eBay Marketplace Architecture

Architectural Strategies, Patterns, and Forces

Randy Shoup, eBay Distinguished Architect

QCon SF 2007
November 9, 2007

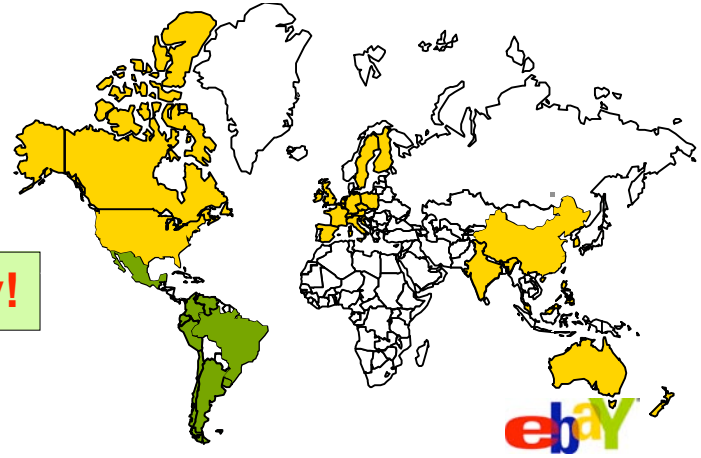
What we're up against

- eBay manages …
  - **Over 248,000,000 registered users**
  - **Over 1 Billion photos**
  - eBay users worldwide trade on average $1812 in goods every second
  - eBay averages well over 1 billion page views per day
  - At any given time, there are over 100 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 4.4 billion API calls per month
- In a dynamic environment
  - 300+ features per quarter
  - We roll 100,000+ lines of code every two weeks
- In 38 countries, in seven languages, 24x7

>44 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 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"*

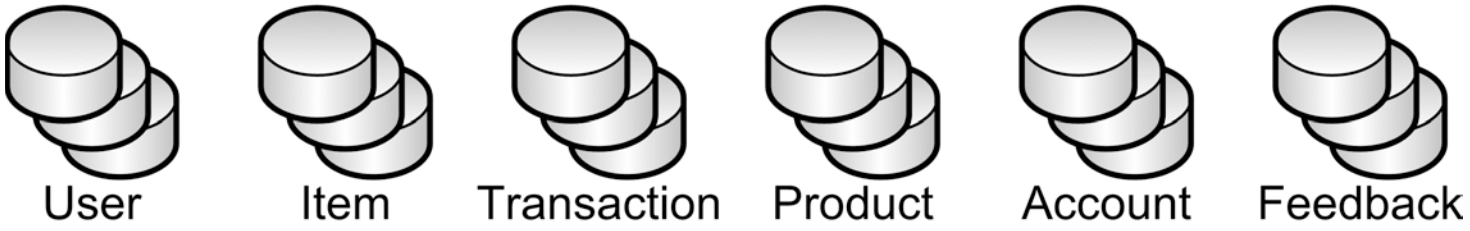
- Motivations
  - Scalability: can scale horizontally and independently
  - Availability: can isolate failures
  - Manageability: can decouple different segments and functional areas
  - Cost: can use less expensive hardware

