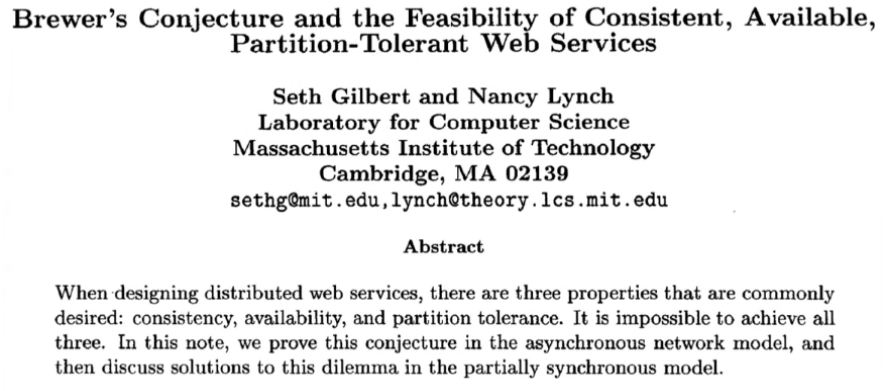
At AWS, we obsess over operational excellence. We have a deep understanding of system availability, informed by over a decade of experience operating the cloud and our roots of operating Amazon.com for nearly a quarter-century. One thing we've learned is that failures come in many forms, some expected, and some unexpected. It's vital to build from the ground up and embrace failure. A core consideration is how to minimize the "blast radius" of any failures. In this talk, we discuss a range of blast radius reduction design techniques that we employ, including cell-based architecture, shuffle-sharding, availability zone independence, and region isolation. We also discuss how blast radius reduction infuses our operational practices.

This is the speed of light limits distributed systems



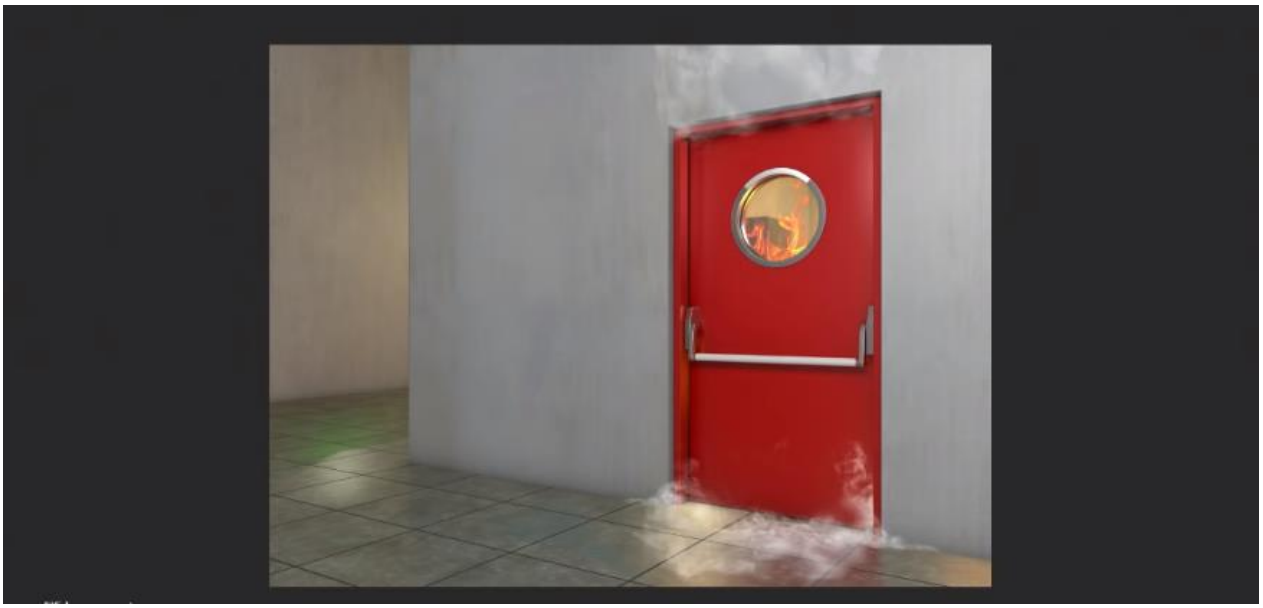
Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services

Seth Gilbert and Nancy Lynch
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
sethg@mit.edu,lynch@theory.lcs.mit.edu

Abstract

When designing distributed web services, there are three properties that are commonly desired: consistency, availability, and partition tolerance. It is impossible to achieve all three. In this note, we prove this conjecture in the asynchronous network model, and then discuss solutions to this dilemma in the partially synchronous model.

The CAP theorem also limits distributed systems



We will see the techniques AWS uses to limit blasts in their systems when Murphy's law doe strike

"As a thought exercise, how could you cut the blast radius for a similar event in half?"

