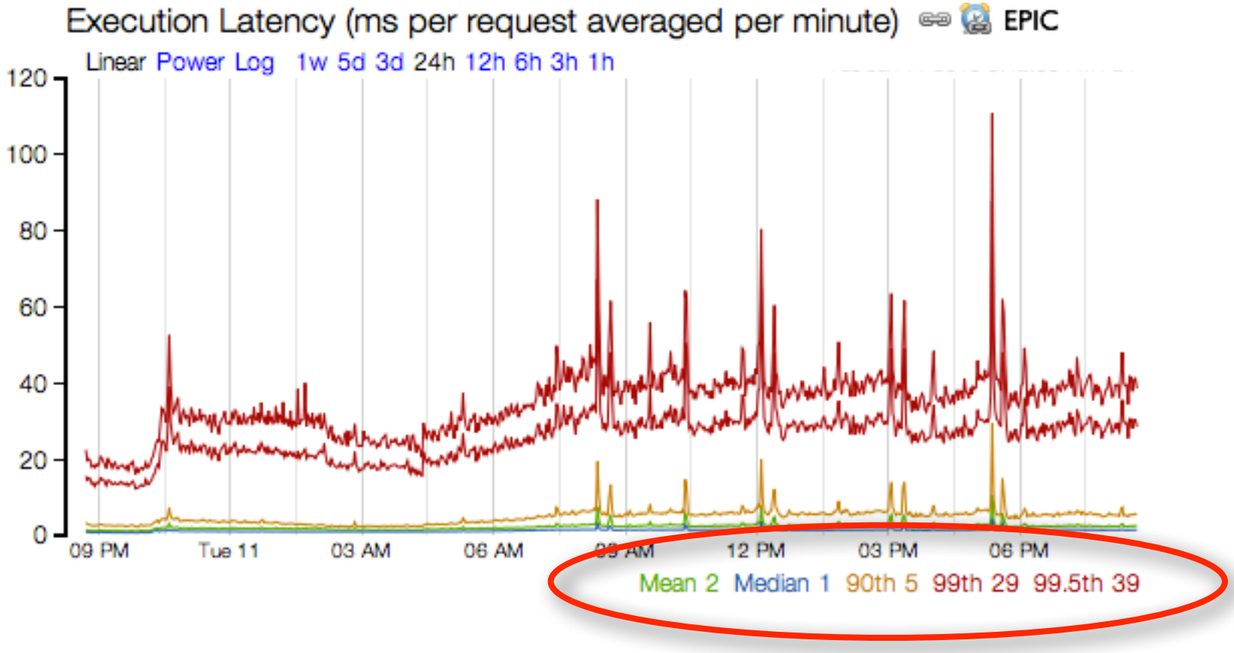


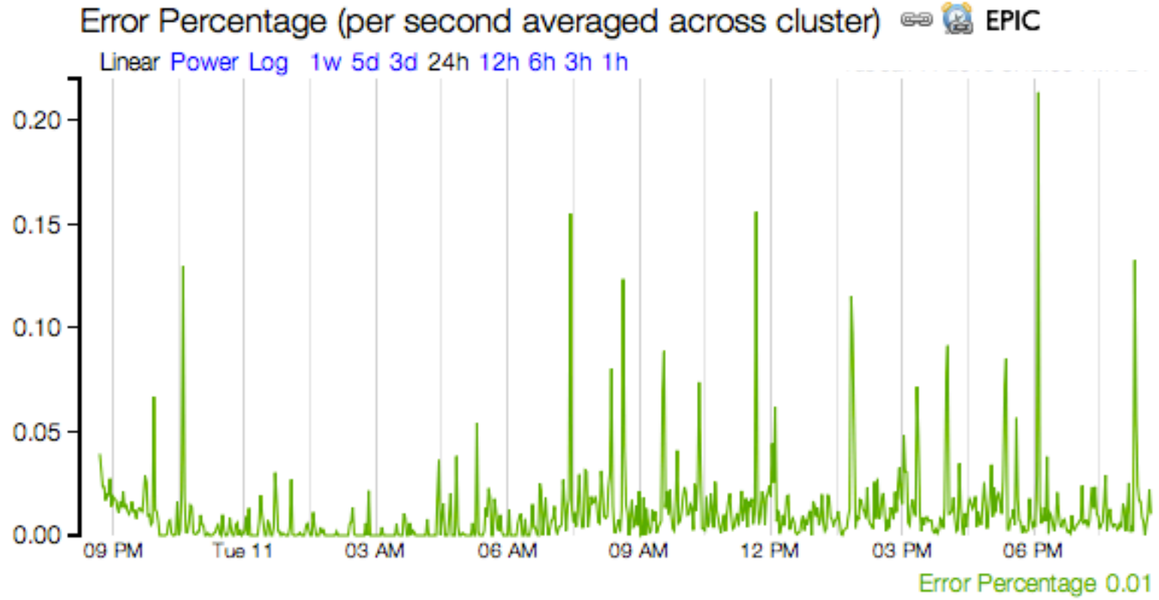


The rejection spikes on the left correlate with and do in fact represent the cause of the fallback spikes on the right.



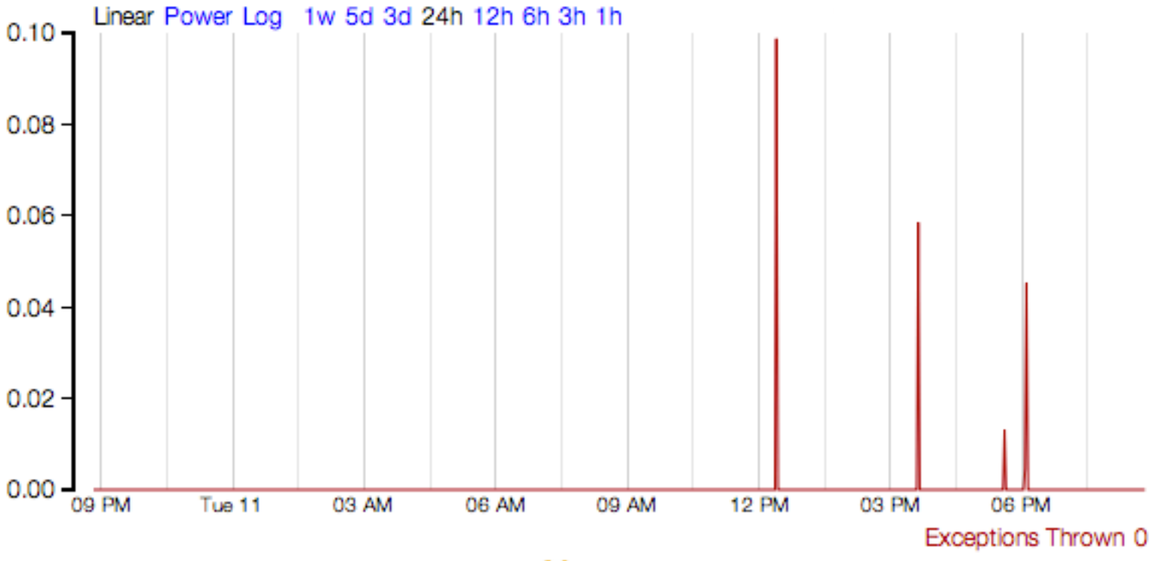




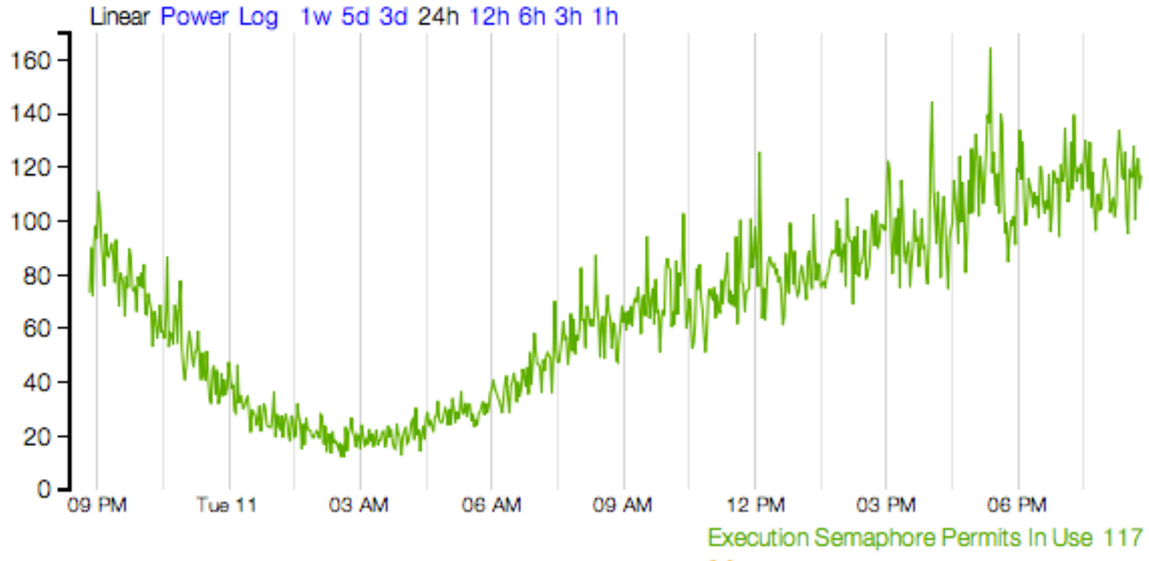


Latency percentiles are captured at every 5th percentile and a few extra such as 99.5th (though this graph is only showing 50th/99th/99.5th).

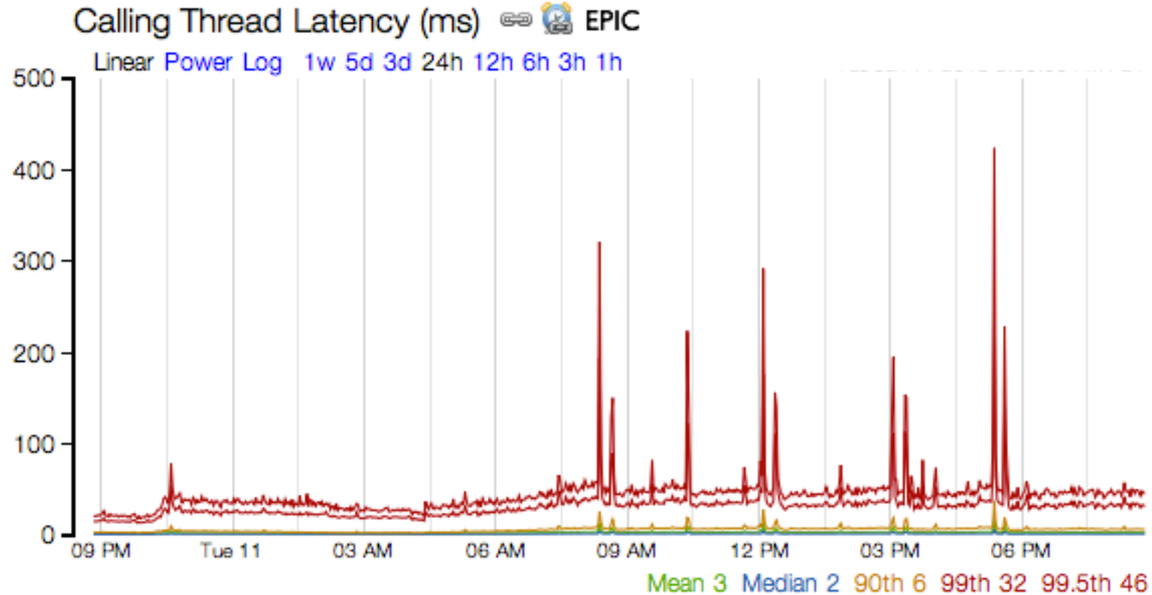
Exceptions Thrown (per second for cluster) EPIC



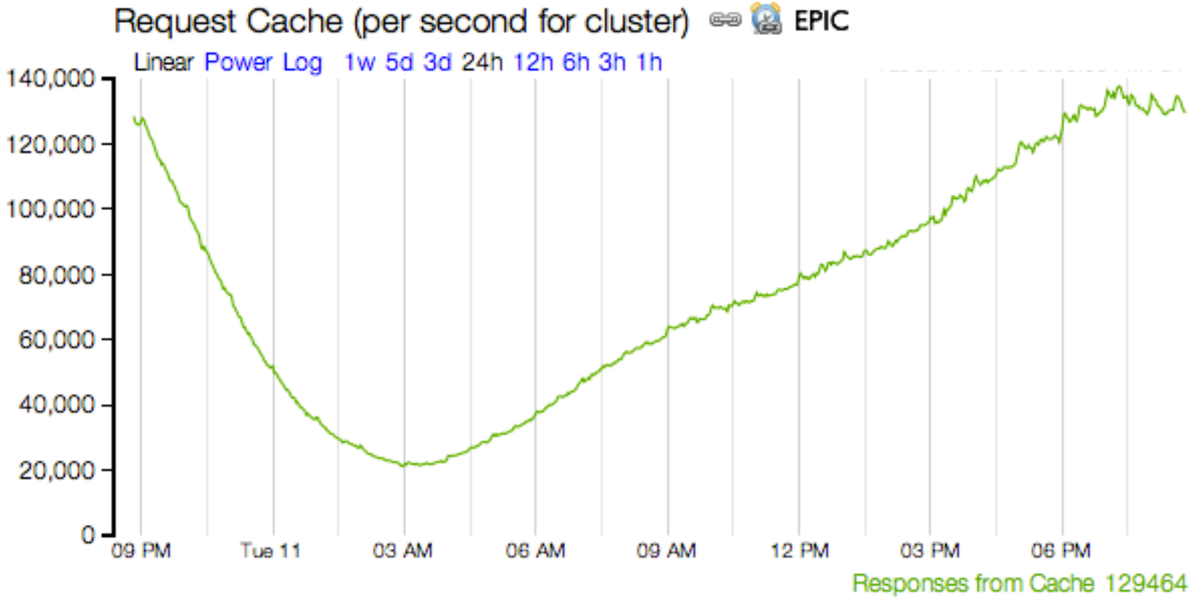
Semaphore Permits In Use (per second for cluster) EPIC

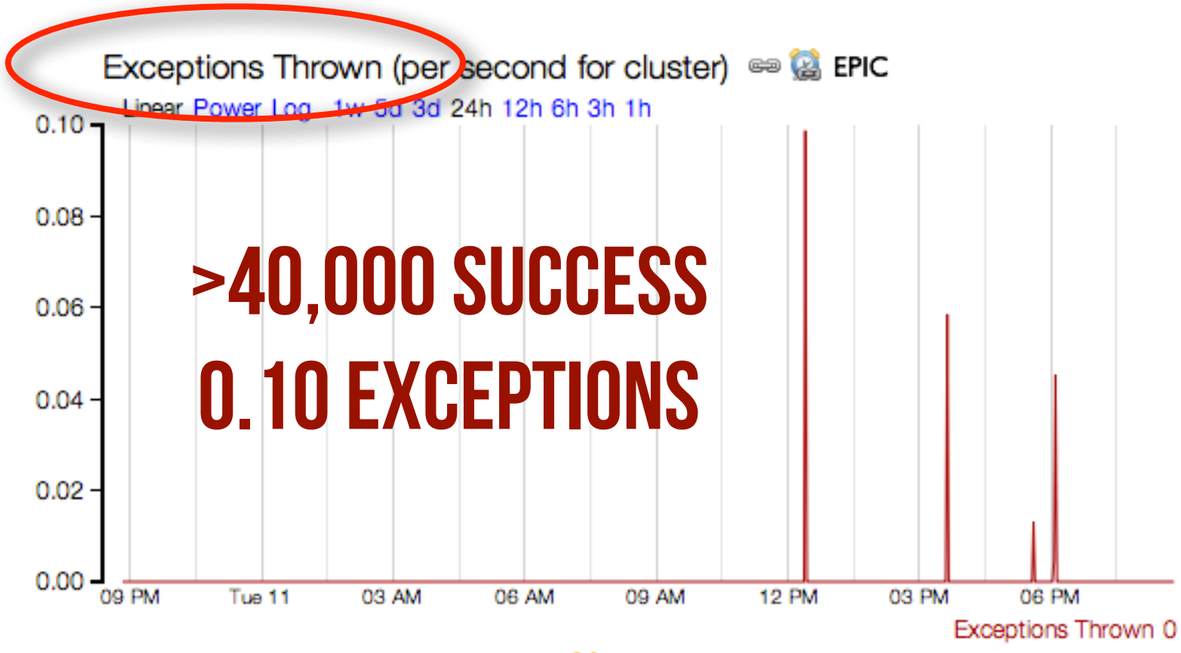


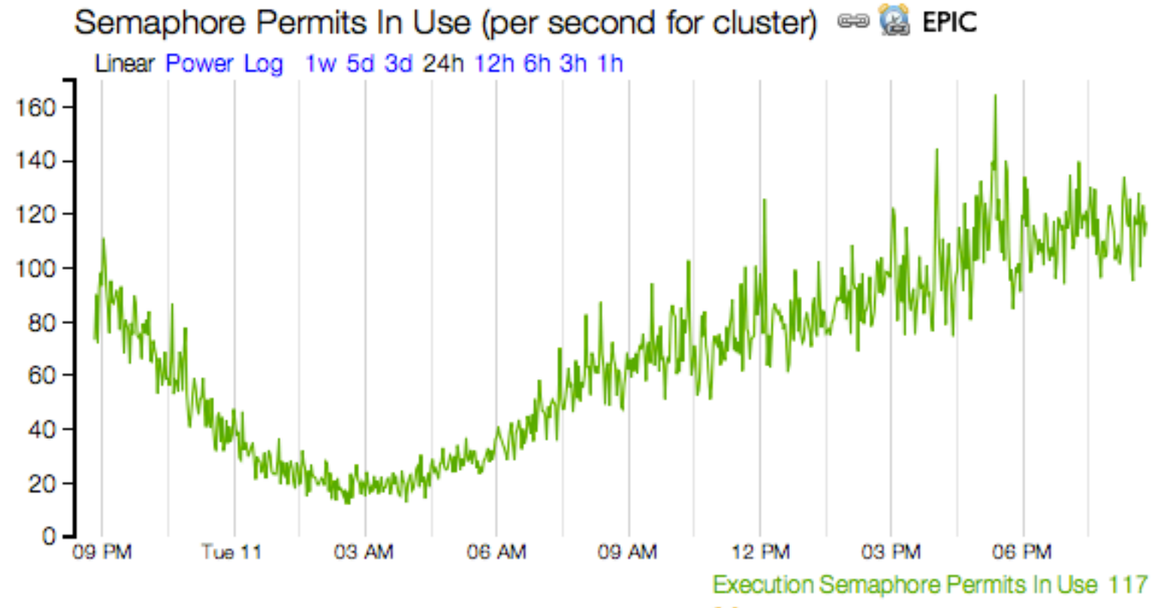
Calling Thread Latency (ms) EPIC



Request Cache (per second for cluster) EPIC

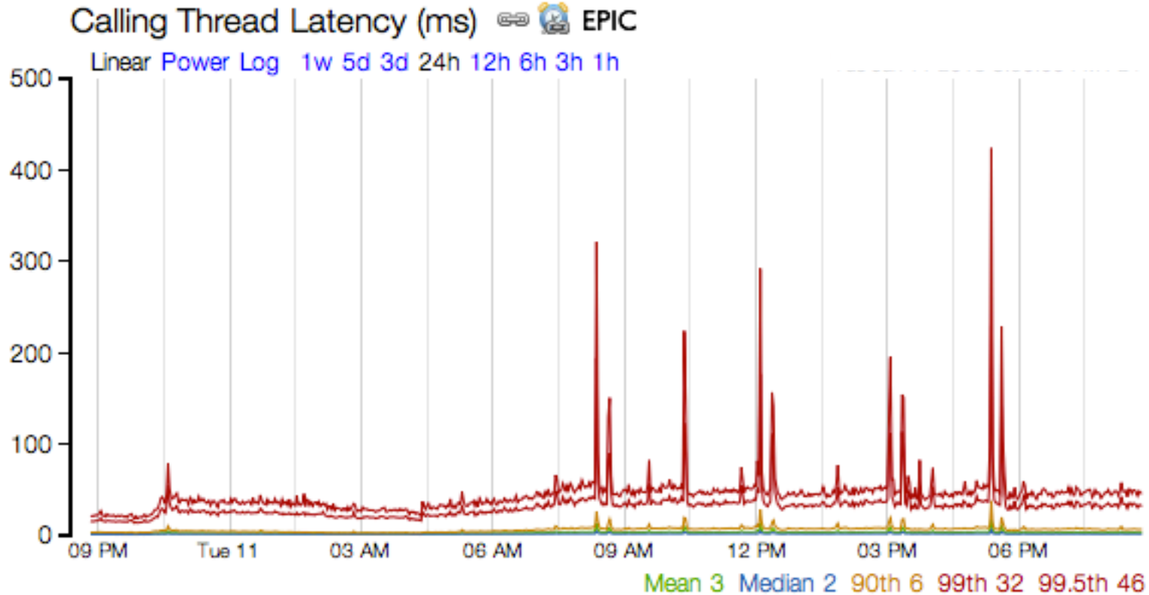


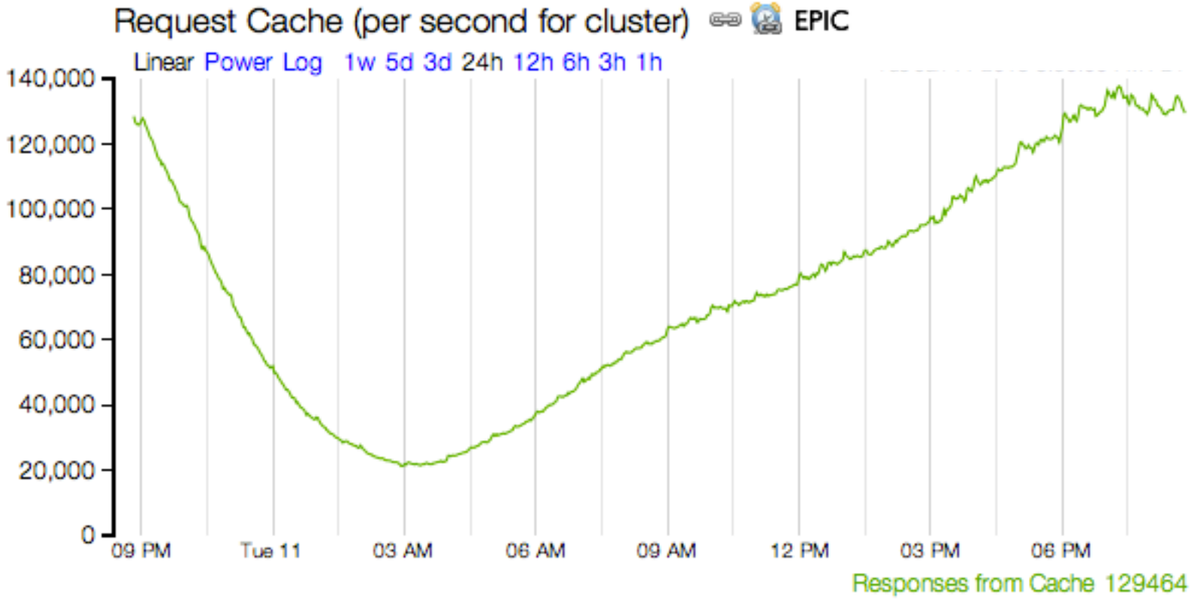




>40,000 SUCCESS

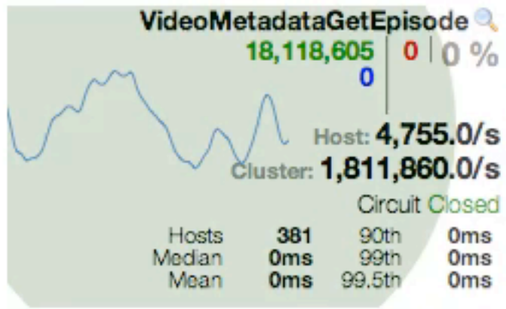
0.10 EXCEPTIONS

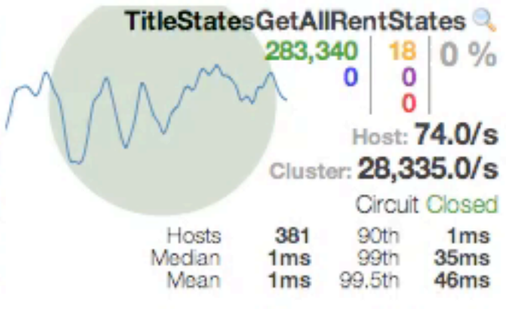


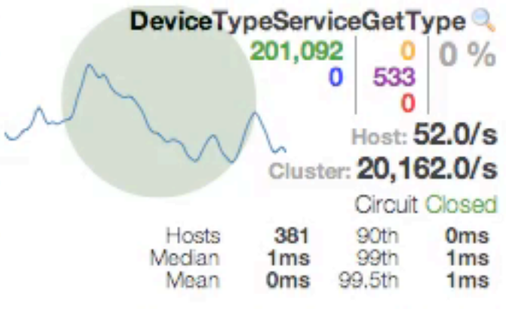


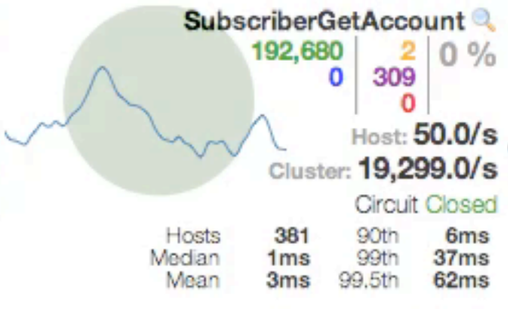
Exceptions Thrown helps to identify if a failure state is being handled by a fallback successfully or not. In this case we are seeing < 0.1 exceptions per second being thrown but on the previous set of metrics saw 5-40 fallbacks occurring each second, thus we can see that the fallbacks are doing their job but we may want to look for very small number of edge cases where fallbacks fail resulting in an exception.

