eBay’s Architectural Principles

Architectural Strategies, Patterns, and Forces for Scaling a Large eCommerce Site

Randy Shoup
eBay Distinguished Architect

QCon London 2008
March 14, 2008

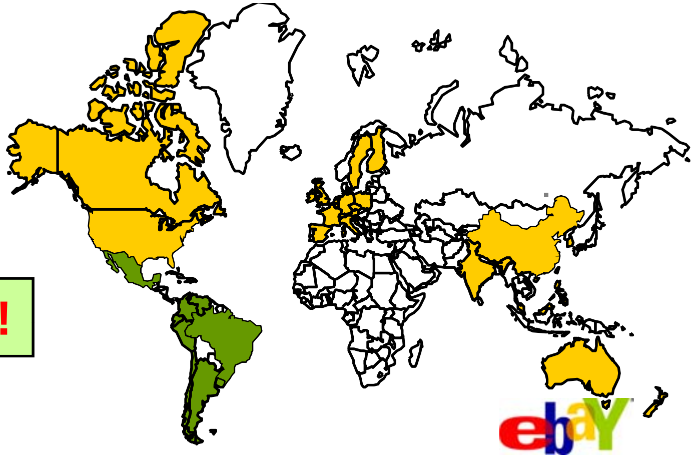
What we're up against

- eBay manages …
  - **Over 276,000,000 registered users**
  - **Over 2 Billion photos**
  - eBay users worldwide trade on average $2039 in goods every second
  - eBay averages well over 1 billion page views per day
  - At any given time, there are over 113 million items for sale in over 50,000 categories
  - eBay stores over 2 Petabytes of data – over 200 times the size of the Library of Congress!
  - The eBay platform handles 5.5 billion API calls per month
- In a dynamic environment
  - 300+ features per quarter
  - We roll 100,000+ lines of code every two weeks
- In 39 countries, in 7 languages, 24x7x365

>48 Billion SQL executions/day!

Over ½ Million pounds of Kimchi are sold every year!

Auction www.auction.co.kr an ebay company



Architectural Forces: What do we think about?

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

Architectural Strategies: How do we do it?

- Strategy 1: Partition Everything
  - *"How do you eat an elephant? … One bite at a time"*

- Strategy 2: Async Everywhere
  - *"Good things come to those who wait"*

- Strategy 3: Automate Everything
  - *"Give a man a fish and he eats for a day …*

    Teach a man to fish and he eats for a lifetime"

- Strategy 4: Remember Everything Fails
  - *"Be Prepared"*

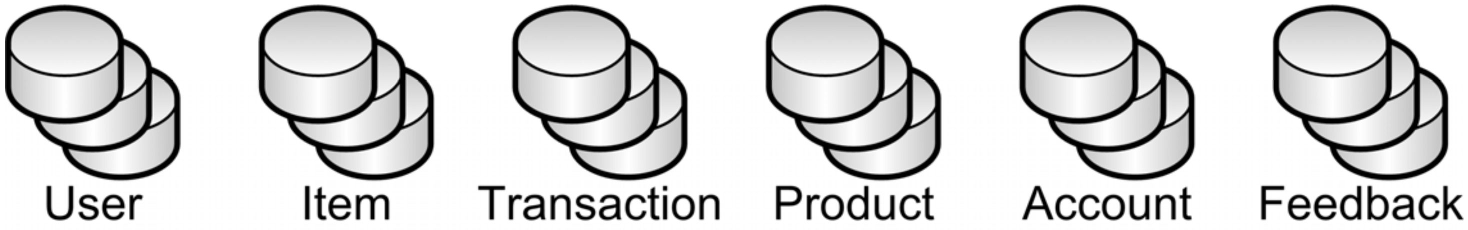
Strategy 1: Partition Everything

- 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
- Partitioning Patterns
  - *Functional Segmentation*
  - *Horizontal Split*