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

Partition Everything: Databases

Pattern: Functional Segmentation

- Segment databases into functional areas



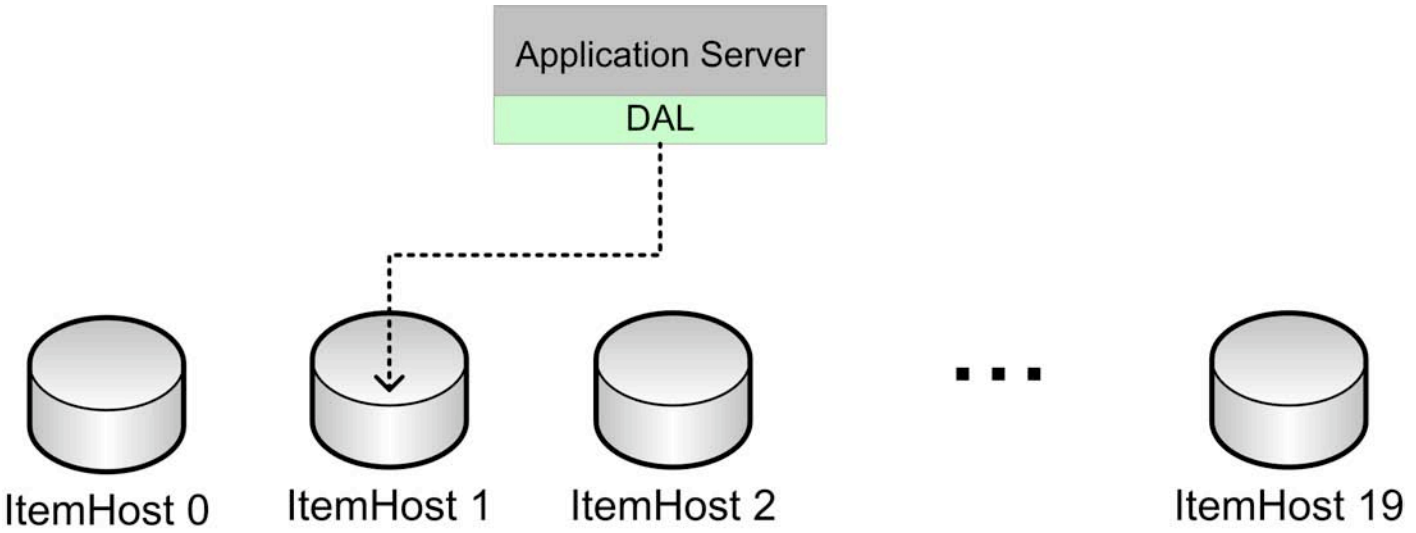
- Group data using standard data modeling techniques
  - Cardinality (1:1, 1:N, M:N)
  - Relationships
  - Usage characteristics
- Logical hosts
  - Abstract application's logical representation from host's physical location
  - Support combining and splitting without code change

Over 1000 logical databases on ~400 physical hosts

Partition Everything: Databases

Pattern: Horizontal Split

- Split databases horizontally along primary access path
- Multiple split approaches 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

- Transaction policy
  - Absolutely no client side transactions, two-phase commit, etc.
  - Auto-commit for vast majority of DB writes
  - Anonymous PL/SQL blocks for transactions within single database
- Consistency without transactions
  - Careful ordering of DB operations
  - Recovery through
    - Asynchronous recovery event
    - Reconciliation batch
    - Failover to guaranteed async flow
- Additional benefits
  - Avoids deadlocks
  - Avoids coupling availability
  - Maximizes update concurrency
- All consistency is not created equal (!)

