CHICAGO
INTERNATIONAL
SOFTWARE DEVELOPMENT
CONFERENCE 2016

goto; conference

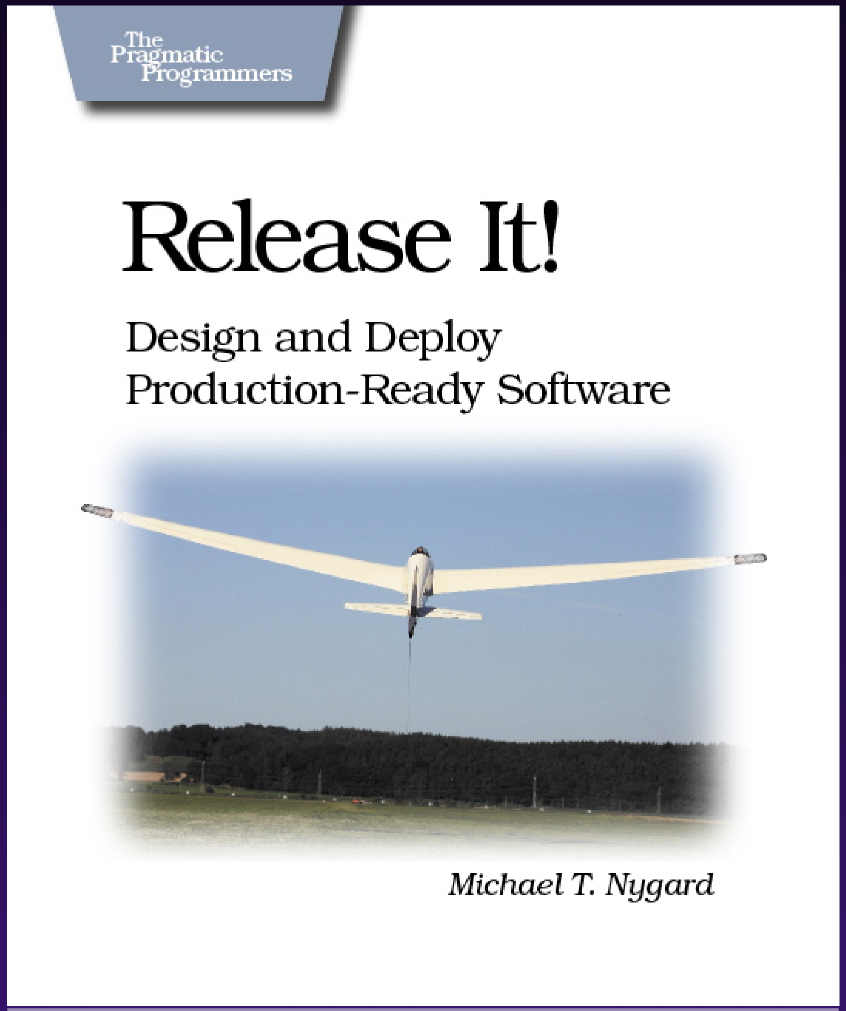
# Stability Patterns and Antipatterns

*Michael Nygard*

follow us @gotochgo

Conference: May 24th-25th / Workshops: 23th-26th

# A Developer Sojourns in Operations


The Pragmatic Programmers
Release It!
Design and Deploy
Production-Ready Software
Michael T. Nygard

**Availability**

Probability that system is operating at time *t*.

**Stability**

Architectural characteristic producing availability despite faults and errors.

**Fault**

Incorrect internal state. Initiated via defect or injection.

**Error**

Observably incorrect operation.

**Failure**

Loss of availability. System unresponsive.

# Stability Antipatterns

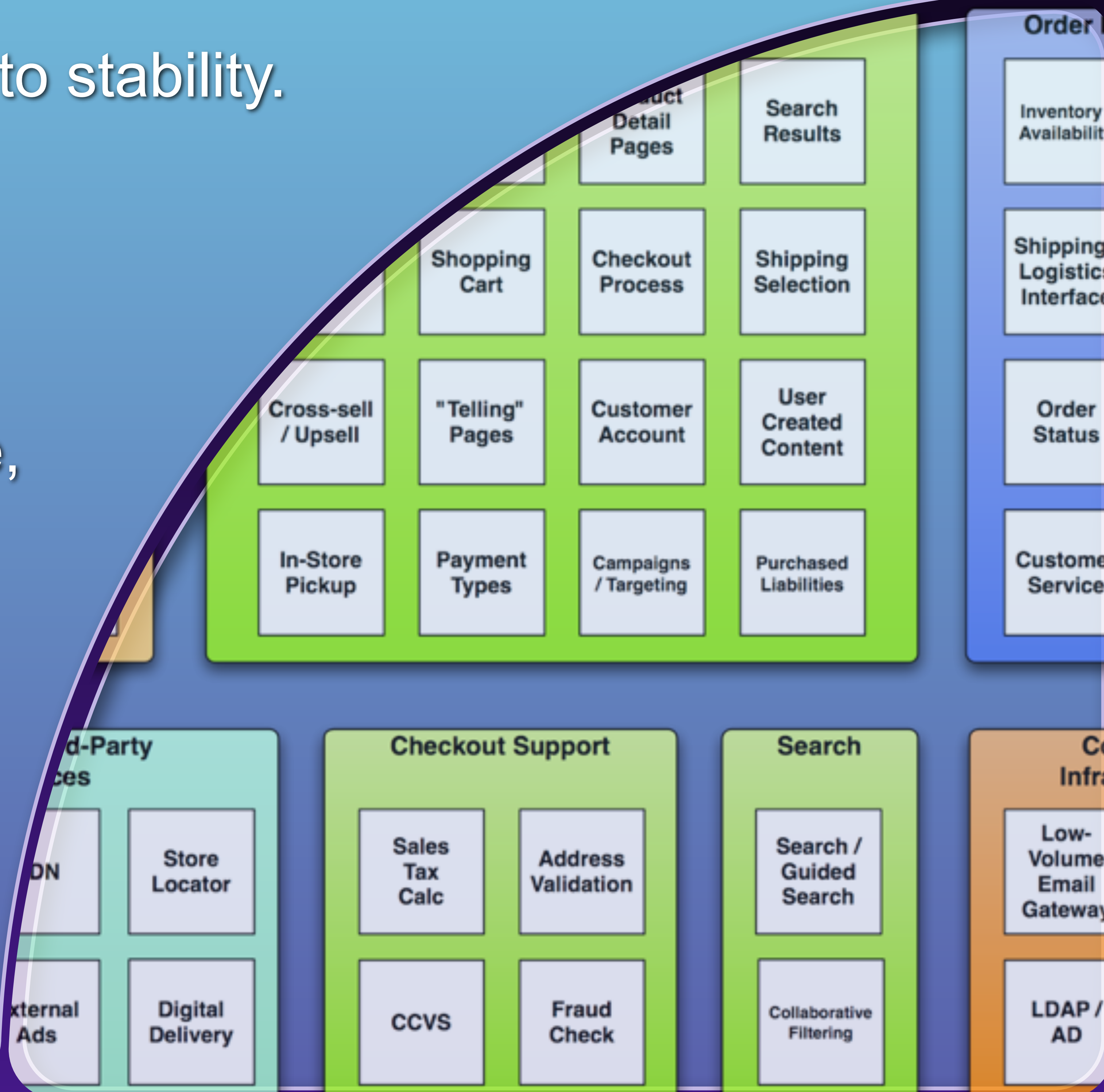
# Integration Points

Integrations are the #1 risk to stability.