Correction of Errors Template

Agenda

Region isolation

Availability Zone independence

Cell-based architecture

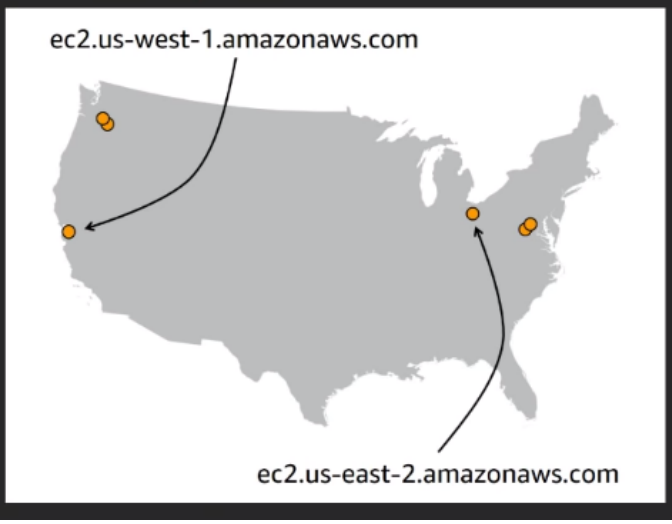
Shuffle sharding

Operational practices

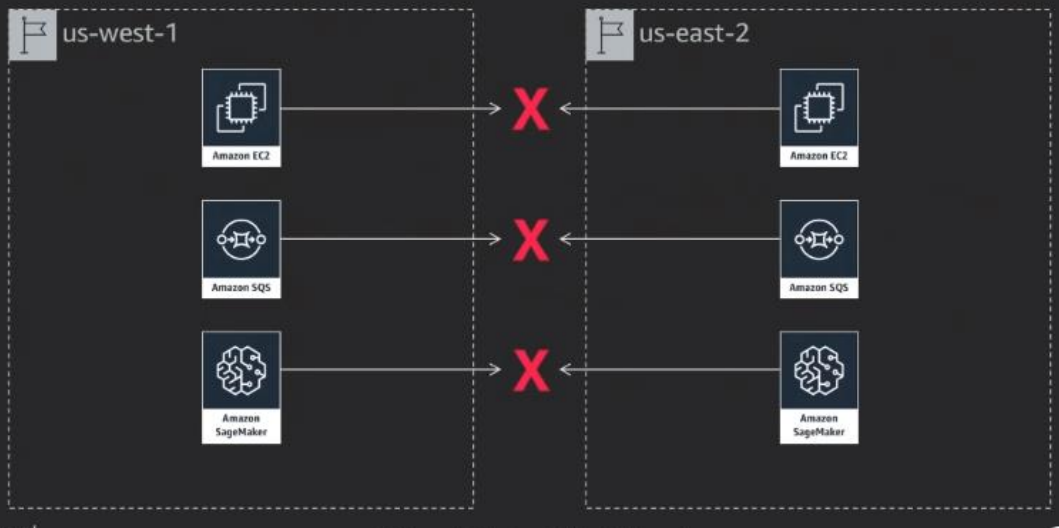
Region isolation

AWS regions

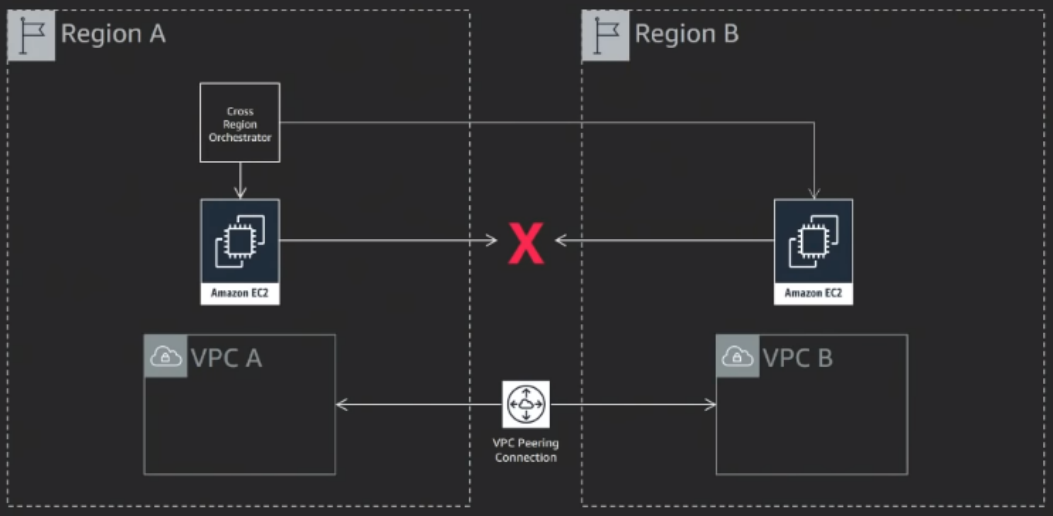
Regional service endpoints



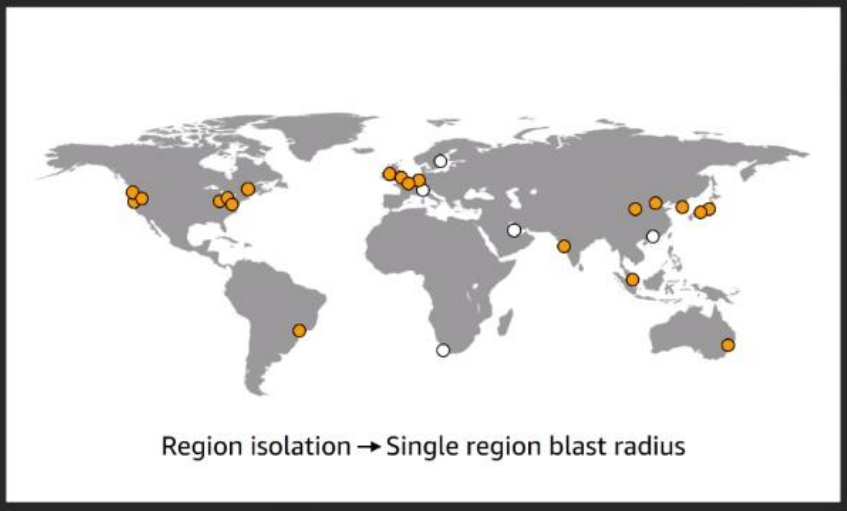
Shared nothing architecture



Inter-Region Virtual Private Cloud (VPC) Peering



AWS regions

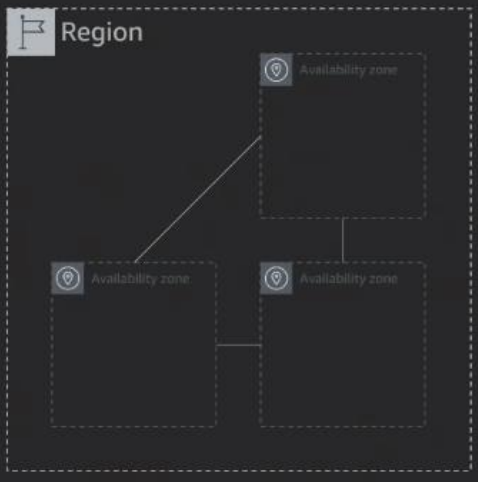


Region isolation → Single region blast radius

Availability Zone independence

Availability Zones

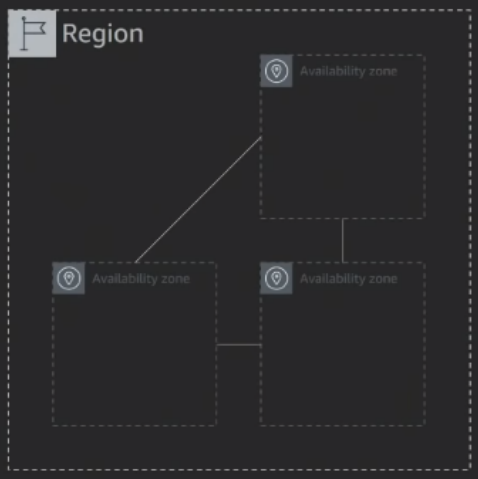
- Geographically spread to avoid correlated failure
- Geographically close for low latency



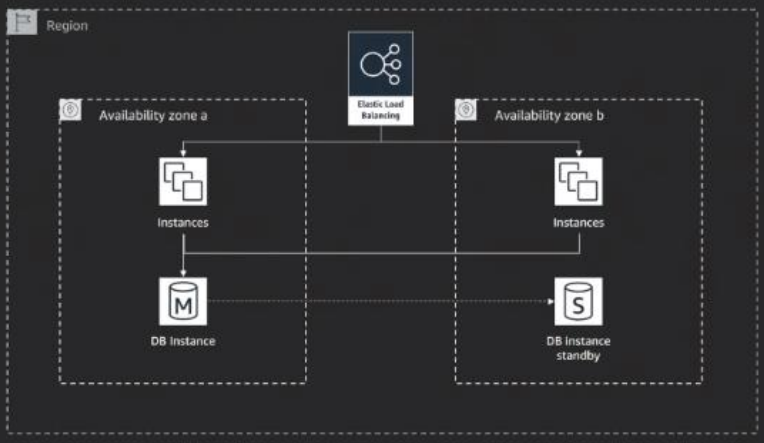
Availability Zones

- *n* data centers per AZ (1 or more)
- *n* AZs per region (typically 3+)

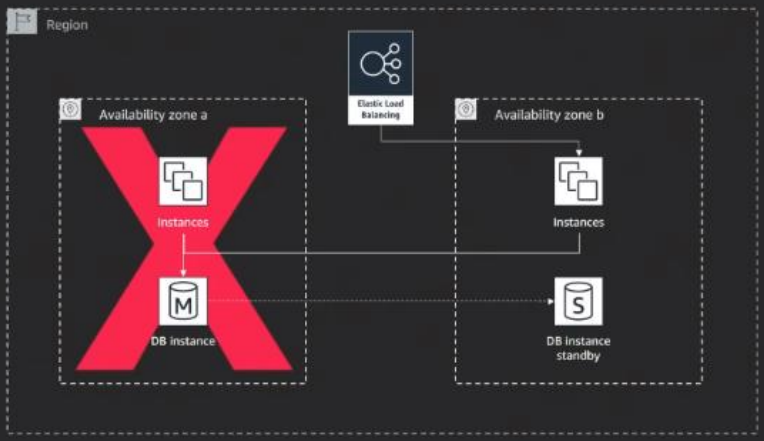
- 57 AZs across 19 regions globally
- Coming soon: 15 additional AZs across 5 new regions



Multi-AZ architecture



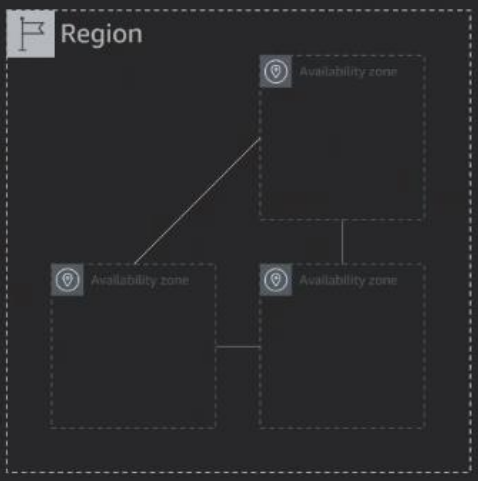
Multi-AZ architecture



Multi-AZ architecture

- Enables fault-tolerant applications
- AWS regional services designed to withstand AZ failures
- Leveraged in S3's design for 99.999999999% durability

Multi-AZ → Zero blast radius!



Zonal services

- Amazon EC2 instances
- Amazon EBS volumes
- Amazon EMR clusters
- AWS CloudHSM instances
- NAT gateways
- etc.

- Zone-specific resources
- Zone-local control planes
- Availability Zone independence

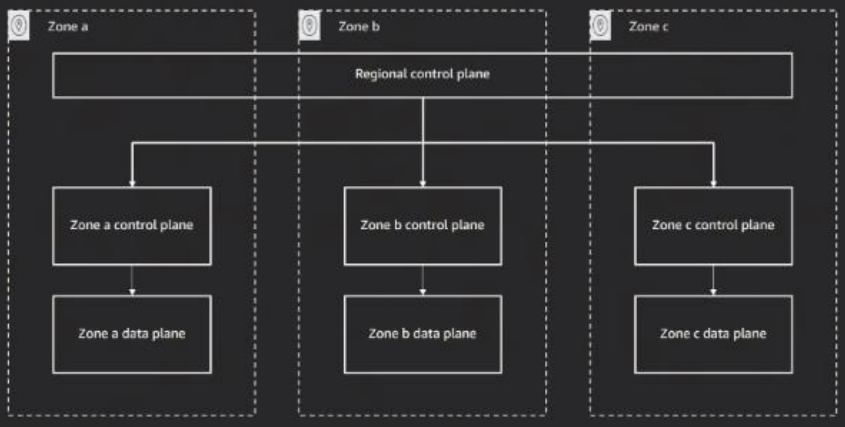
Control planes and data planes

- Control plane: administration of resources
- Data plane: usage of resources

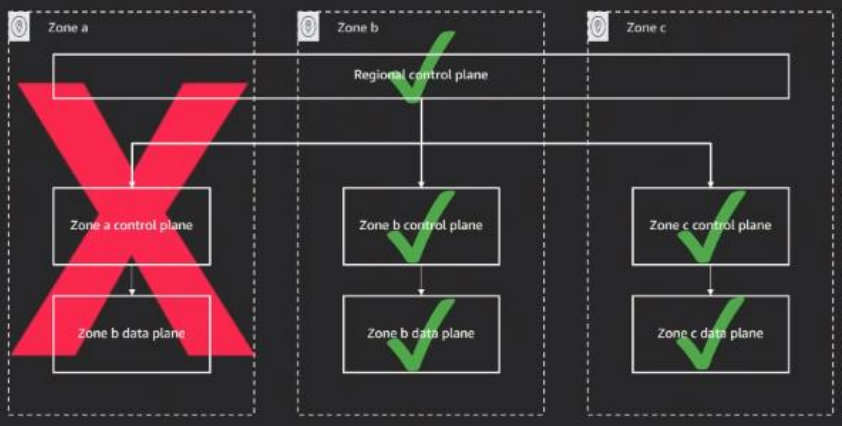
Service	Control plane	Data plane
Amazon DynamoDB	DescribeTable API	Query API
Amazon EC2	RunInstances API	A running EC2 instance
AWS Lambda	CreateFunction API	Invoke API

