

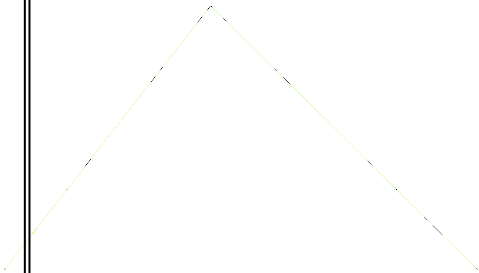
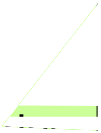
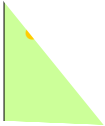
# **Best Practices for Scaling Websites**

## **Lessons from eBay**

**Randy Shoup**  
**eBay Distinguished Architect**

QCon Asia 2009





# Architectural Forces at Internet Scale

- Scalability
  - Resource usage should increase linearly (or better!) with load
  - Design for 10x growth in data, traffic, users, etc.
- Availability
  - Resilience to failure (MTBF)
  - Rapid recoverability from failure (MTTR)
  - Graceful degradation
- Latency
  - User experience latency
  - Data latency
- Manageability
  - Simplicity
  - Maintainability
  - Diagnostics
- Cost
  - Development effort and complexity
  - Operational cost (TCO)



# Best Practices for Scaling

1. Partition Everything
2. Asynchrony Everywhere
3. Automate Everything
4. Remember Everything Fails
5. Embrace Inconsistency



# Best Practice 1: Partition Everything

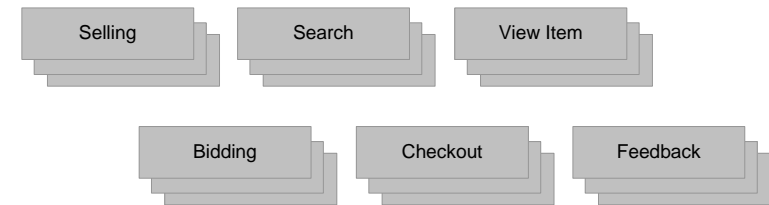
- Split every problem into manageable chunks
  - By data, load, and/or usage pattern
  - *“If you can’t split it, you can’t scale it”*
- Motivations
  - Scalability: can scale horizontally and independently
  - Availability: can isolate failures
  - Manageability: can decouple different segments and functional areas
  - Cost: can use less expensive hardware



# Best Practice 1: Partition Everything

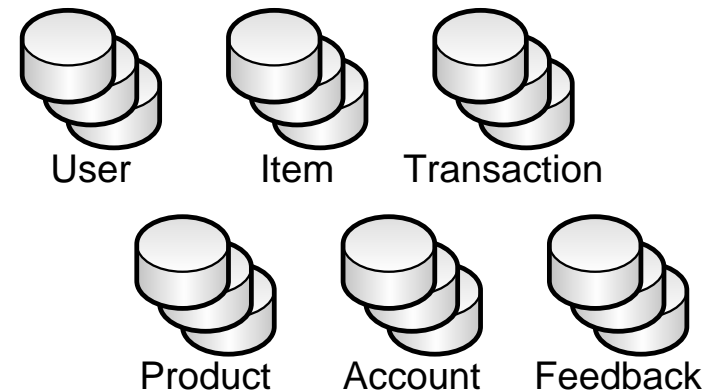
## **Pattern: Functional Segmentation**

- Segment processing into pools, services, and stages
- Segment data along usage boundaries



## **Pattern: Horizontal Split**

- Load-balance processing
  - Within a pool, all servers are created equal
- Split (or “*shard*”) data along primary access path
  - Partition by range, modulo of a key, lookup, etc.