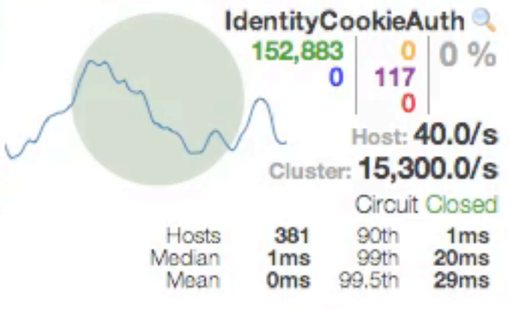
Circuit Breakers Sort: Error then Volume | Alphabetical | Volume | Error | Mean | Median | 90 | 99 | 99.5 Success | Latent | Short-Circuited | Timeout | Rejected | Failure | Error %

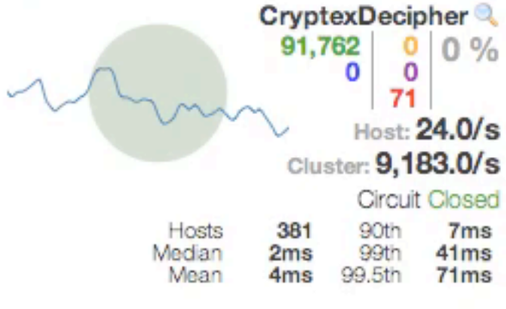


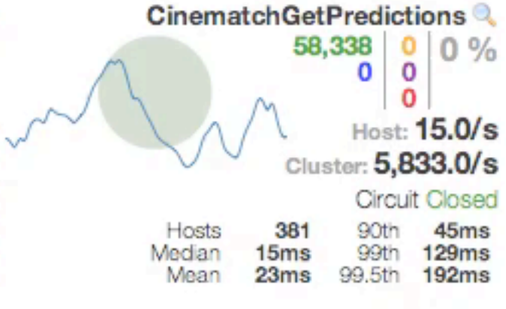


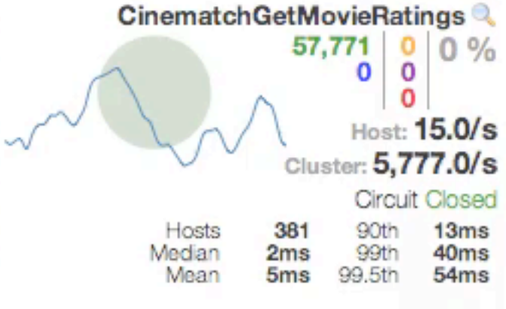


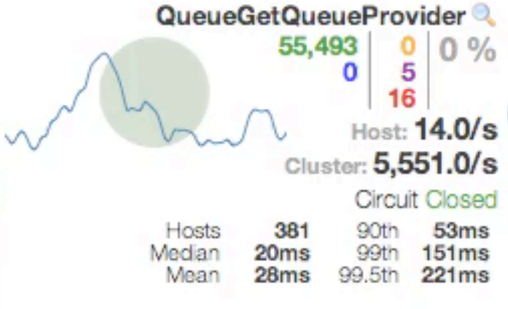


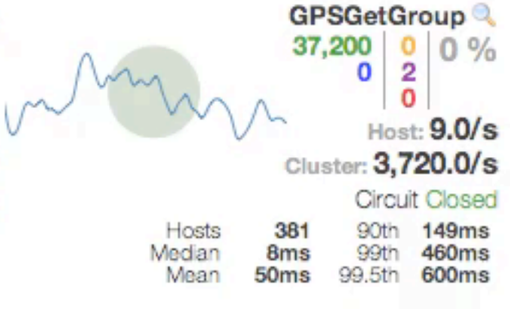


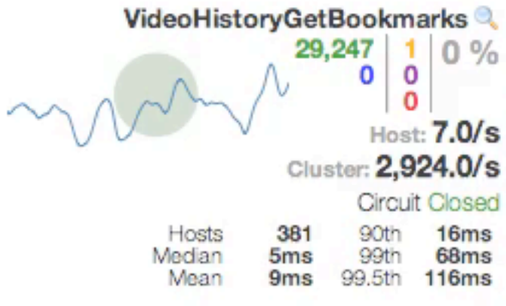


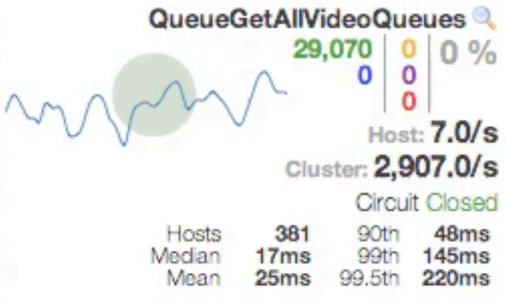


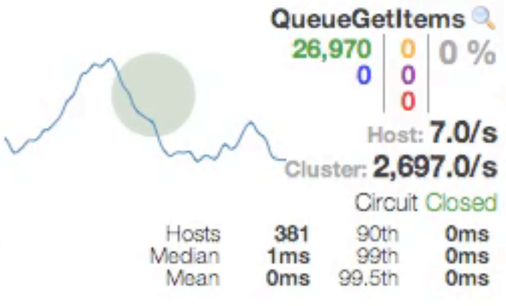


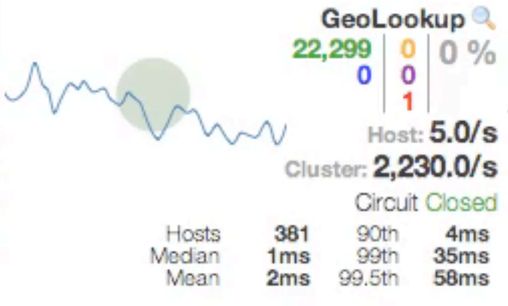


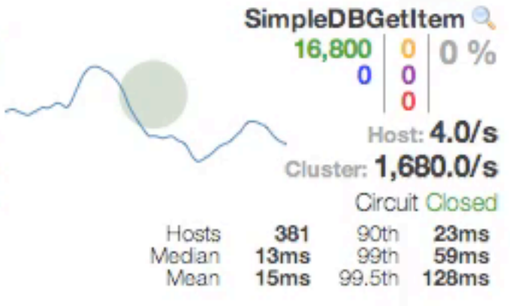


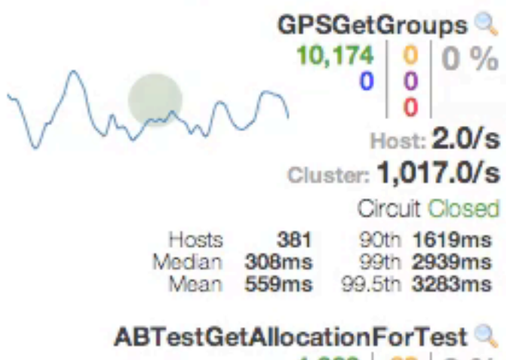


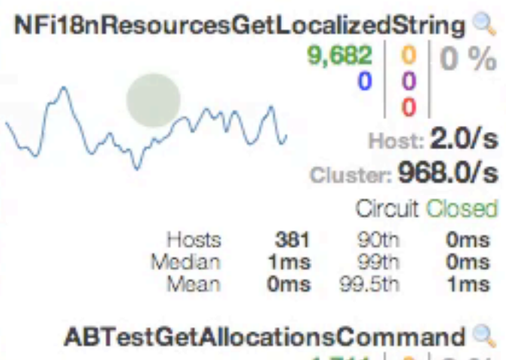


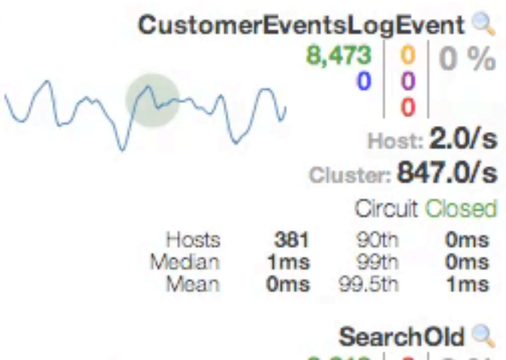


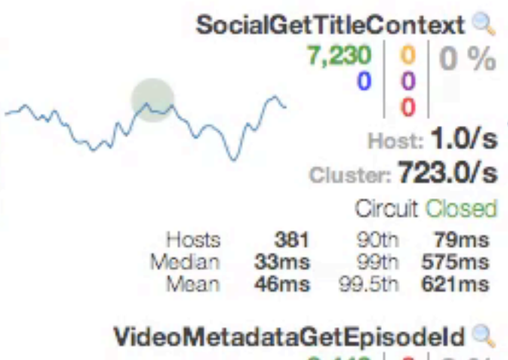


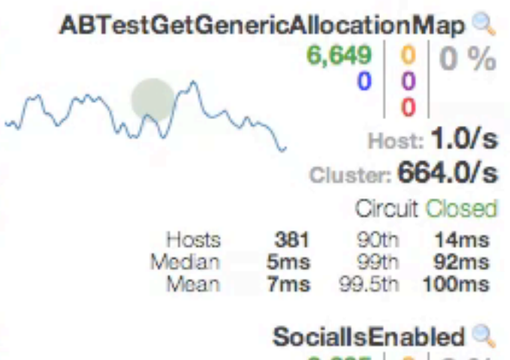


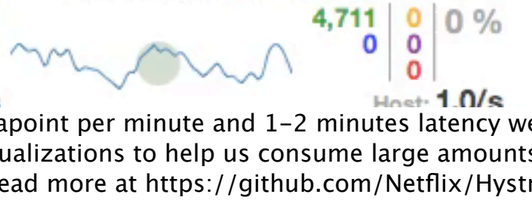


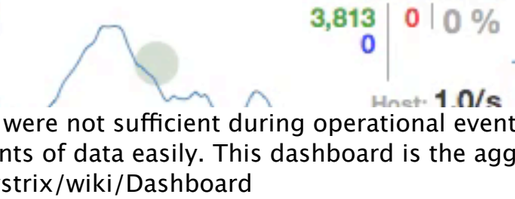


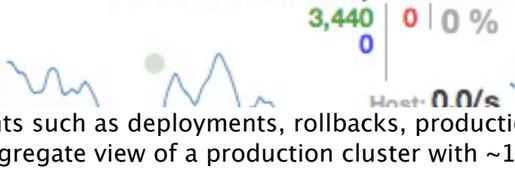


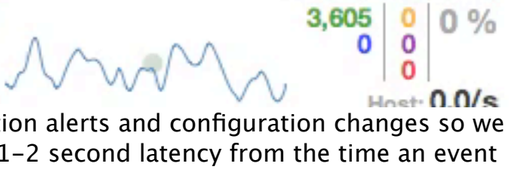




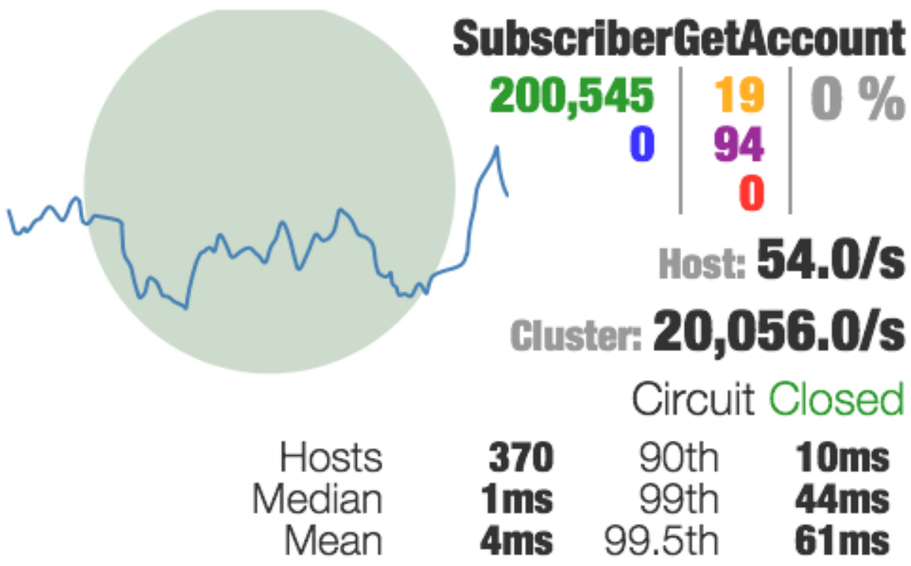






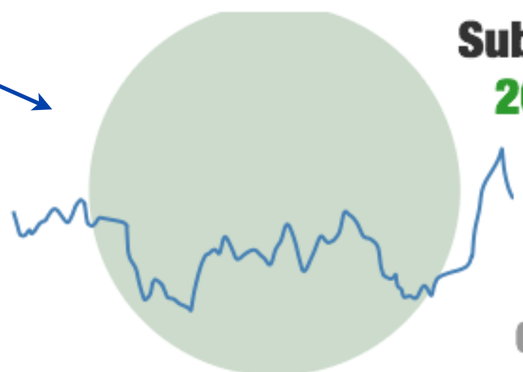


We found that historical metrics with 1 datapoint per minute and 1-2 minutes latency were not sufficient during operational events such as deployments, rollbacks, production alerts and configuration changes so we built near realtime monitoring and data visualizations to help us consume large amounts of data easily. This dashboard is the aggregate view of a production cluster with ~1-2 second latency from the time an event occurs to being rendered in the browser. Read more at https://github.com/Netflix/Hystrix/wiki/Dashboard



Each bulkhead is represented with a visualization like this.

CIRCLE COLOR AND SIZE REPRESENT HEALTH AND TRAFFIC VOLUME



SubscriberGetAccount

200,545	19	0 %
0	94	
	0	

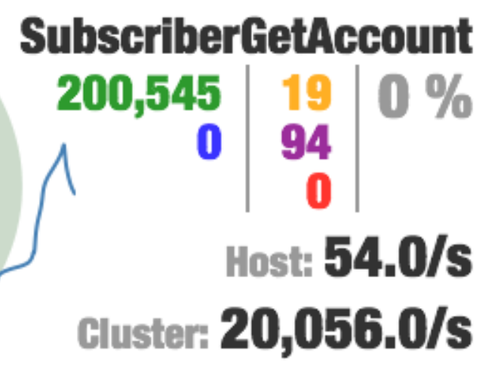
Host: **54.0/s**

Cluster: **20,056.0/s**

Circuit Closed

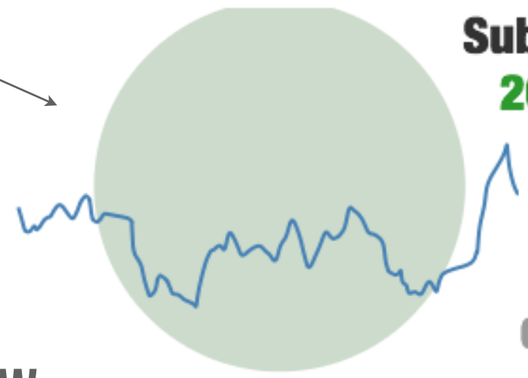
Hosts	**370**	90th	**10ms**
Median	**1ms**	99th	**44ms**
Mean	**4ms**	99.5th	**61ms**

CIRCLE COLOR AND SIZE REPRESENT HEALTH AND TRAFFIC VOLUME



2 MINUTES OF REQUEST RATE TO SHOW RELATIVE CHANGES IN TRAFFIC

CIRCLE COLOR AND SIZE REPRESENT HEALTH AND TRAFFIC VOLUME



2 MINUTES OF REQUEST RATE TO SHOW RELATIVE CHANGES IN TRAFFIC

SubscriberGetAccount

200,545	19	0 %
0	94	
	0	

Host: **54.0/s**

Cluster: **20,056.0/s**

Circuit Closed

Hosts	**370**	90th	**10ms**
Median	**1ms**	99th	**44ms**
Mean	**4ms**	99.5th	**61ms**

HOSTS REPORTING FROM CLUSTER

CIRCLE COLOR AND SIZE REPRESENT HEALTH AND TRAFFIC VOLUME

SubscriberGetAccount

200,545	19	0 %
0	94	
	0	

Host: **54.0/s**

Cluster: **20,056.0/s**

Circuit Closed

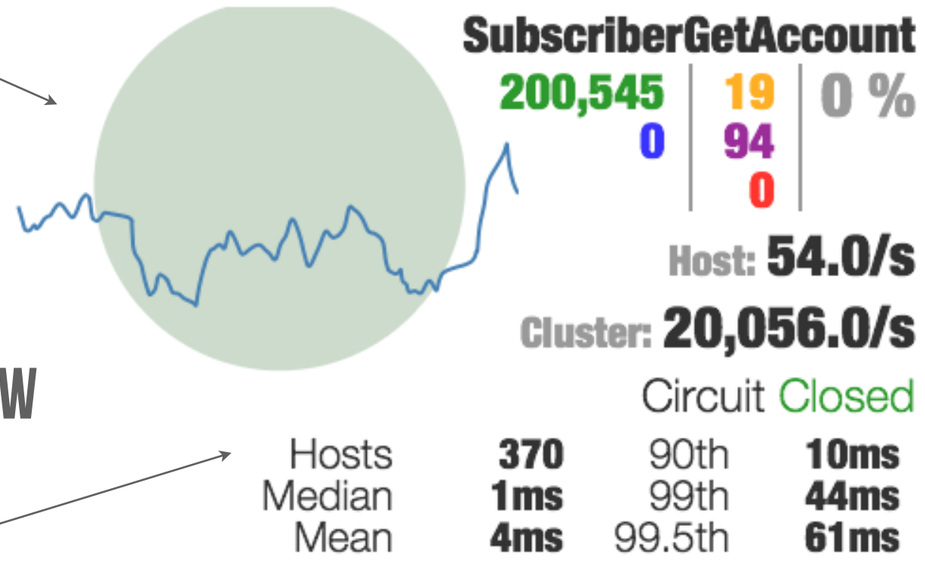
Hosts	**370**	90th	**10ms**
Median	**1ms**	99th	**44ms**
Mean	**4ms**	99.5th	**61ms**

2 MINUTES OF REQUEST RATE TO SHOW RELATIVE CHANGES IN TRAFFIC

HOSTS REPORTING FROM CLUSTER

LAST MINUTE LATENCY PERCENTILES

CIRCLE COLOR AND SIZE REPRESENT HEALTH AND TRAFFIC VOLUME



2 MINUTES OF REQUEST RATE TO SHOW RELATIVE CHANGES IN TRAFFIC

CIRCUIT-BREAKER STATUS

HOSTS REPORTING FROM CLUSTER

LAST MINUTE LATENCY PERCENTILES

CIRCLE COLOR AND SIZE REPRESENT HEALTH AND TRAFFIC VOLUME

SubscriberGetAccount

200,545	19	0 %
0	94	
	0	

Host: 54.0/s

Cluster: 20,056.0/s

REQUEST RATE

2 MINUTES OF REQUEST RATE TO SHOW RELATIVE CHANGES IN TRAFFIC

Circuit Closed

CIRCUIT-BREAKER STATUS

Hosts	**370**	90th	**10ms**
Median	**1ms**	99th	**44ms**
Mean	**4ms**	99.5th	**61ms**

HOSTS REPORTING FROM CLUSTER

LAST MINUTE LATENCY PERCENTILES

CIRCLE COLOR AND SIZE REPRESENT HEALTH AND TRAFFIC VOLUME

2 MINUTES OF REQUEST RATE TO SHOW RELATIVE CHANGES IN TRAFFIC

HOSTS REPORTING FROM CLUSTER

ERROR PERCENTAGE OF LAST 10 SECONDS

REQUEST RATE

CIRCUIT-BREAKER STATUS

LAST MINUTE LATENCY PERCENTILES

SubscriberGetAccount

200,545	19	0 %
0	94	
	0	

Host: 54.0/s

Cluster: 20,056.0/s

Circuit Closed

Hosts	370	90th	10ms
Median	1ms	99th	44ms
Mean	4ms	99.5th	61ms

CIRCLE COLOR AND SIZE REPRESENT HEALTH AND TRAFFIC VOLUME

SubscriberGetAccount

200,545	19	0 %
0	94	
	0	

Host: 54.0/s

Cluster: 20,056.0/s

Circuit Closed

Hosts	370	90th	10ms
Median	1ms	99th	44ms
Mean	4ms	99.5th	61ms

ERROR PERCENTAGE OF LAST 10 SECONDS

REQUEST RATE

2 MINUTES OF REQUEST RATE TO SHOW RELATIVE CHANGES IN TRAFFIC

CIRCUIT-BREAKER STATUS

HOSTS REPORTING FROM CLUSTER

LAST MINUTE LATENCY PERCENTILES

ROLLING 10 SECOND COUNTERS
WITH 1 SECOND GRANULARITY

SUCCESSES	200,545	19	THREAD TIMEOUTS
SHORT-CIRCUITED (REJECTED)	0	94	THREAD-POOL REJECTIONS
		0	FAILURES/EXCEPTIONS

LOW LATENCY GRANULAR METRICS

Success	23	47	26	48	38	42	59	46	39	12
Timeout	5	8	4	9	4	6	11	5	3	1
Failure	2	1	0	4	2	7	5	2	5	0
Rejection	0	0	0	0	0	0	1	0	0	0

23	47	26	48	38	42	59	46	39	45	1
5	8	4	9	4	6	11	5	3	6	0
2	1	0	4	2	7	5	2	5	2	0
0	0	0	0	0	0	1	0	0	0	0

ROLLING 10 SECOND WINDOW

1 SECOND RESOLUTION

All metrics are captured in both absolute cumulative counters and rolling windows with 1 second granularity. Read more at https://github.com/Netflix/Hystrix/wiki/Metrics-and-Monitoring

LOW LATENCY GRANULAR METRICS

Success	23	47	26	48	38	42	59	46	39	12
Timeout	5	8	4	9	4	6	11	5	3	1
Failure	2	1	0	4	2	7	5	2	5	0
Rejection	0	0	0	0	0	0	1	0	0	0

23	47	26	48	38	42	59	46	39	45	1
5	8	4	9	4	6	11	5	3	6	0
2	1	0	4	2	7	5	2	5	2	0
0	0	0	0	0	0	1	0	0	0	0

ROLLING 10 SECOND WINDOW

1 SECOND RESOLUTION

The rolling counters default to 10 second windows with 1 second buckets.