Your first job is to protect against integration points.

Every socket, process, pipe, or remote procedure call can and will eventually kill your system.

Even database calls can hang, in obvious and not-so-obvious ways.



# Example: Wicked database hang

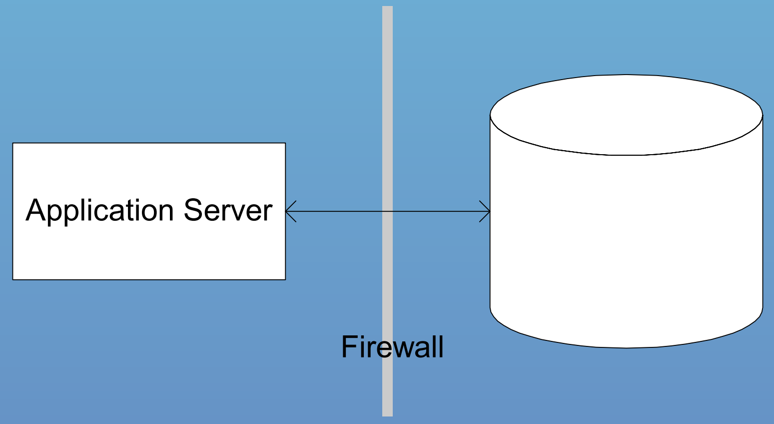
**Not at all obvious:** Firewall idle connection timeout

"Connection" is an abstraction.

The firewall only sees packets.

It keeps a table of "live" connections.

When the firewall sees a TCP teardown sequence, it removes that connection from the table.




To avoid resource leaks, it will drop entries from table after idle period timeout.

Causes broken database connections after long idle period, like 2 a.m. to 5 a.m.

**Simple solution:** Enable "dead connection detection" (Oracle) or similar feature to keep connection alive.

**Alternative solution:** timed job to periodically issue trivial query.

**What about prevention?**

# “In Spec” vs. “Out of Spec”

Example: Request-Reply using XML over HTTP

**“In Spec” failures**

- TCP connection refused
- HTTP response code 500
- Error message in XML response

Well-Behaved Errors

**“Out of Spec” failures**

- TCP connection accepted, but no data sent
- TCP window full, never cleared
- Server never ACKs TCP, causing very long delays as client retransmits
- Connection made, server replies with SMTP hello string
- Server sends HTML “link-farm” page
- Server sends one byte per second
- Server sends Weird Al catalog in MP3

Wicked Errors

# Remember This

Know when to open up abstractions.

Failures propagate quickly.

Large systems fail faster than small ones.

Apply “Circuit Breaker”, “Use Timeouts”, “Use Decoupling Middleware”, and “Handshaking” to contain and isolate failures.

Use “Test Harness” to find problems in development.

# Chain Reaction

Failure in one component raises probability of failure in its peers

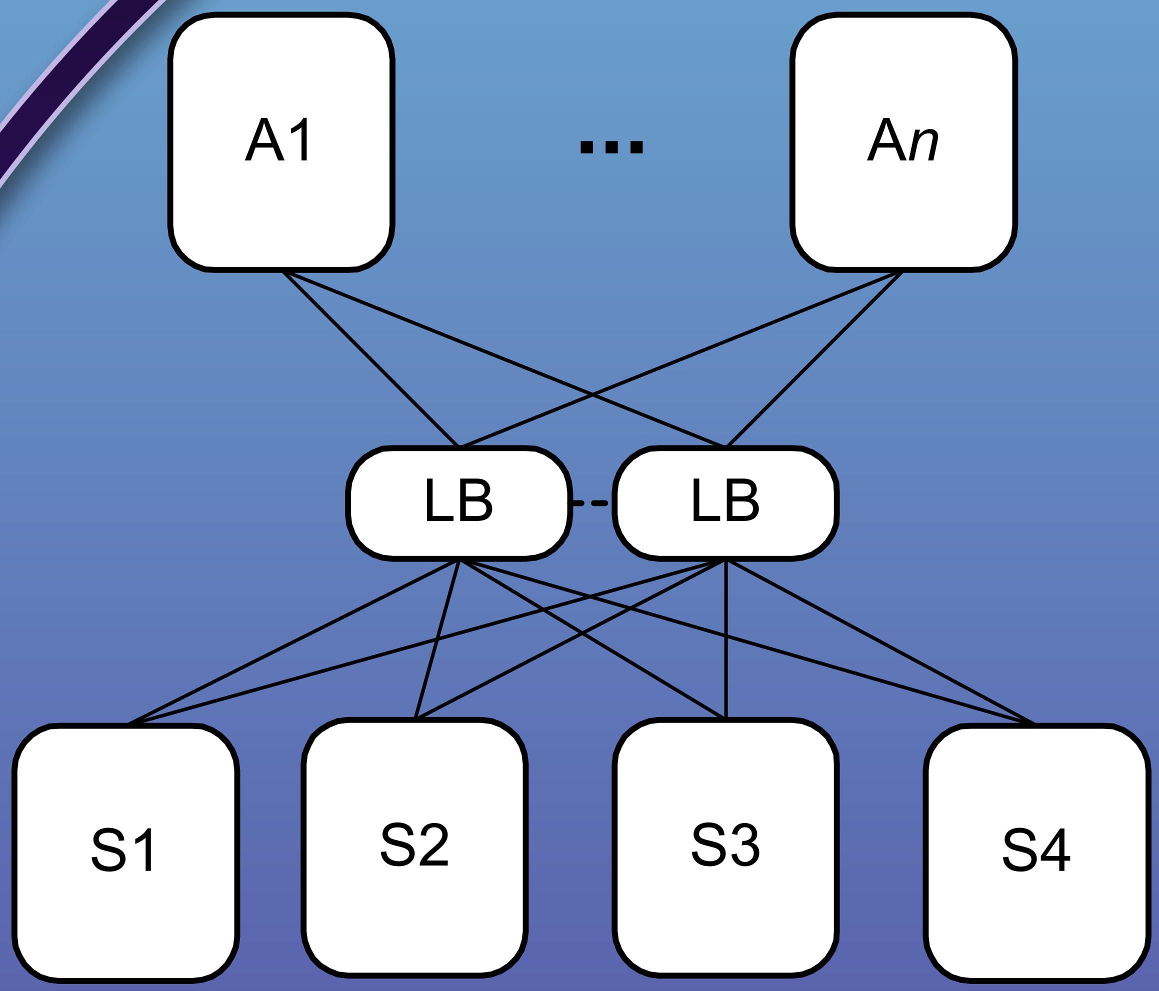
Example:

- Suppose S4 goes down
- S1 - S3 go from 25% of total to 33% of total
- That's 33% more load

Each one dies faster

Failure moves horizontally across tier

Common in search engines and application servers



# Remember This

One server down jeopardizes the rest.