## **Corollary: No Session State**

- User session flow moves through multiple application pools
- Absolutely no session state in application tier

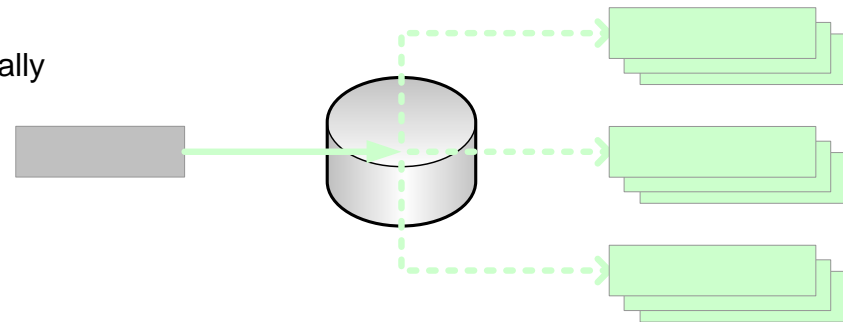
## Best Practice 2: Asynchrony Everywhere

- Prefer Asynchronous Processing
  - Move as much processing as possible to asynchronous flows
  - Where possible, integrate disparate components asynchronously
- Motivations
  - Scalability: can scale components independently
  - Availability
    - Can decouple availability state
    - Can retry operations
  - Latency
    - Can significantly improve user experience latency at cost of data/execution latency
    - Can allocate more time to processing than user would tolerate
  - Cost: can spread peak load over time

# Best Practice 2: Asynchrony Everywhere

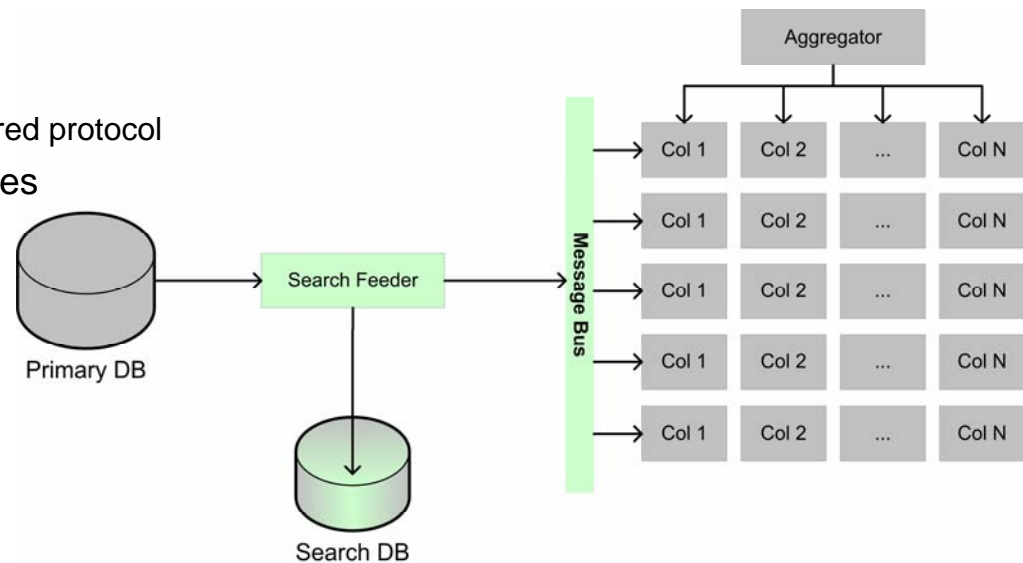
## Pattern: Event Queue

- Primary use-case produces event
  - Create event (*ITEM.NEW*, *ITEM.SOLD*) transactionally with primary insert/update
- Consumers subscribe to event
  - At least once delivery
  - No guaranteed order
  - Idempotency and readback



## Pattern: Message Multicast

- Search Feeder publishes item updates
  - Reads item updates from primary database
  - Publishes sequenced updates via SRM-inspired protocol
- Nodes listen to assigned subset of messages
  - Update in-memory index in real time
  - Request recovery (NAK) when messages are missed