Partition Everything: Databases

Pattern: Functional Segmentation

- Segment databases into functional areas



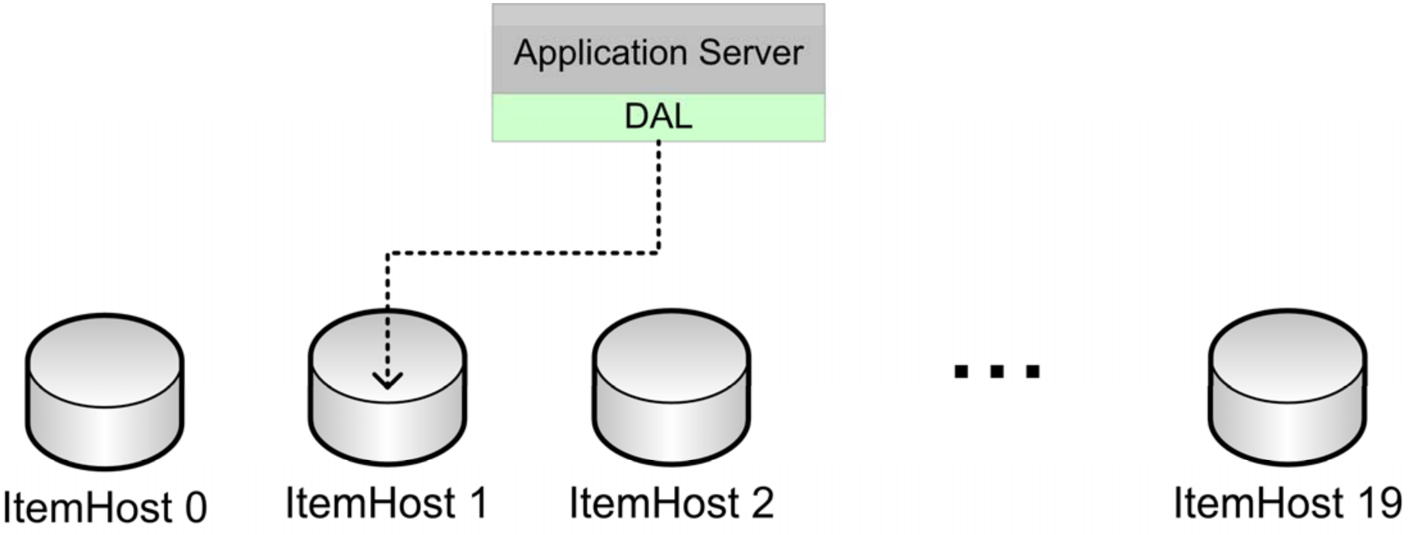
- Group data using standard data modeling techniques
  - Cardinality (1:1, 1:N, M:N)
  - Data relationships
  - Usage characteristics
- Logical hosts
  - Abstract application's logical representation from host's physical location
  - Support collocating and separating hosts without code change

Over 1000 logical databases on ~400 physical hosts

Partition Everything: Databases

Pattern: Horizontal Split

- Split (or *"shard"*) databases horizontally along primary access path
- Different split strategies for different use cases
  - Modulo on key (item id, user id, etc.)
  - Lookup- or range-based
- Aggregation / routing in Data Access Layer (DAL)
  - Abstracts developers from split logic, logical-physical mapping
  - Routes CRUD operation(s) to appropriate split(s)
  - Supports rebalancing through config change



Partition Everything: Databases

Corollary: No Database Transactions

- eBay’s transaction policy
  - Absolutely no client side transactions, two-phase commit, etc.
  - Auto-commit for vast majority of DB writes
  - Anonymous PL/SQL blocks for multi-statement transactions within single DB
- *Consistency is not always required or possible (!)*
  - To guarantee availability and partition-tolerance, we are forced to trade off consistency (Brewer’s CAP Theorem)
  - Leads unavoidably to systems with BASE semantics rather than ACID guarantees
  - Consistency is a spectrum, not binary
- Consistency without transactions
  - Careful ordering of DB operations
  - Eventual consistency through asynchronous event or reconciliation batch