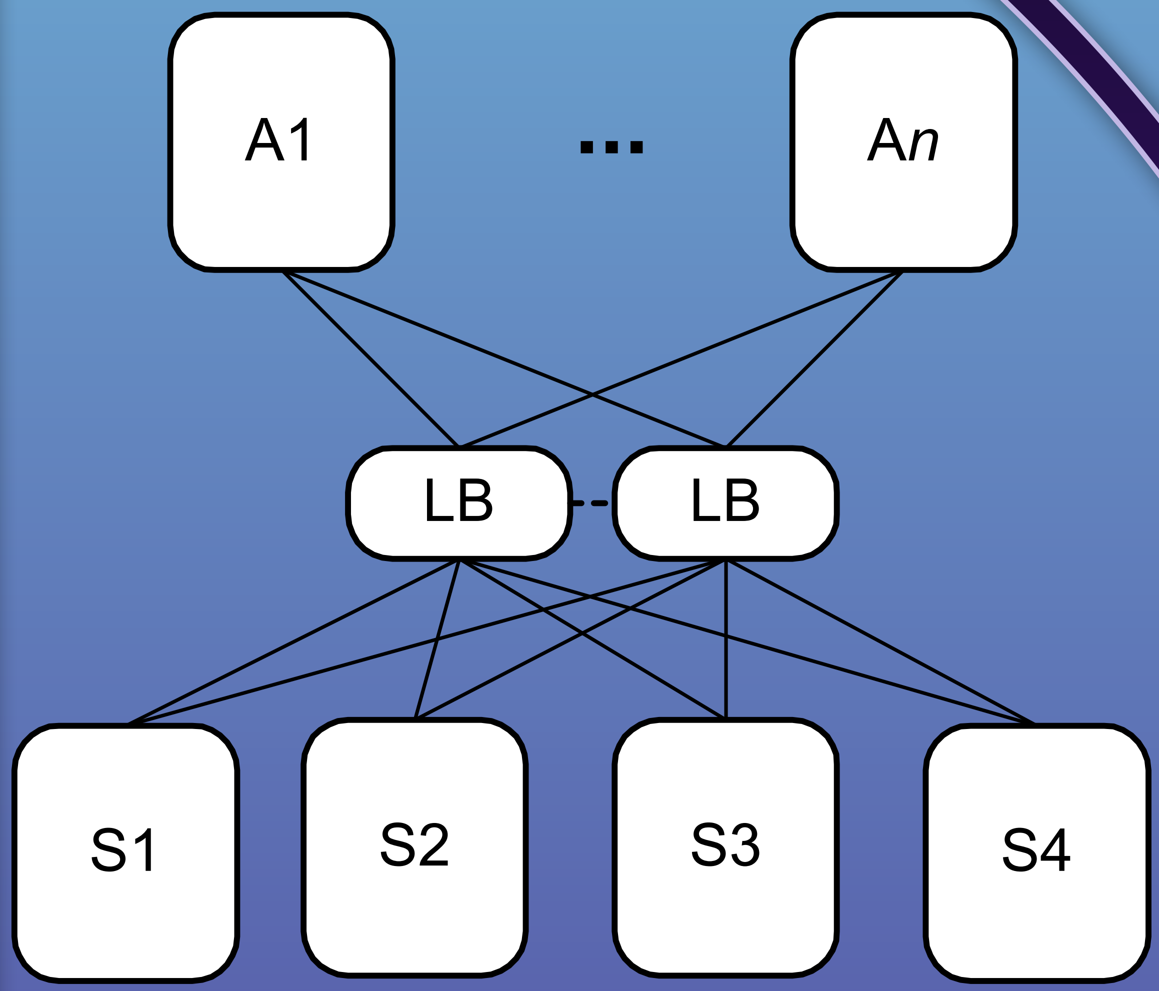
Hunt for Resource Leaks.

Defend with “Bulkheads”.

# Cascading Failure

Failure in one system causes calling systems to be jeopardized

## The Microservice Failure Mode




Example:

System S goes down, causing calling system A to get slow or go down.

# Remember This

Damage Containment

Scrutinize resource pools

Defend via Timeouts & Circuit Breakers

# Attacks of Self-Denial

Good marketing can kill your system at any time.

Send promotion to a “select group”

About 10,000,000 times more show up

Get crushed

# Defending the Ramparts

Avoid deep links

Set up static landing pages

Only allow the user's second click to reach application servers

Allow throttling of incoming users

Set up lightweight versions of dynamic pages.

Use your CDN to divert users

Use shared-nothing architecture

# Remember This

Keep lines of communication open

Protect shared resources

Expect instantaneous distribution of exploits

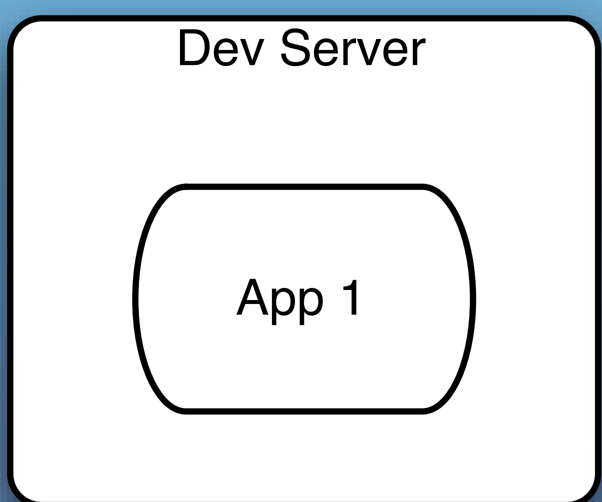
# Scaling Effects

Understand which end of the lever you are sitting on.

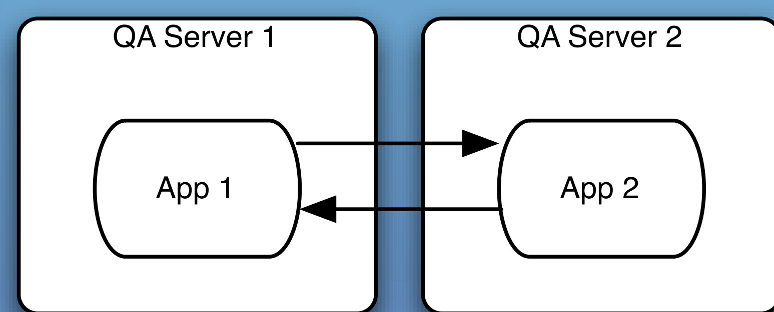
- Ratios in dev and QA tend to be 1:1
  - Web server to app server
  - Front end to back end
- Production is wildly different