LOW LATENCY GRANULAR METRICS

Success	23	47	26	48	38	42	59	46	39	12
Timeout	5	8	4	9	4	6	11	5	3	1
Failure	2	1	0	4	2	7	5	2	5	0
Rejection	0	0	0	0	0	0	1	0	0	0

23	47	26	48	38	42	59	46	39	45	1
5	8	4	9	4	6	11	5	3	6	0
2	1	0	4	2	7	5	2	5	2	0
0	0	0	0	0	0	1	0	0	0	0

ROLLING 10 SECOND WINDOW

1 SECOND RESOLUTION

As each second passes the oldest bucket is dropped (to soon be overwritten since it is a ring buffer)...

LOW LATENCY GRANULAR METRICS

Success	23	47	26	48	38	42	59	46	39	12
Timeout	5	8	4	9	4	6	11	5	3	1
Failure	2	1	0	4	2	7	5	2	5	0
Rejection	0	0	0	0	0	0	1	0	0	0

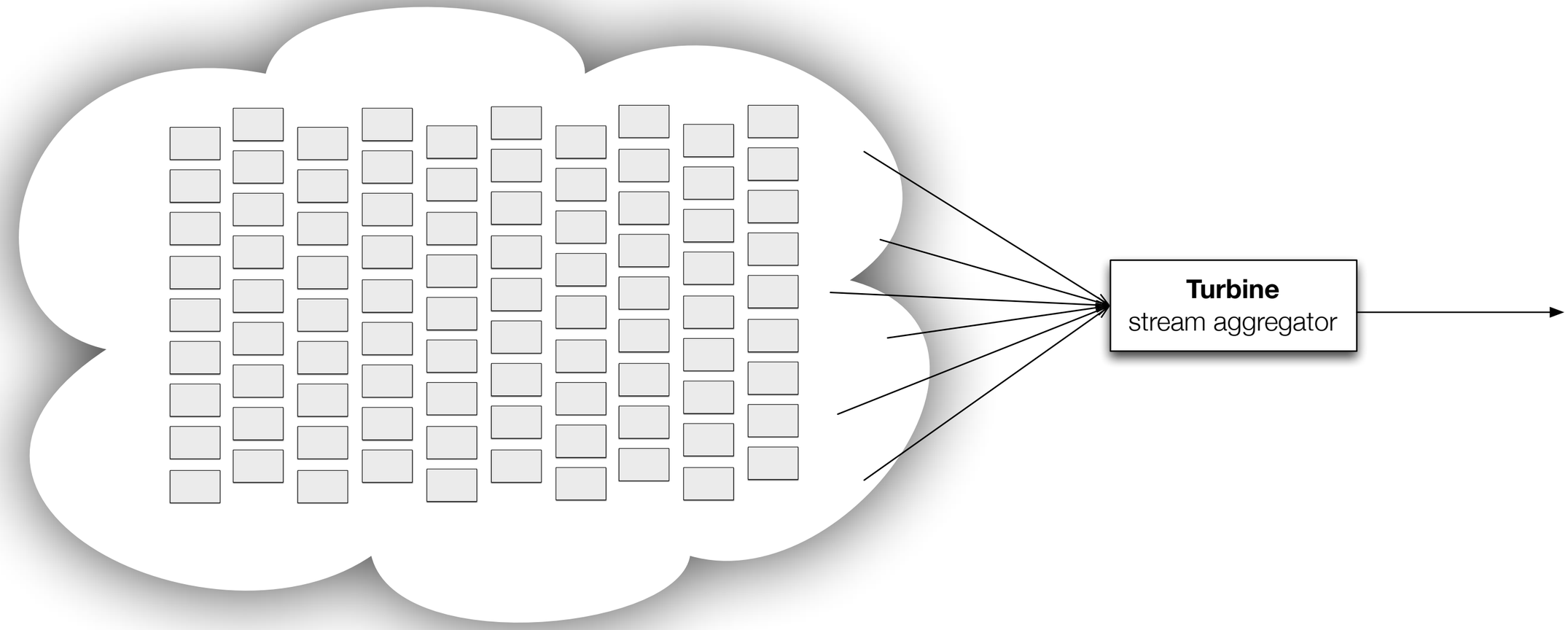
23	47	26	48	38	42	59	46	39	45	1
5	8	4	9	4	6	11	5	3	6	0
2	1	0	4	2	7	5	2	5	2	0
0	0	0	0	0	0	1	0	0	0	0

ROLLING 10 SECOND WINDOW

1 SECOND RESOLUTION

... and a new bucket is created.

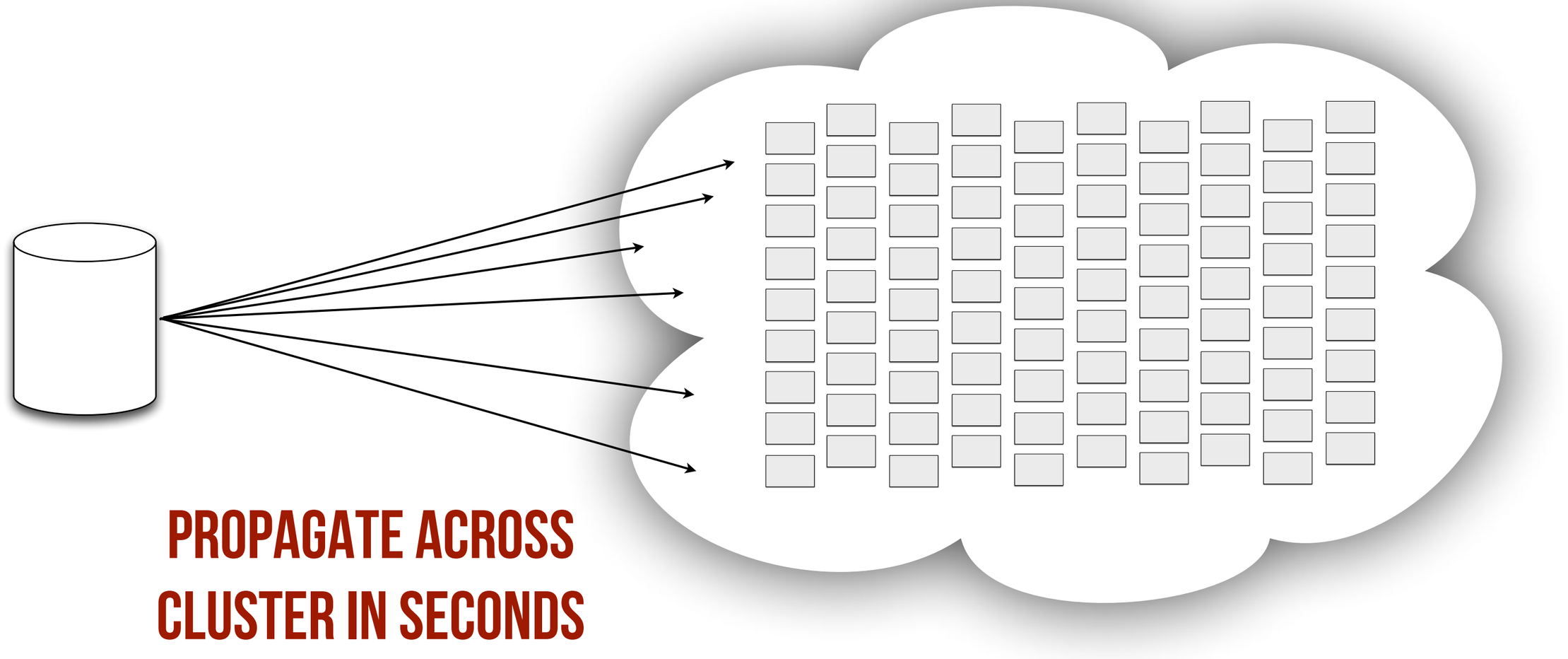
LOW LATENCY GRANULAR METRICS



~1 SECOND LATENCY AGGREGATED STREAM

Metrics are subscribed to from all servers in a cluster and aggregated with ~1 second latency from event to aggregation. This stream can then be consumed by the dashboard, an alerting system or anything else wanting low latency metrics.

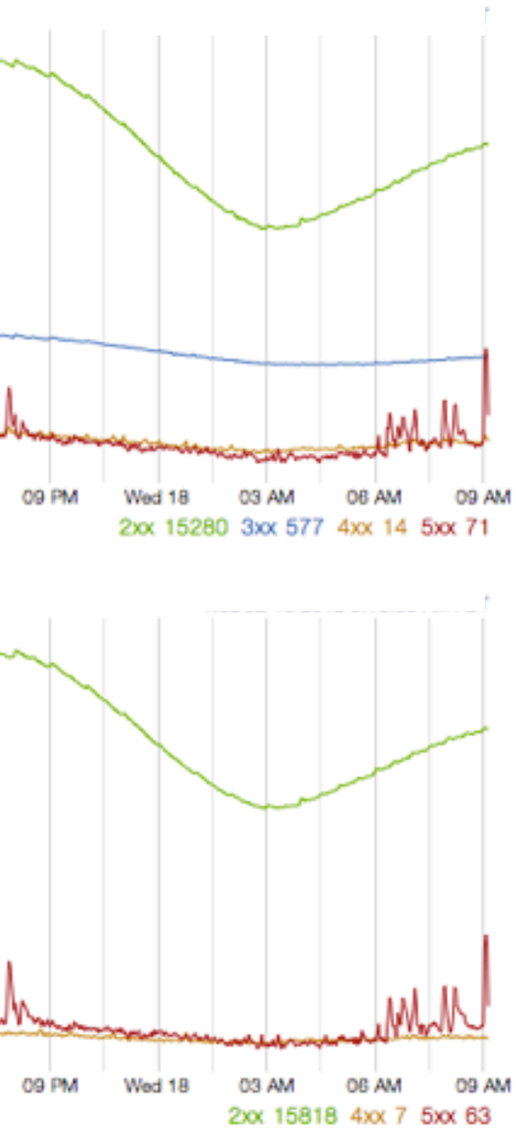
LOW LATENCY CONFIGURATION CHANGES

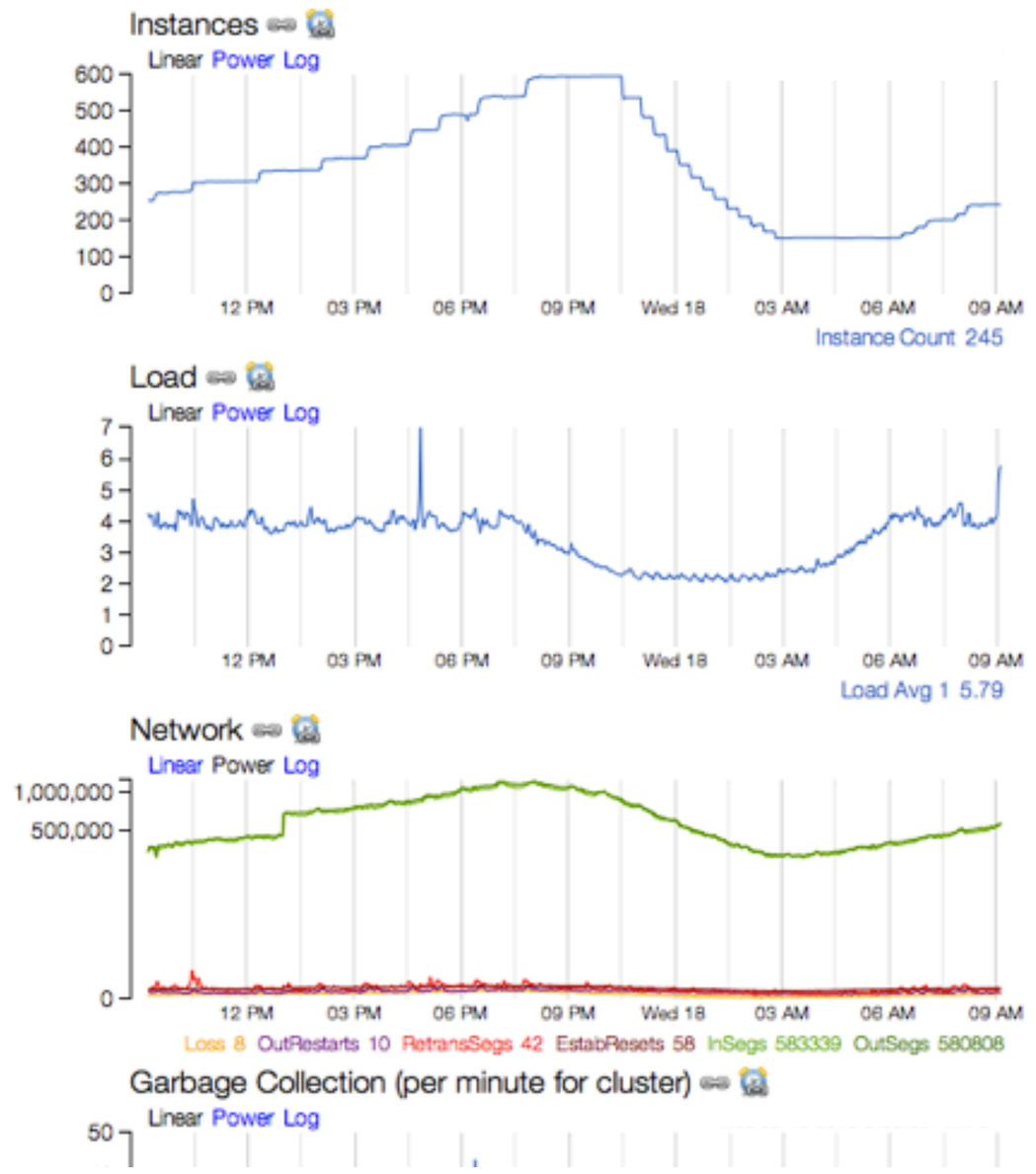


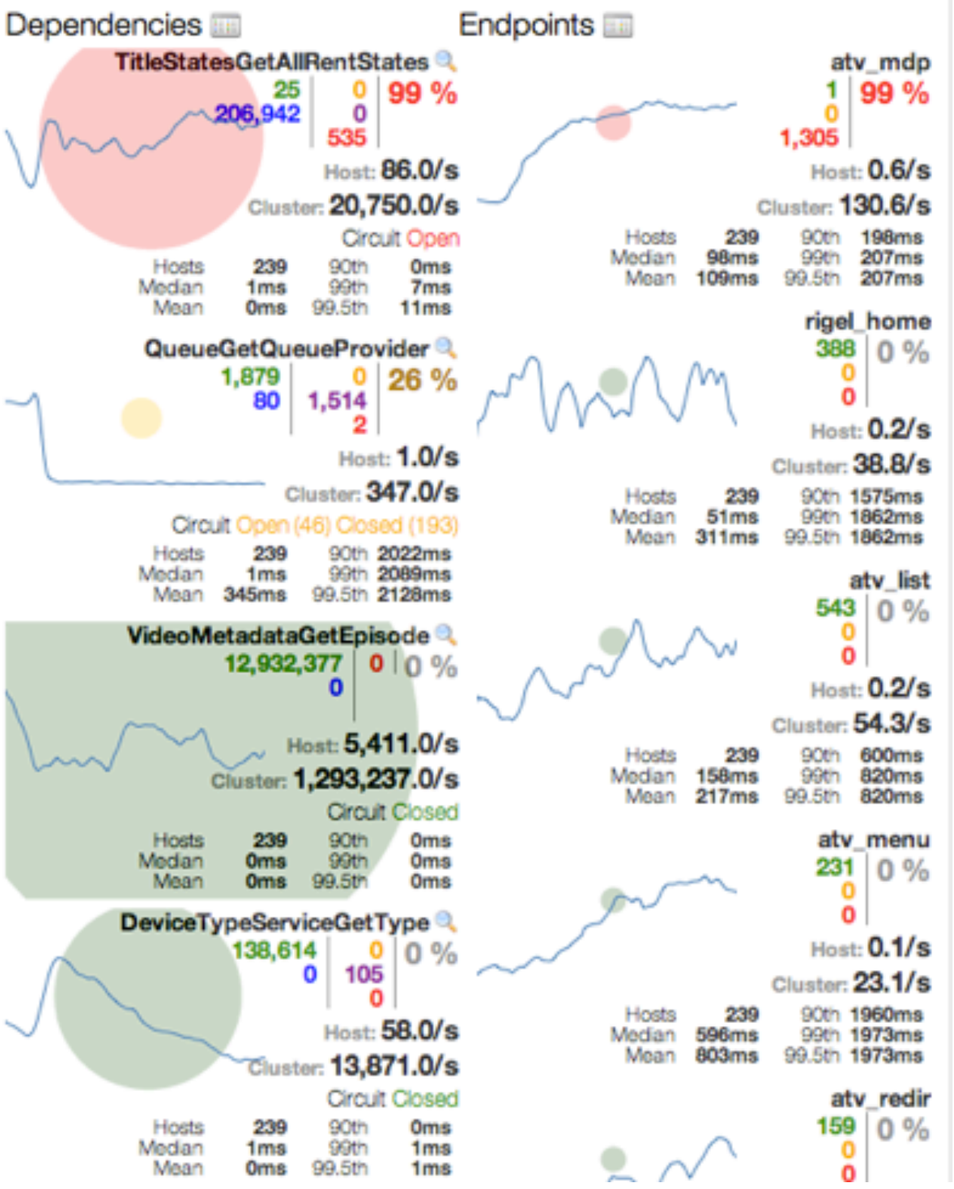
PROPAGATE ACROSS CLUSTER IN SECONDS

The low latency loop is completed with the ability to propagate configuration changes across a cluster in seconds. This enables rapid iterations of seeing behavior in production, pushing config changes and then watching them take effect immediately as the changes roll across a cluster of servers. Low latency operations requires both the visibility into metrics and ability to affect change operating with similar latency windows.