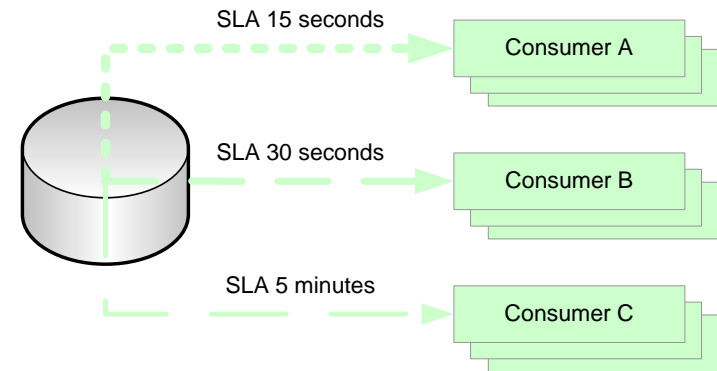
# Best Practice 3: Automate Everything

- Prefer Adaptive / Automated Systems to Manual Systems
- Motivations
  - Scalability
    - Can scale with machines, not humans
  - Availability / Latency
    - Can adapt to changing environment more rapidly
  - Cost
    - Machines are far less expensive than humans
    - Can learn / improve / adjust over time without manual effort

# Best Practice 3: Automate Everything

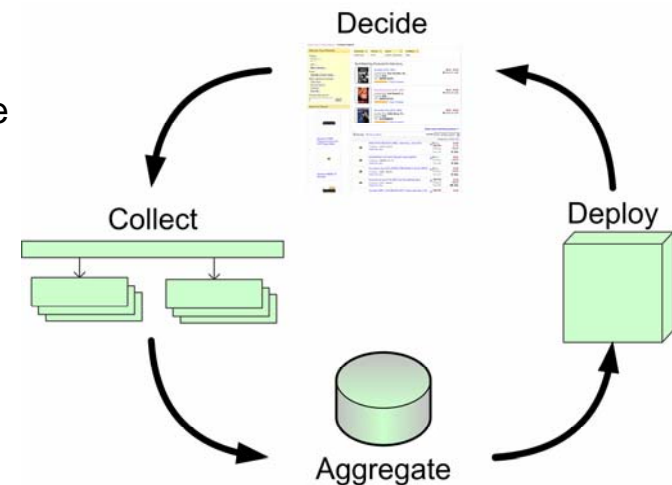
## **Pattern: Adaptive Configuration**

- Define SLA for a given logical consumer
  - E.g., 99% of events processed in 15 seconds
- Dynamically adjust config to meet defined SLA



## **Pattern: Machine Learning**

- Dynamically adapt search experience
  - Determine best inventory and assemble optimal page for that user and context
- Feedback loop enables system to learn and improve over time
  - Collect user behavior
  - Aggregate and analyze offline
  - Deploy updated metadata
  - Decide and serve appropriate experience
- Perturbation and dampening



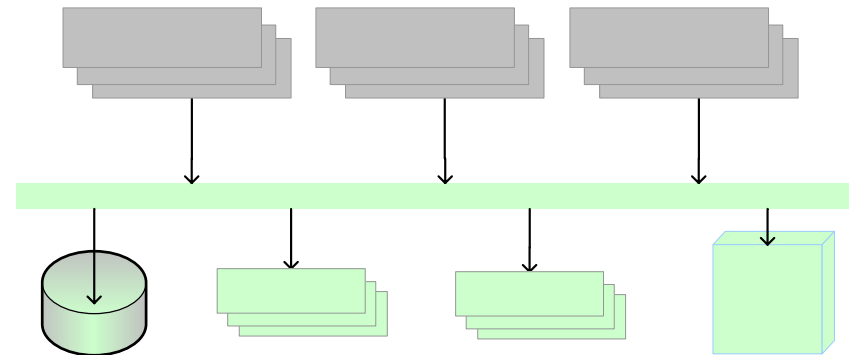
## Best Practice 4: Remember Everything Fails

- Build all systems to be tolerant of failure
  - Assume every operation will fail and every resource will be unavailable
  - Detect failure as rapidly as possible
  - Recover from failure as rapidly as possible
  - Do as much as possible during failure
- Motivation
  - Availability

# Best Practice 4: Remember Everything Fails

## **Pattern: Failure Detection**

- Servers log all requests
  - Log all application activity, database and service calls on multicast message bus
  - **Over 2TB of log messages per day**
- Listeners automate failure detection and notification



## **Pattern: Rollback**

- *Absolutely no changes to the site which cannot be undone (!)*
- Every feature has on / off state driven by central configuration
  - Feature can be immediately turned off for operational or business reasons
  - Features can be deployed “wired-off” to unroll dependencies

## **Pattern: Graceful Degradation**

- Application “marks down” an unavailable or distressed resource
- Non-critical functionality is removed or ignored
- Critical functionality is retried or deferred

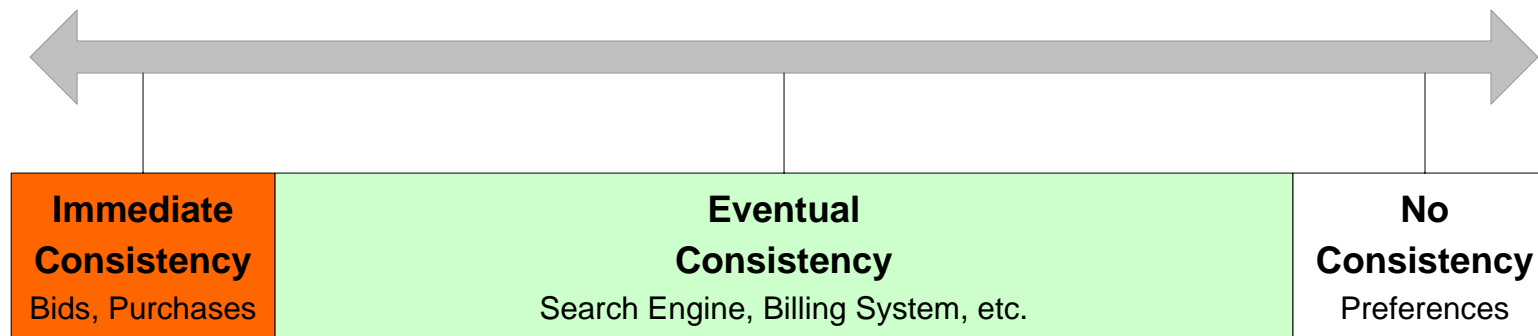
# Best Practice 5: Embrace Inconsistency

- Brewer's CAP Theorem
  - Any shared-data system can have at most two of the following properties:
    - **Consistency:** *All clients see the same data, even in the presence of updates*
    - **Availability:** *All clients will get a response, even in the presence of failures*
    - **Partition-tolerance:** *The system properties hold even when the network is partitioned*
  - This trade-off is fundamental to all distributed systems

# Best Practice 5: Embrace Inconsistency

## *Choose Appropriate Consistency Guarantees*

- To guarantee availability and partition-tolerance, we trade off immediate consistency
- Most real-world systems (even financial systems!) do not require immediate consistency
- Consistency is a spectrum
- Prefer eventual consistency to immediate consistency



## *Avoid Distributed Transactions*

- eBay does absolutely no distributed transactions – no two-phase commit
- Minimize inconsistency through state machines and careful ordering of operations
- Eventual consistency through asynchronous event or reconciliation batch

## Recap: Best Practices for Scaling

1. Partition Everything
2. Asynchrony Everywhere
3. Automate Everything
4. Remember Everything Fails
5. Embrace Inconsistency



# Questions?

## About the Presenter

**Randy Shoup** has been the primary architect for eBay's search infrastructure since 2004. Prior to eBay, Randy was Chief Architect and Technical Fellow at Tumbleweed Communications, and has also held a variety of software development and architecture roles at Oracle and Informatica.

[rshoup@ebay.com](mailto:rshoup@ebay.com)



© 2009 eBay Inc.