# Example: Point to Point Cache Invalidation

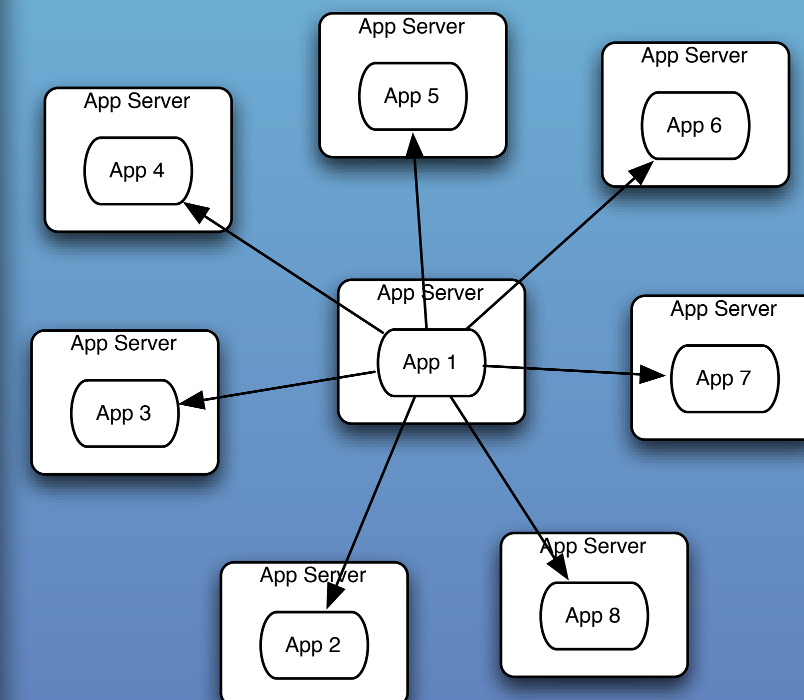
## Development



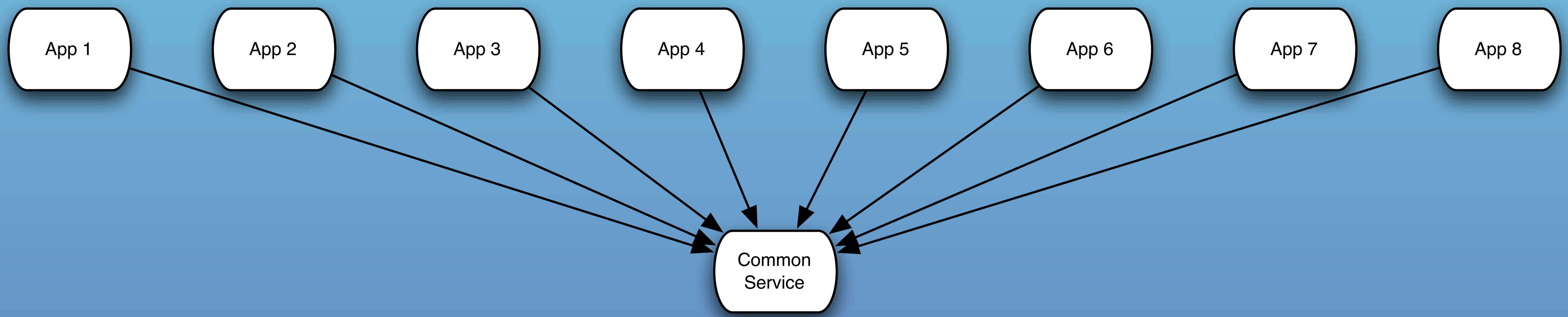

## QA




## Production



# Example: Shared Resources




Examine services you call. Are they sized correctly?

# Remember This

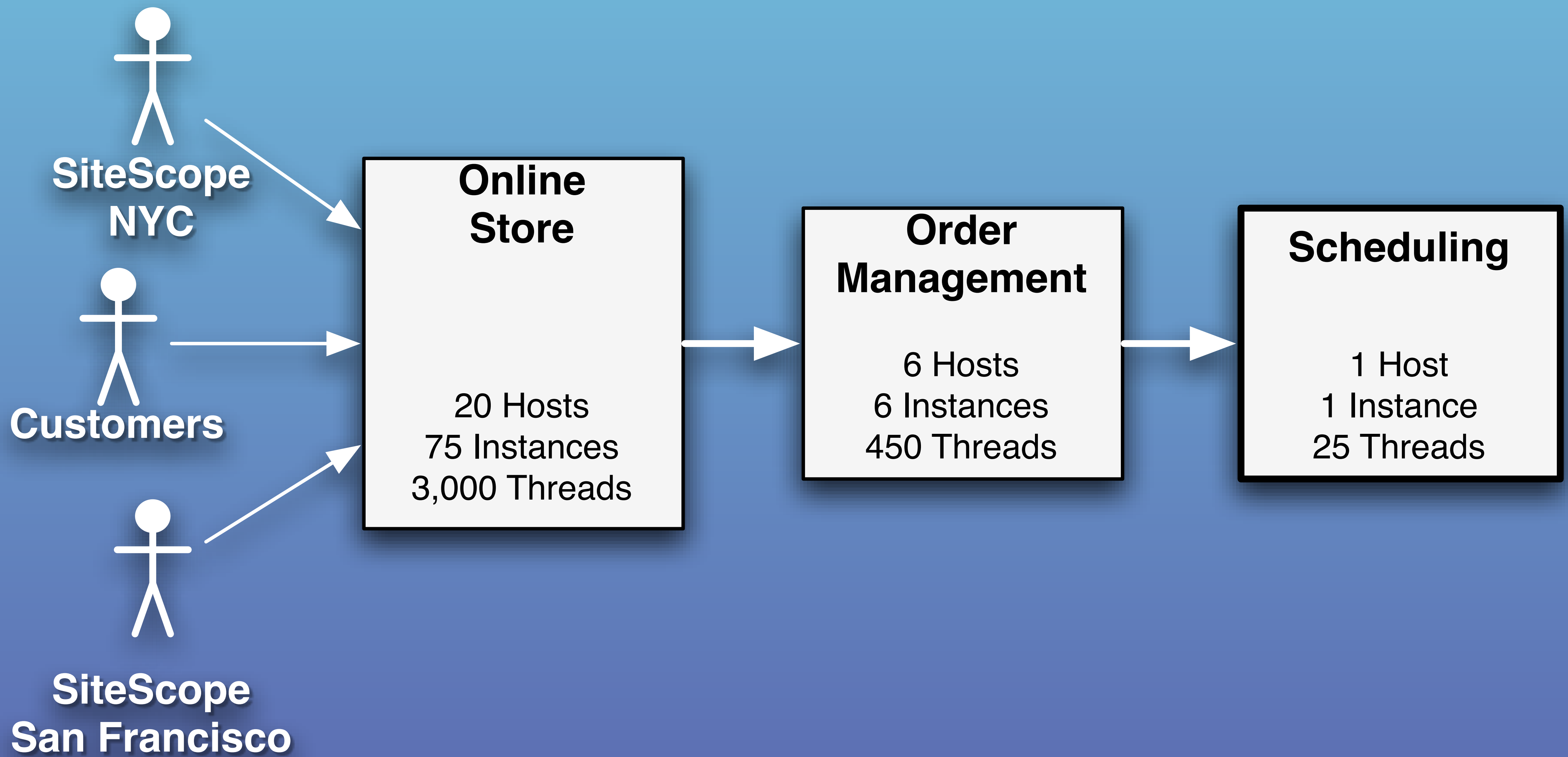
Desk check ratios

Broadcast instead of point-to-point

Watch out for shared resources

# Unbalanced Capacities

Traffic floods sometimes start inside the data center walls.



# Unbalanced Capacities

Unbalanced capacities is a type of scaling effect that occurs between systems in an enterprise.

May appear after changes in traffic patterns

# Remember This

Examine server and thread counts

Watch out for changes in traffic patterns

Stress both sides of the interface in QA

Simulate back end failures during testing

# Slow Responses

Slow response is worse than no response

What does your server do when it’s overloaded?

- “Connection refused” is a fast failure, the caller’s thread is released right away
- A slow response ties up the caller’s thread, makes the user wait
- It uses capacity on caller and receiver
- If the caller times out, then the work was wasted

# Slow Responses

- Too much load on system
- Transient network saturation
- Firewall overloaded
- Protocol with retries built in (NFS, DNS)
- Chatty remote protocols

# Remember This

Slow responses trigger cascading failures

Slow responses invite more traffic

Don’t send a slow response; fail fast

Hunt for memory leaks or resource contention

# Unbounded Result Sets

Limited resources, unlimited data volume

Development and testing is done with small data sets

Test databases get reloaded frequently

Queries that are OK in dev bonk badly with production data volume.

# Unbounded Result Sets: Databases

SQL queries have no inherent limits

ORM tools are bad about this

It starts as a degenerating performance problem, but can tip the system over

# Unbounded Result Sets: SOA

- Often found in chatty remote protocols, together with the N+1 query problem
- Causes problems on the client and the server
  - On server: constructing results, marshalling XML
  - On client: parsing XML, iterating over results.
- This is a breakdown in handshaking.  The client knows how much it can handle, not the server.

# Remember This

- Test with realistic data volumes
  - Scrubbed production data is the best.
  - Generated data also works.
- Don’t rely on the data producers.  Their behavior can change overnight.
- Put limits in your application-level protocols:
  - WS, RMI, DCOM, XML-RPC, etc.

# Stability Patterns

# Use Timeouts

Don’t hold your breath.

- In any server-based application, request handling threads are your most precious resource
  - When all are busy, you can’t take new requests
  - When they stay busy, your server is down
  - Busy time determines overall capacity
- Protect request handling threads at all costs

# Considerations

Calling code must be prepared for timeouts.

Better error handling is a good thing anyway.

Beware third-party libraries and vendor APIs.

# Remember This

Apply to Integration Points, Blocked Threads, and Slow Responses

Apply to recover from unexpected failures.

Consider **delayed** retries.

# Circuit Breaker

Defend yourself.

Have you ever seen a remote call wrapped with a retry loop?

```
int remainingAttempts = MAX_RETRIES;

while(--remainingAttempts >= 0) {
  try {
    doSomethingDangerous();
    return true;
  } catch(RemoteCallFailedException e) {
    log(e);
  }
}
return false;
```

Why?

# Retries Hurt Users and Systems

## Users:

Retries make the user wait even longer to get an error response.

After the final retry, what happens to the users' work?

The target service may be non-critical, so why damage critical features for it?

## Systems:

Ties up caller's resources, reducing overall capacity.

If target service is busy, retries increase its load at the worst time.

Every single request will go through the same retry loop, letting a back-end problem cause a front-end brownout.

# Stop Banging Your Head

Circuit Breaker:

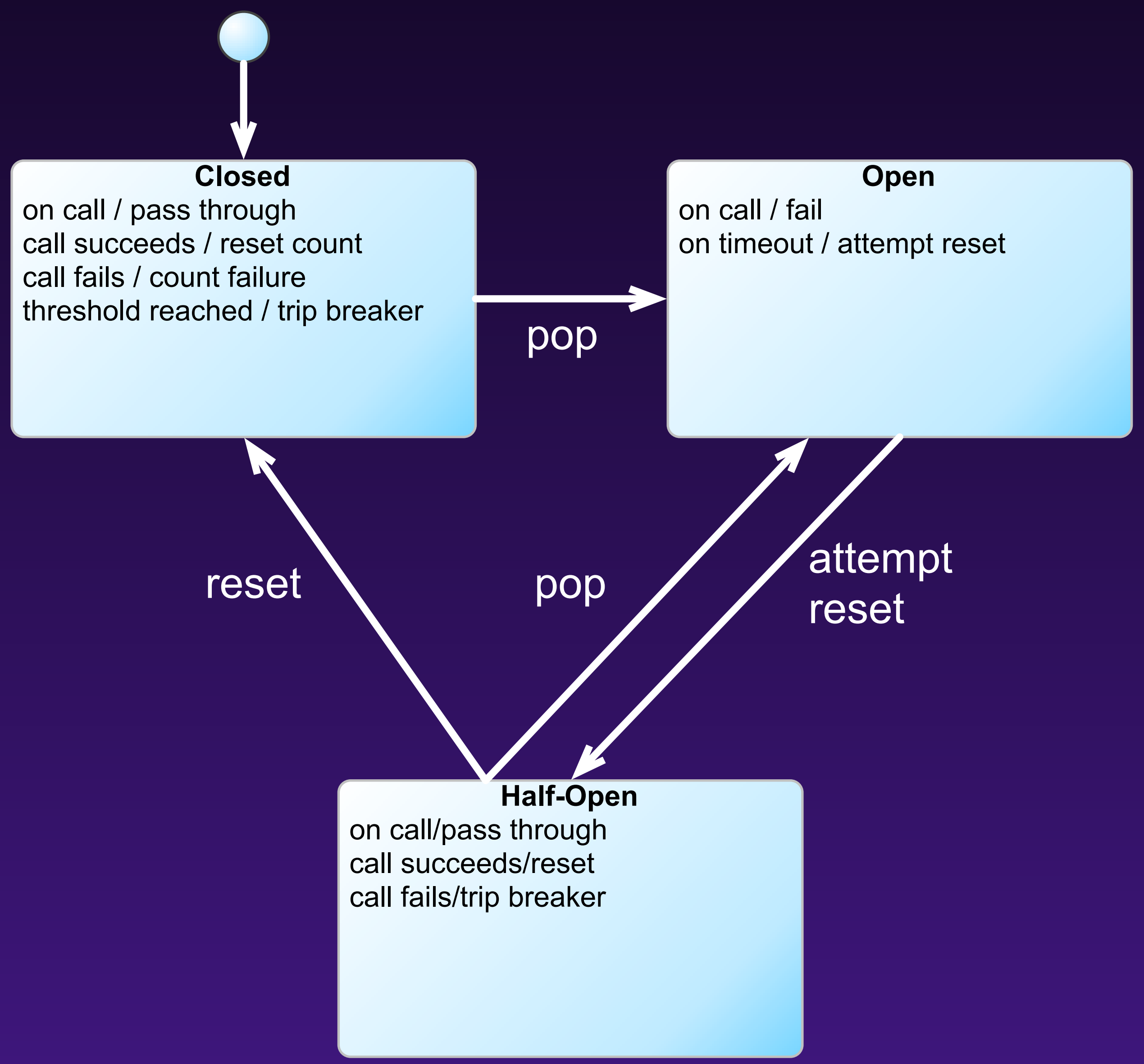
Wraps a “dangerous” call

Counts failures

After too many failures, stop passing calls through

After a “cooling off” period, try the next call

If it fails, wait for another cooling off time before calling again



# Remember This

Use Circuit Breakers together with Timeouts

Expose, track, and report state changes

Circuit Breakers prevent Cascading Failures

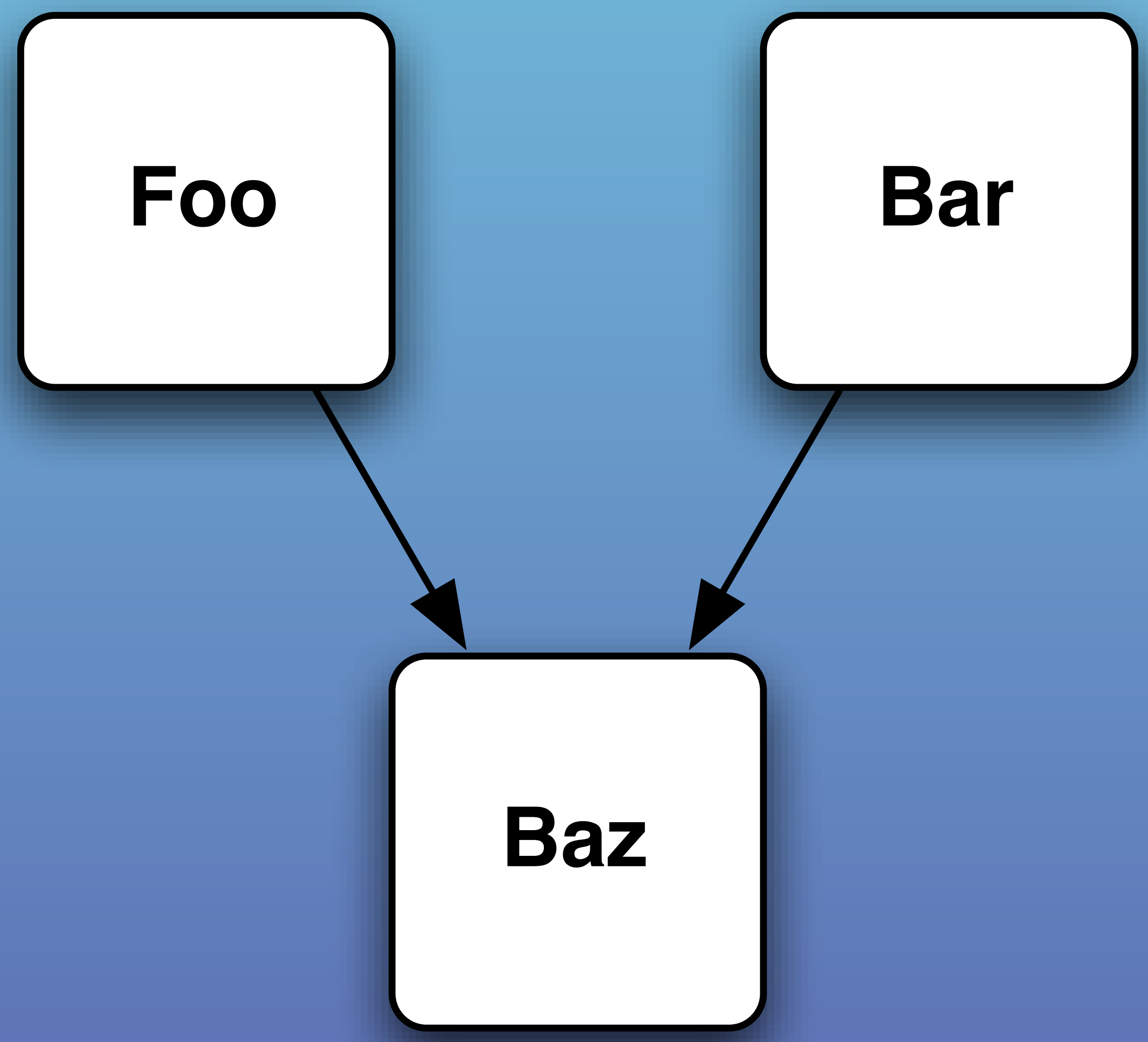
They protect against Slow Responses

# Bulkheads

Save part of the ship, at least.

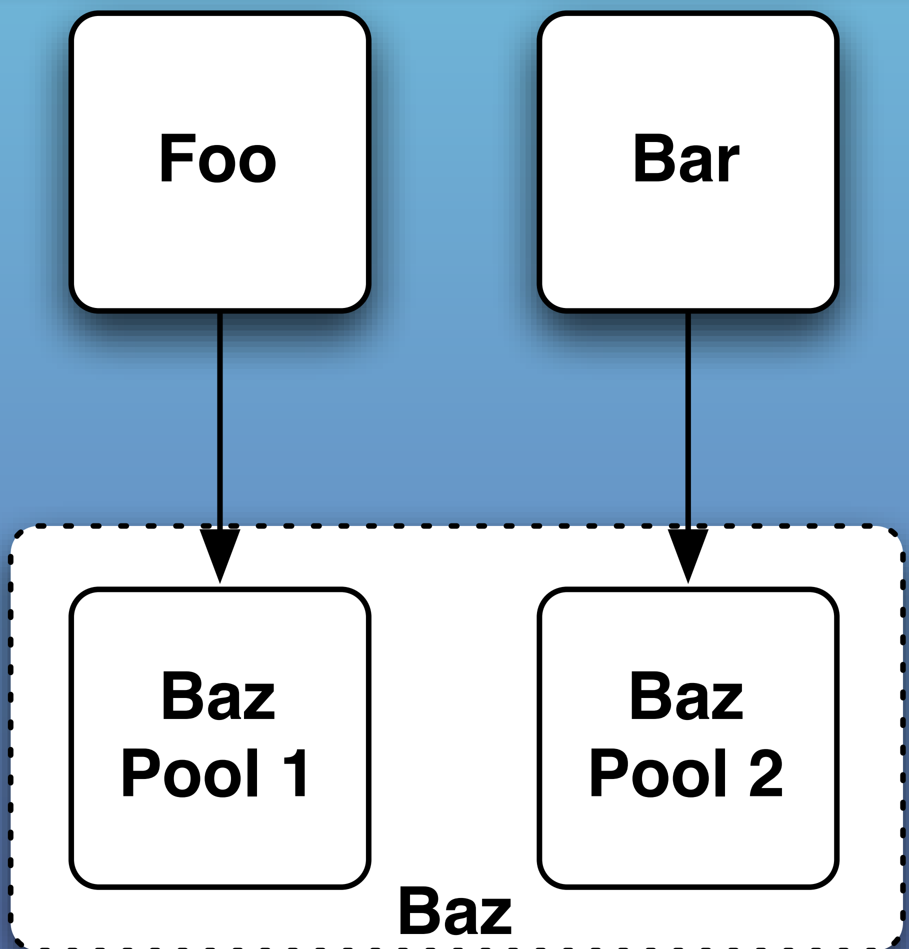
- Increase resilience by partitioning (compartmentalizing) the system
  - One part can go dark without losing service entirely
- Apply at several levels
  - Thread pools within a process
  - CPUs in a server (CPU binding)
  - Server pools for priority clients

# Common Mode Dependency: Service-Oriented Architecture




Foo and Bar are coupled by their shared use of Baz

# SOA with Bulkheads




Foo and Bar each have dedicated resources from Baz.

Each pool can be rebooted, or upgraded, independently.

Surging demand–or bad code–in Foo only harms Foo.

# Remember This

Save part of the ship

Pick a useful granularity

Very important with SaaS and microservices

Monitor each partitions performance to SLA

# Steady State

Run indefinitely without fiddling.

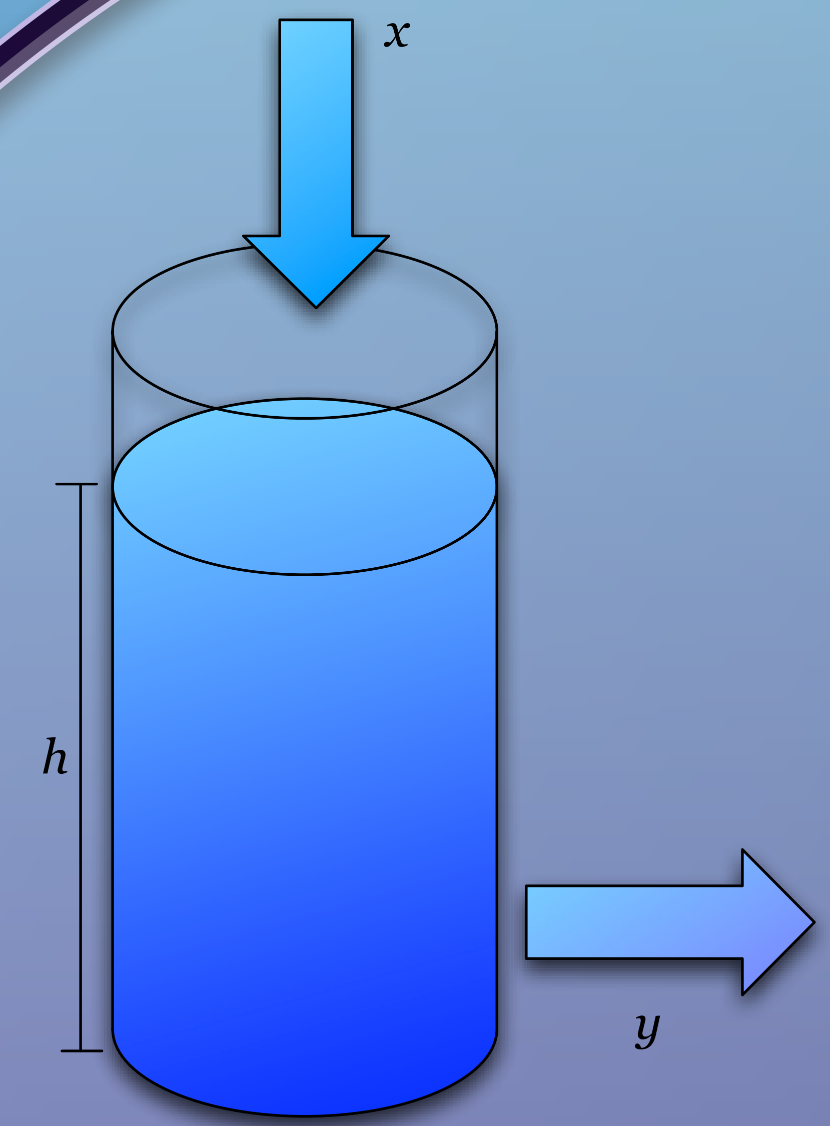
Run without crank-turning and hand-holding

Human error is a leading cause of downtime

If regular intervention is needed, then missing the schedule will cause downtime

# Routinely Recycle Resources

- All computing resources are finite
- For every mechanism that accumulates resources, there must be some mechanism to reclaim those resources
  - In-memory caching
  - Database storage
  - Log files



# Three Common Violations of Steady State

## Runaway Caching

Meant to speed up response time

When memory low, can cause more GC

∴ Limit cache size, make “elastic”

## Database Sludge

Rising I/O rates

Increasing latency

DBA action ⇒

application errors

- Gaps in collections
- Unresolved references

∴ Build purging into app

## Log File Filling

Most common ticket in Ops

Best case: lose logs

Worst case: errors

∴ Compress, rotate, purge

∴ Limit by size, not time

**How long is your shortest fuse?**

In crunch mode, it's hard to make time for housekeeping functions.

Features always take priority over data purging.

This is a false economy: one-time development cost for ongoing operational costs.

# Remember This

Avoid fiddling

Purge data with application logic

Limit caching

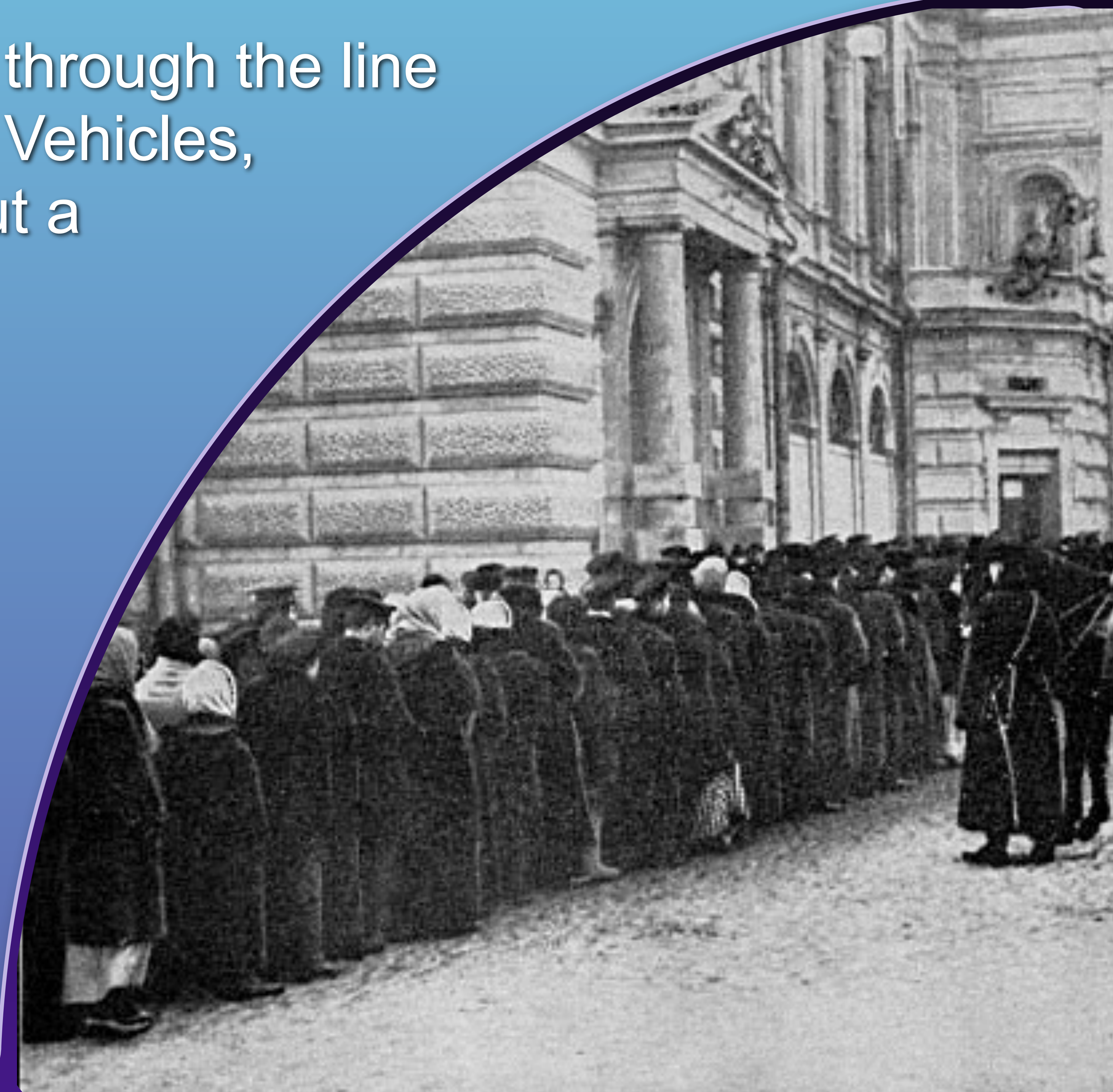
Roll the logs

# Fail Fast

Don't make me wait to receive an error.

Imagine waiting all the way through the line at the Department of Motor Vehicles, just to be sent back to fill out a different form.

Don't burn cycles, occupy threads and keep callers waiting, just to slap them in the face.



# Predicting Failure

- Several ways to determine if a request will fail, before actually processing it:
  - Good old parameter-checking
  - Acquire critical resources early
  - Check on internal state:
    - Circuit Breakers
    - Connection Pools
    - Average latency vs. committed SLAs

# Being a Good Citizen by Failing Fast

- In a multi-tier application or SOA, Fail Fast avoids common antipatterns:
  - Slow Responses
  - Blocked Threads
  - Cascading Failure
- Helps preserve capacity when parts of system have already failed.

# Remember This

- Avoid Slow Responses; Fail Fast
- Reserve resources, verify integration points early
- Validate input; fail fast if not possible to process request

# Decoupling Middleware

Fire and forget.

Async avoids risk.

# Spectrum of Coupling

| Same Time<br>Same Host<br>Same Process | | Same Time<br>Different Host<br>Different Process | | Different Time<br>Different Host<br>Different Process |
|---|---|---|---|---|
| **In-Process Method Calls** | **Interprocess Communication** | **Remote Procedure Calls** | **Message-Oriented Middleware** | **Tuple Spaces** |
| C Functions<br>Java Calls<br>Dynamic Libs | Shared Memory<br>Pipes<br>Semaphores<br>Windows Events | DCE RPC<br>DCOM<br>RMI<br>XML-RPC<br>HTTP | MQ<br>Pub-Sub<br>SMTP<br>SMS | JavaSpaces<br>TSpaces<br>GigaSpaces |

Request-reply: logical simplicity, operational complexity

Message passing: logical complexity, operational simplicity

Tuple Spaces: logical complexity, operational complexity

# Consideration

Changing middleware usually implies a rewrite.

Changing from synchronous to asynchronous semantics implies business rule discussions.

Middleware decisions are often handed down from the ivory tower.

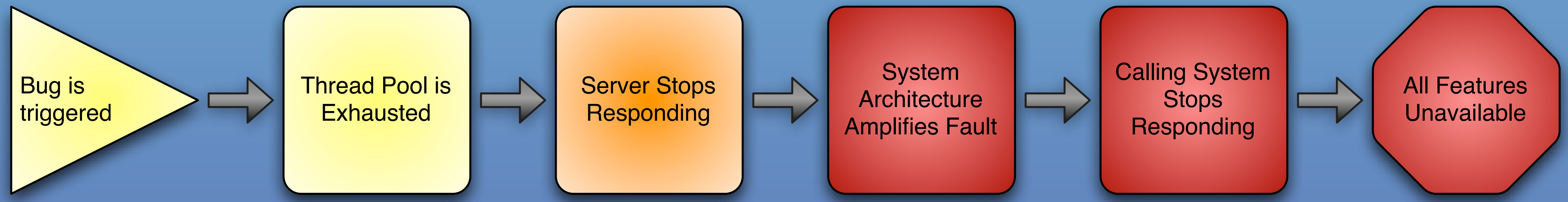
# Remember This

Decide at the last *responsible* moment.

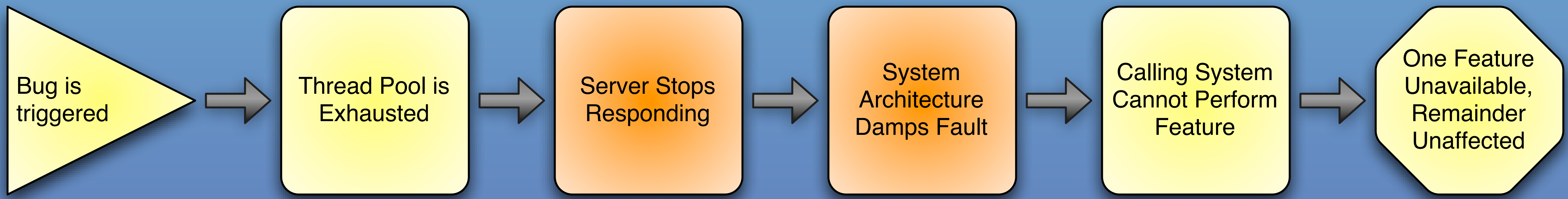
Avoid many failure modes at once by total decoupling.

Learn many architecture styles, choose among them as appropriate.

# Propagation of Problems



# Nullification of Problems



Michael T. Nygard
@mtnygard
mtnygard@cognitect.com

CHICAGO
INTERNATIONAL
SOFTWARE DEVELOPMENT
CONFERENCE 2016

goto; conference


Please

# Remember to rate this session

Thank you!