AUDITING VIA SIMULATION

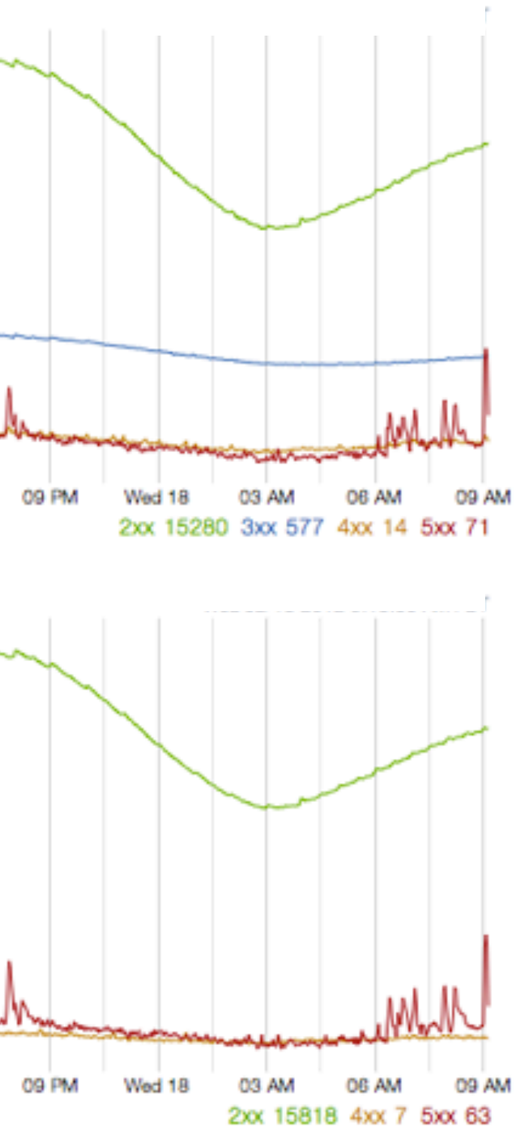


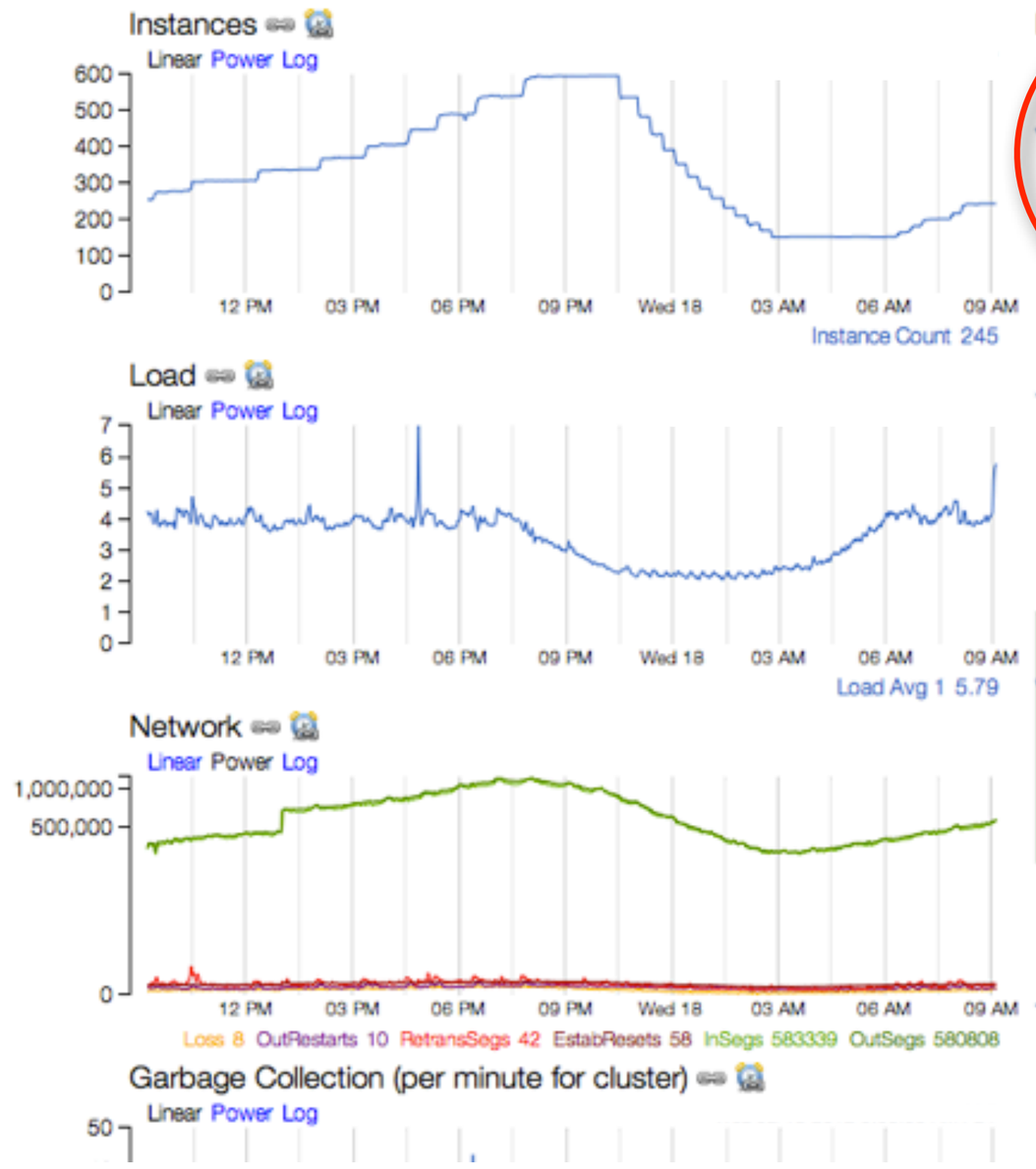


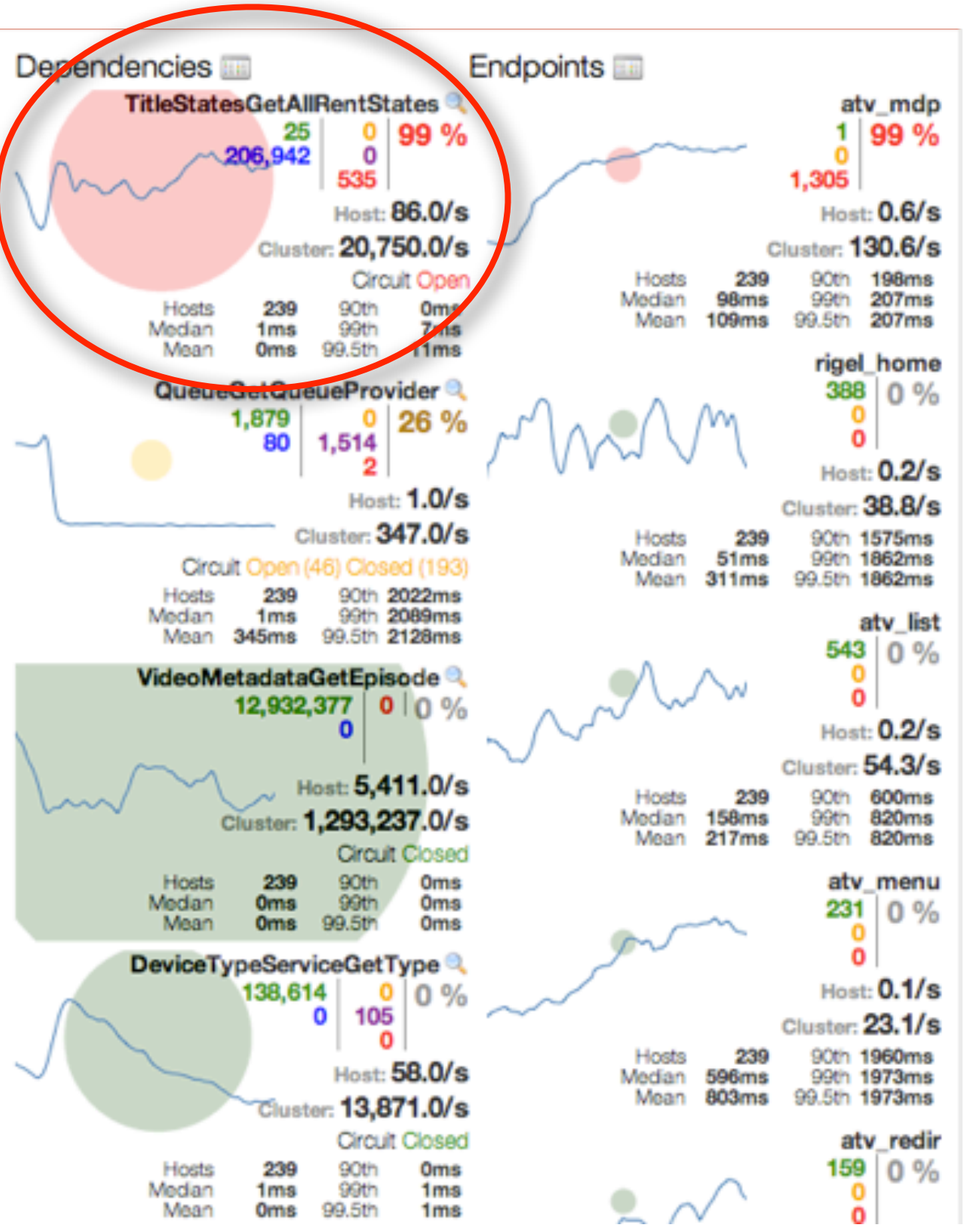


Simulating failure states in production has proven an effective approach for auditing our applications to either prove resilience or find weakness.

AUDITING VIA SIMULATION

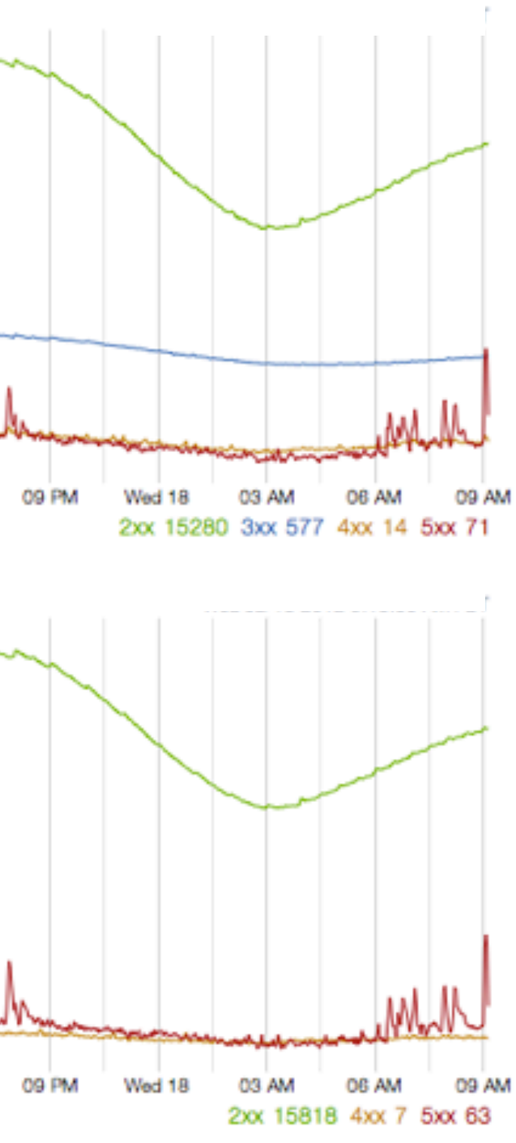


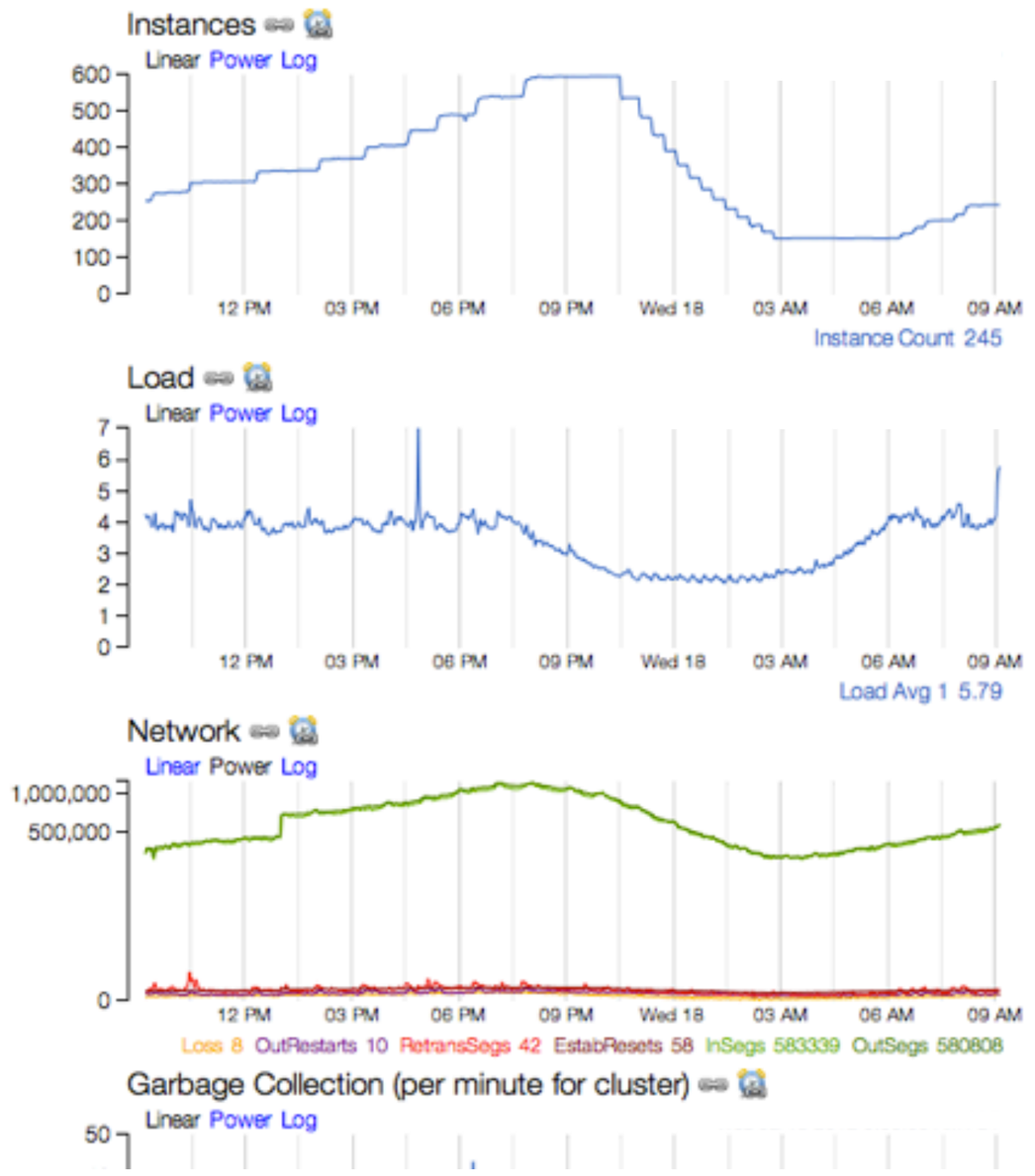


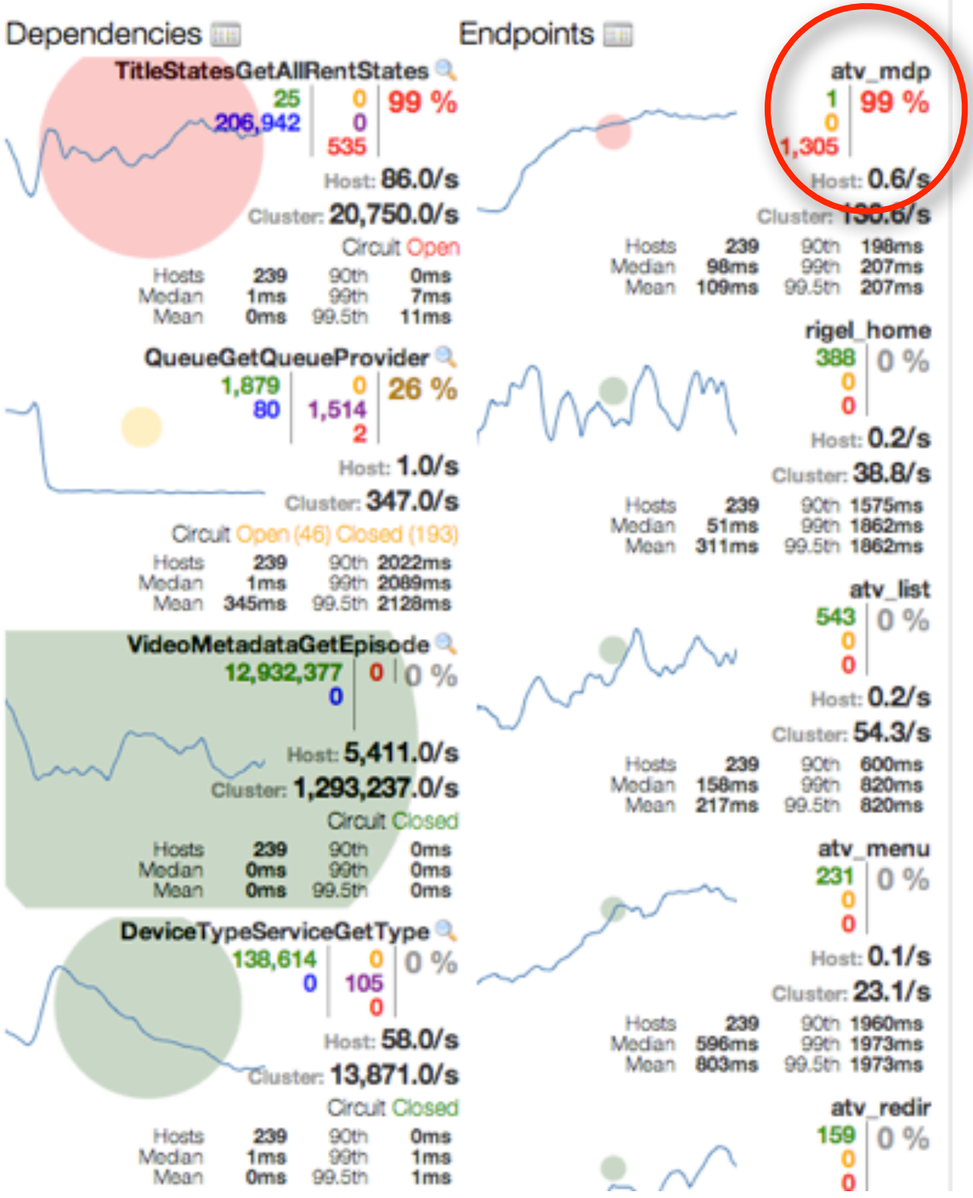


In this example failure was injected into a single dependency which caused the bulkhead to return fallbacks and trip all circuits since the failure rate was almost 100%, well above the threshold for circuits to trip.

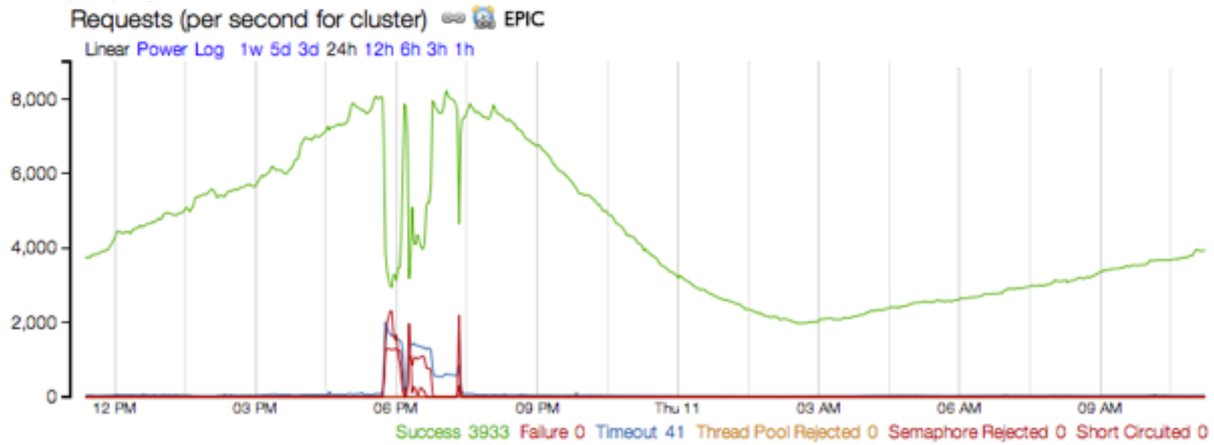
AUDITING VIA SIMULATION

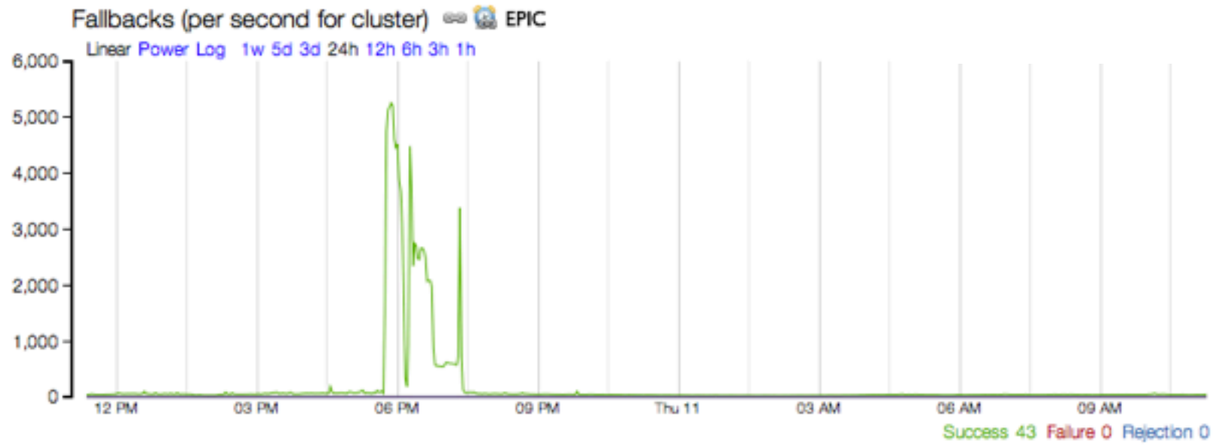


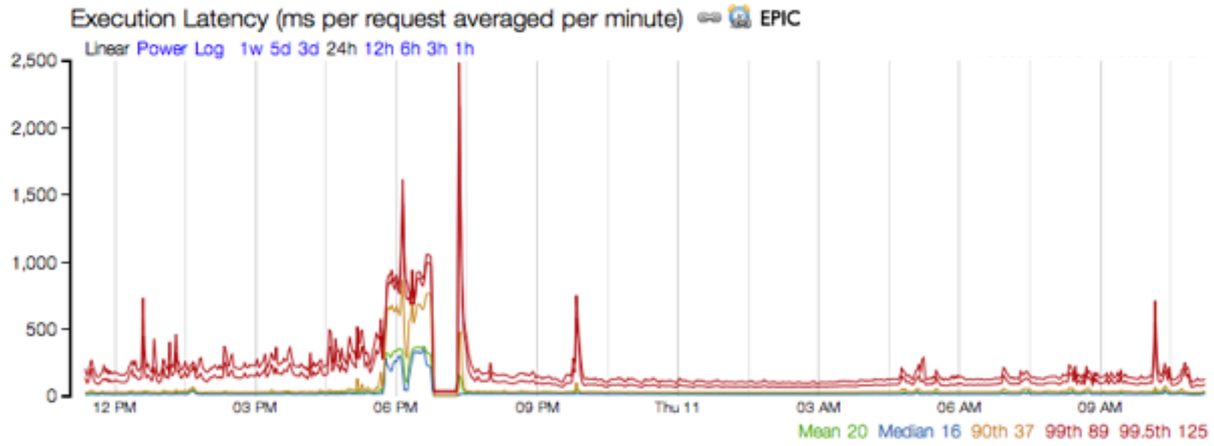


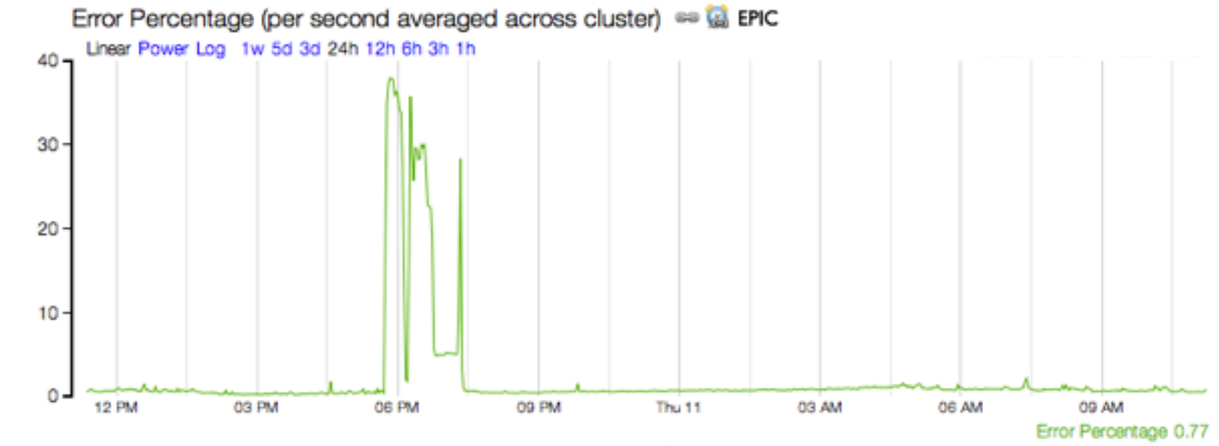


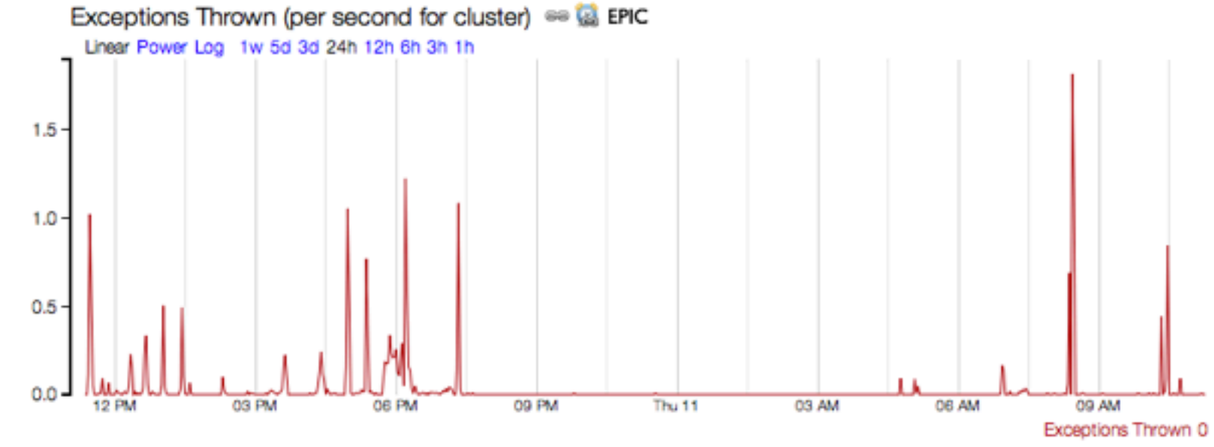
When the 'TitleStatesGetAllRentStates` bulkhead began returning fallbacks the 'atv_mdp' endpoint shot to the top of the dashboard with 99% error rate. There was a bug in how the fallback was handled so we immediately stopped the simulation, fixed the bug over the coming days and repeated the simulation to prove it was fixed and the rest of the system remained resilient. This was caught in a controlled simulation where we could catch and act in less than a minute rather than a true production incident where we likely wouldn't have been able to do anything.

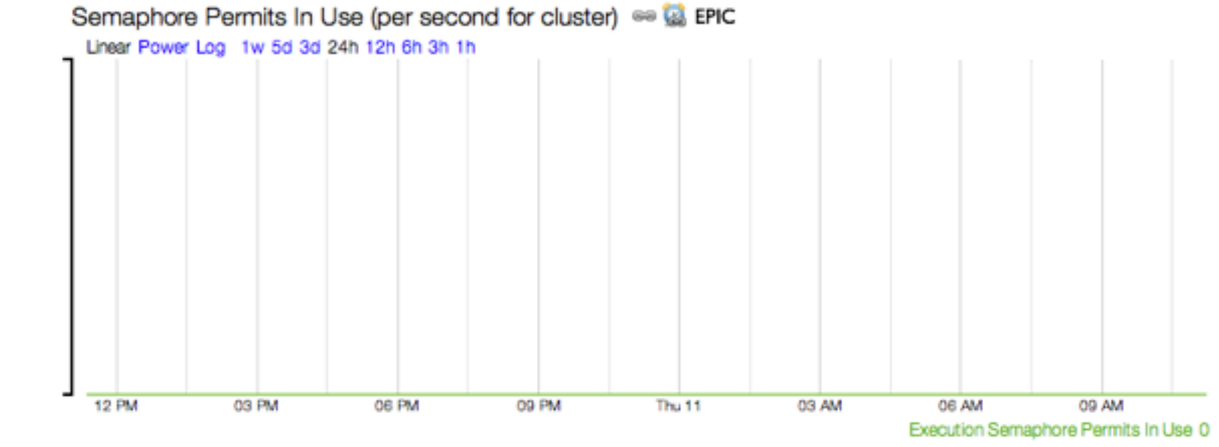




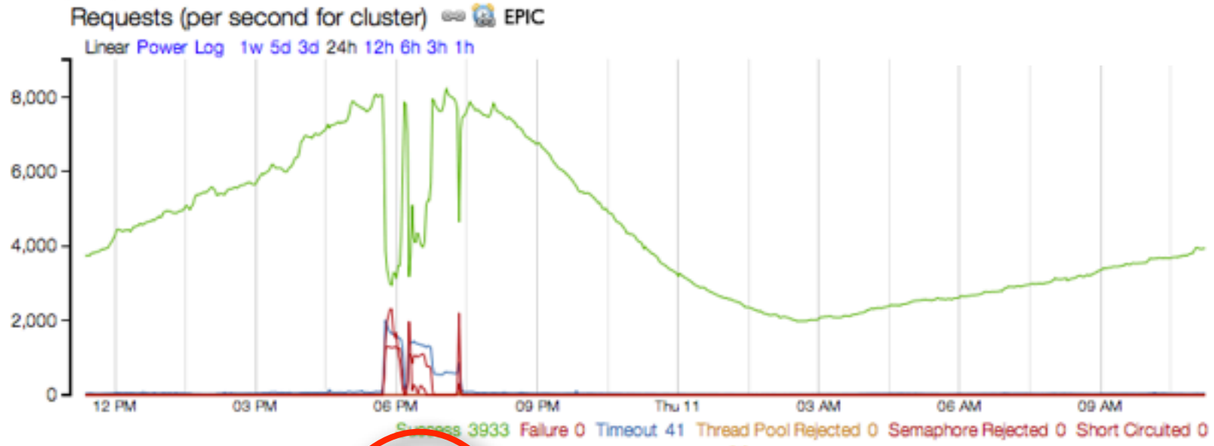


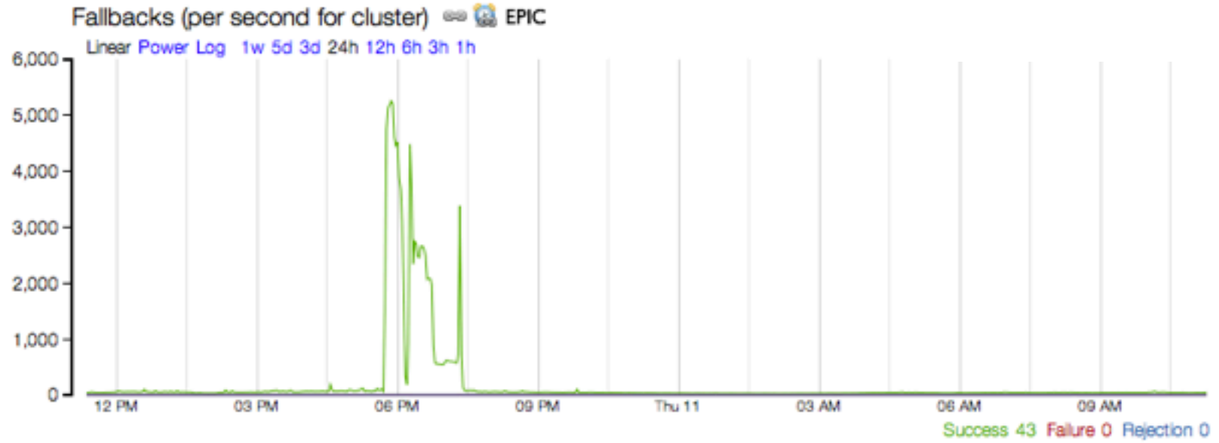


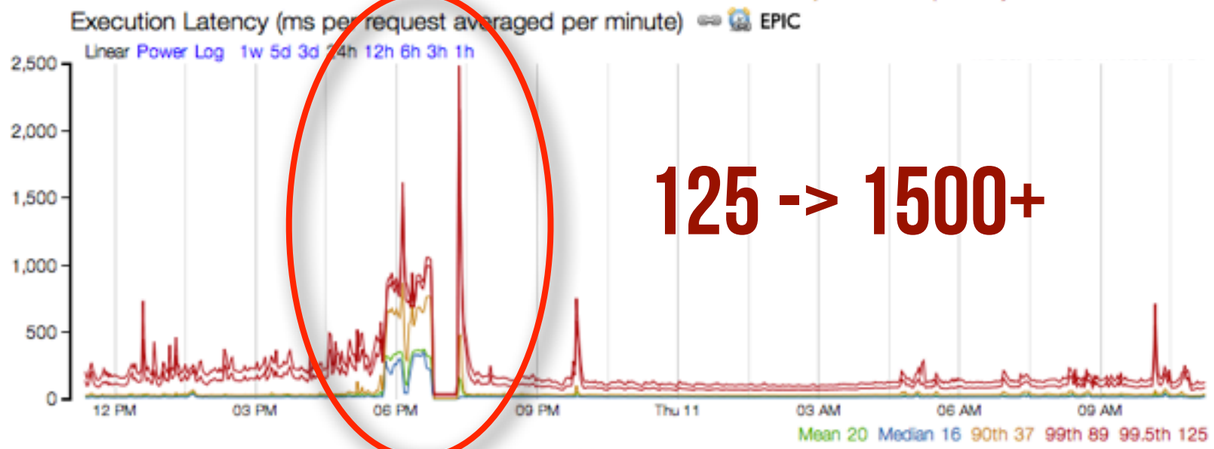




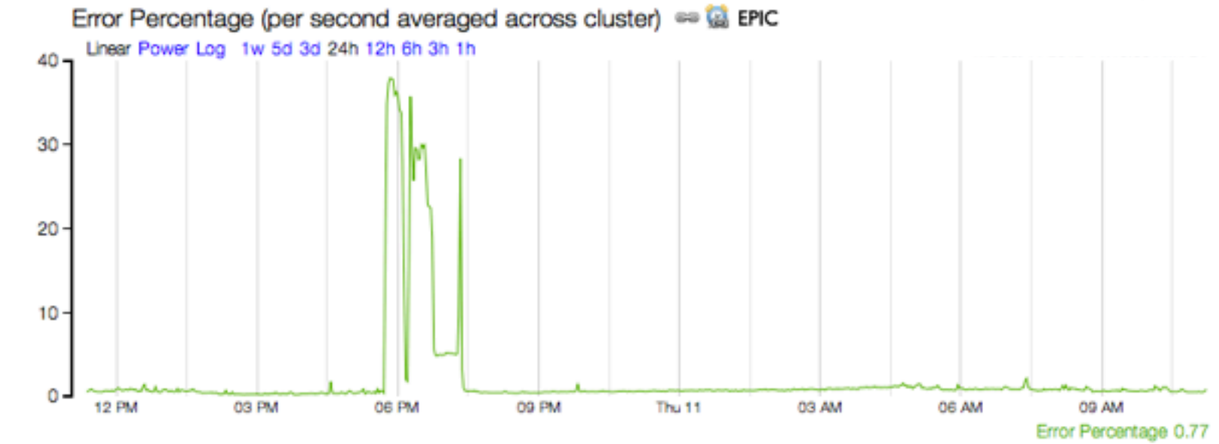
This shows another simulation where latency was injected. Read more at http://techblog.netflix.com/2011/07/netflix-simian-army.html

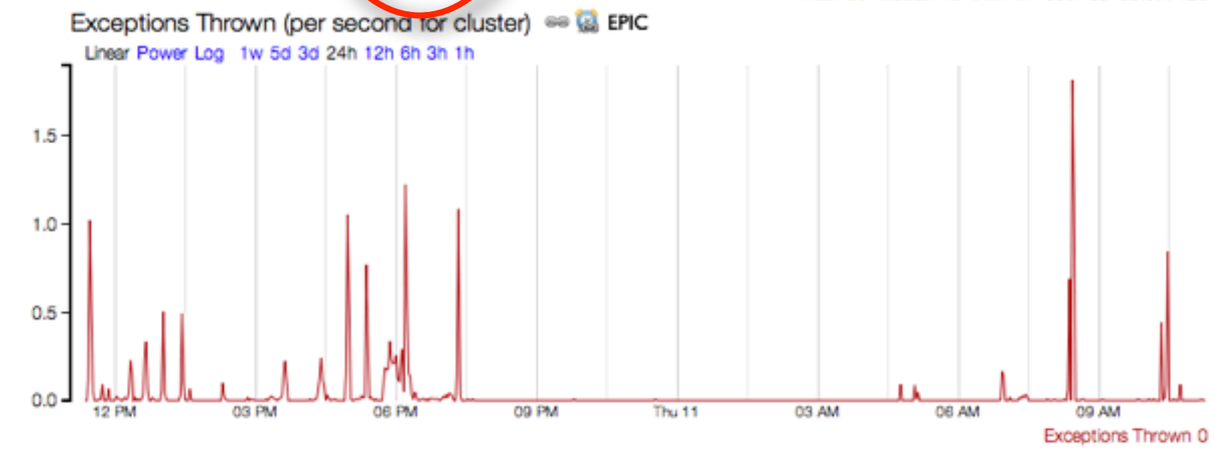


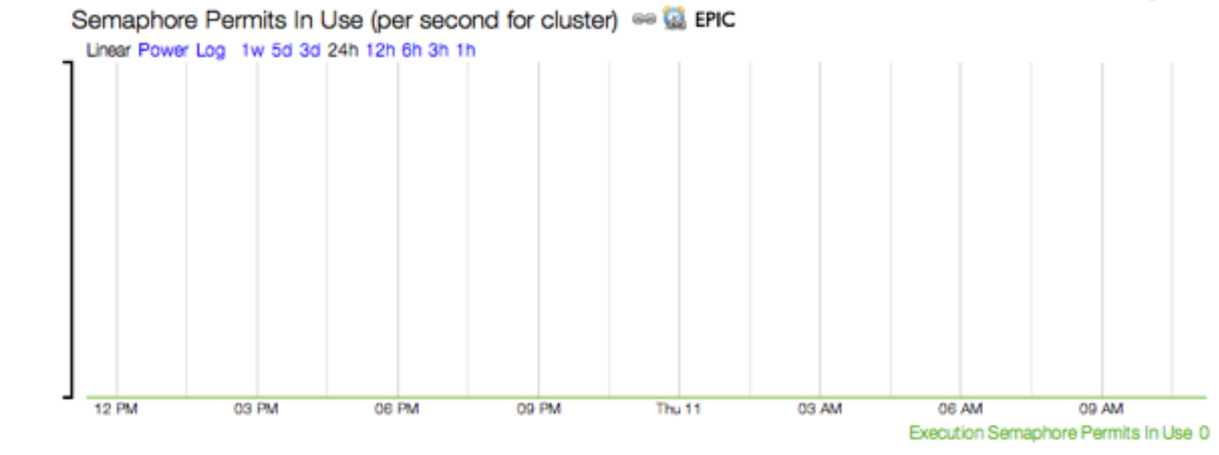




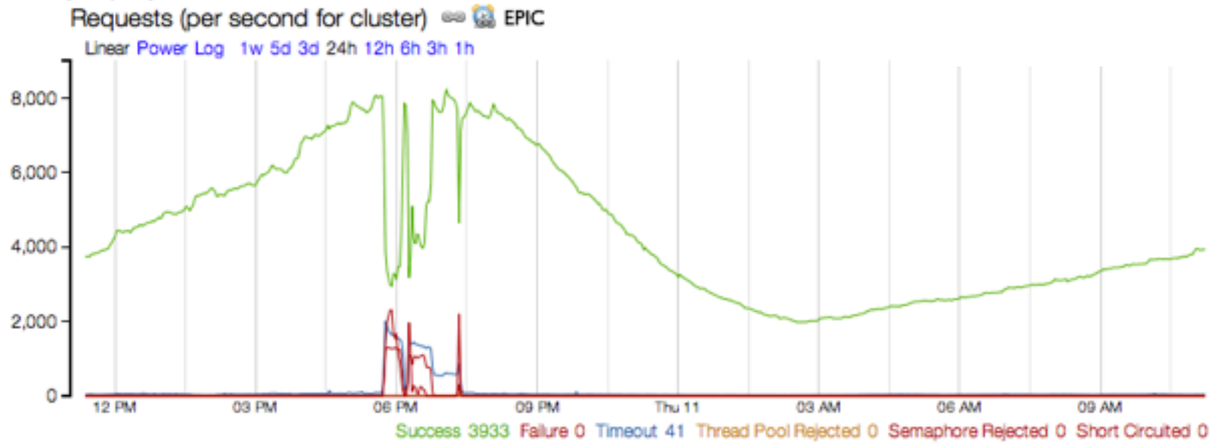
125 -> 1500+

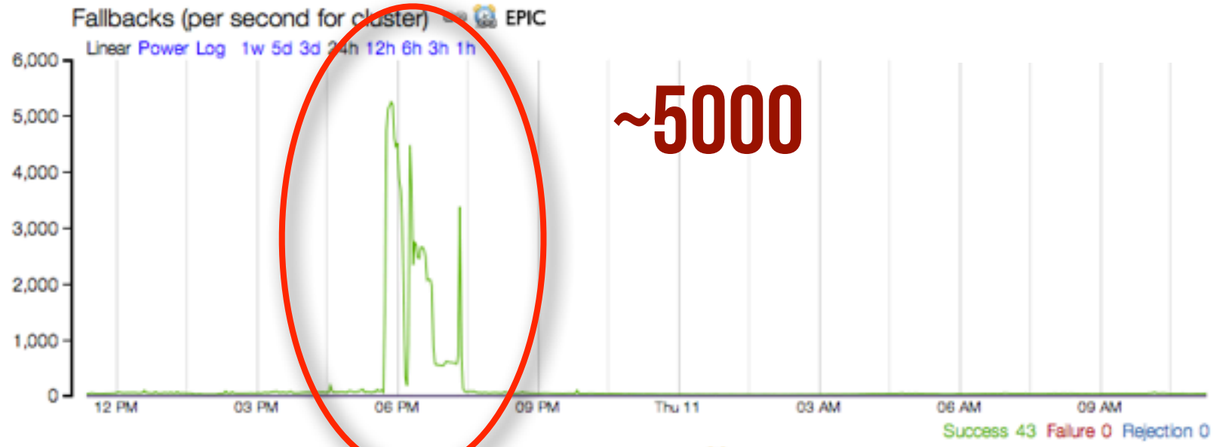




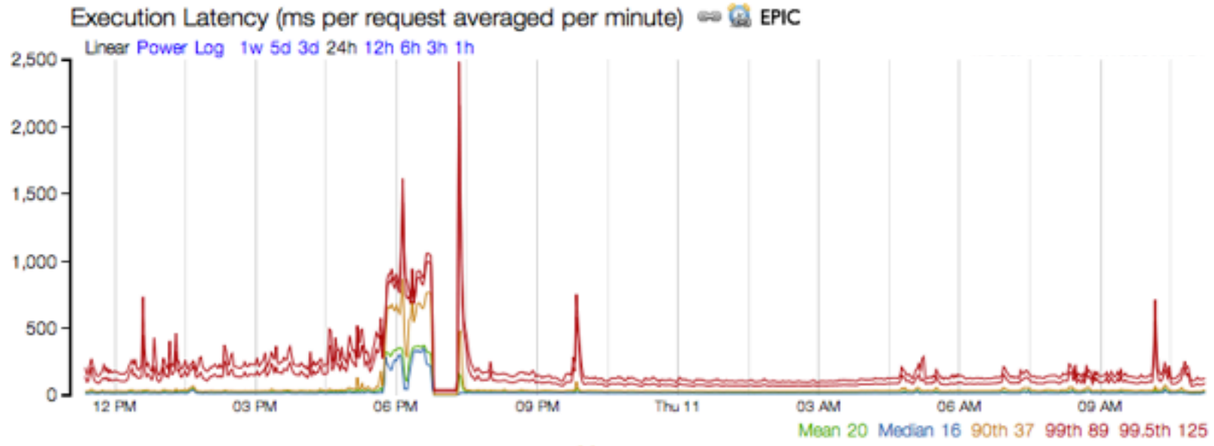


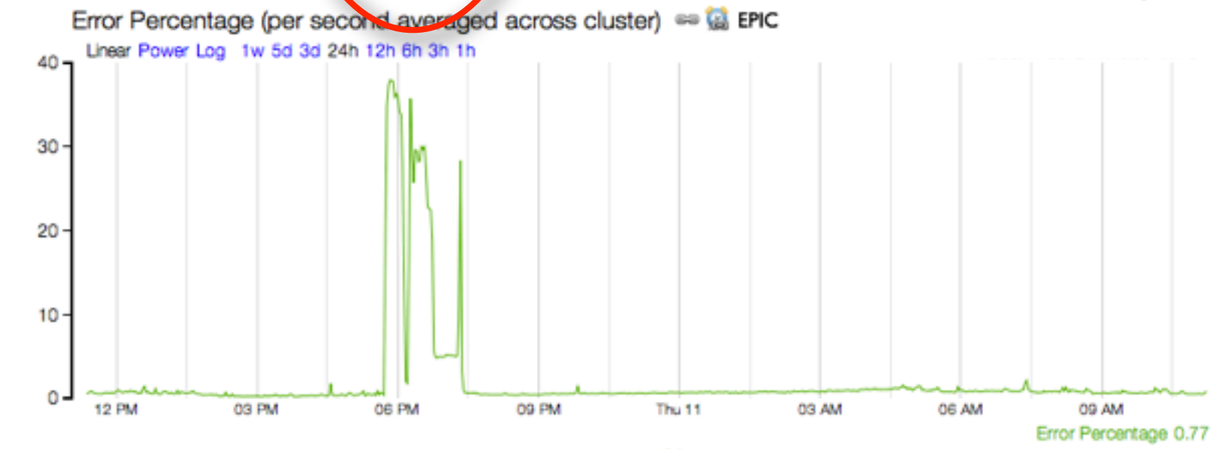
1000+ ms of latency was injected into a dependency that normally completes with a median latency of ~15-20ms and 99.5th of 120-130ms.

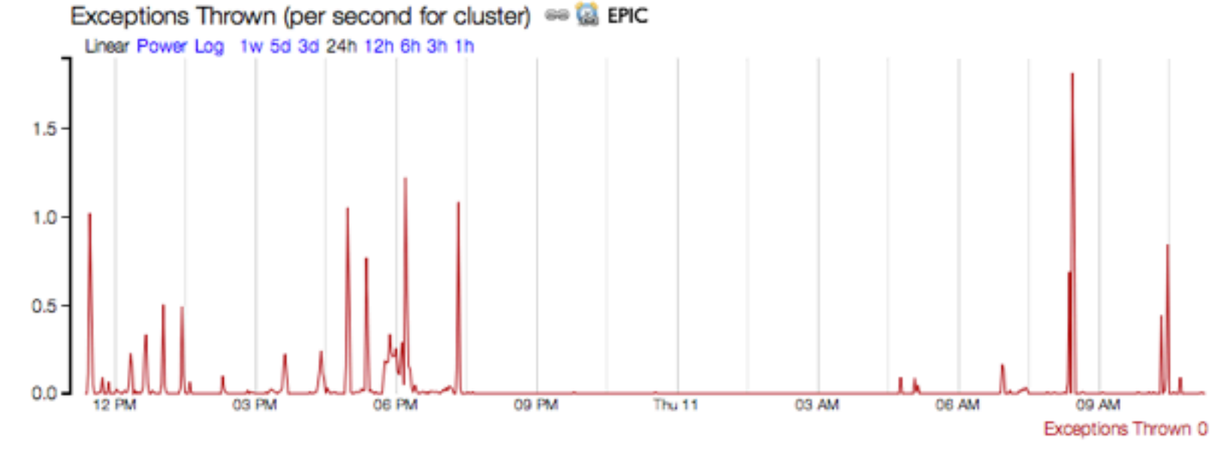


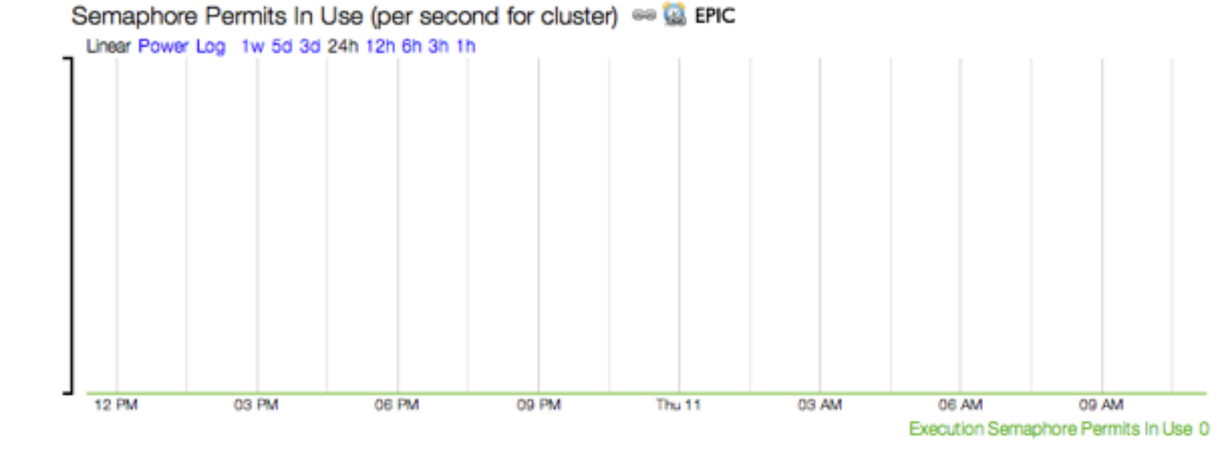


~5000

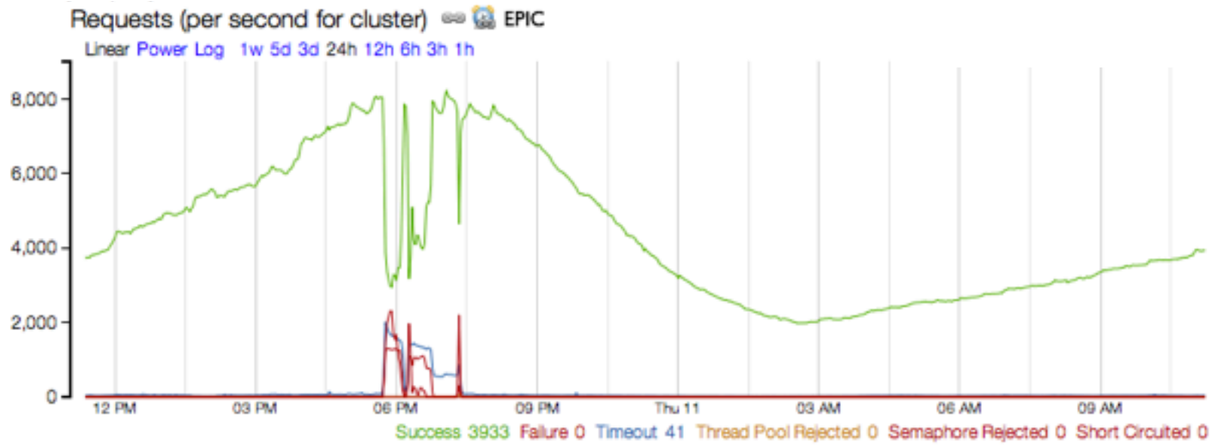


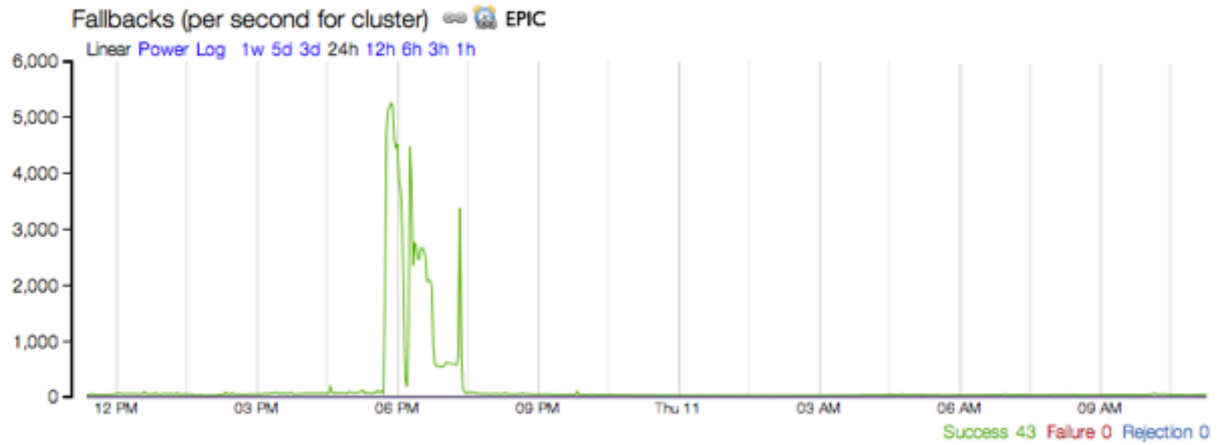


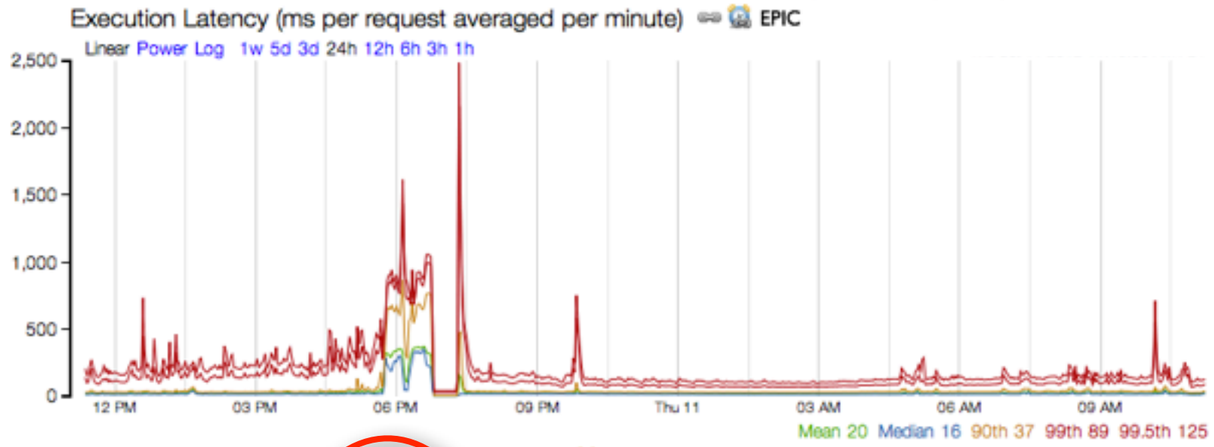


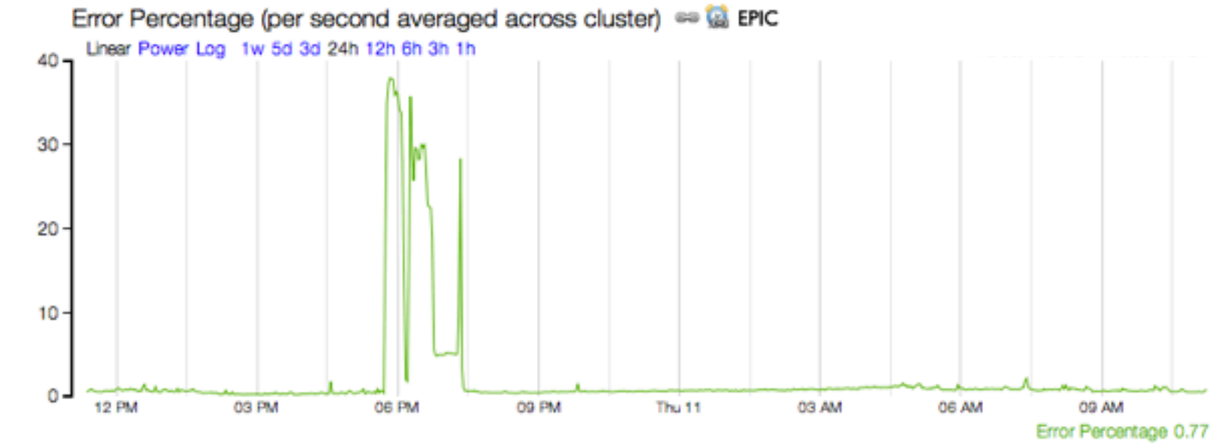


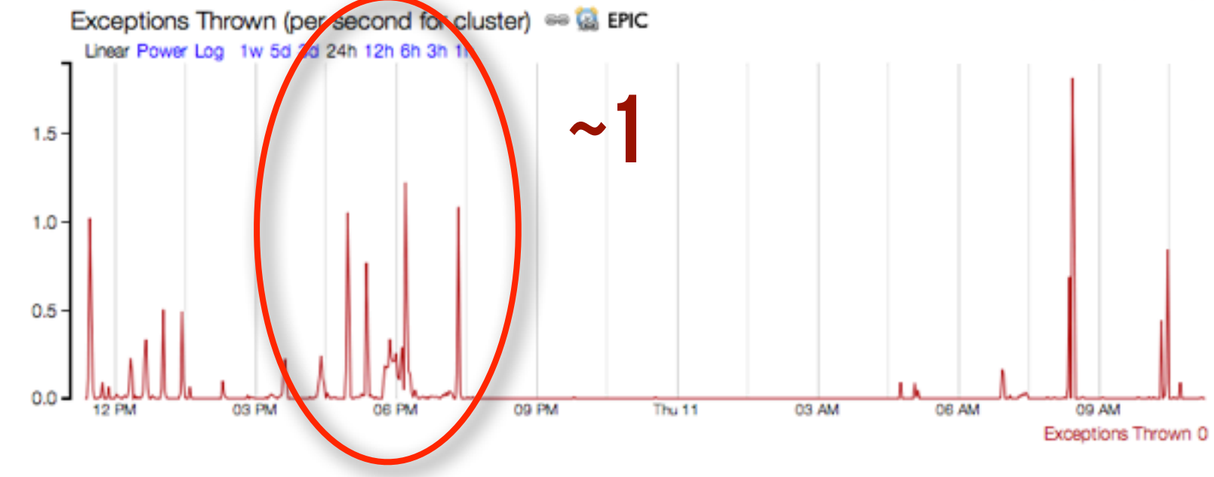
The latency spike caused timeouts, short-circuiting and rejecting and up to ~5000 fallbacks per second as a result of these various failure states.

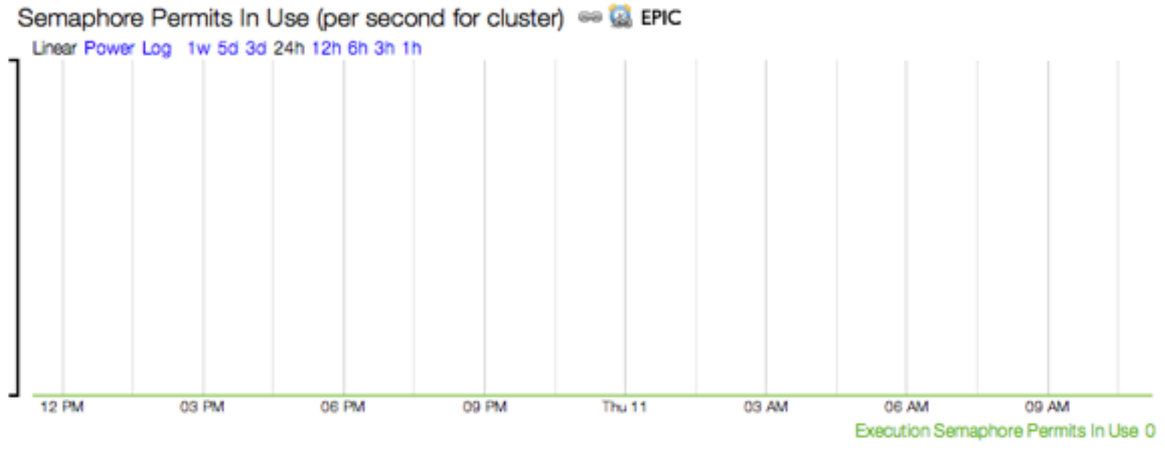




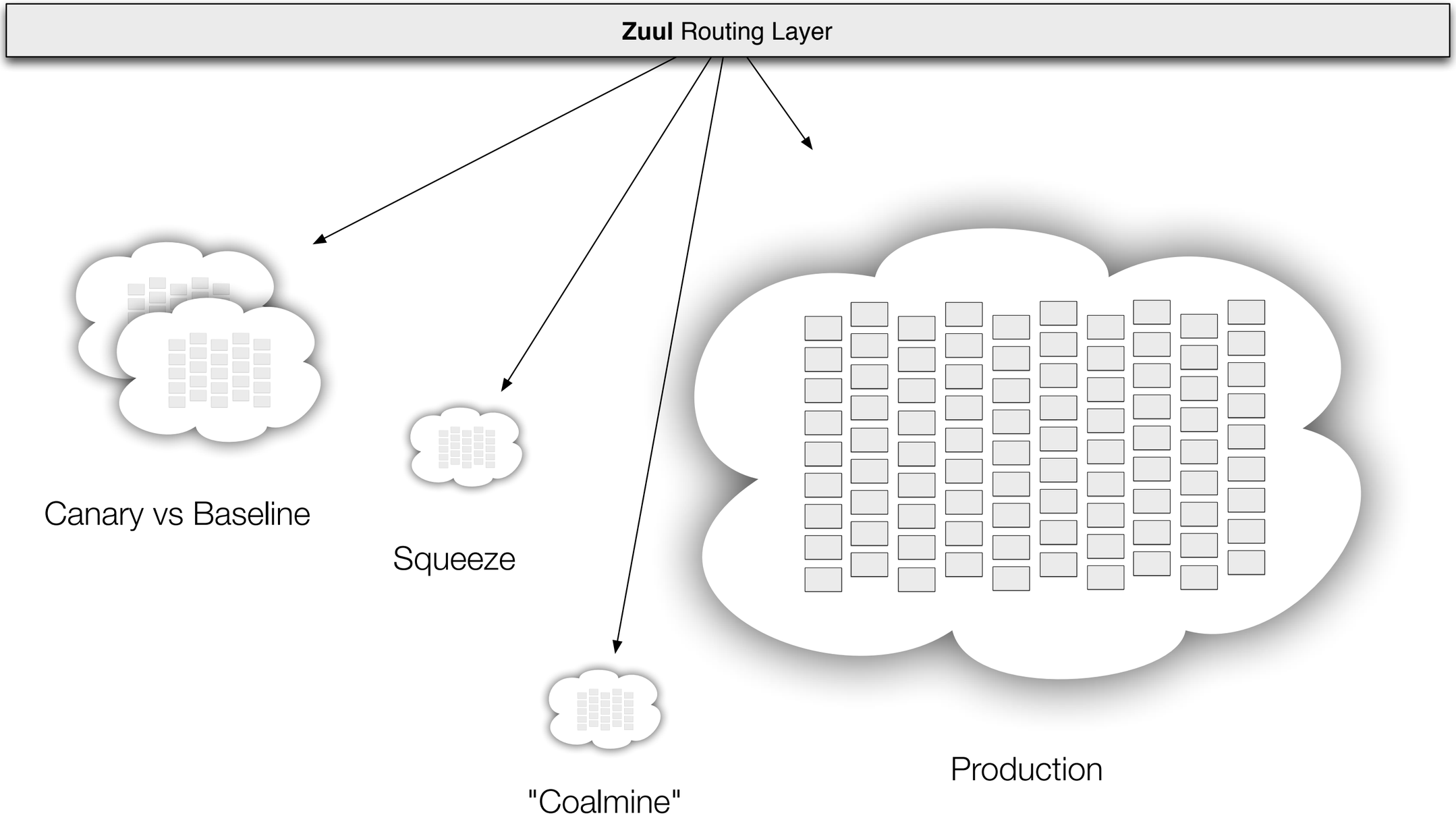




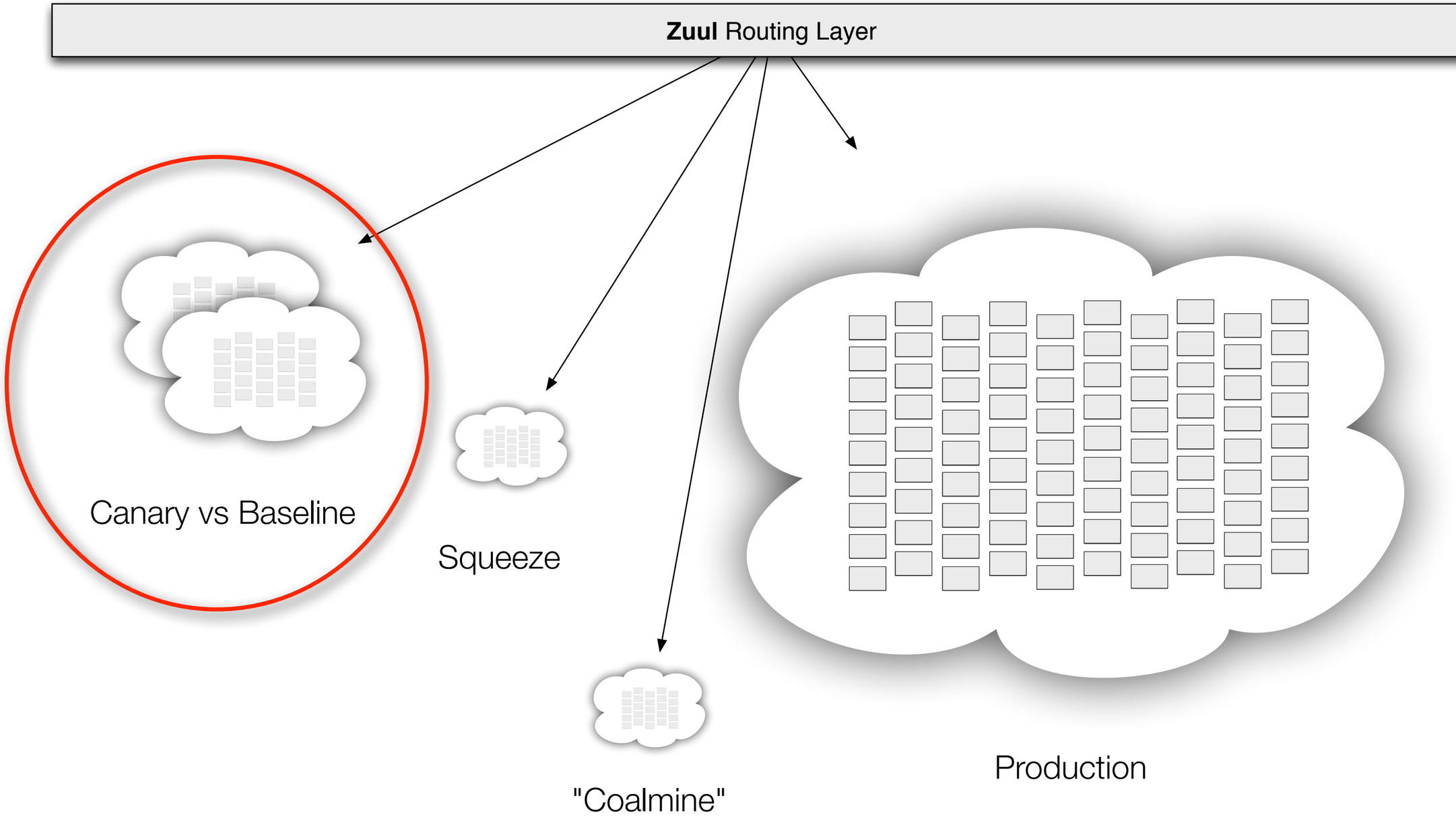




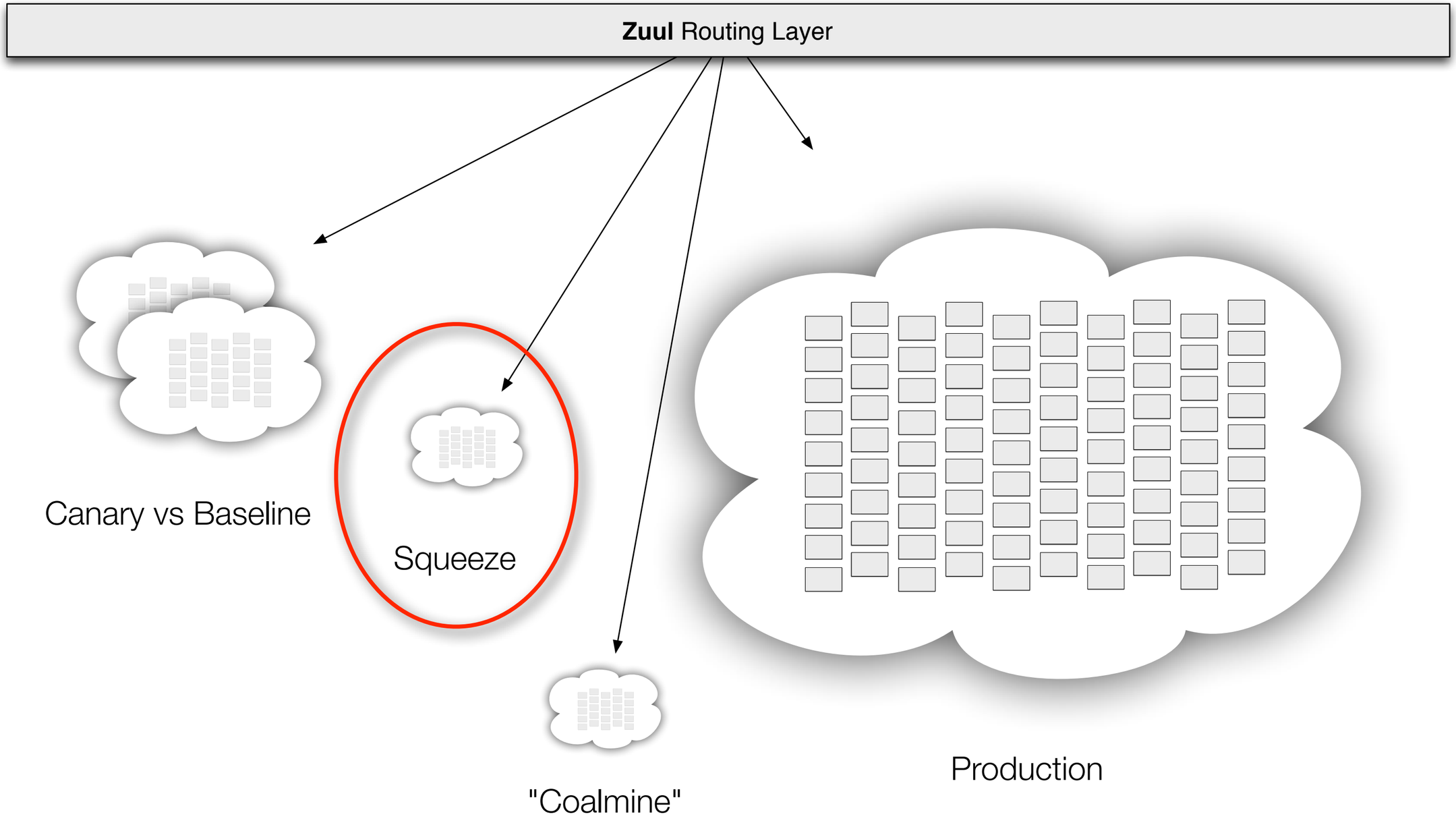
While delivering the ~5000 fallbacks per second the exceptions thrown didn't go beyond ~1 per second demonstrating that user impact was negligible (as perceived from the server, the client behavior must also be validated during a simulation but is not part of this dataset).



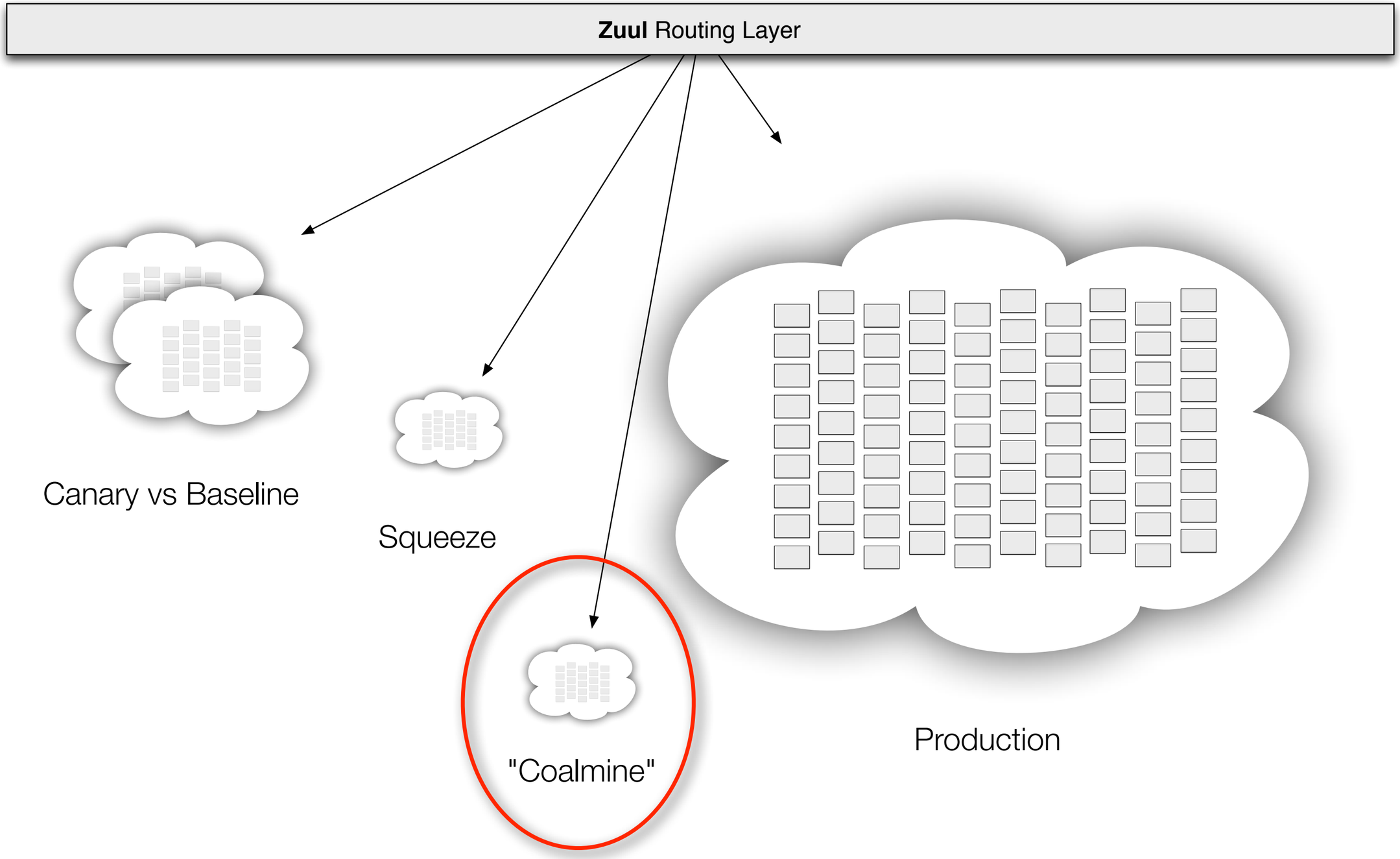
Other approaches to auditing take advantage of our routing layer to route traffic to different clusters. Read more at http://techblog.netflix.com/2013/06/announcing-zuul-edge-service-in-cloud.html



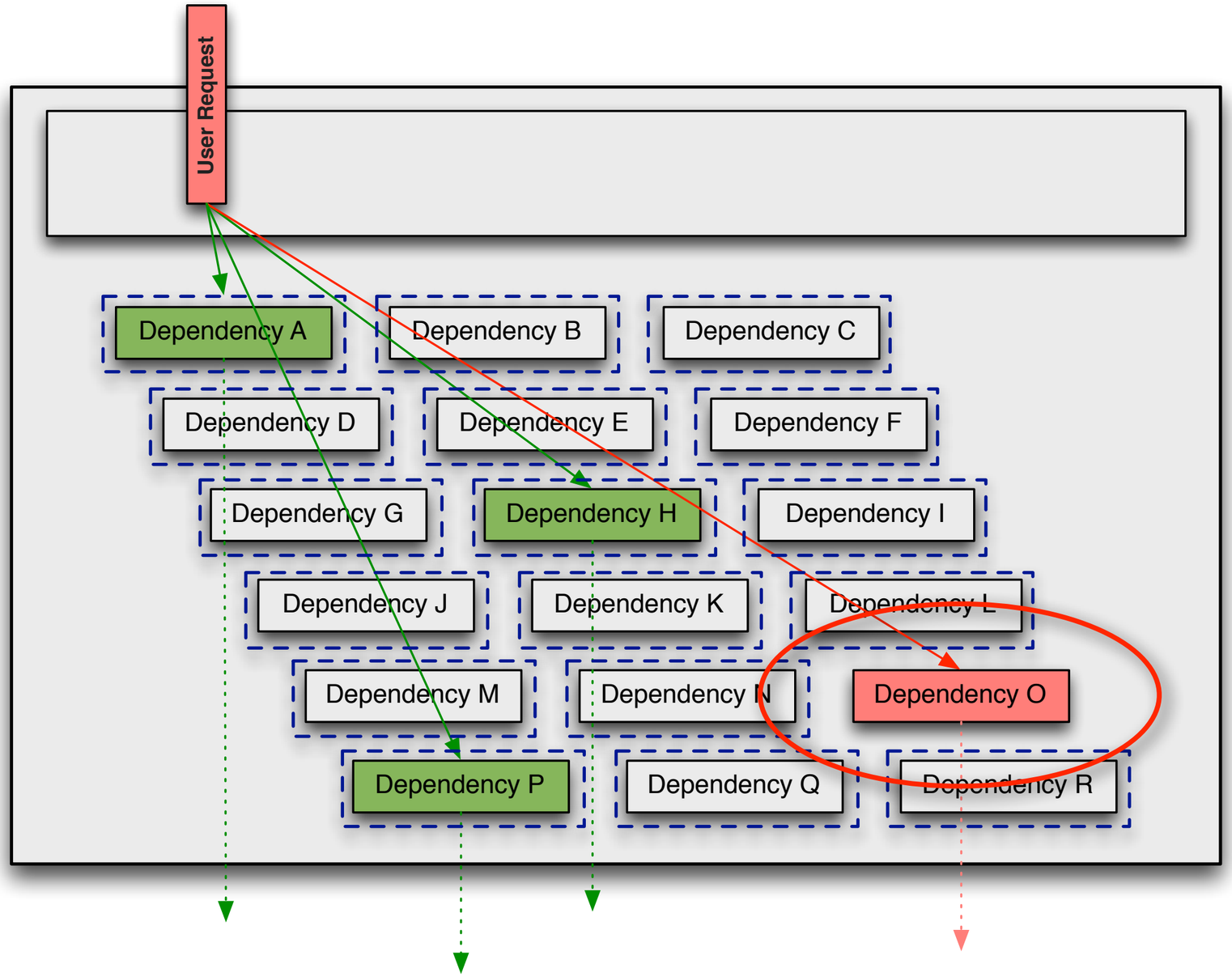
Every code deployment is preceded by a canary test where a small number of instances are launched to take production traffic, half with new code (canary), half with existing production code (baseline) and compared for differences. Thousands of system, application and bulkhead metrics are compared to make a decision on whether the new code should continue to full deployment. Many issues are found via production canaries that are not found in dev and test environments.



New instances are also put through a squeeze test before full rollout to find the point at which the performance degrades. This is used to identify performance and throughput changes of each deployment.



Long-term canaries are kept in a cluster we call “coalmine” with agents intercepting all network traffic. These run the same code as the production cluster and are used to identify network traffic without a bulkhead that starts happening due to unknown code paths being enabled via configuration, AB test and other changes. Read more at https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib/hystrix-network-auditor-agent



SYSTEM RELATIONSHIP OVER NETWORK WITHOUT BULKHEAD

For example, a network relationship could exist in production code but not be triggered in dev, test or production canaries but then be enabled via a condition that changes days after deployment to production. This can be a vulnerability and we use the “coalmine” to identity these situations and inform decisions.

Zuul Routing Layer

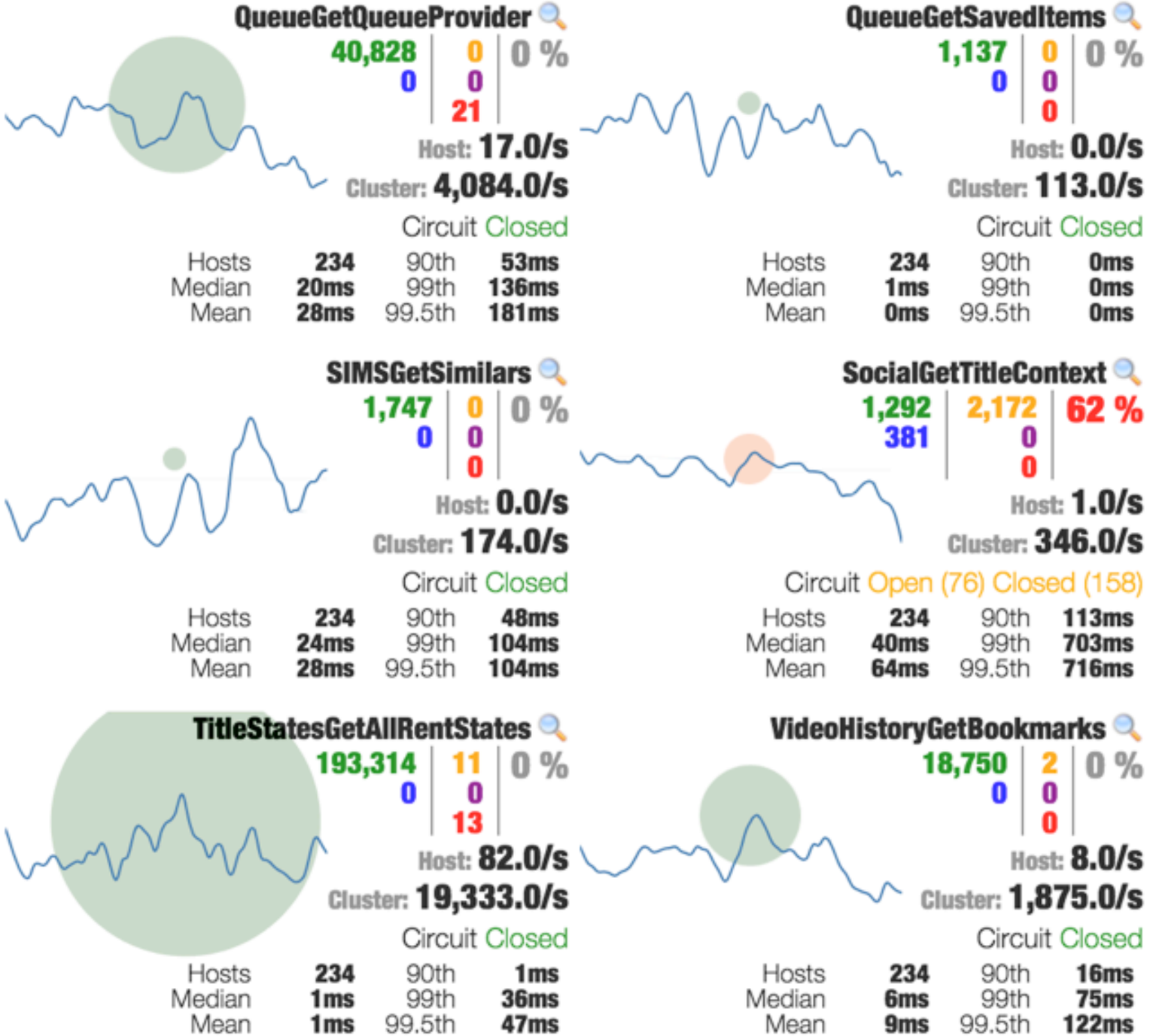
Canary vs Baseline

Squeeze

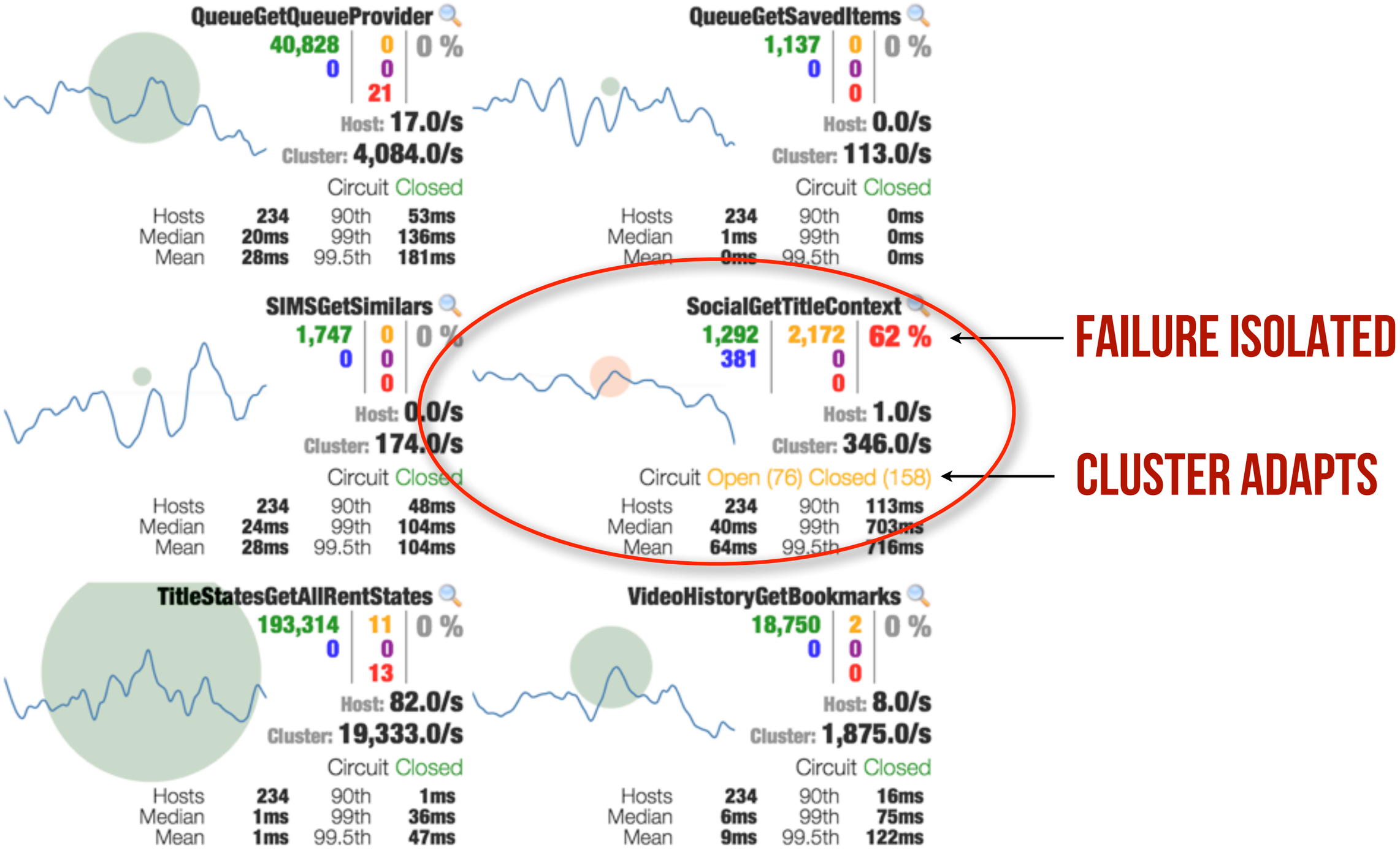
"Coalmine"

Production

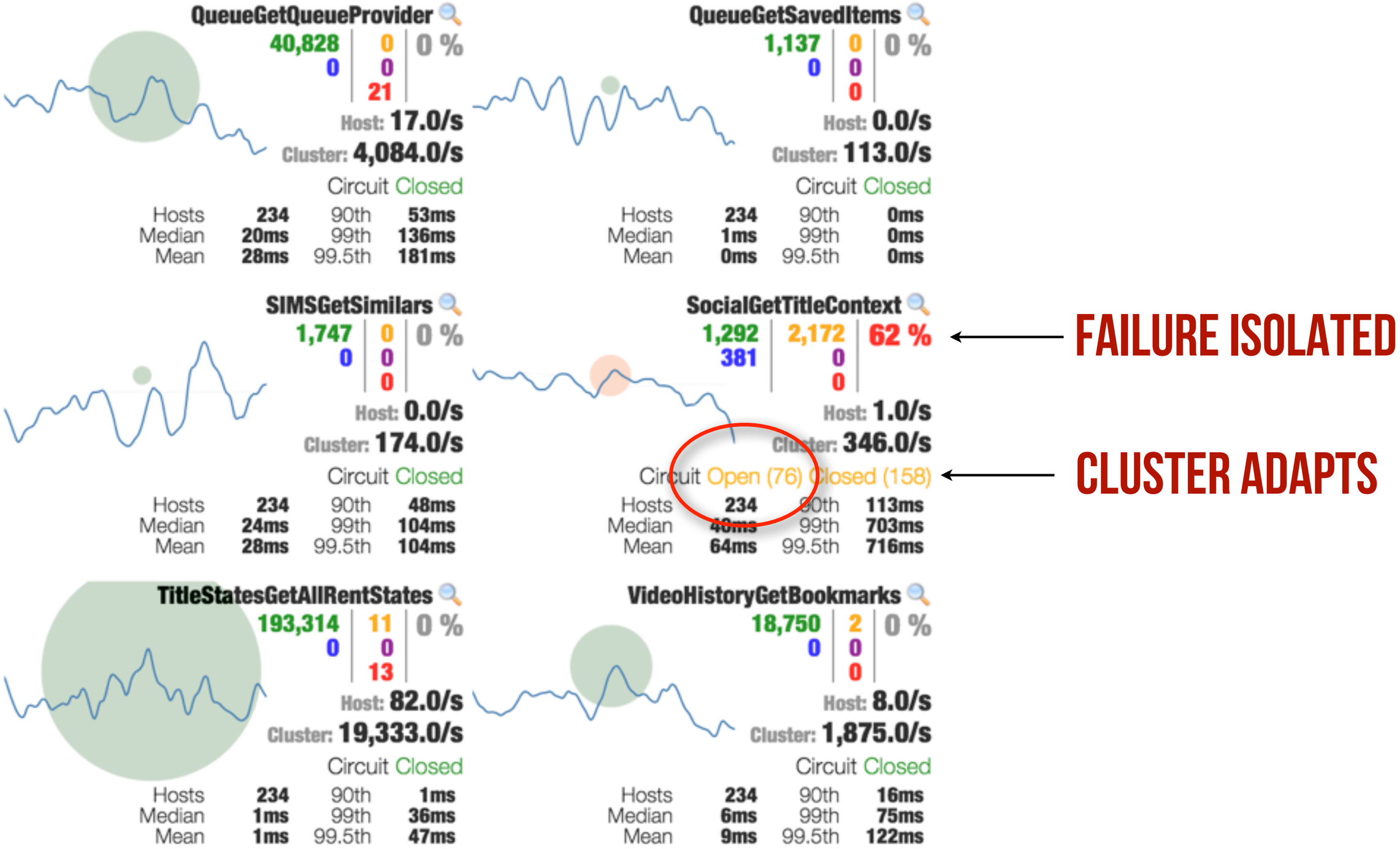
FAILURE INEVITABLY HAPPENS ...



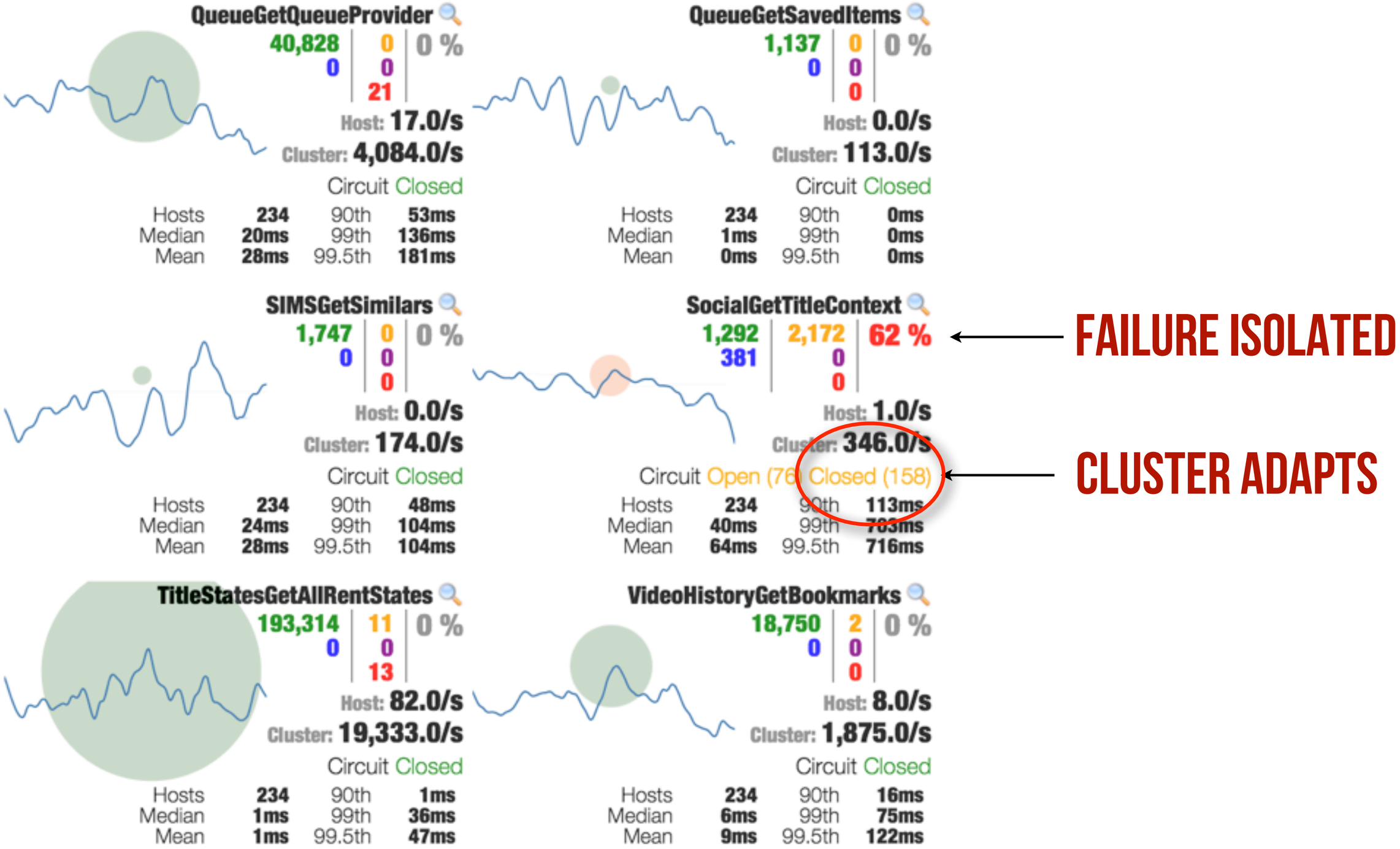
When the backing system for the ‘SocialGetTitleContext’ bulkhead became latent the impact was contained and fallbacks returned.



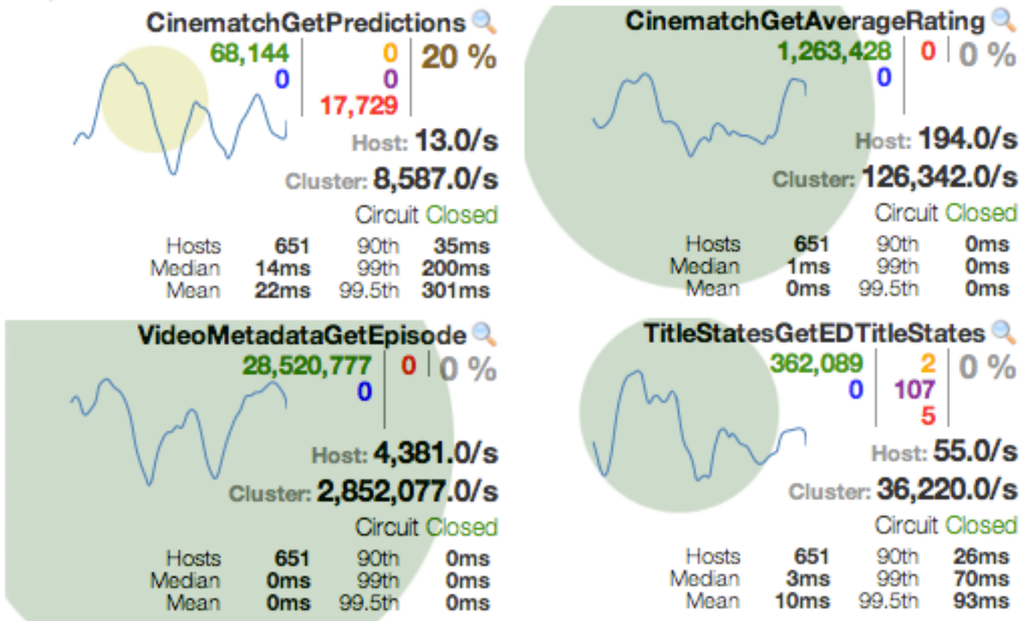
When the backing system for the ‘SocialGetTitleContext’ bulkhead became latent the impact was contained and fallbacks returned.

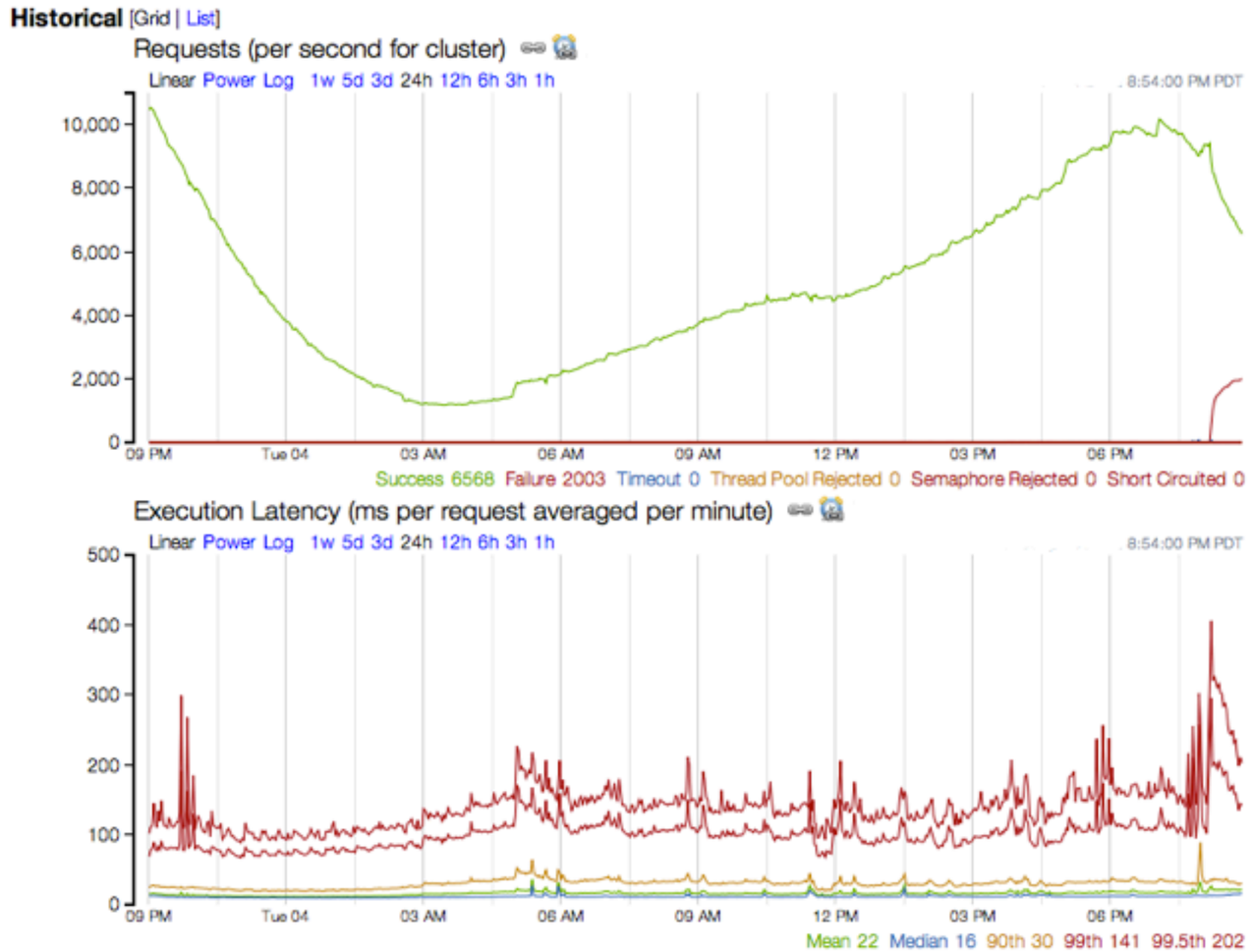


Since the failure rate was above the threshold circuit breakers began tripping. As a portion of the cluster tripped circuits it released pressure on the underlying system so it could successfully perform some work.

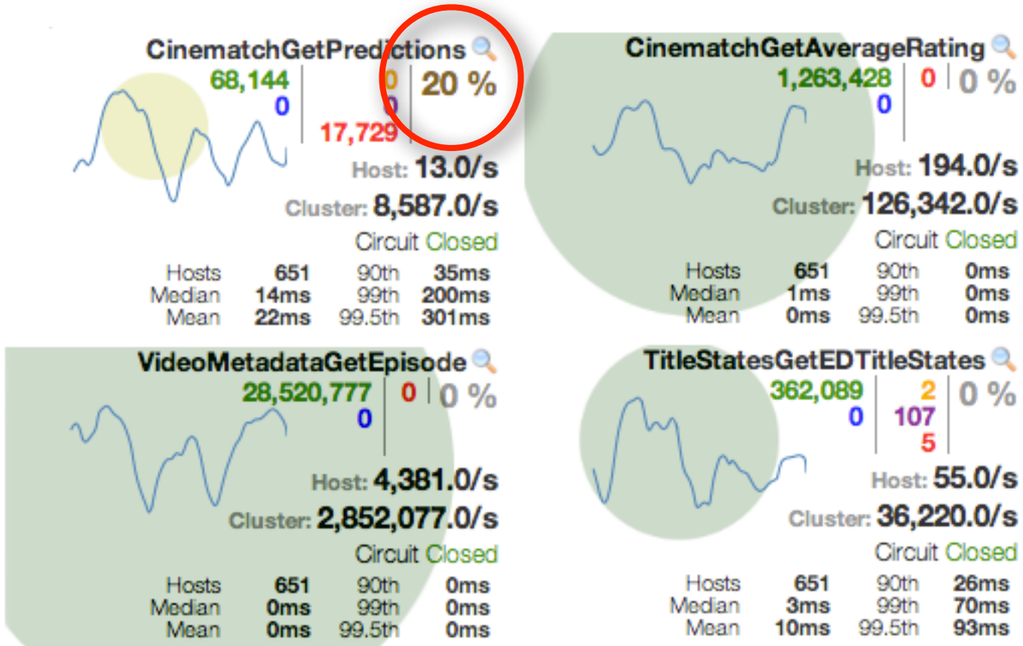


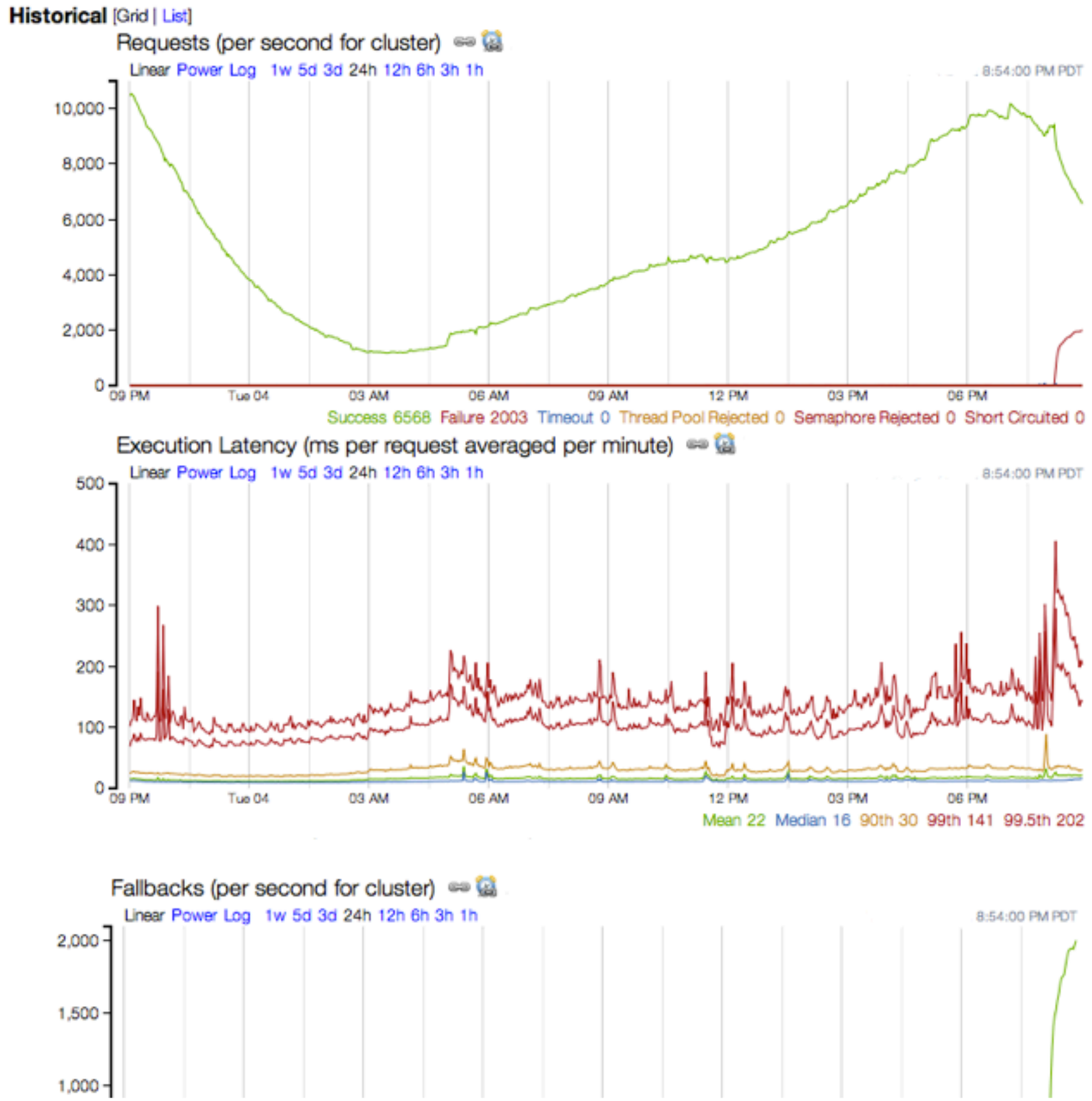
The cluster naturally adapts as bulkheads constrain throughput and circuits open and close in a rolling manner across the instances in the cluster.



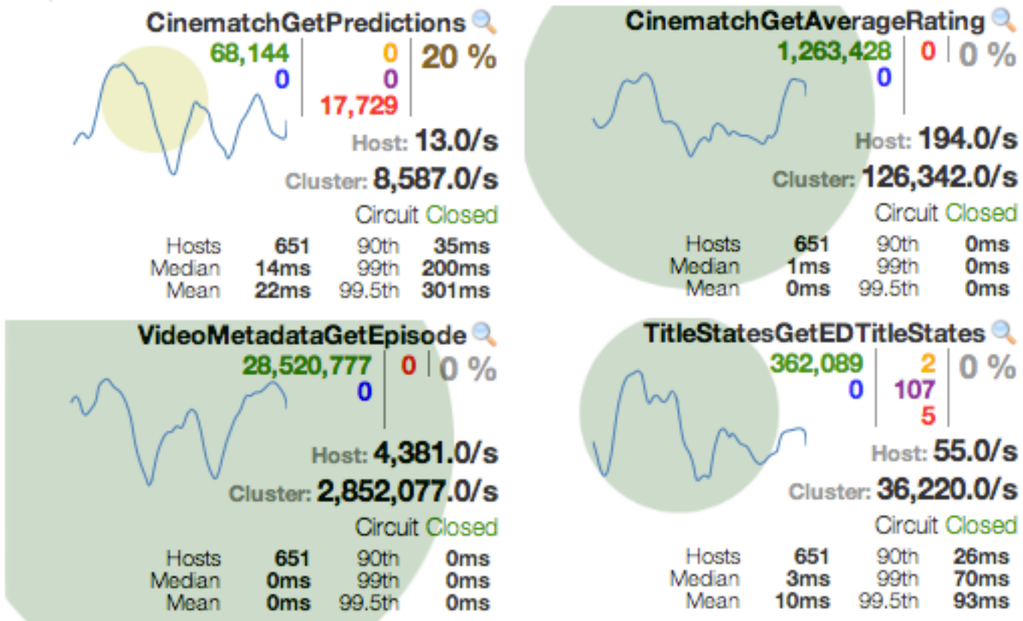


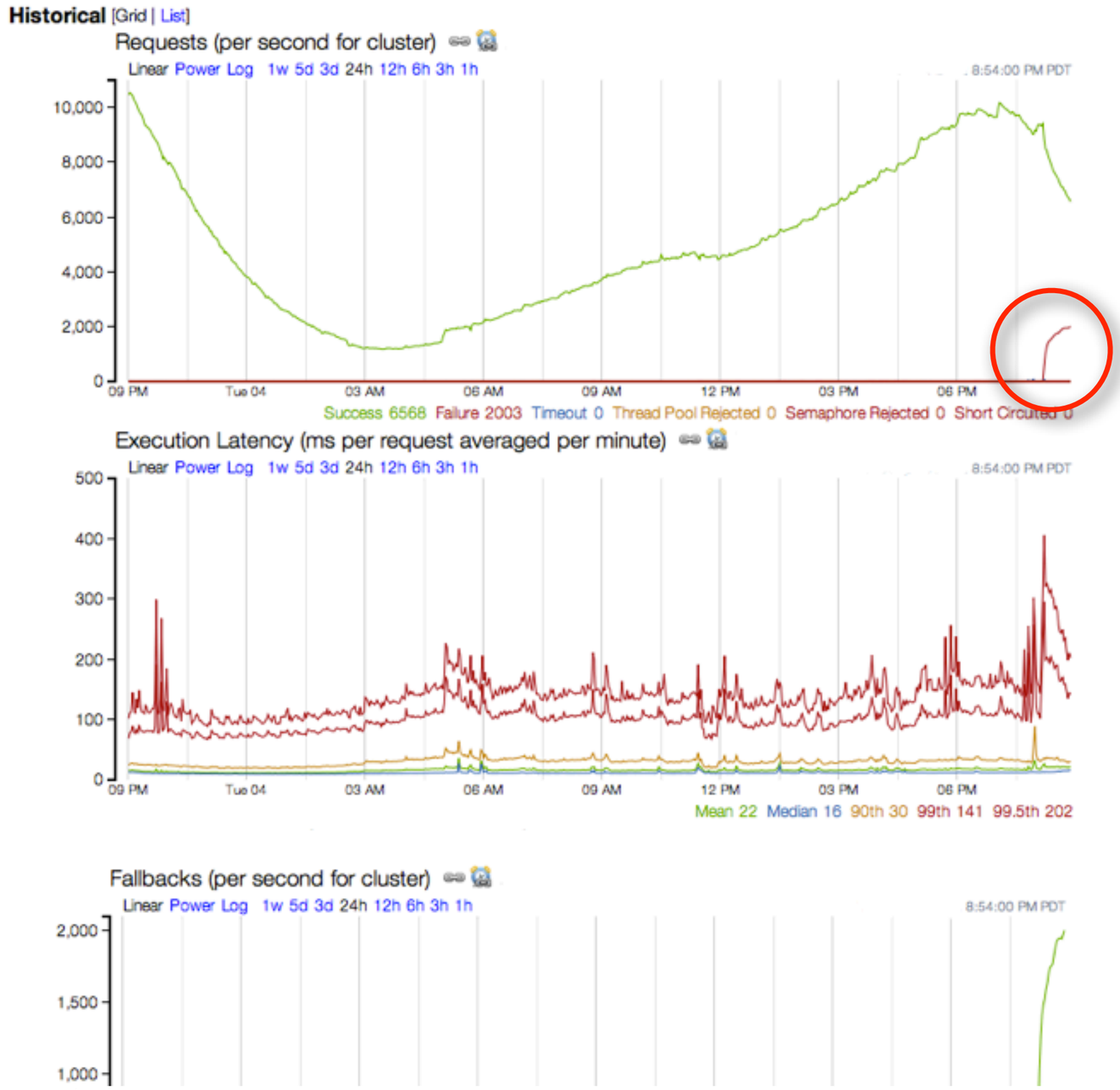
In this example the ‘CinematchGetPredictions’ functionality began failing.



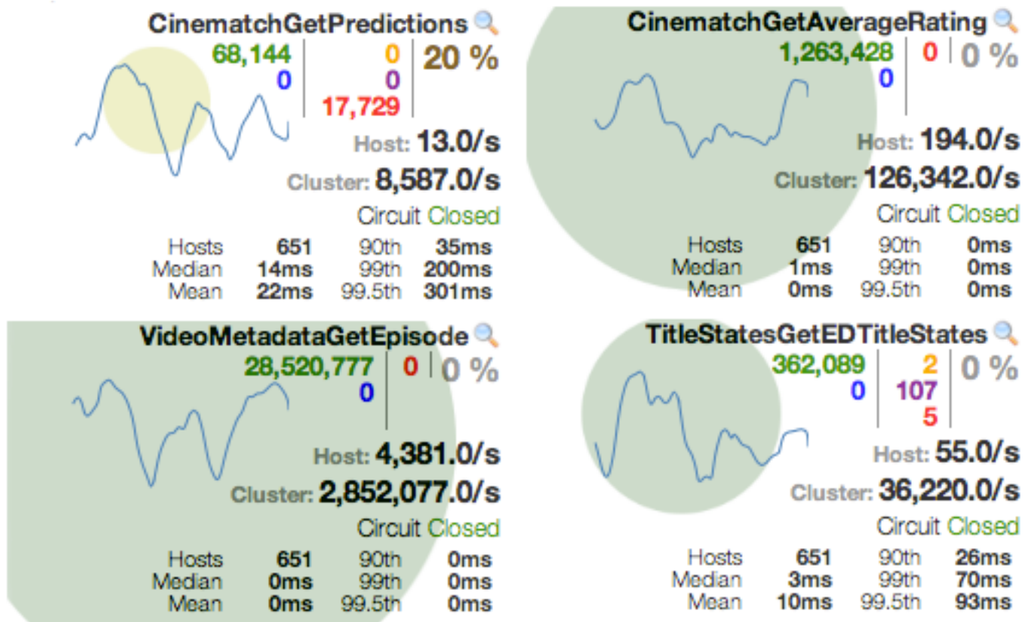


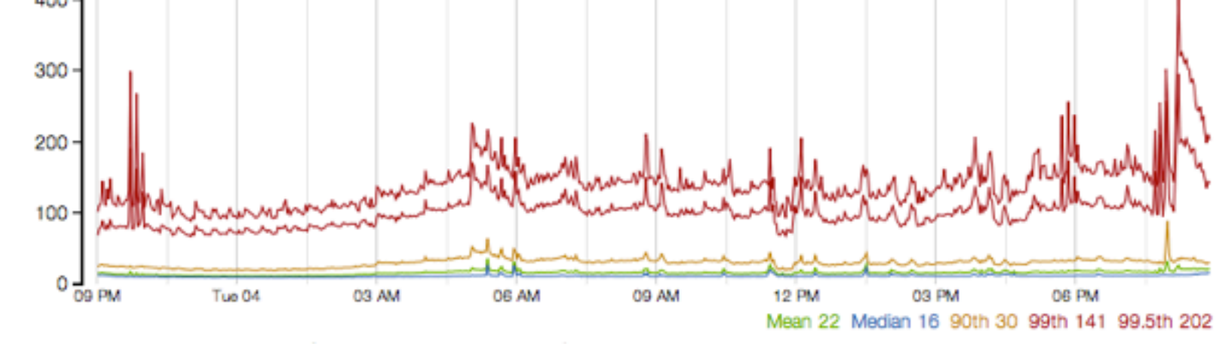
The red metric shows it was exceptions thrown by the client, not latency or concurrency constraints.

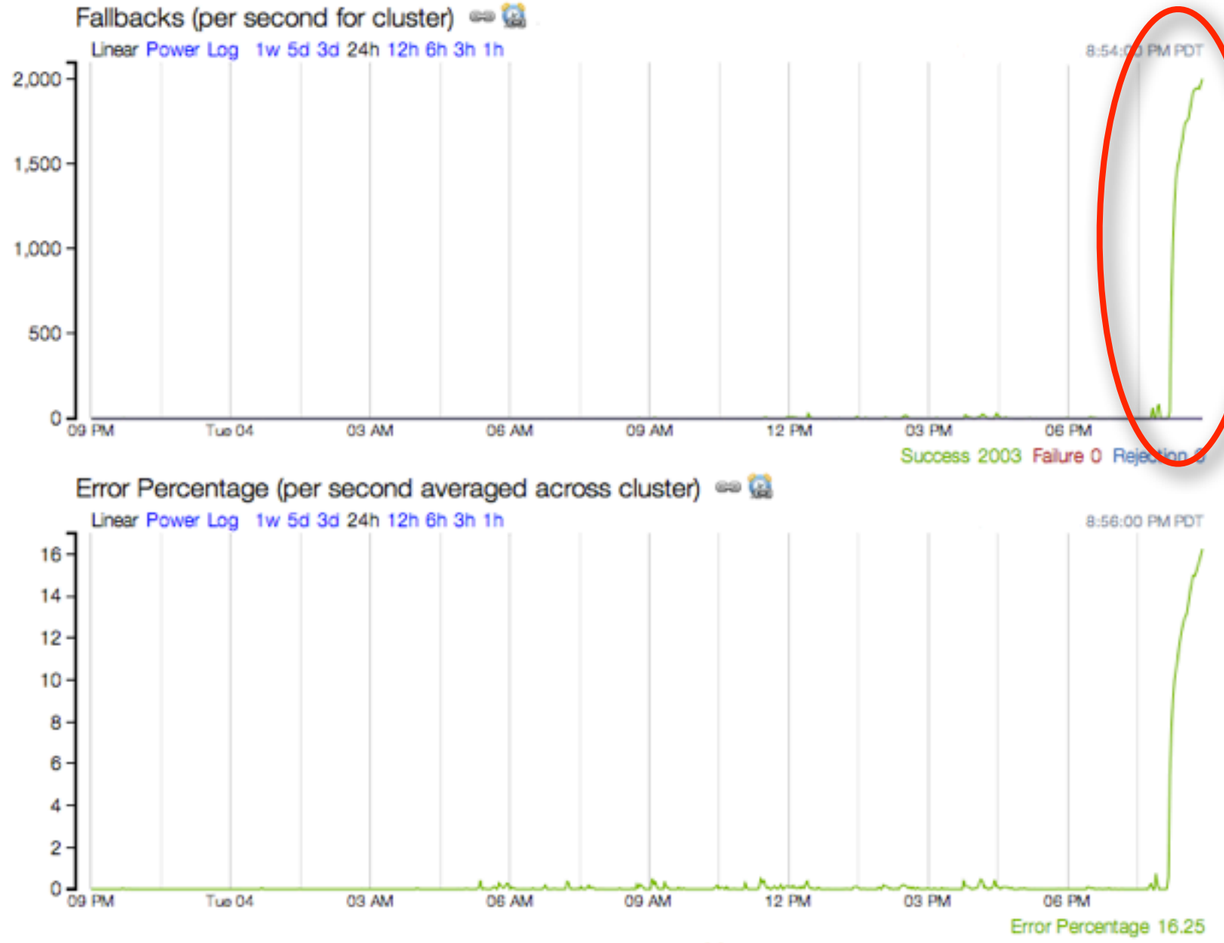




The 20% error rate from the realtime visualization is also seen in the historical metrics with accompanying drop in successes.







Matching the increase in failures is the increase of fallbacks being delivered for every failure.

DISTRIBUTED SYSTEMS ARE COMPLEX

HYSTRIX

DEFEND YOUR APP

Distributed applications need to be treated as complex systems and we must recognize that no machine or human can comprehend all of the state or interactions.

ISOLATE RELATIONSHIPS

HYSTRIX

DEFEND YOUR APP

One way to dealing with the complex system is to isolate the relationships so they can each fail independently of each other. Bulkheads have proven an effective approach for isolating and managing failure.

AUDITING & OPERATIONS ARE ESSENTIAL

HYSTRIX

DEFEND YOUR APP

Resilient code is only part of the solution. Systems drift and have latent bugs and failure states emerge from the complex interactions of the many relationships. Constant auditing can be part of the solution. Human operations must handle everything the system can't which by definition means it is unknown so the system must strive to expose clear insights and effective tooling so humans can make informed decisions.

HYSTRIX

https://github.com/Netflix/Hystrix

APPLICATION RESILIENCE IN A SERVICE-ORIENTED ARCHITECTURE

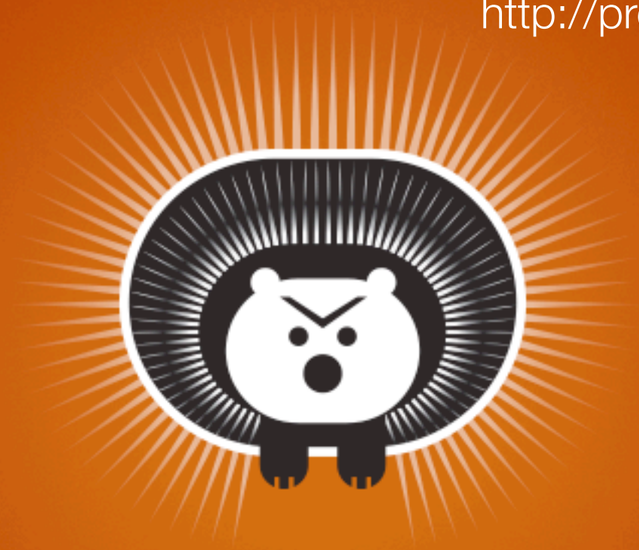
http://programming.oreilly.com/2013/06/application-resilience-in-a-service-oriented-architecture.html

FAULT TOLERANCE IN A HIGH VOLUME, DISTRIBUTED SYSTEM

http://techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html

MAKING THE NETFLIX API MORE RESILIENT

http://techblog.netflix.com/2011/12/making-netflix-api-more-resilient.html



HYSTRIX

DEFEND YOUR APP

BEN CHRISTENSEN

@BENJCHRISTENSEN

HTTP://WWW.LINKEDIN.COM/IN/BENJCHRISTENSEN

NETFLIX

JOBS.NETFLIX.COM