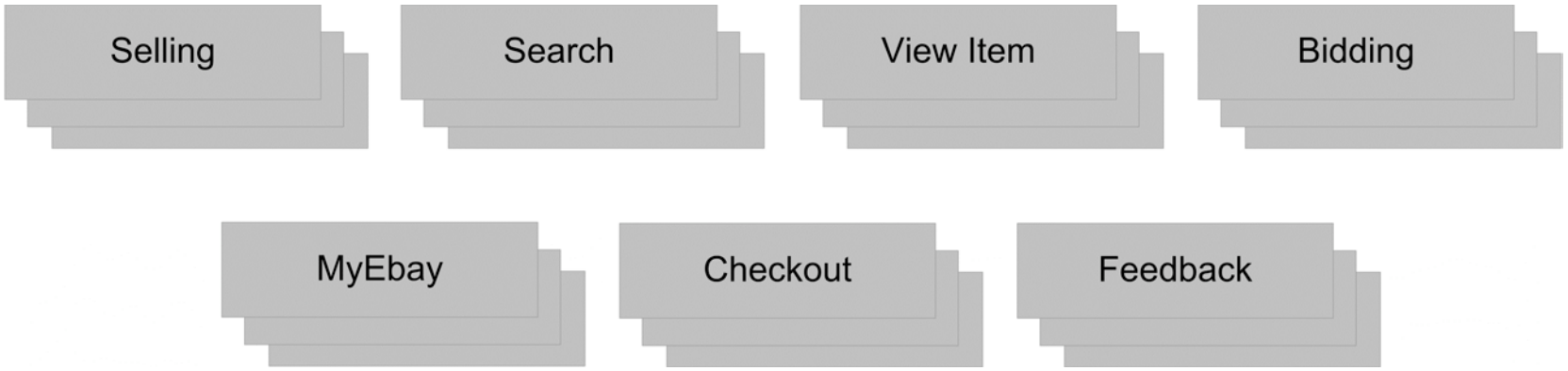
Partition Everything: Application Tier

Pattern: Functional Segmentation

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

Pattern: Horizontal Split

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



[9]*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

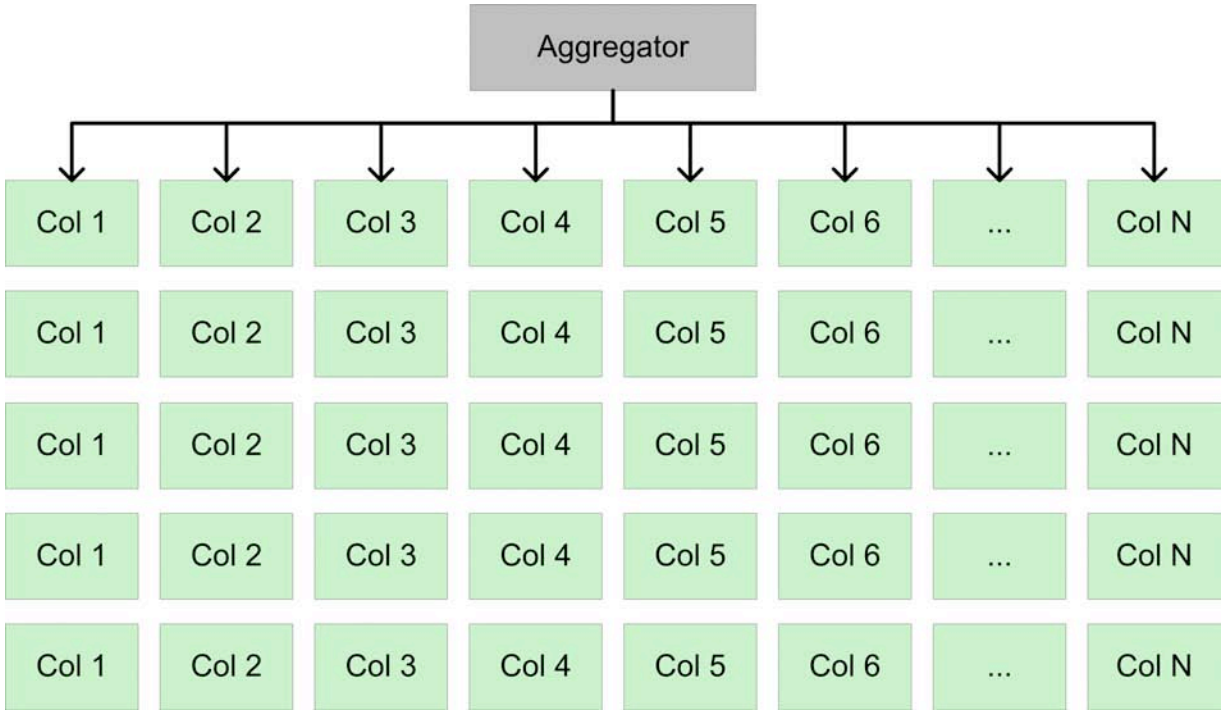
Partition Everything: Search

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 over all N slices, load-balances over M instances



Strategy 2: Async Everywhere

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

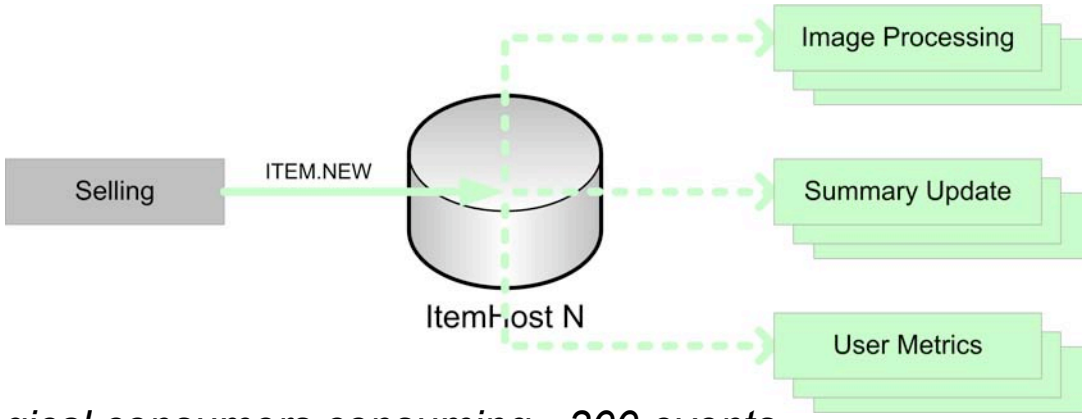
- Motivations
  - Scalability: can scale components independently
  - Availability
    - Can decouple availability state (up/down)
    - Can decouple availability characteristics (always available, best effort)
    - Can retry operations
  - Latency
    - Can improve user experience latency at cost of 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 processes event independently
  - 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

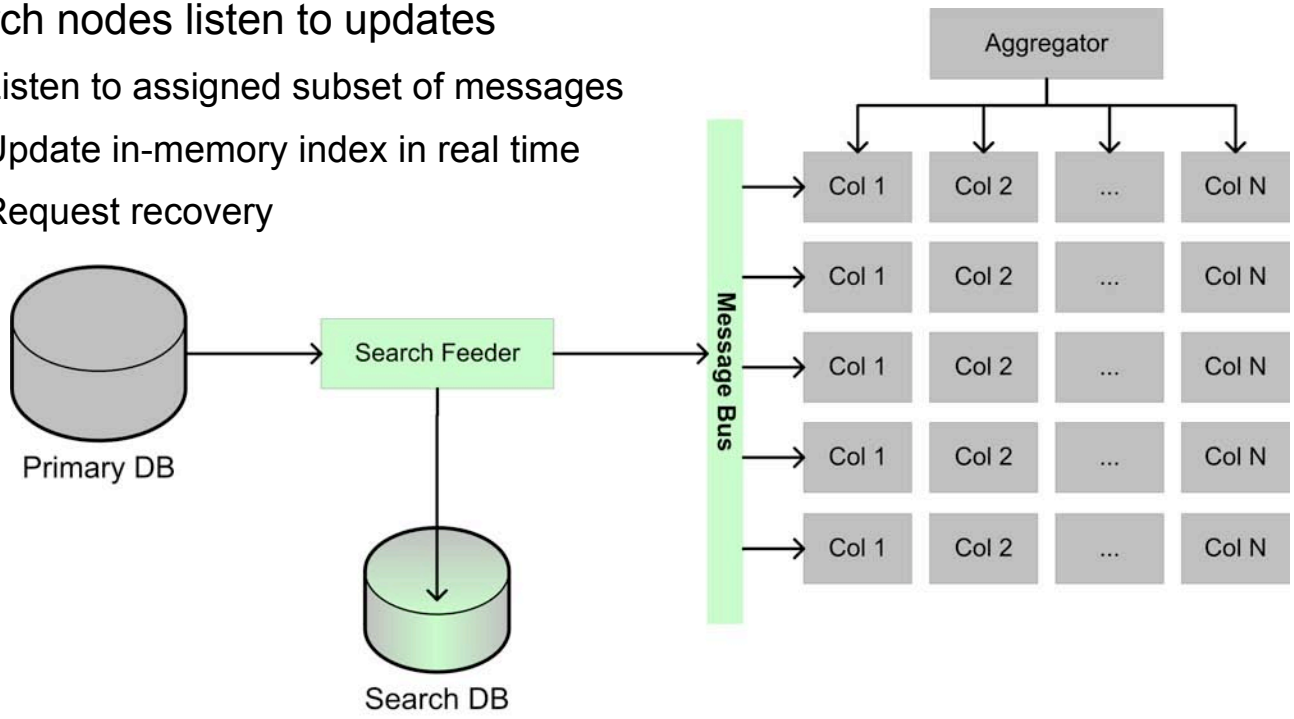


Over 100 logical consumers consuming ~300 events

Async Everywhere: Search Feeder Infrastructure

Pattern: Message Dispatch

- Read and transform item updates from primary database
  - Normalize text, augment with metadata, augment with additional inferences
- Reliable multicast
  - Publish updates to search service
  - Persist messages in intermediate data store for recovery
  - 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
  - Generate recommendations (items, products, searches, etc.)
  - Import third-party data (catalogs, currency, etc.)
  - Compute sales rank
  - Archive / purge deleted items
- Often drives further downstream processing through ***Message Dispatch***

Strategy 3: Automate Everything

- Prefer Adaptive / Automated Systems to Manual Systems

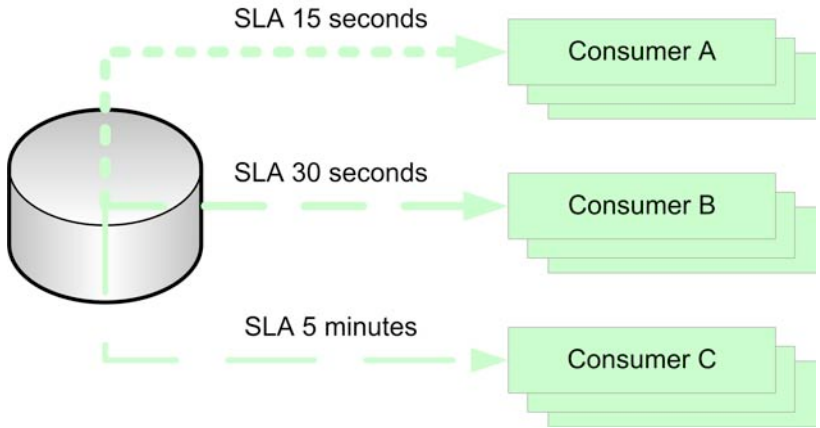
- Motivations
  - Scalability
    - Can scale with machines, not humans
  - Availability / Latency
    - Can adapt to changing environment more rapidly
  - Cost
    - Machines are far less expensive than humans
    - Can learn / improve / adjust over time without manual effort
  - Functionality
    -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

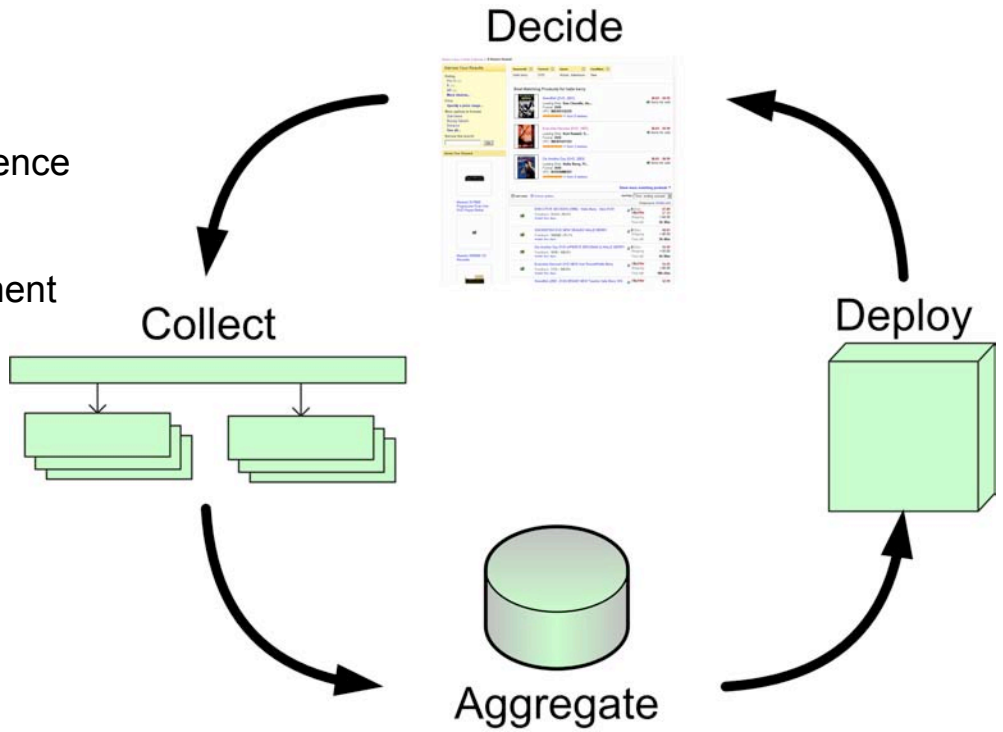
- Define service-level agreement (SLA) for a given logical event consumer
  - E.g., 99% of events processed in 15 seconds
- Consumer dynamically adjusts to meet defined SLA with minimal resources
  - Event polling size and polling frequency
  - Number of processor threads
- Automatically adapts to changes 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 and serve appropriate experience
- Best Practices
  - "Perturbation" for continual improvement
  - Dampening of positive feedback



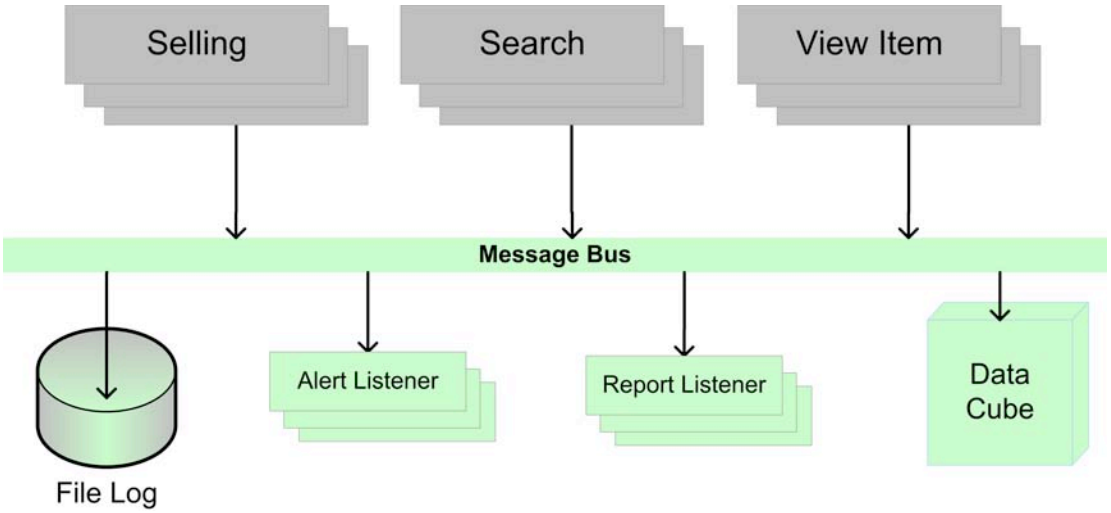
Strategy 4: Remember Everything Fails

- Build all systems to be tolerant of failure
  - Assume every operation will fail
  - Assume 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, especially 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
  - Transaction reports by application server pool, URL, database, etc.



- *Over 1.5TB of log messages per day*

Everything Fails: Code and Feature Deployment

Pattern: Rollback

- Absolutely no changes to the site which cannot be undone
- Code Deployment: Rollout / Rollback
  - Entire site rolled every 2 weeks: 16,000 application servers in 220 pools
  - Many deployments 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
- Feature Deployment: Wire-on / Wire-off
  - Every feature has on / off state driven by central configuration
  - Allows feature to be immediately turned off for operational or business reasons
  - Decouples code deployment from feature deployment
  - Applications can check for feature “availability” in the same way as they check for resource availability

Everything Fails: 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
  - Sends alert
- Non-critical functionality is removed or ignored
  - "Limp mode" operation
- Critical functionality is retried or deferred
  - Failover to alternate resource
  - