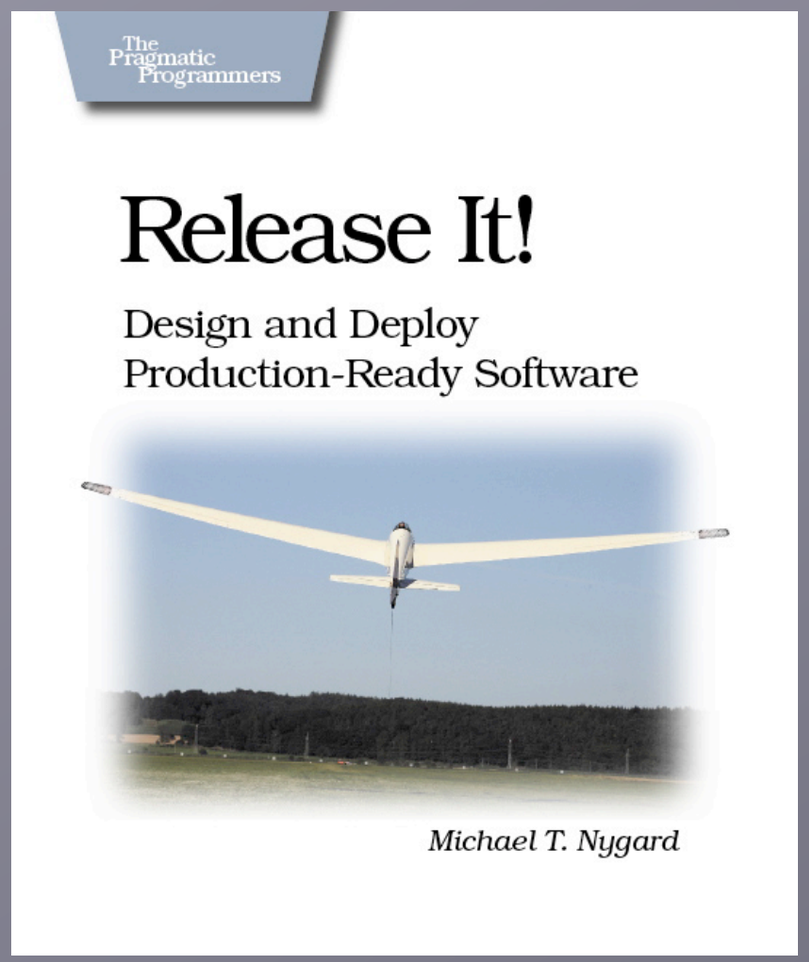
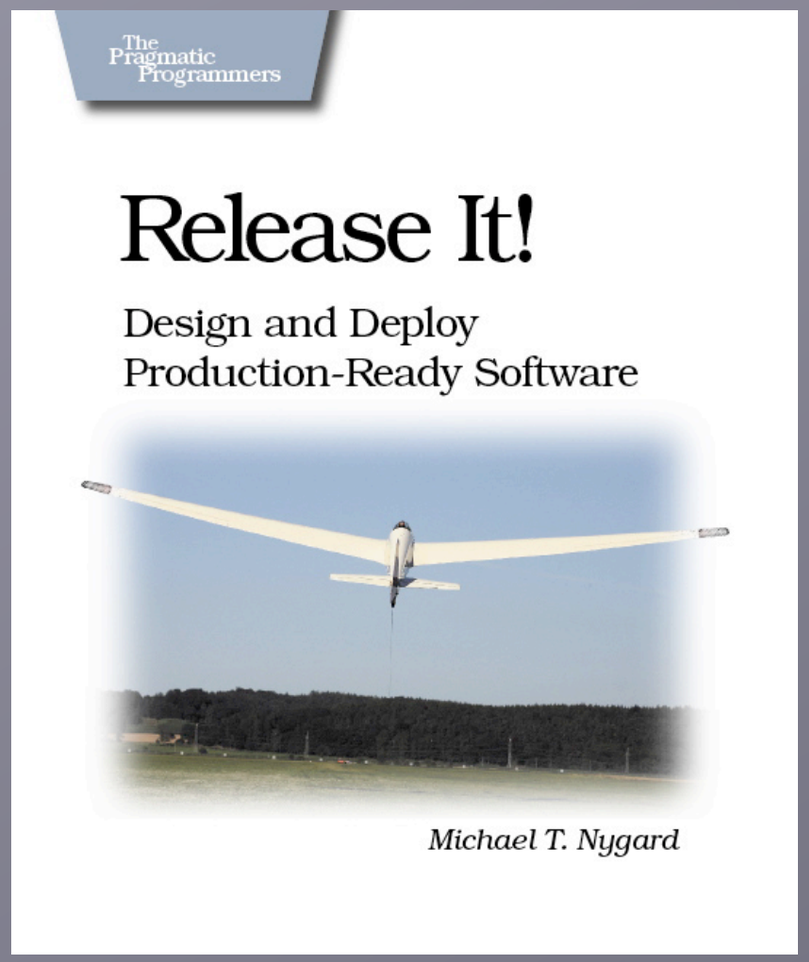
Failure Comes in Flavors
Part I: Antipatterns

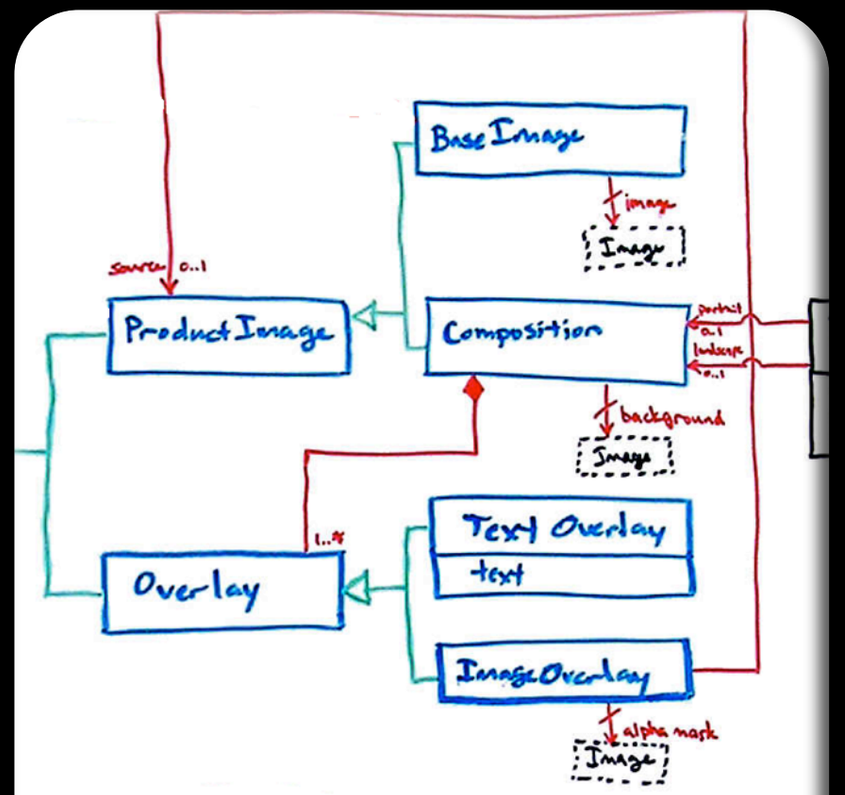


Michael Nygard
mtnygard@gmail.com
www.michaelnygard.com

Failure Comes in Flavors
Part I: Antipatterns



Michael Nygard
mtnygard@gmail.com
www.michaelnygard.com



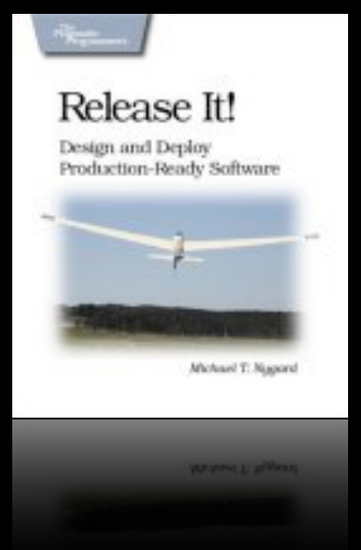
My Rap Sheet

C
C++
Object Pascal
Objective-C
Perl
Java
Smalltalk
Ruby

1989 - 2008: Application Developer
Time served: 19 years

1995: Web Development
Time served: 13 years

2003: IT Operations
Time served: 5 Years



High-Consequence Environments

Users in the thousands and tens of thousands

24 hours a day, 365 days a year

Millions in hardware and software

Millions (or billions) in revenue

Highly interdependent systems

Actively malicious environment

What downtime means for a few of my clients

Manufacturer:
Over 500,000 products and media

Financial services broker:
Average transaction $10,000,000

Top 10 online retailer:
$1,000,000 per hour of downtime

Airline:
Downtime grounds planes and strands travelers

Aiming for the Wrong Target

Many projects get cancelled before release.

Most developers roll off after release 1.0.
(Also known as “The exodus of the consultants.”)

Many developers have little experience with systems in production.

Quality metrics focus attention on functional requirements.

Hence, development efforts usually “aim for QA”.

Aiming for the Wrong Target

Many projects get cancelled before release.

Most developers roll off after release 1.0.
(Also known as “The exodus of the consultants.”)

Many developers have little experience with systems in production.

Quality metrics focus attention on functional requirements.

Hence, development efforts usually “aim for QA”.

Question:

How close is “feature-complete” to “production-ready”?

Points of Leverage

Small decisions at every level can have a huge impact:

- Architecture
- Design
- Implementation
- Build & Deployment
- Administration

Good News

Some large improvements are available with little to no added development cost.

Bad News

Leverage points come early. The cost of choosing poorly comes much, much later.

Assumptions

Users care about the things they do (features), not the software or hardware you run.

Severability: Limit functionality instead of crashing completely.

Resilience: Recover from transient effects automatically.

Recoverability: Allow component-level restarts instead of rebooting the world.

Tolerance: Absorb shocks, but do not transmit them.

Together, these qualities produce stability–the consistent, long-term availability of features.

Stability Under Stress

Stability under stress is resilience to transient problems

- User load
- Back-end outages
- Network slowdowns
- Other “exogenous impulses”

There is no such thing as perfect stability; you are buying time

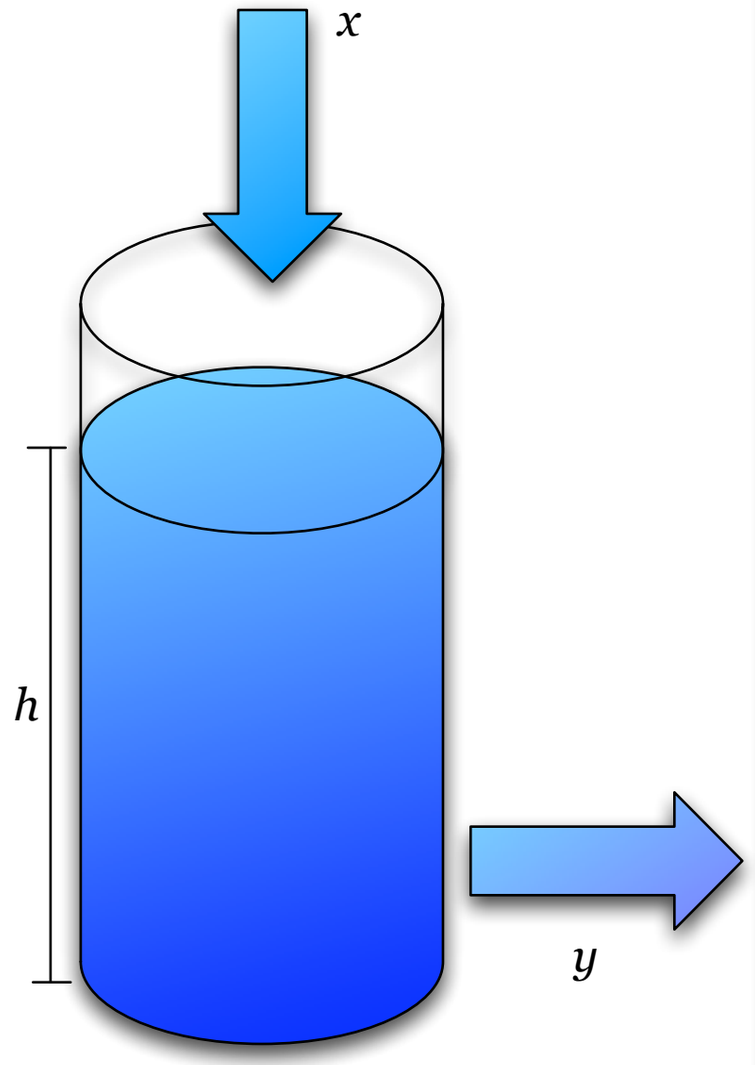
How long is your shortest fuse?

Stability Over Time

How long can a process or server run before it needs to be restarted?

Is data produced and purged at the same rate?

Usually not tested in development or QA. Too many rapid restarts.



The Bitterness of Failure: Stability Antipatterns

Integration Points

Resource Leaks

Chain Reactions

Cascading Failures

Users

Blocked Threads

Attacks of Self-Denial

Scaling Effects

Unbalanced Capacities

Slow Responses

SLA Inversion

Unbounded Result Sets

Integration Points

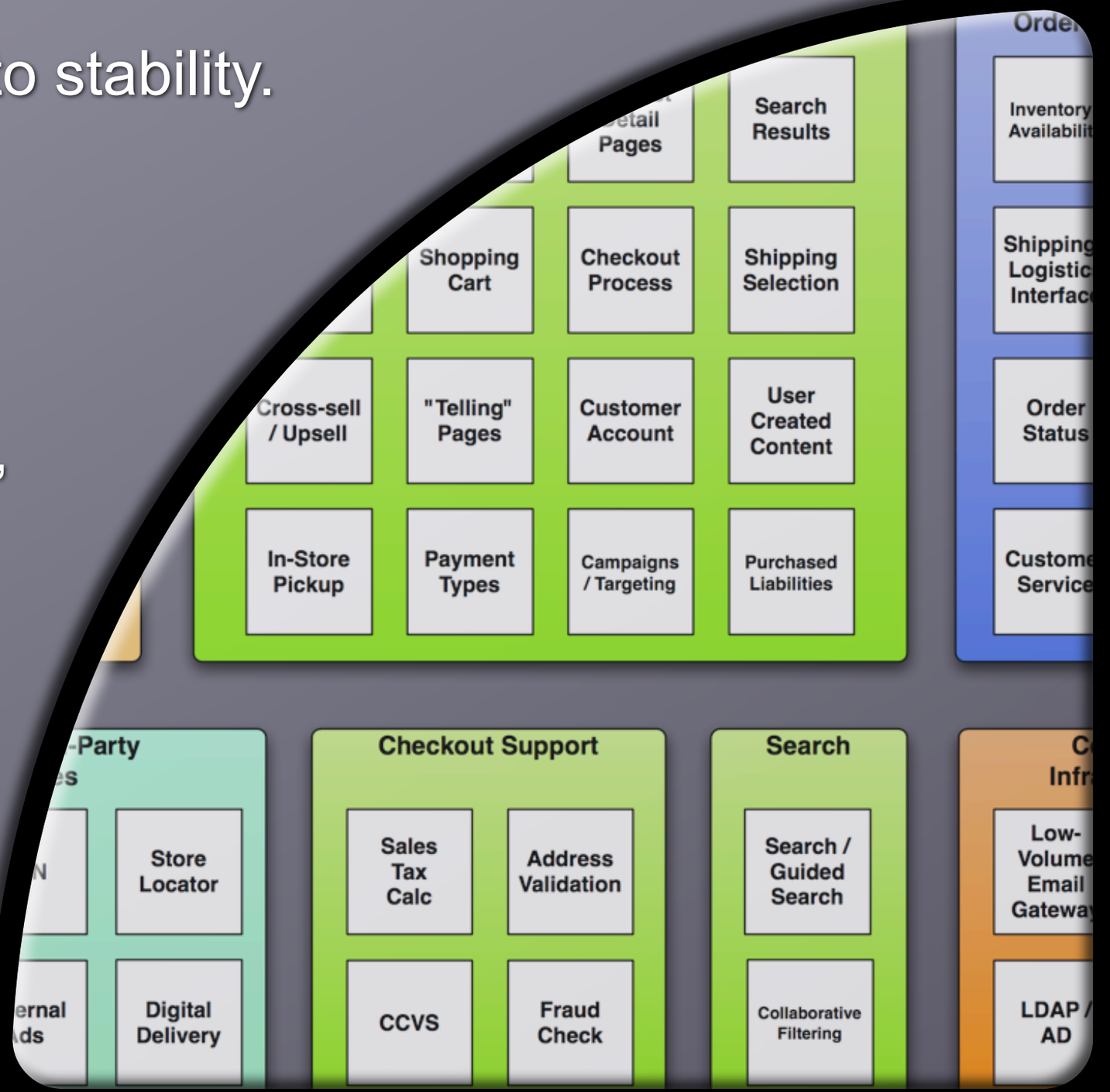
Examine every arrow in the architecture diagram with deep suspicion

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: Database hang

Obvious version: Deadly Embrace

Textbook solution: Uniform transaction ordering

Pragmatic solution: Use a database server that can detect deadlocks and break them.

- Your application logic must be prepared for commit failures!

Example: Database hang

Obvious version: Deadly Embrace

Textbook solution: Uniform transaction ordering

Pragmatic solution: Use a database server that can detect deadlocks and break them.

Your application logic must be prepared for commit failures!

Less obvious: Multiple connection pools, cross-tier deadlock

Thread A holds last connection from pool A; needs to lock rows in table T.

Thread B holds rows locks in T; cannot commit until it gets a connection from pool A.

Simple solution: Use one connection pool

Example: Wicked database hang

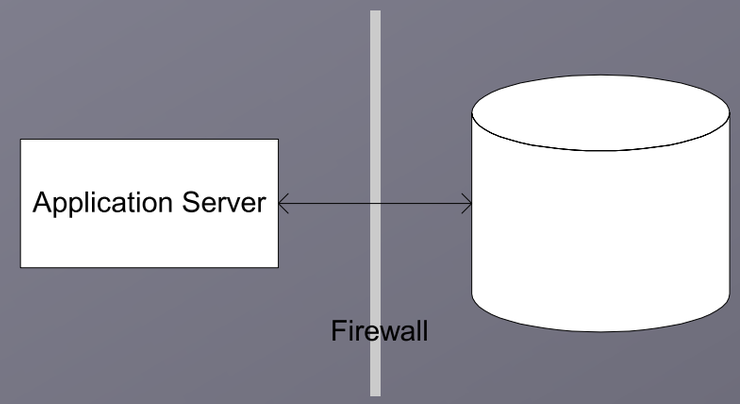
Not at all obvious: Firewall idle connection timeout

“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?

Remember This

Beware this necessary evil.

Beware vendor libraries.

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

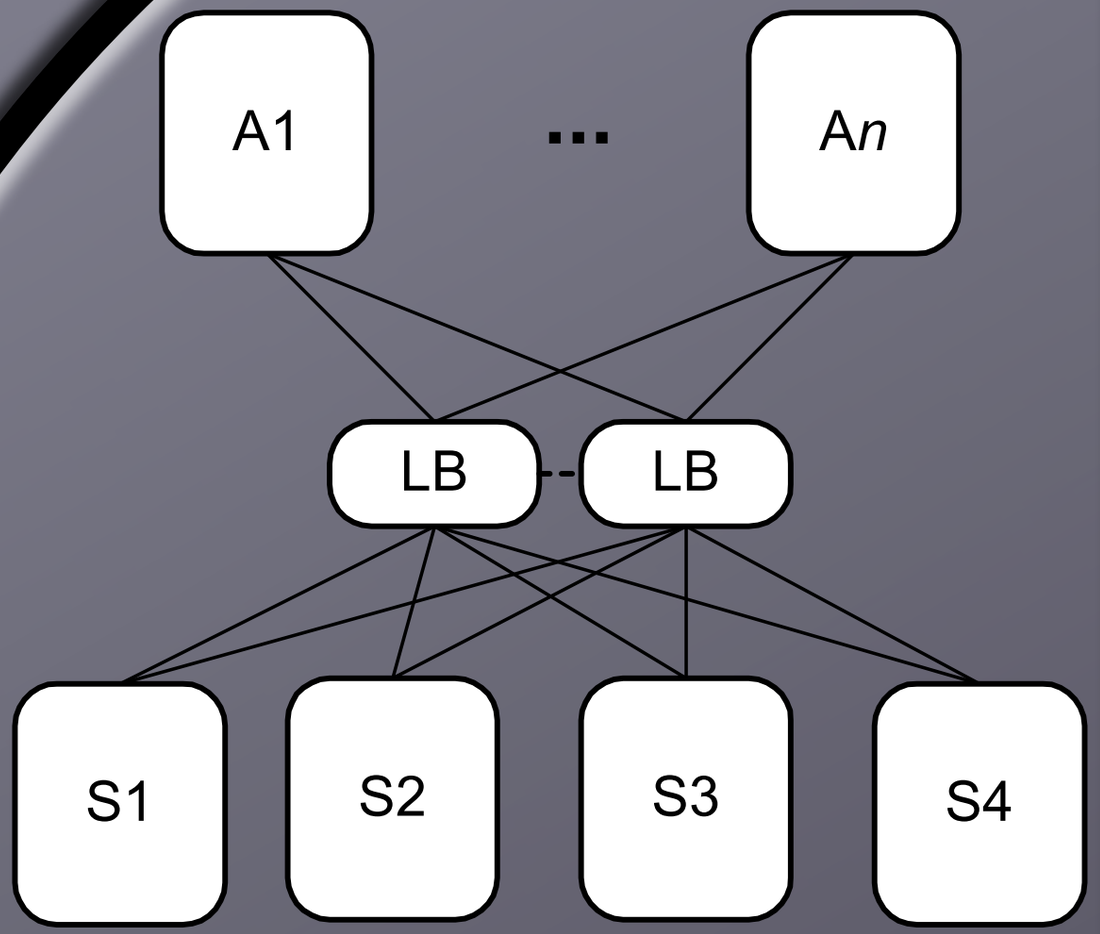
Example:

- 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

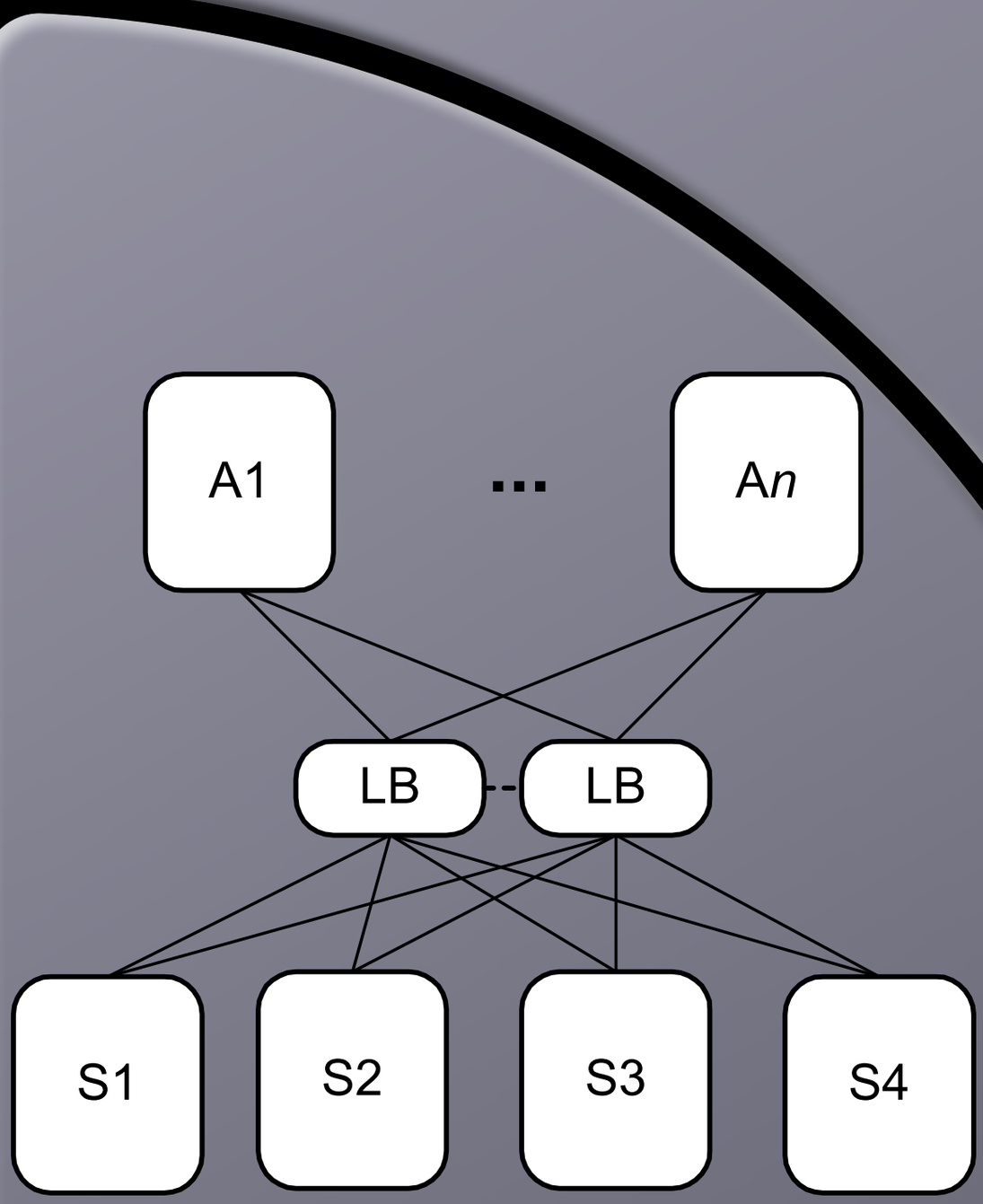
One server down jeopardizes the rest.

Hunt for Resource Leaks.

Defend with “Bulkheads”.

Cascading Failure

Failure in one system causes calling systems to be jeopardized



Example:

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

Failure moves vertically across tiers

Common in enterprise services and SOAs

Remember This

Prevent Cascading Failure to stop cracks from jumping the gap.

Think “Damage Containment”

Scrutinize resource pools, they get exhausted when the lower layer fails.

Defend with “Use Timeouts” and “Circuit Breaker”.

Users

Can’t live with them...

Ways that users cause instability

- Sheer traffic
  - Flash mobs
  - Click-happy
- Malicious users
  - Screen-scrapers
  - Badly configured proxy servers

Two types of “bad” user

- Front-page viewer
  - Application servers are all fragile to sessions
- Buyers
  - High conversion rate is bad for the systems.

Handle Traffic Surges Gracefully

Turn off expensive features when the system is busy.

Divert or throttle users. Preserve a good experience for some when you can't serve all.

Reduce the burden of serving each user. Be especially frugal with memory.

- Hold IDs, not object graphs.
- Hold query parameters, not result sets.

Differentiate people from bots. Don't keep sessions for bots.

Remember This

Minimize the memory you devote to each user.

Malicious users are out there.

But, so are weird random ones.

Users come in clumps: one, a few, or *way* too many.

Blocked Threads

Request handling threads are precious. Protect them.

Most common form of “crash”: all request threads blocked

Very difficult to test for:

- Combinatoric permutation of code pathways.
- Safe code can be extended in unsafe ways.
- Errors are sensitive to timing and difficult to reproduce
- Dev & QA servers never get hit with 10,000 concurrent requests.

Best bet: keep threads isolated. Use well-tested, high-level constructs for cross-thread communication.

- Learn to use java.util.concurrent or System.Threading

Pernicious and Cumulative

Hung request handlers reduce the server's capacity.

Eventually, a restart will be required.

Each hung request handler indicates a frustrated user or waiting caller

The effect is non-linear and accelerating

- Each remaining thread serves 1/N-1 extra requests

Example: Blocking calls

Example: Blocking calls

Example:

In a request-processing method:

```
String key = (String)request.getParameter(PARAM_ITEM_SKU);
Availability avl = globalObjectCache.get(key);
```

Example: Blocking calls

Example:

In a request-processing method:

```
String key = (String)request.getParameter(PARAM_ITEM_SKU);
Availability avl = globalObjectCache.get(key);
```

GlobalObjectCache.get(String id) is synchronized method, with a plugin factory

```
Object obj = items.get(id);
if(obj == null) {
  obj = remoteSystem.lookup(id);
}
…
```

Example: Blocking calls

Example:

In a request-processing method:

```
String key = (String)request.getParameter(PARAM_ITEM_SKU);
Availability avl = globalObjectCache.get(key);
```

GlobalObjectCache.get(String id) is synchronized method, with a plugin factory

```
Object obj = items.get(id);
if(obj == null) {
  obj = remoteSystem.lookup(id);
}
…
```

Remote system stopped responding due to “Unbalanced Capacities”

Example: Blocking calls

Example:

In a request-processing method:

```
String key = (String)request.getParameter(PARAM_ITEM_SKU);
Availability avl = globalObjectCache.get(key);
```

GlobalObjectCache.get(String id) is synchronized method, with a plugin factory

```
Object obj = items.get(id);
if(obj == null) {
  obj = remoteSystem.lookup(id);
}
…
```

Remote system stopped responding due to “Unbalanced Capacities”

Threads piled up like cars on a foggy freeway.

Remember This

Scrutinize resource pools.

Never wait forever.

Use proven constructs.

Beware the code you cannot see.

Defend with “Use Timeouts”.

Attacks of Self-Denial

Good marketing can kill your system at any time.

Ever heard this one?

A retailer offered a great promotion to a “select group of customers”.

Approximately a bazillion times the expected customers show up for the offer.

The retailer gets crushed, disappointing the avaricious and legitimate.

It’s a self-induced Slashdot effect.

Attacks of Self-Denial

Good marketing can kill your system at any time.

Ever heard this one?

A retailer offered a great promotion to a “select group of customers”.

Approximately a bazillion times the expected customers show up for the offer.

The retailer gets crushed, disappointing the avaricious and legitimate.

It’s a self-induced Slashdot effect.

Victoria’s Secret: Online Fashion Show

BestBuy: XBox 360 Preorder

Amazon: XBox 360 Discount

Anything on FatWallet.com

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

One email I saw went out with a deep link that bypassed Akamai. Worse, it encoded a specific server and included a session ID.

Another time, an email went out with a promo code. It could be used an unlimited number of times.

Once a vulnerability is found, it will be flooded within seconds.

Remember This

Keep lines of communication open

Support the marketers. If you don't, they'll invent their way around you, and might jeopardize the systems.

Protect shared resources

Expect instantaneous distribution of exploits

Scaling Effects

Understand which end of the lever you are sitting on.

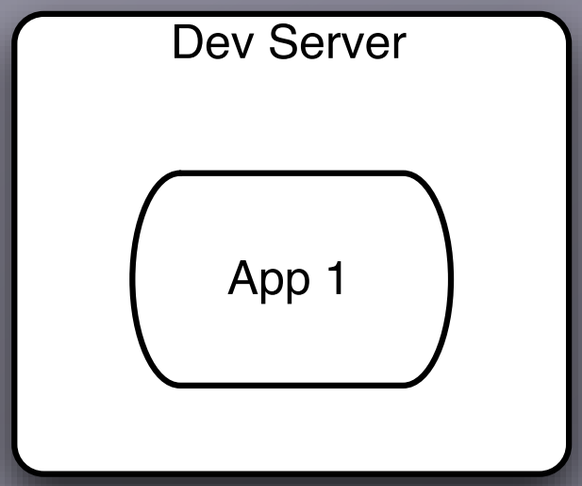
Ratios in dev and QA tend to be 1:1

- Web server to app server
- Front end to back end

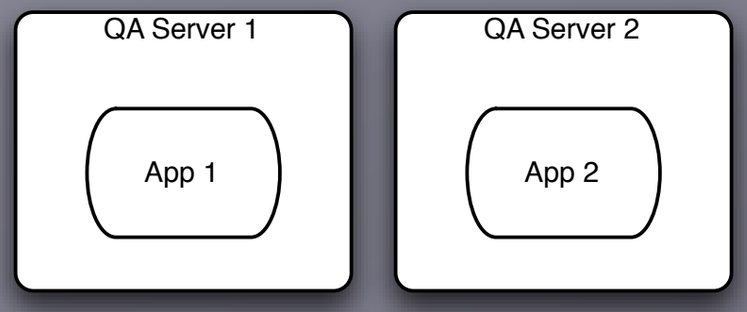
They differ wildly in production, so designs and architectures may not be appropriate

Example: Point to Point Cache Invalidation

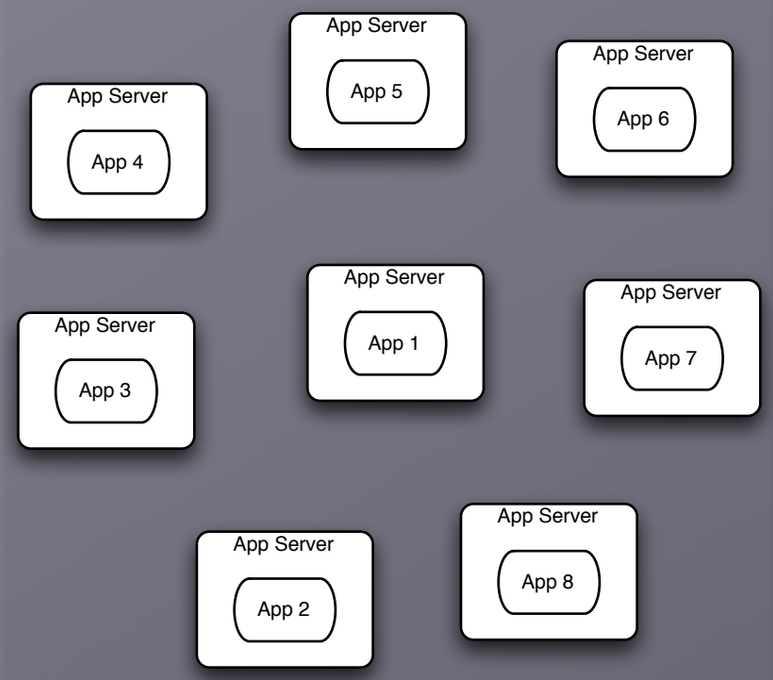
Development



QA

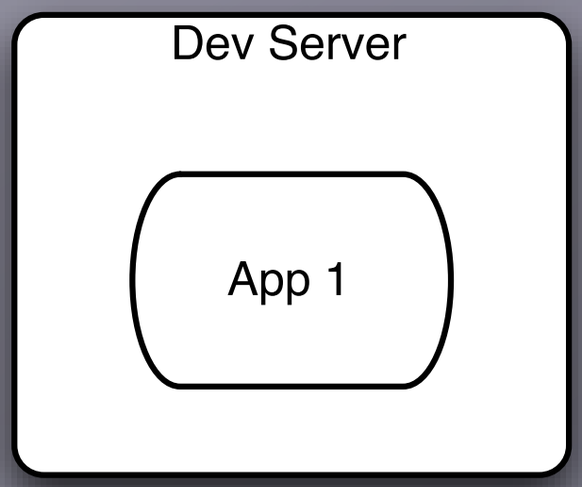


Production

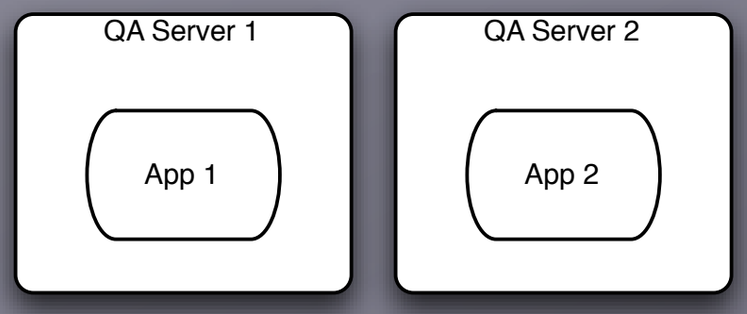


Example: Point to Point Cache Invalidation

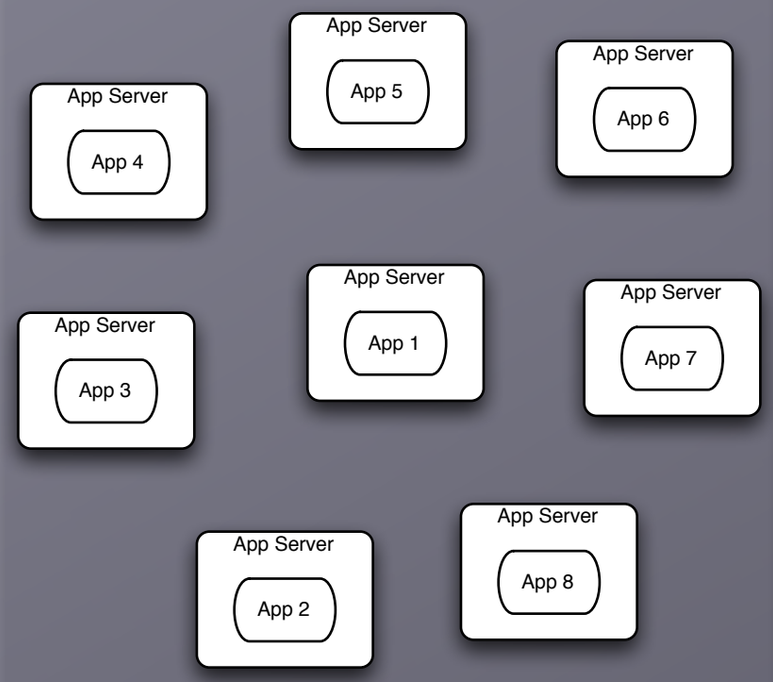
Development



QA



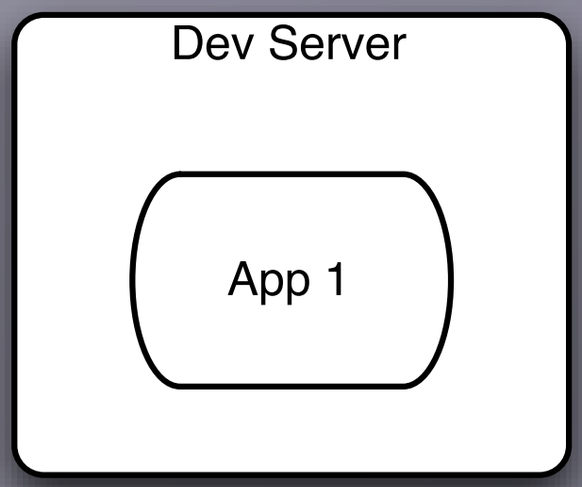
Production



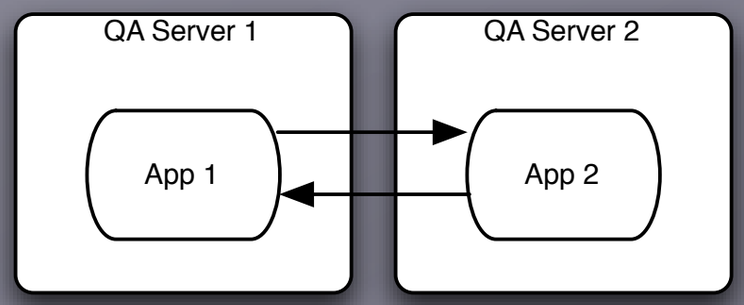
1 server
1 local call
No TCP connections

Example: Point to Point Cache Invalidation

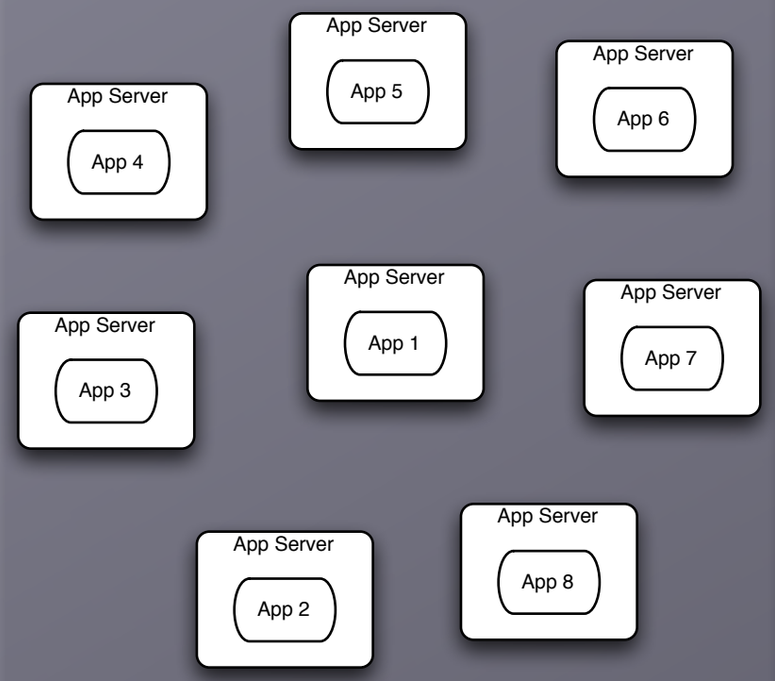
Development



QA



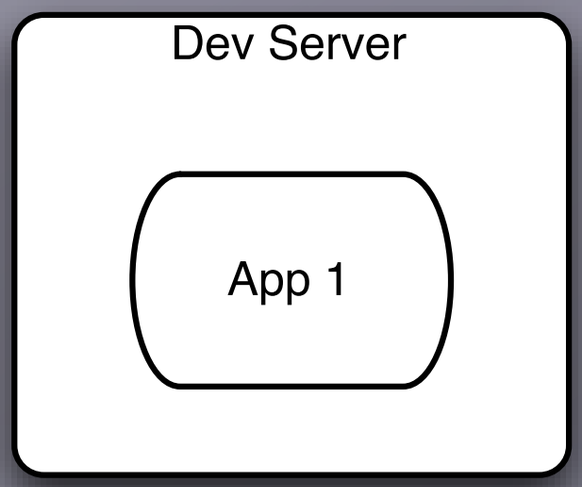
Production



1 server
1 local call
No TCP connections

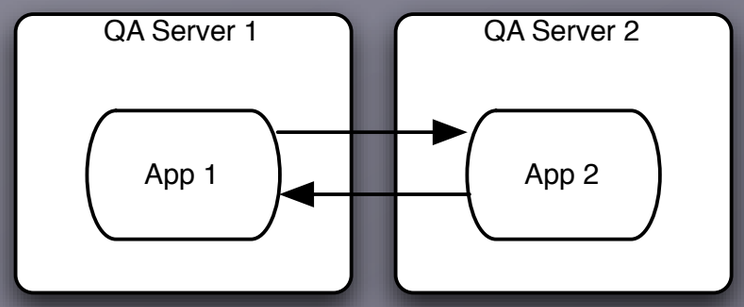
Example: Point to Point Cache Invalidation

Development



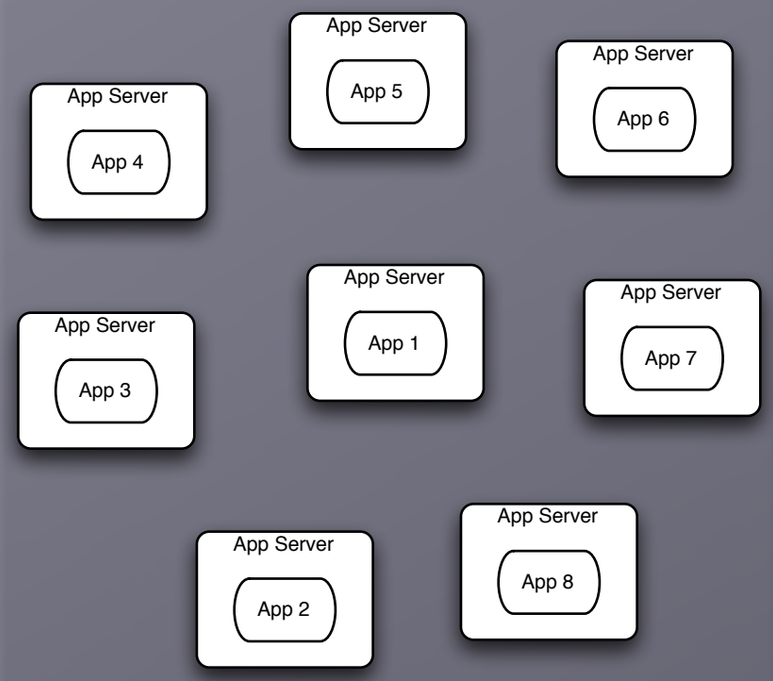
1 server
1 local call
No TCP connections

QA



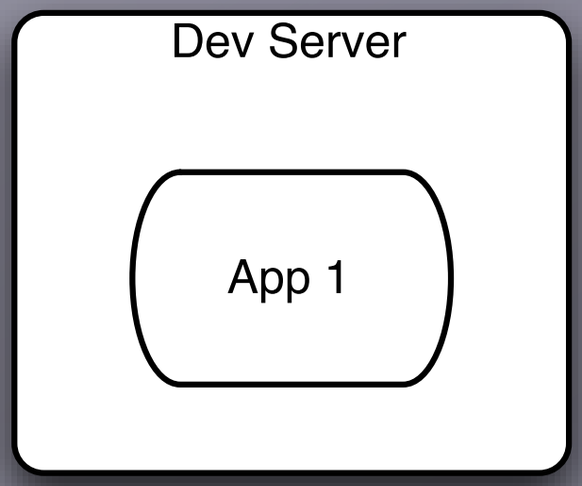
2 servers
1 local call
1 TCP connection

Production



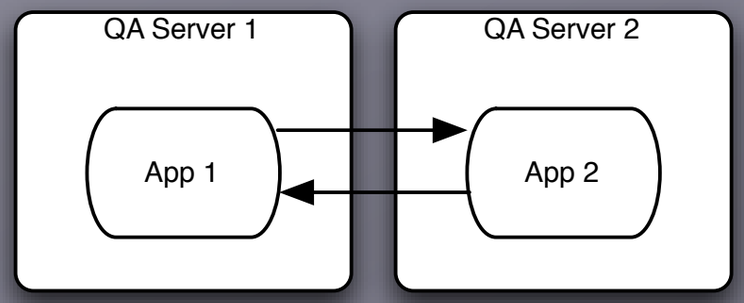
Example: Point to Point Cache Invalidation

Development



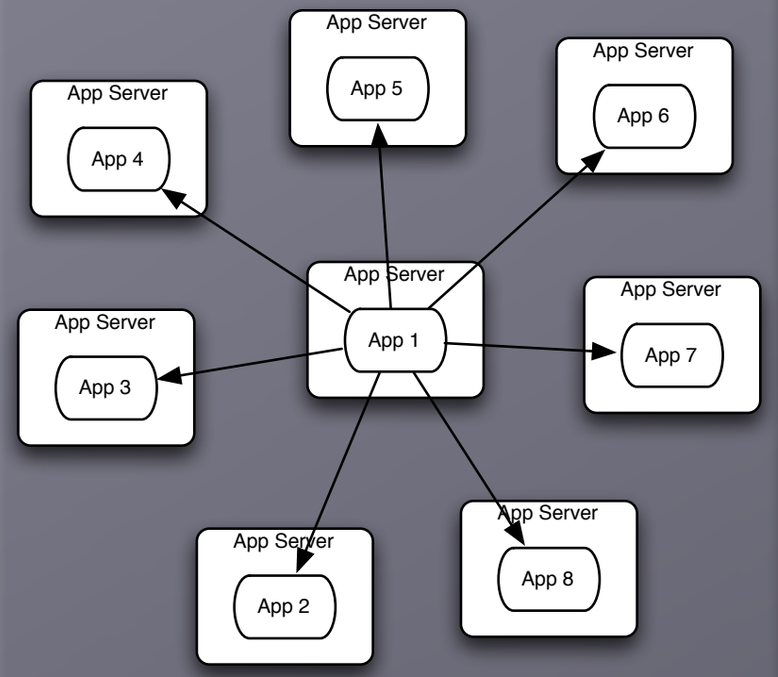
1 server
1 local call
No TCP connections

QA



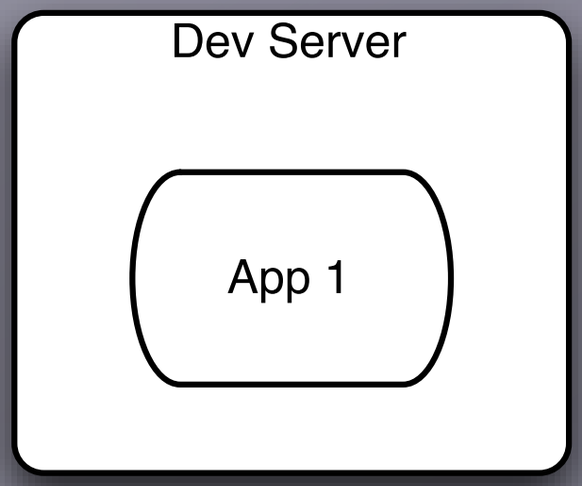
2 servers
1 local call
1 TCP connection

Production



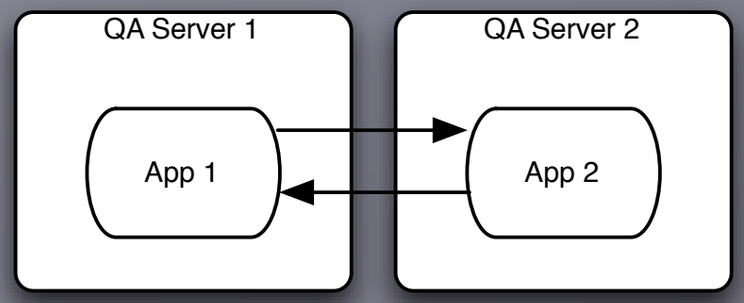
Example: Point to Point Cache Invalidation

Development



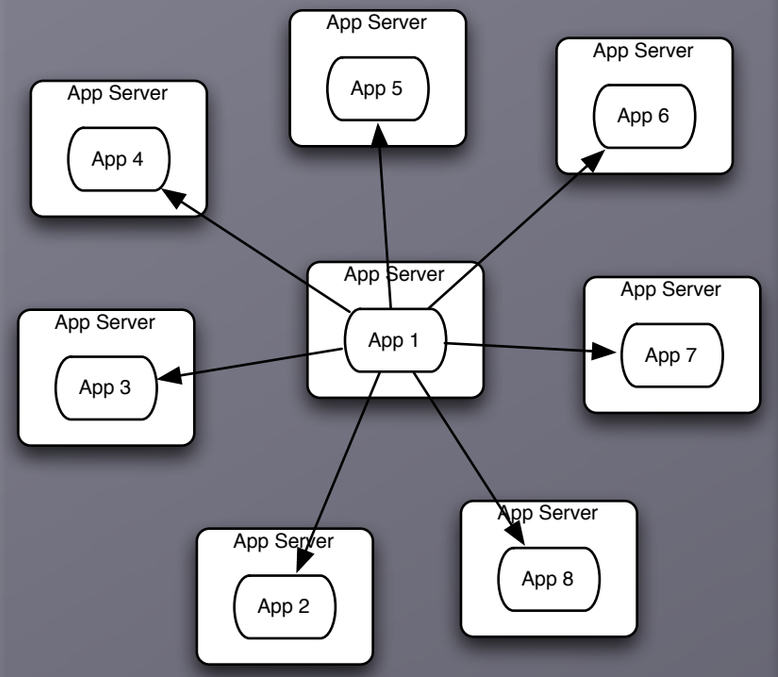
1 server
1 local call
No TCP connections

QA



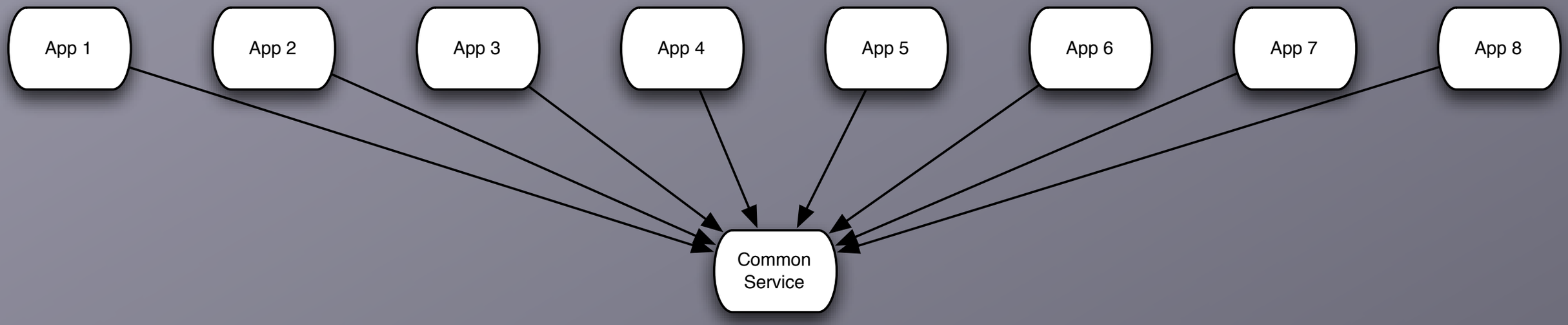
2 servers
1 local call
1 TCP connection

Production



8 servers
1 local call
7 TCP connection

Example: Shared Resources



Shared resources commonly appear as lock managers, load managers, query distributors, cluster managers, and message gateways. They're all vulnerable to scaling effects.

Remember This

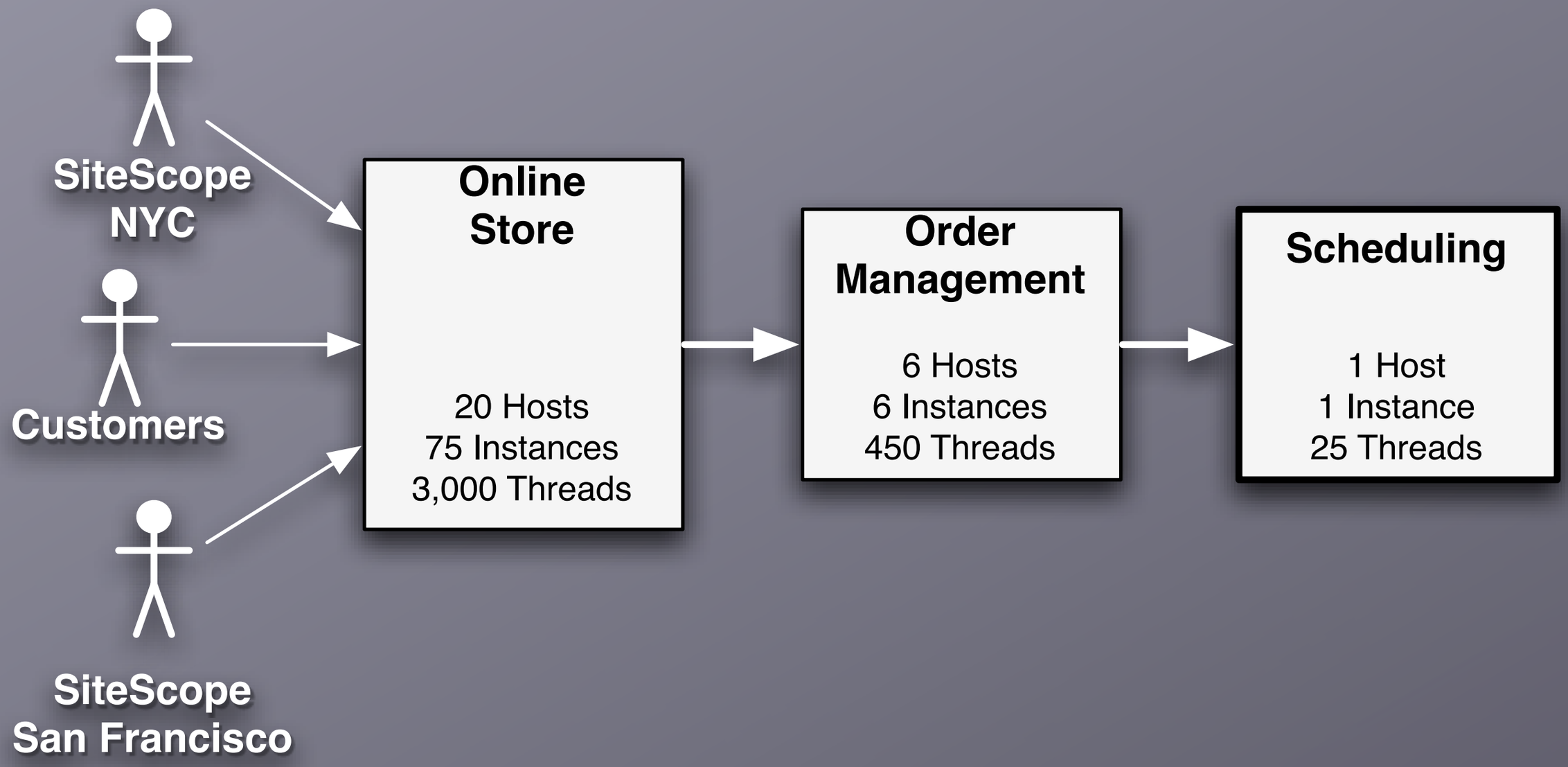
Examine production versus QA environments to spot scaling effects.

Watch out for point-to-point communications. It rarely belongs in production.

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.

It happens because

- All dev systems are one server
- Almost all QA environments are two servers
- Production environments may be 10:1 or 100:1

May be induced by changes in traffic or behavior 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

Look at the latency:

- TCP connection refused comes back in ~10 ms
- TCP packets not acknowledged, sender retransmits for 1 – 10 min

Causes of 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

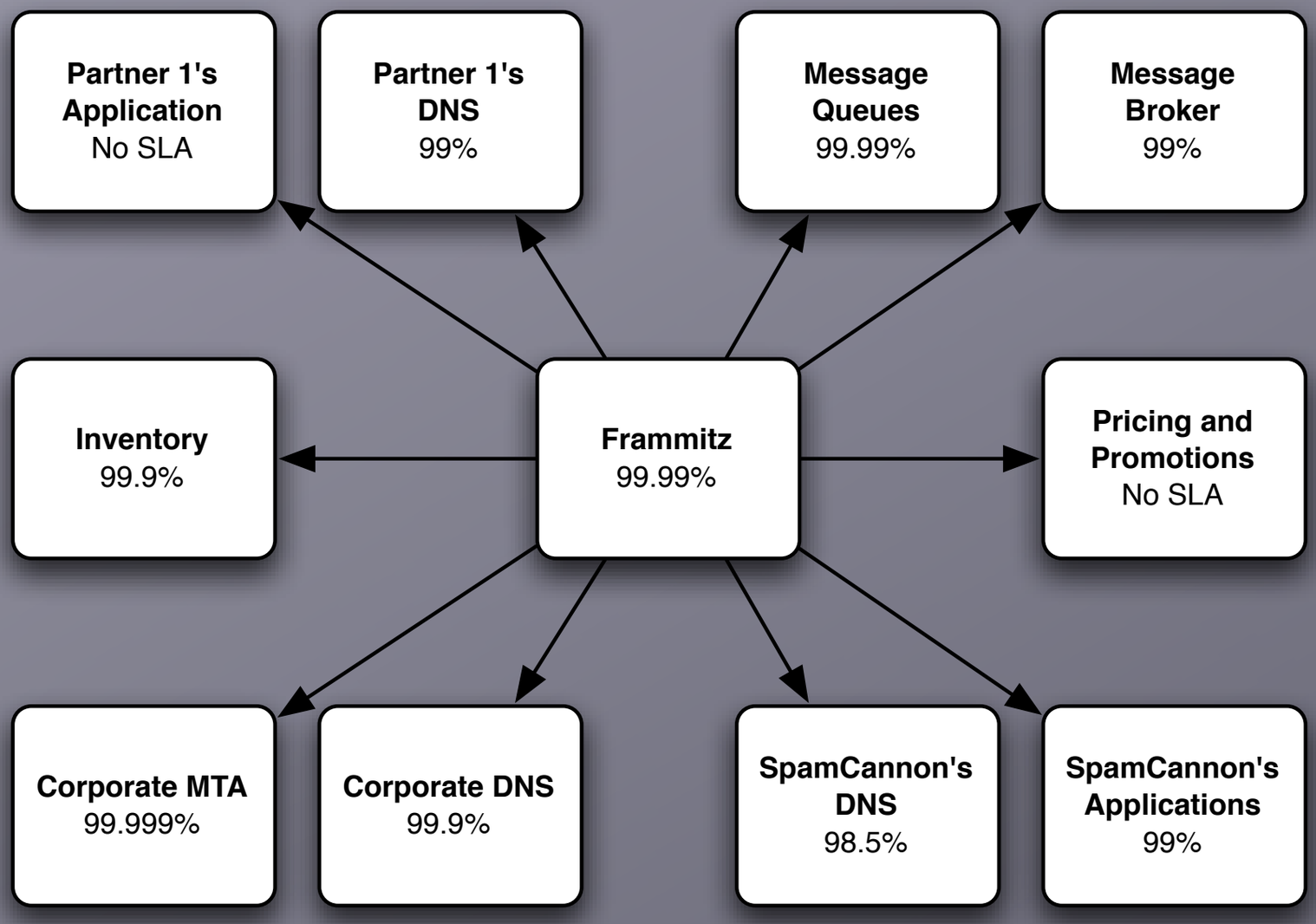
For websites, slow responses invite more traffic as the users pound “reload”

Don’t send a slow response; fail fast

Hunt for memory leaks or resource contention

SLA Inversion

Surviving by luck alone.



What SLA can Frammitz *really* guarantee?

Absent other protections, the best SLA you can offer is the *worst* SLA provided by your dependencies.

The dreaded SPOF is a special case of SLA Inversion.

Do your web servers have to ask DNS to find the application server's IP address?

Remember This

Don’t make empty promises. Be sure you can deliver the SLA you commit to.

Examine every dependency. Verify that *they* can deliver on their promises.

Decouple your SLAs from your dependencies’.

Measure availability by feature, not by server.

Be wary of “enterprise” services such as DNS, SMTP, and LDAP.

Unbounded Result Sets

Limited resources, unlimited data volume

Development and testing is done with small data sets

Test databases get reloaded frequently

Queries that perform acceptably in development and test bonk badly with production data volume.

- Bad access patterns can make them very slow
- Too many results can use up all your server's RAM or take too long to process
- You never know when somebody else will mess with your data

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.

For example:

- Application server using database table to pass message between servers.
- Normal volume 10 – 20 events at a time.
- Time-based trigger on every user generated 10,000,000+ events at midnight.
- Each server trying to receive **all** events at startup.
- Out of memory errors at startup.

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

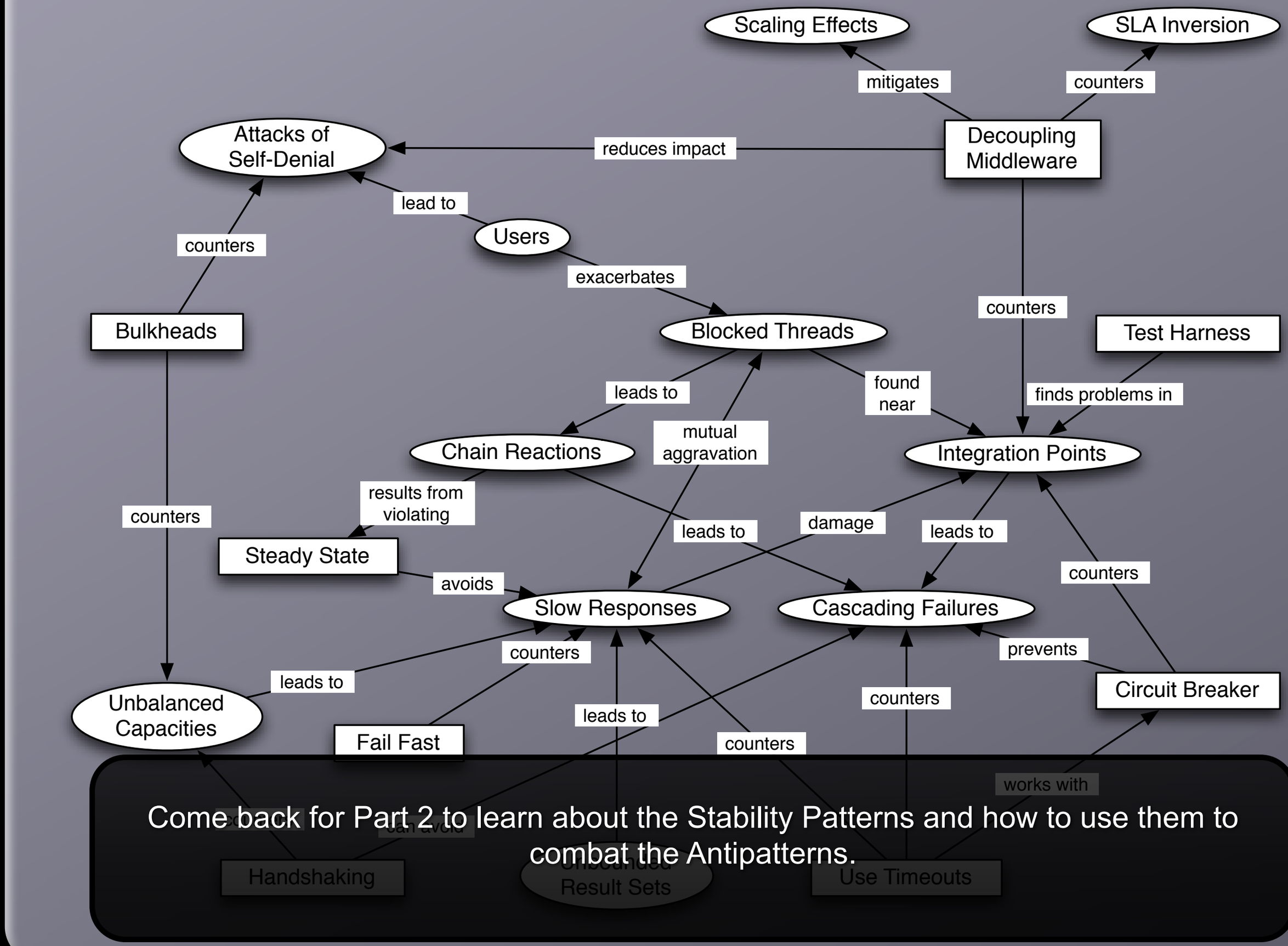
Test with realistic data volumes

- Scrubbed production data is the best.
- Generated data also works.

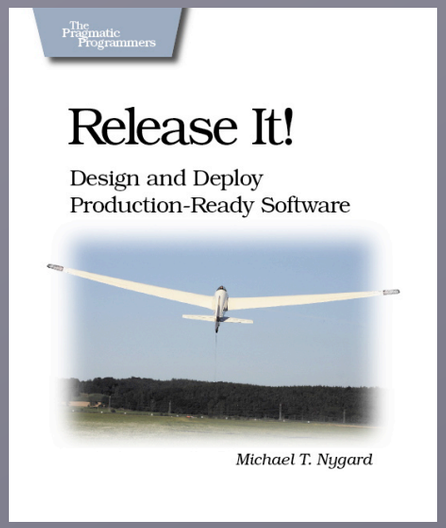
Don’t rely on the data producers. Their behavior can change overnight.

Put limits in your application-level protocols:

- WS, RMI, DCOM, XML-RPC, etc.


Scaling Effects
SLA Inversion
mitigates
counters
Attacks of Self-Denial
reduces impact
Decoupling Middleware
lead to
Users
counters
exacerbates
counters
Bulkheads
Blocked Threads
Test Harness
leads to
found near
finds problems in
mutual aggravation
Chain Reactions
Integration Points
results from violating
counters
leads to
damage
leads to
Steady State
counters
avoids
Slow Responses
Cascading Failures
counters
prevents
leads to
Circuit Breaker
Unbalanced Capacities
counters
leads to
Fail Fast
counters
works with
Handshaking
Unbounded Result Sets
Use Timeouts
Come back for Part 2 to learn about the Stability Patterns and how to use them to combat the Antipatterns.

Thank You



Michael Nygard
mtnygard@gmail.com
www.michaelnygard.com