- Separating control plane from data plane reduces blast radius

Availability Zone independence



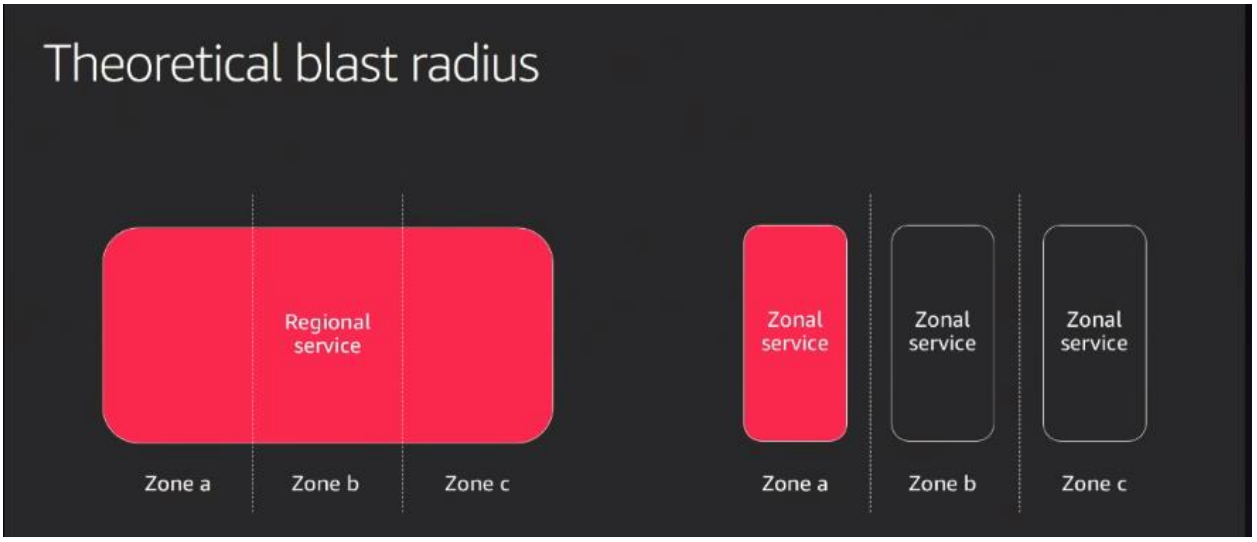
Availability Zone independence

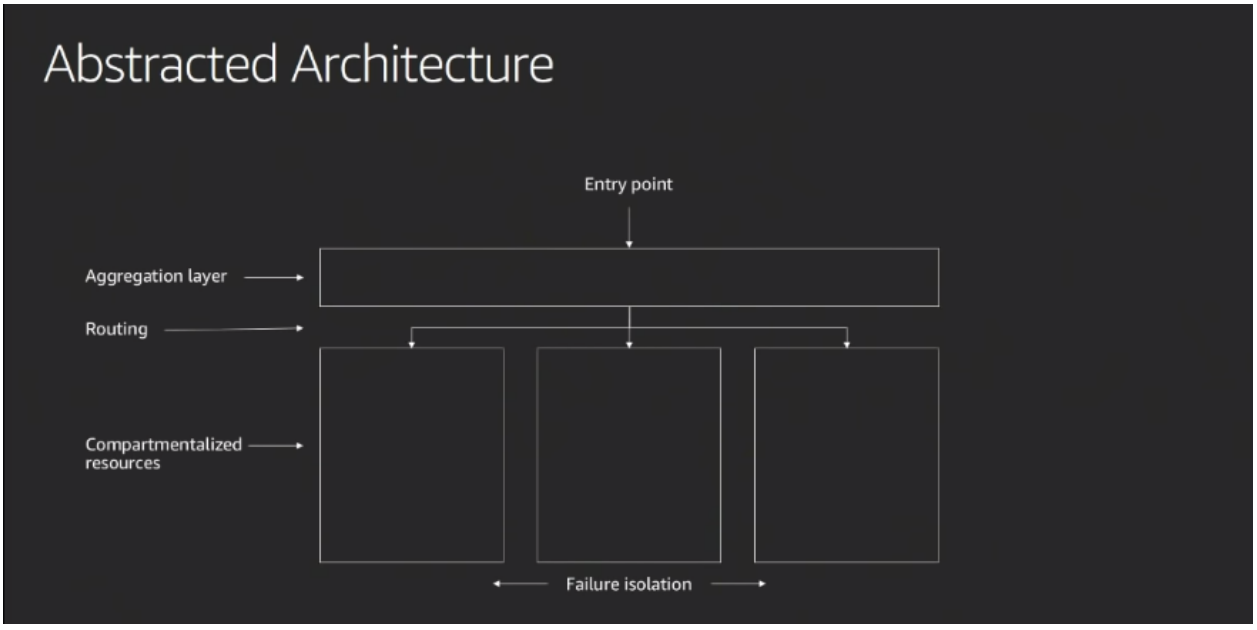


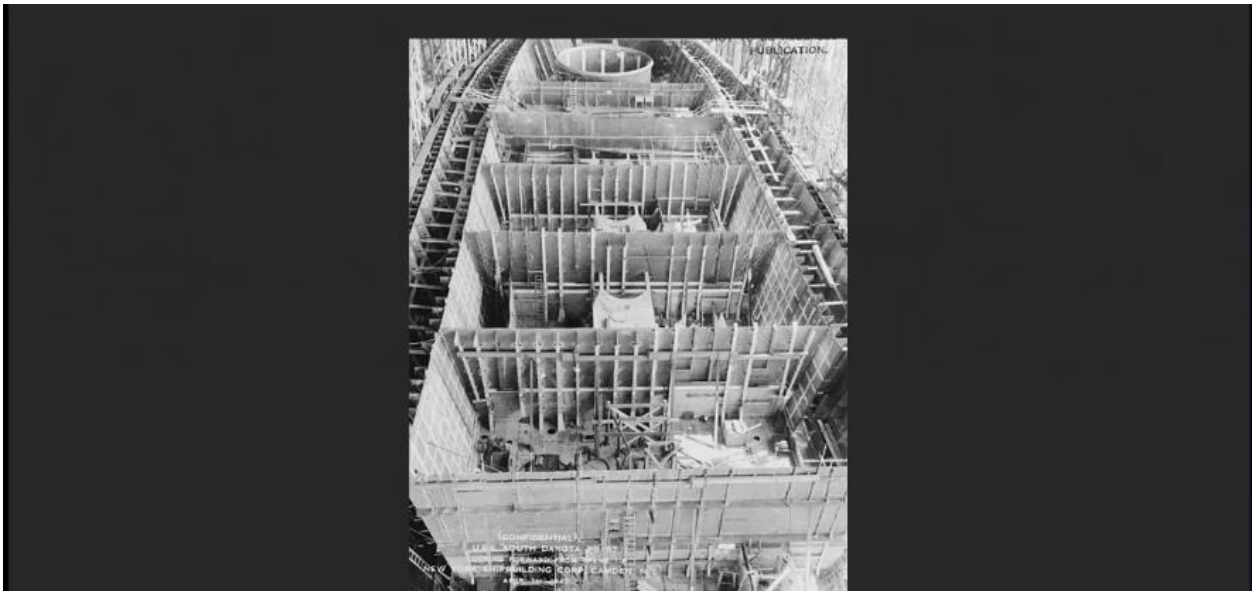
Blast Radius with Availability Zones

Availability zone failure

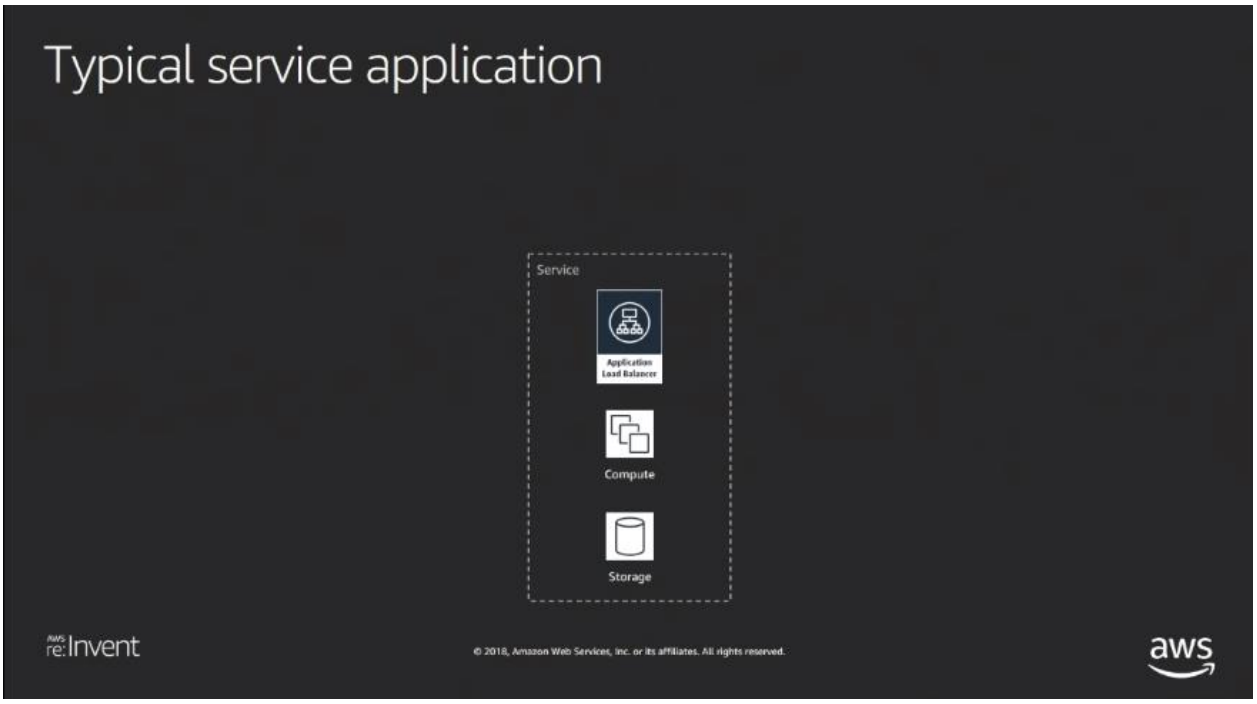
Theoretical blast radius

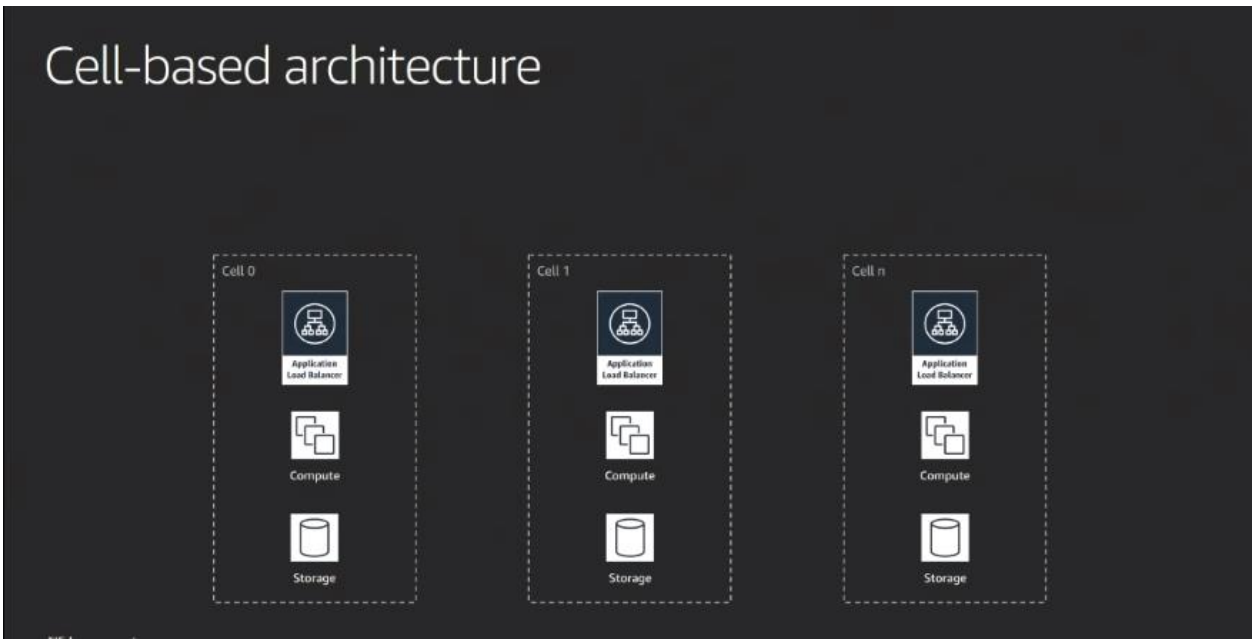




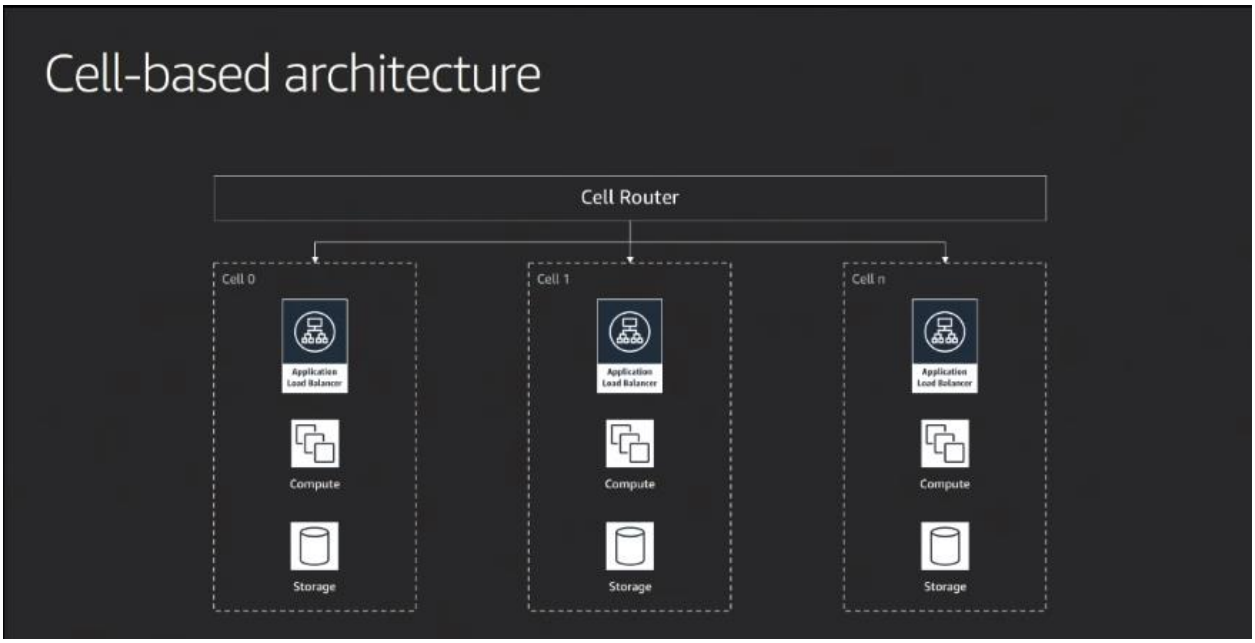


We have taken this idea and applied it to our regional based services as a Cell-based architecture approach

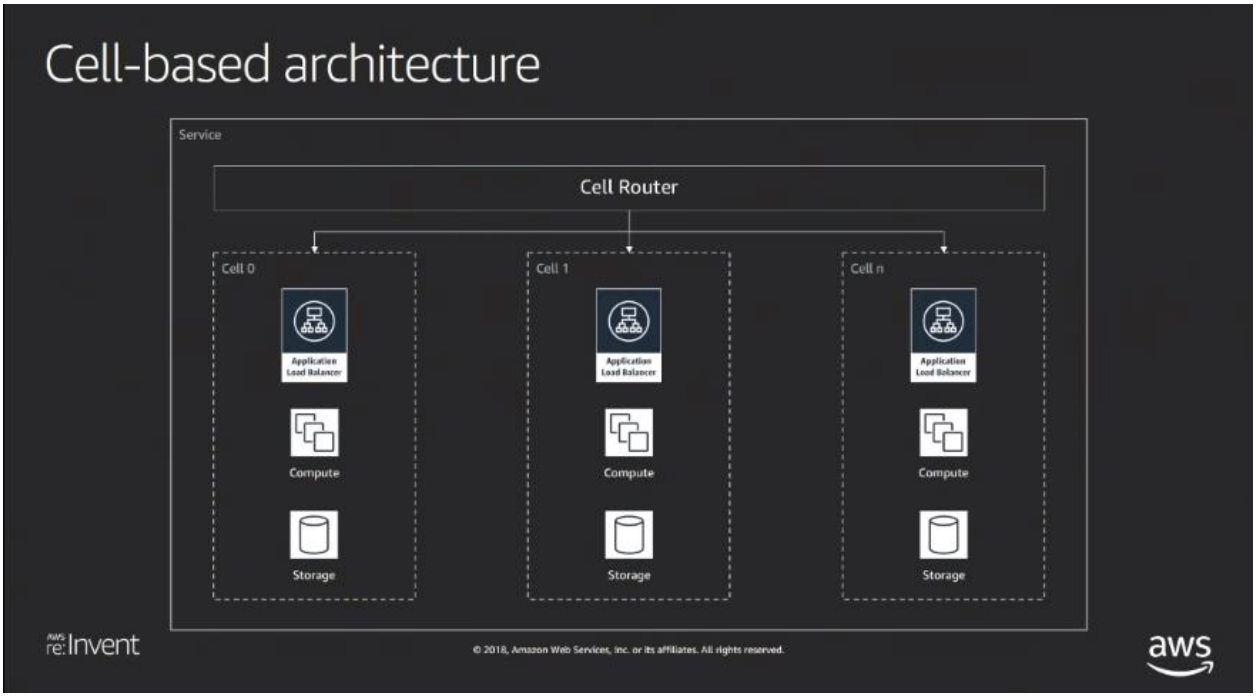




We create isolated instances of the service stack as individual cells



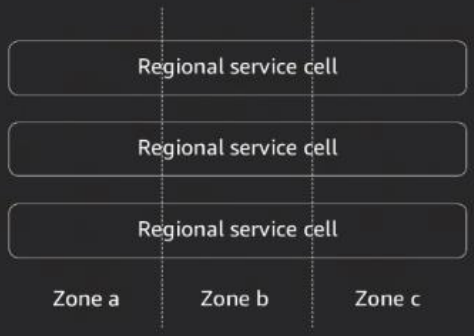
We then put a router on them to make routing decisions



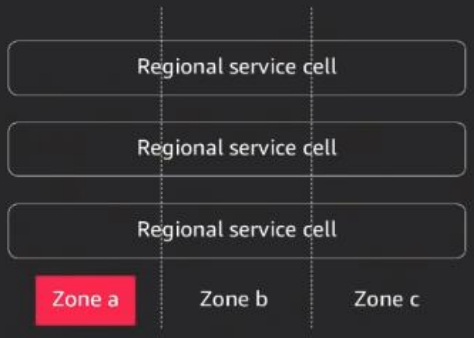
We now have a service with a cell-based architecture

Cells and Availability zones

Cells and Availability zones

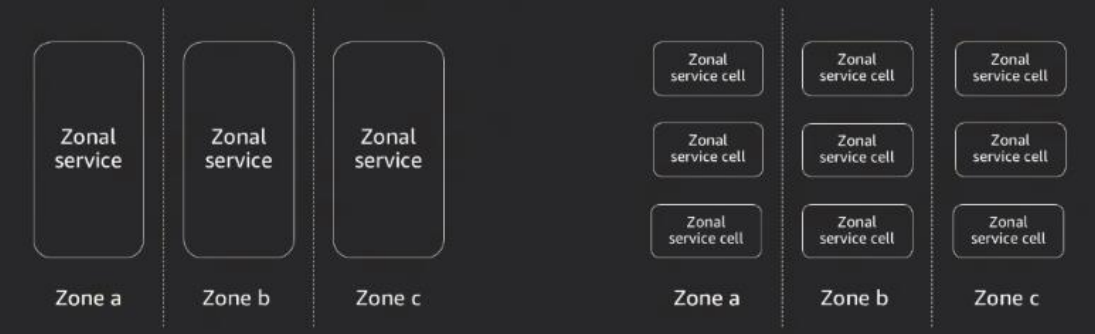


Availability Zone failure

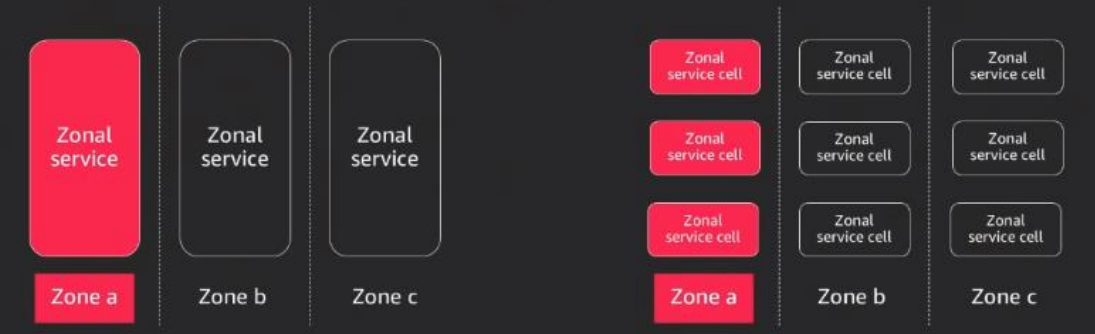


Theoretical blast radius

Cells and Availability Zones – zonal services



Availability zone failure

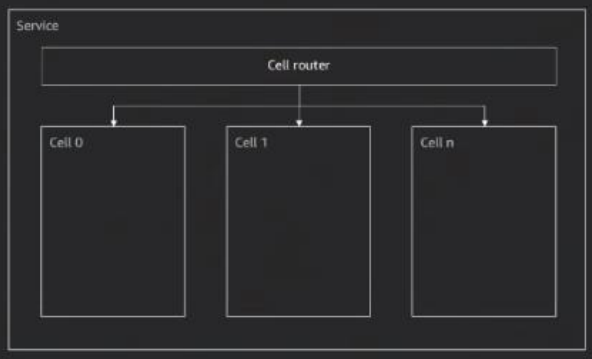


Partial availability zone failure

Theoretical blast radius

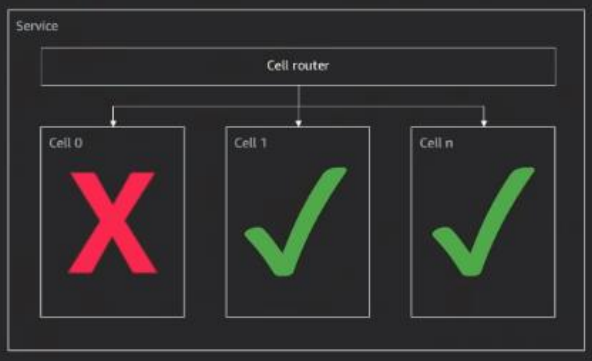
System properties

- Workload isolation
- Failure containment
- Scale-out vs. scale-up
- Testability
- Manageability



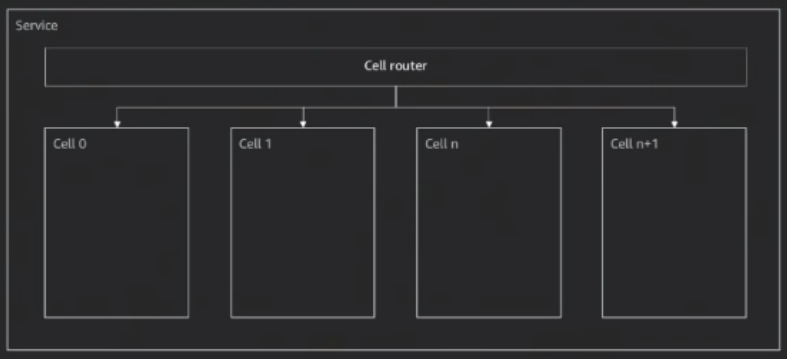
System properties

- Workload isolation
- Failure containment
- Scale-out vs. scale-up
- Testability
- Manageability



System properties

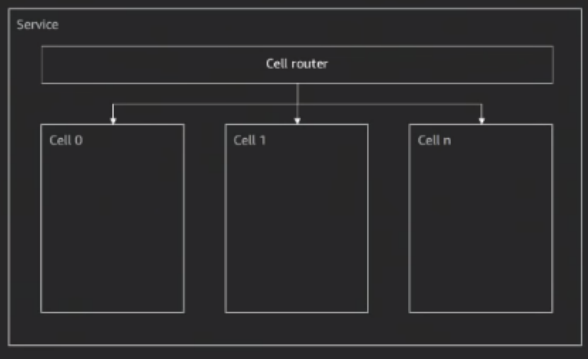
- Workload isolation
- Failure containment
- Scale-out vs. scale-up
- Testability
- Manageability



Cells have a maximum size beyond which we will have to scale out by adding a new cell instance, this helps with manageability of each cells too

Core considerations

- Cell size
- Router robustness
- Partitioning dimension
- Cross-cell use cases
- Cell migration



Cell size tradeoffs

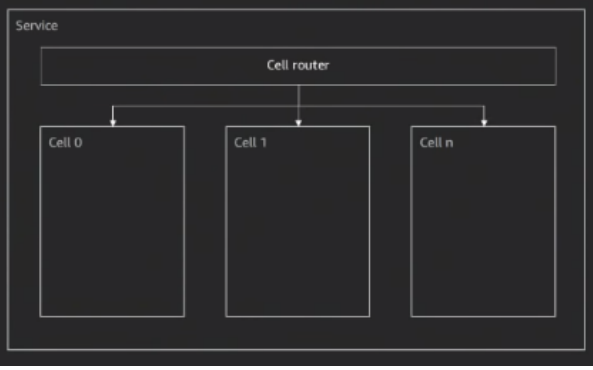
Smaller cells
- Reduced blast radius
- Easier to test
- Cells easier to operate

Larger cells
- Cost efficiency
- Reduced splits
- System easier to operate

Router robustness

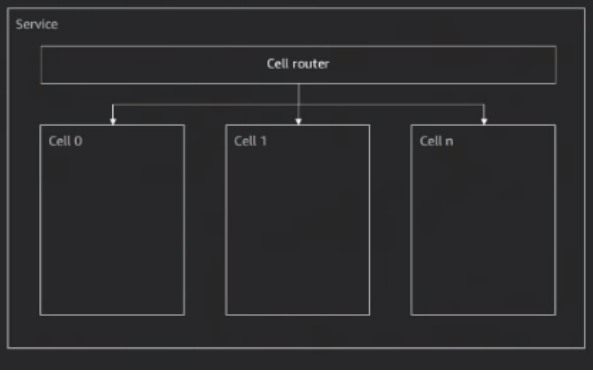
- Remaining shared component
- Stress testing
- Battle hardening
- Keep it simple!

- "Thinnest Possible Layer"



Partitioning dimension

- System-specific
- Needs careful analysis
- Cut with grain of domain

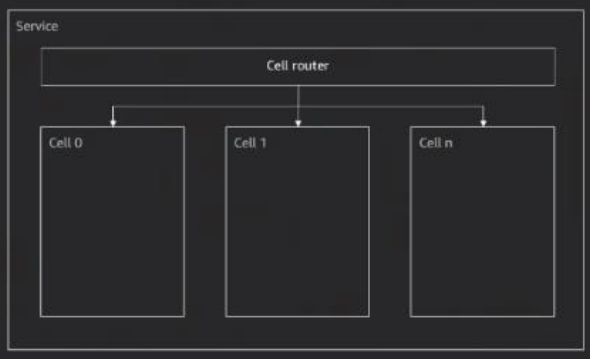


Cut with the grain

vs.

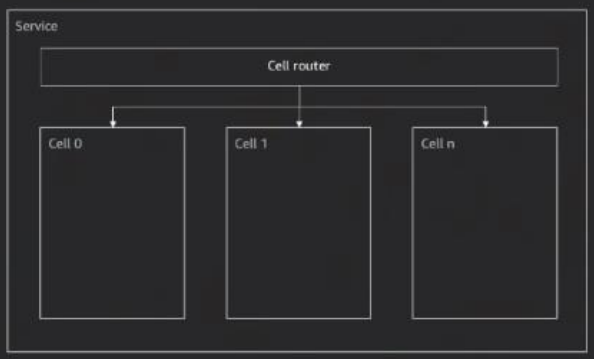
Cross-cell use cases

- May be unavoidable…keep to small subset of workload
- Scatter/gather queries
- Batch operations
- Coordinated writes
- Cell migration

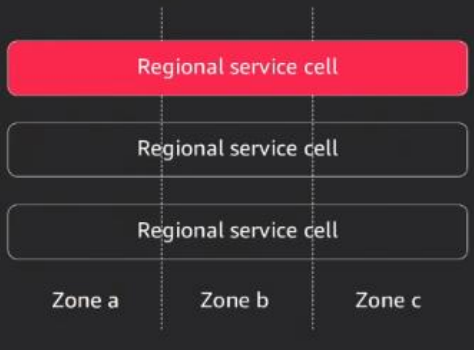


Cell migration

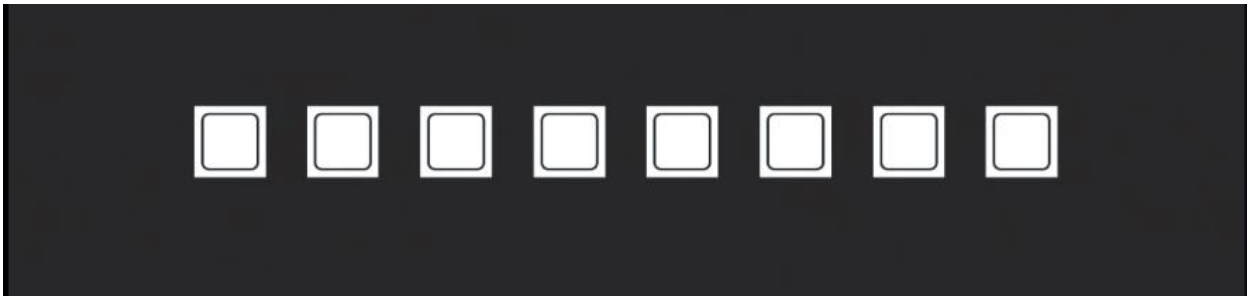
- Relocation of workloads
- Driven by
  - Heat management
  - Size balancing
  - Scale-out
- Similar to VM live migration
- Careful coordination between router and cells



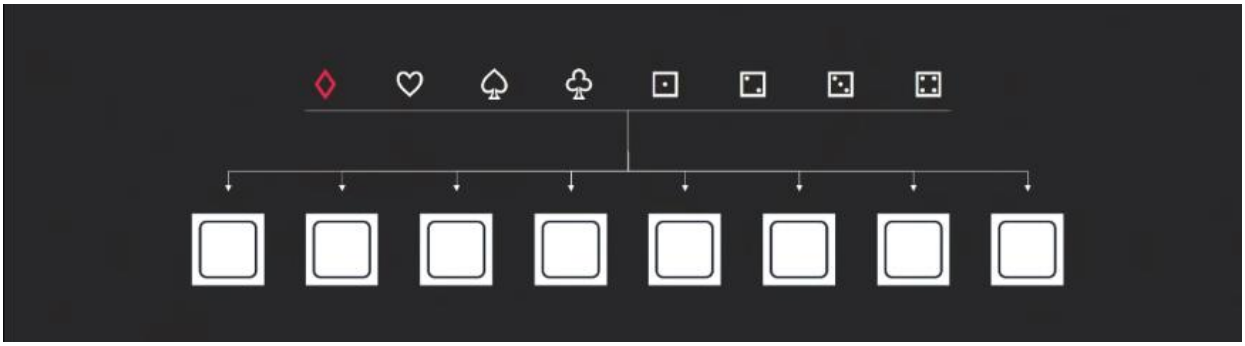
Theoretical blast radius



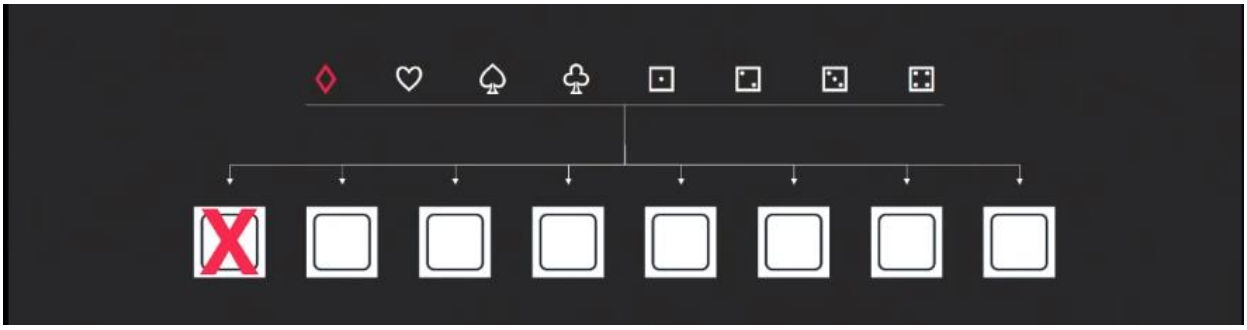
This looks like cell-based architecture but more focused on self-service or stateless services.

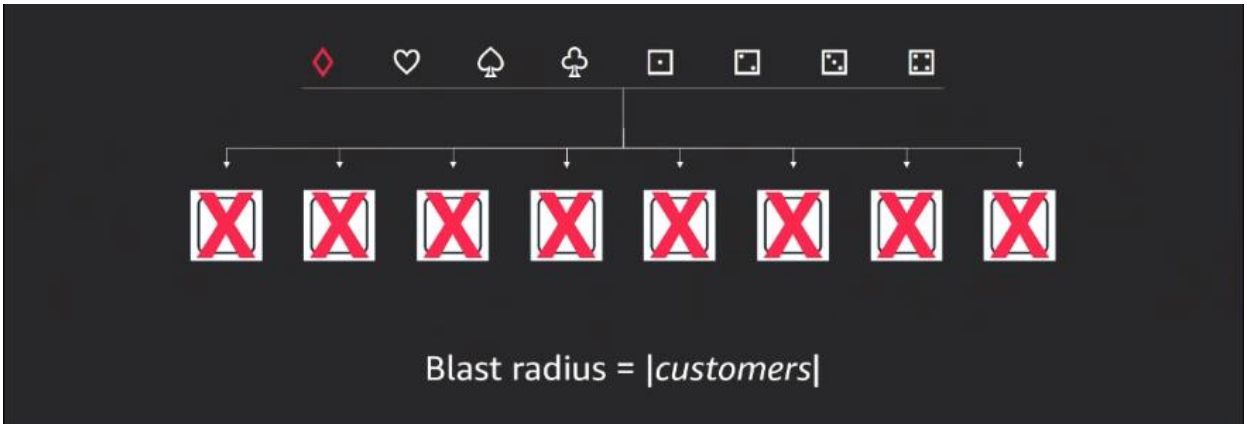


We have 8 nodes for our service

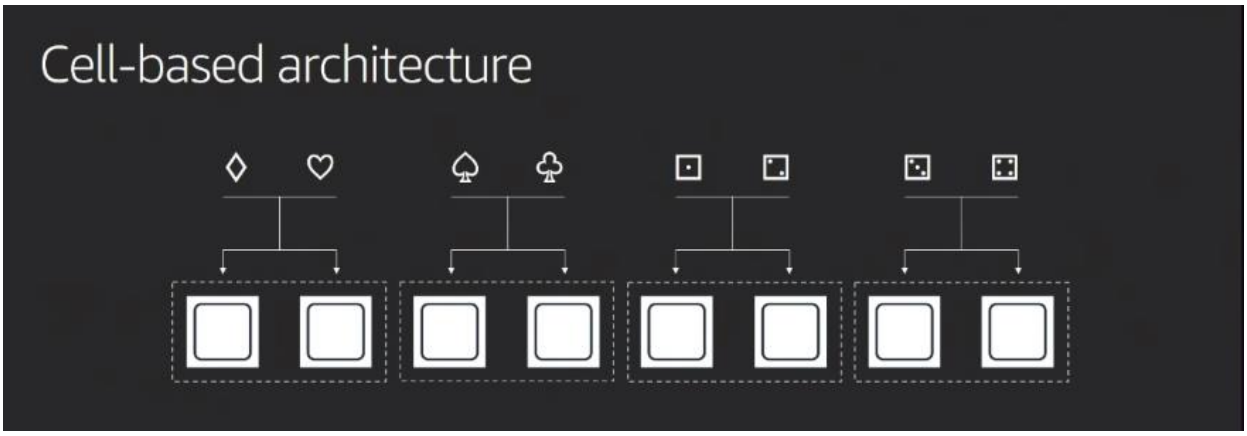


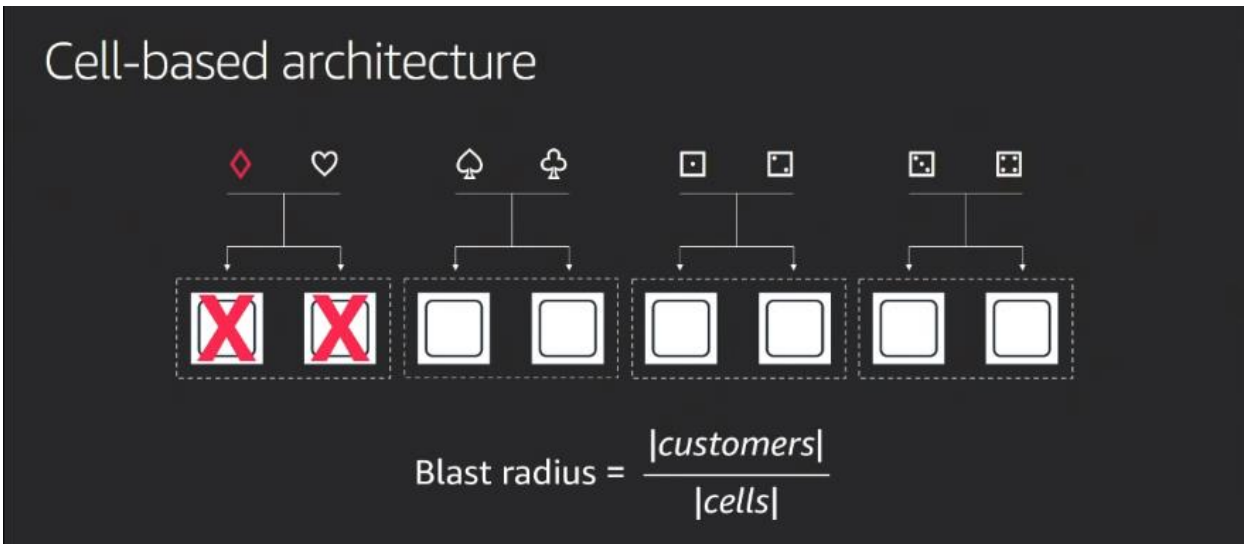
We now have 8 different customers sending requests, the diamond customer is sending a bad request

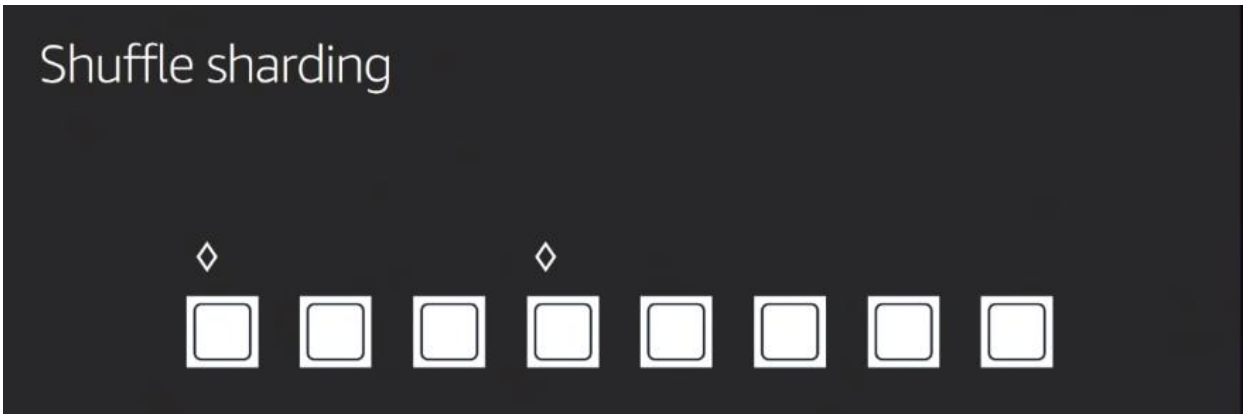


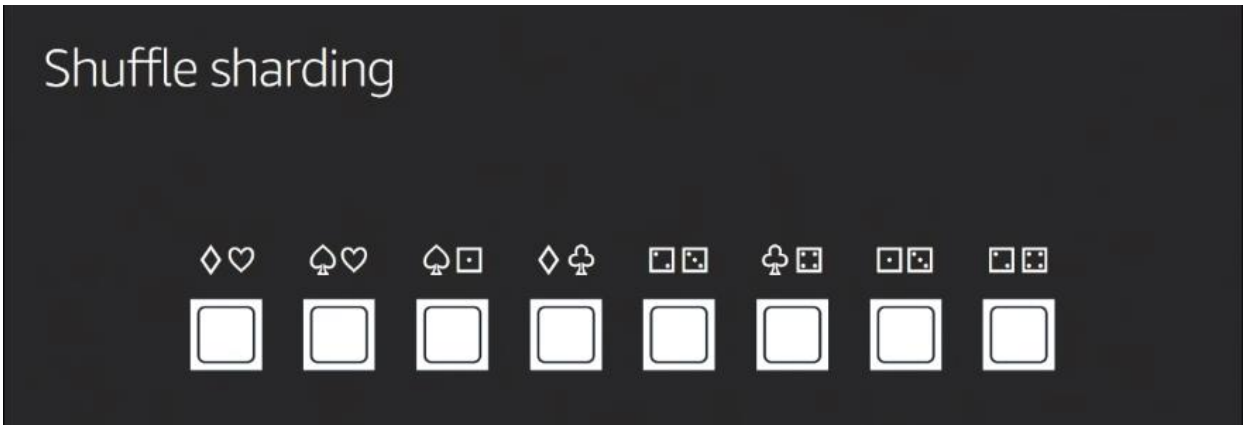


This might eventually take out the entire system

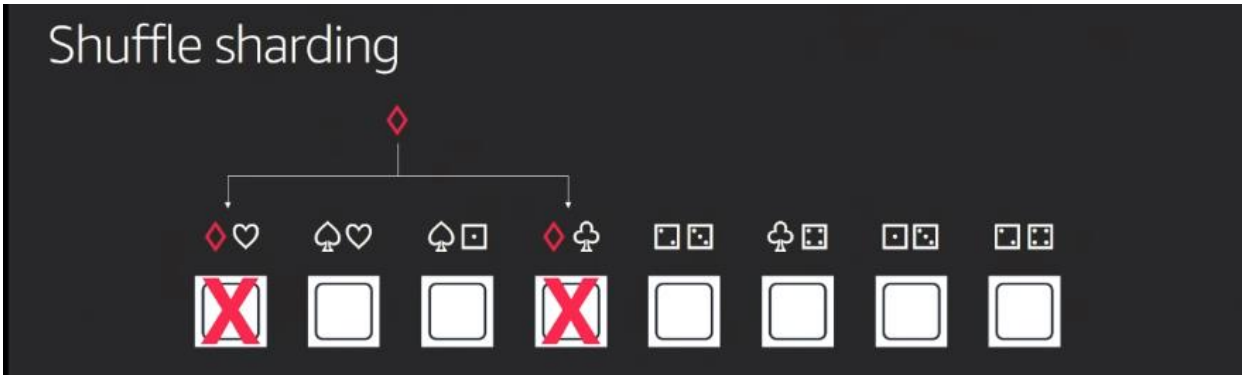


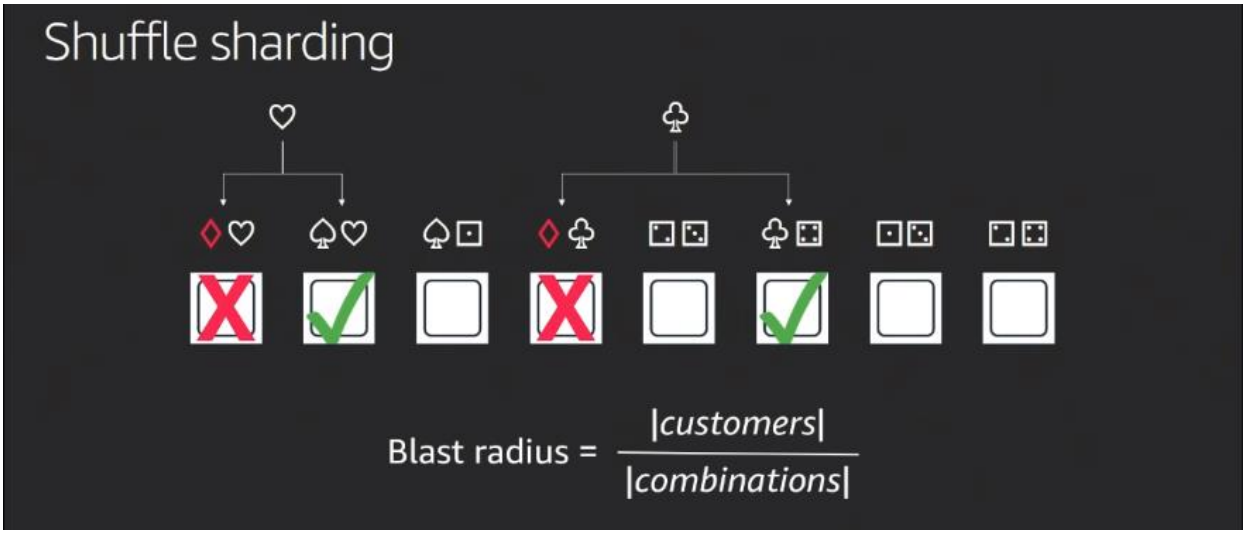




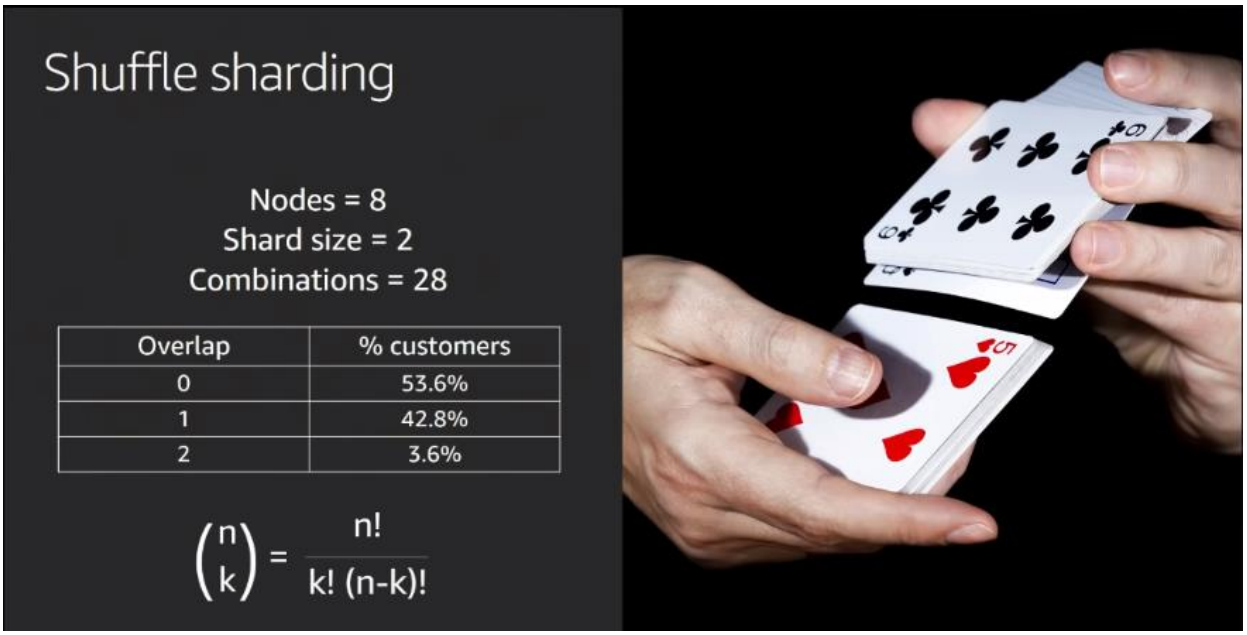


We take each customer and assign them to 2 nodes based on some hashing approach

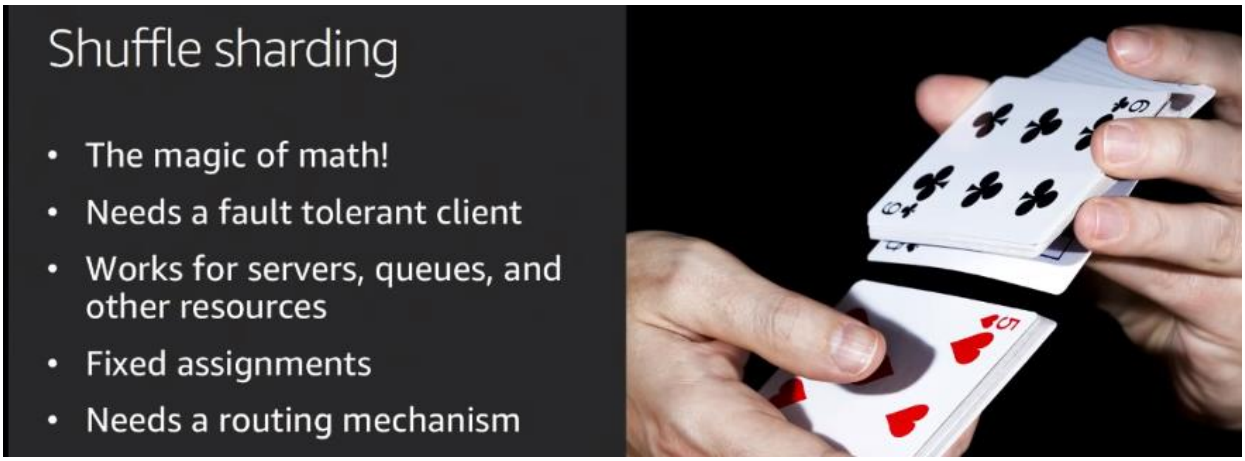




The heart and spade customers can retry and still get service



Cell-based architecture gave us 25% failure rate but shuffle sharding gives us 3.6% failure rate.



We have used the shuffle sharding approach to create a single-tenant experience in a multi-tenant distributed system. We are using fixed assignments of client requests to a possible set of nodes within the system. It also requires a fault-tolerant client that can do retries automatically.

Operational practices

"I'd wager that every new AWS engineer knows within their first week, if not their first day, that we never want to touch more than one zone at a time."

Werner Vogels

Software deployments

- Staggered deployments
  - Across regions
  - Across availability zones
  - Across cells
- Fractional deployments
  - Within deployment units
- Automated tests and canaries
- Automated rollback

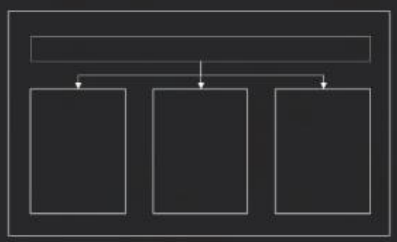
Automation is key

- Reviewable
- Testable
- Predictable
- Repeatable

End-to-end ownership

- Service teams own everything!
- No “throw it over the wall”
- Direct feedback loop to reduce blast radius

In conclusion…



In conclusion…

In conclusion…

Thank you!

Peter Vosshall
vosshall@amazon.com