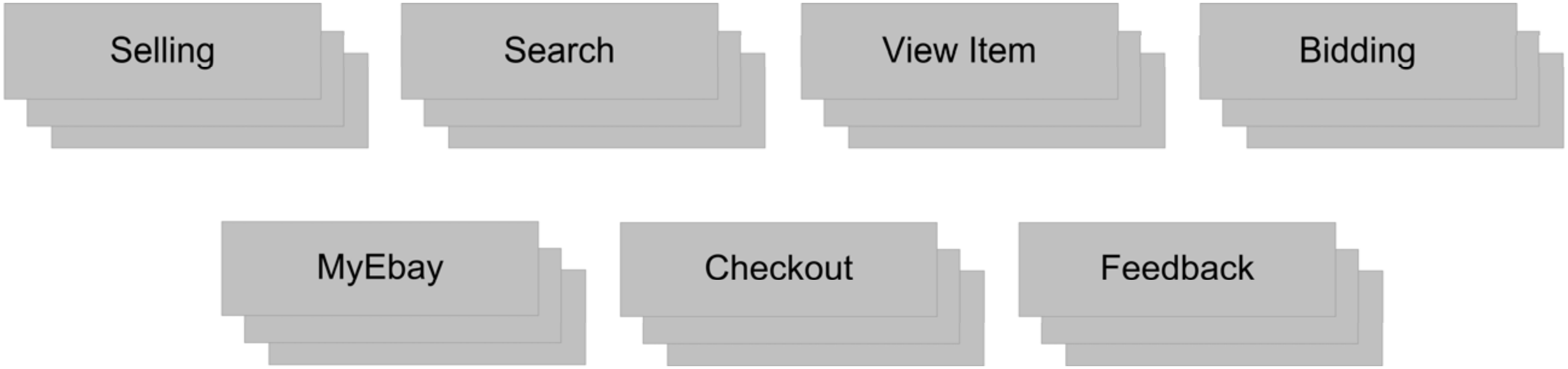
Partition Everything: Application Tier

Pattern: Functional Segmentation

- Segment functions into separate application pools
- Minimizes DB / resource dependencies
- Allows for parallel development, deployment, and monitoring

Pattern: Horizontal Split

- Within pool, all application servers are created equal
- Routing through standard load-balancers
- Allows for rolling updates



Over 16,000 application servers in 220 pools

Partition Everything: Application Tier

Corollary: No Session State

- User session flow moves through multiple application pools
- Absolutely no session state in application tier
- Transient state maintained / referenced by
  - URL
  - Cookie
  - Scratch database

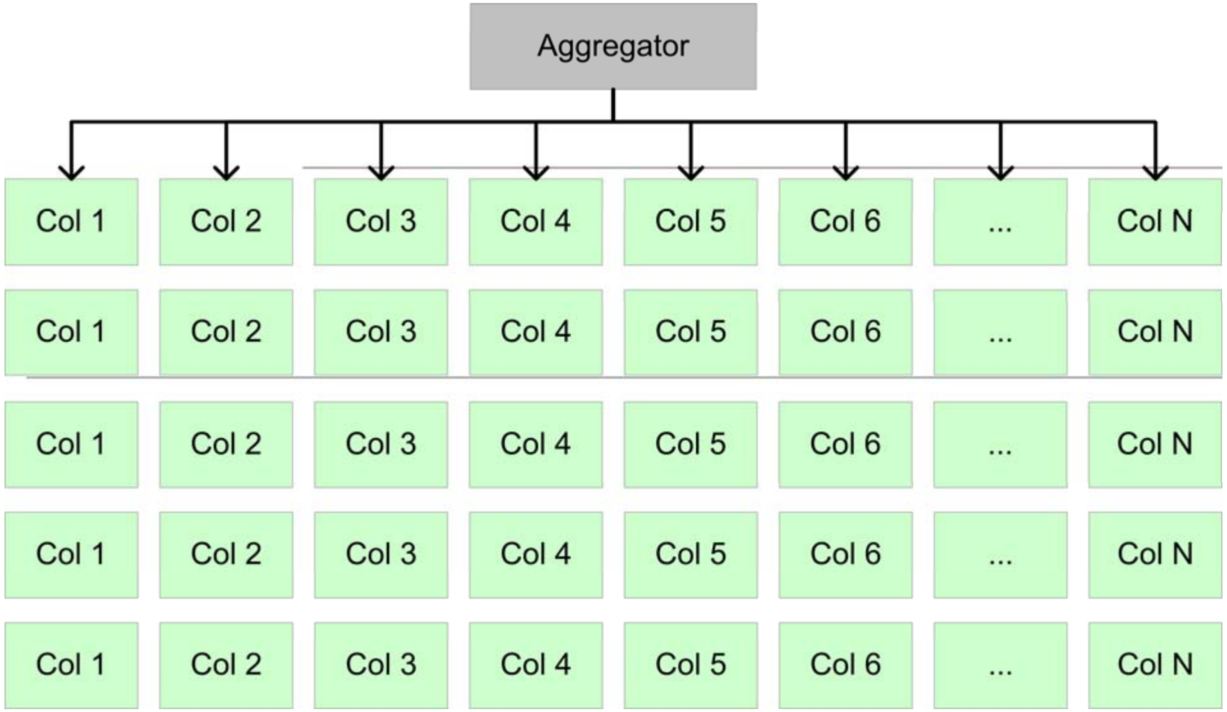
Partition Everything: Search Engine

Pattern: Functional Segmentation

- Read-only search function decoupled from write-intensive transactional databases

Pattern: Horizontal Split

- Search index divided into grid of N slices (“columns”) by modulo of a key
- Each slice is replicated to M instances (“rows”)
- Aggregator parallelizes query to one node in each column, aggregates results



Aggregator

Col 1	Col 2	Col 3	Col 4	Col 5	Col 6	...	Col N
Col 1	Col 2	Col 3	Col 4	Col 5	Col 6	...	Col N
Col 1	Col 2	Col 3	Col 4	Col 5	Col 6	...	Col N
Col 1	Col 2	Col 3	Col 4	Col 5	Col 6	...	Col N
Col 1	Col 2	Col 3	Col 4	Col 5	Col 6	...	Col N

Strategy 2: Async Everywhere

- Prefer Asynchronous Processing
  - Move as much processing as possible to asynchronous flows
  - Where possible, integrate disparate components asynchronously

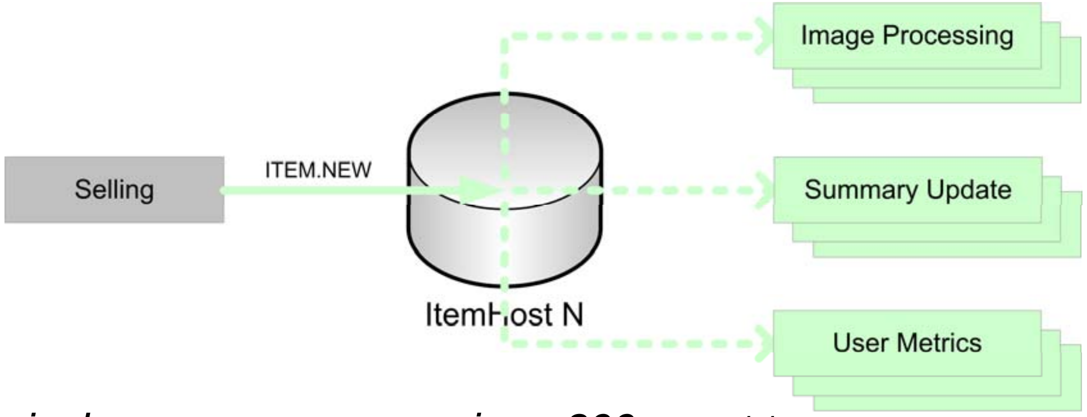
- Motivations
  - Scalability: can scale components independently
  - Availability
    - Can decouple availability state
    - Can retry operations
  - Latency
    - Can significantly improve user experience latency at cost of data/execution latency
    - Can allocate more time to processing than user would tolerate
  - Cost: can spread peak load over time

- Asynchrony Patterns
  - *Message Dispatch*
  - *Periodic Batch*

Async Everywhere: Event Streams

Pattern: Message Dispatch

- Primary use case produces event
  - *E.g.*, *ITEM.NEW*, *BID.NEW*, *ITEM.SOLD*, etc.
  - Event typically created transactionally with insert/update of primary table
- Consumers subscribe to event
  - Multiple logical consumers can process each event
  - Each logical consumer has its own event queue
  - Within each logical consumer, single consumer instance processes event
  - Guaranteed at least once delivery; no guaranteed order
- Managing timing conditions
  - Idempotency: processing event N times should give same results as processing once
  - Readback: consumer typically reads back to primary database for latest data

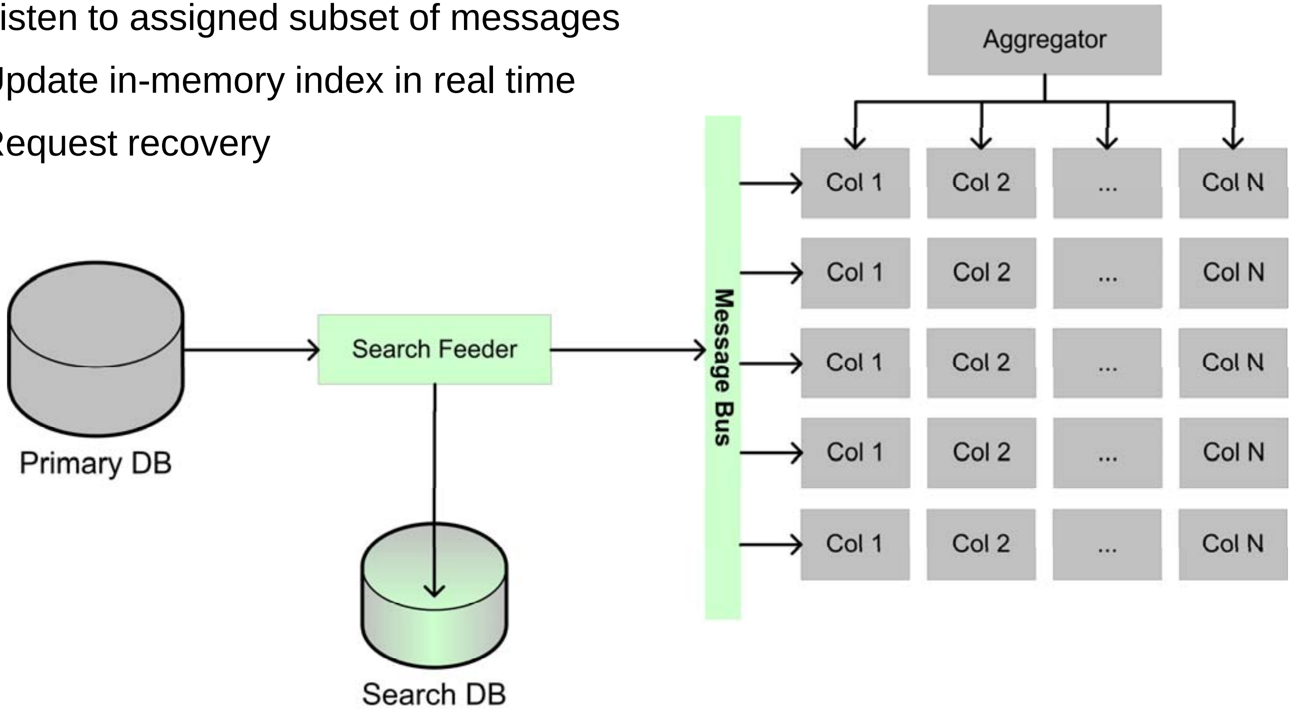


Over 100 logical consumers consuming ~300 event types

Async Everywhere: Search Feeder Infrastructure

Pattern: Message Dispatch

- Feeder reads item updates from primary database
- Feeder publishes updates via reliable multicast
  - Persist messages in intermediate data store for recovery
  - Publish updates to search nodes
  - Resend recovery messages when messages are missed
- Search nodes listen to updates
  - Listen to assigned subset of messages
  - Update in-memory index in real time
  - Request recovery



Async Everywhere: Batch

Pattern: Periodic Batch

- Scheduled offline batch process
- Most appropriate for
  - Infrequent, periodic, or scheduled processing (once per day, week, month)
  - Non-incremental computation (a.k.a. “Full Table Scan”)
- Examples
  - Import third-party data (catalogs, currency, etc.)
  - Generate recommendations (items, products, searches, etc.)
  - Process items at end of auction
- Often drives further downstream processing through ***Message Dispatch***

Strategy 3: Automate Everything

- Prefer Adaptive / Automated Systems to Manual Systems

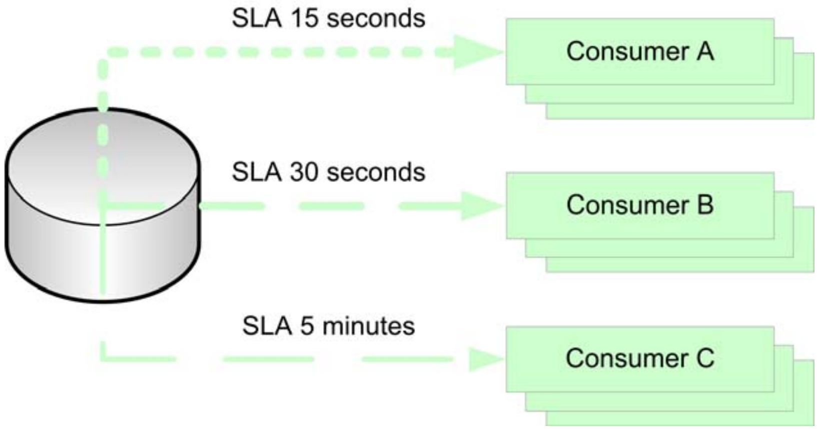
- Motivations
  - Scalability
    - Can scale with machines, not humans
  - Availability / Latency
    - Can adapt to changing environment more rapidly
  - Cost
    - Machines are far less expensive than humans
    - Can learn / improve / adjust over time without manual effort
  - Functionality
    - Can consider more factors in decisions
    - Can explore solution space more thoroughly and quickly

- Automation Patterns
  - *Adaptive Configuration*
  - *Machine Learning*

Automate Everything: Event Consumer Configuration

Pattern: Adaptive Configuration

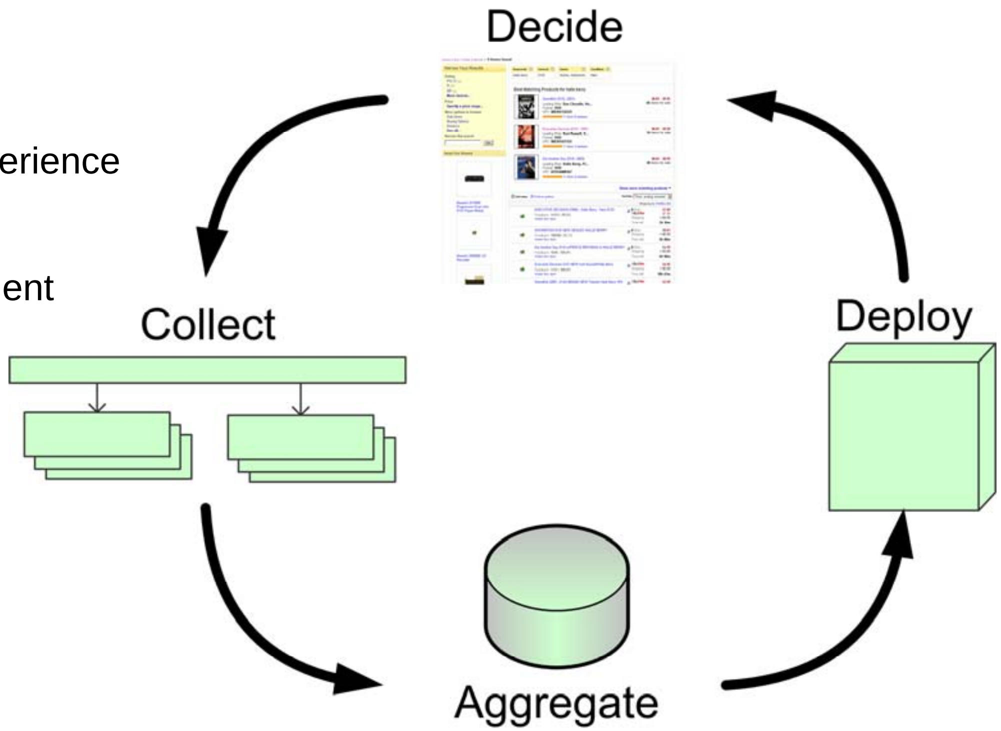
- Define service-level agreement (SLA) for a given logical event consumer
  - E.g., 99% of events processed in 15 seconds
- Consumer dynamically adjusts to meet defined SLA with minimal resources
  - Event polling size and polling frequency
  - Number of processor threads
- Automatically adapts to changes in
  - Load (queue length)
  - Event processing time
  - Number of consumer instances



Automate Everything: Adaptive Finding Experience

Pattern: Machine Learning

- Dynamically adapt experience
  - Choose page, modules, and inventory which provide best experience for that user and context
  - Order results by combination of demand, supply, and other factors (“Best Match”)
- Feedback loop enables system to learn and improve over time
  - Collect user behavior
  - Aggregate and analyze offline
  - Deploy updated metadata
  - Decide on and serve appropriate experience
- Best Practices
  - “Perturbation” for continual improvement
  - Dampening of positive feedback



Strategy 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

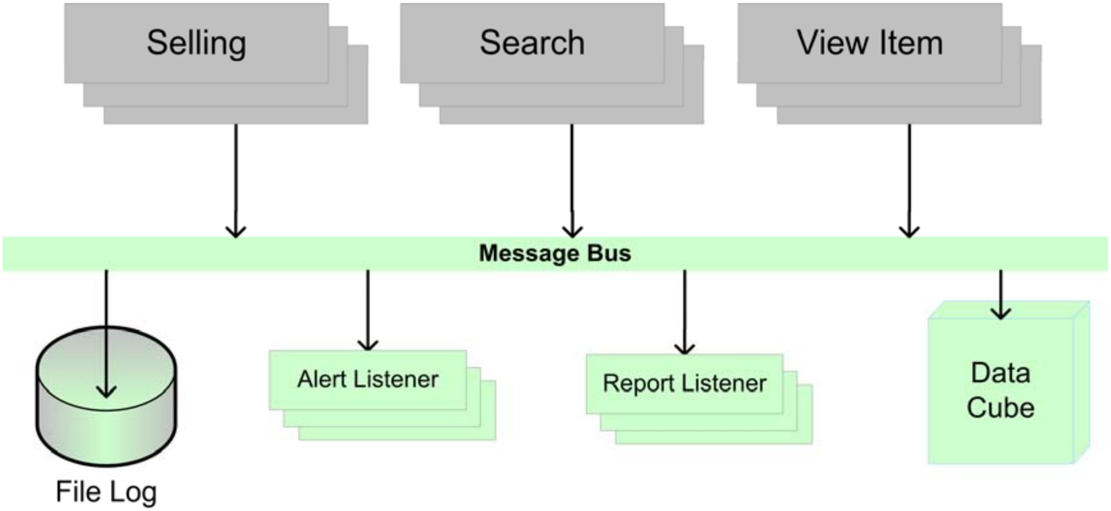
- Motivation
  - Availability

- Failure Patterns
  - *Failure Detection*
  - *Rollback*
  - *Graceful Degradation*

Everything Fails: Central Application Logging

Pattern: Failure Detection

- Application servers log all requests
  - Detailed logging of all application activity, particularly database and other external resources
  - Log request, application-generated information, and exceptions
- Messages broadcast on multicast message bus
- Listeners automate failure detection and notification
  - Real-time application state monitoring: exceptions and operational alerts
  - Historical reports by application server pool, URL, database, etc.



- *Over 1.5TB of log messages per day*

Everything Fails: Code Rollout / Rollback

Pattern: Rollback

Absolutely no changes to the site which cannot be undone (!)

- Entire site rolled every 2 weeks: 16,000 application servers in 220 pools
- Many deployed features have dependencies between pools
- Rollout plan contains explicit set (transitive closure) of all rollout dependencies
- Automated tool executes staged rollout, with built-in checkpoints and immediate rollback if necessary
- Automated tool optimizes rollback, including full rollback of dependent pools

Everything Fails: Feature Wire-on / Wire-off

Pattern: Rollback

- Every feature has on / off state driven by central configuration
  - Allows feature to be immediately turned off for operational or business reasons
  - Allows features to be deployed “wired-off” to unroll dependencies
- Decouples code deployment from feature deployment
- Applications check for feature “availability” in the same way as they check for resource availability

Everything Fails: Resource Markdown

Pattern: Failure Detection

- Application detects when database or other backend resource is unavailable or distressed
  - "Resource slow" is often far more challenging than "resource down" (!)

Pattern: Graceful Degradation

- Application "marks down" the resource
  - Stops making calls to it and sends alert
- Non-critical functionality is removed or ignored
- Critical functionality is retried or deferred
  - Failover to alternate resource
  - Defer processing to async event
- Explicit "markup"
  - Allows resource to be restored and brought online in a controlled way

Recap: Architectural Strategies

- Strategy 1: Partition Everything
- Strategy 2: Async Everywhere
- Strategy 3: Automate Everything
- Strategy 4: Remember Everything Fails

Questions?

- Randy Shoup, eBay Distinguished Architect

  rshoup@ebay.com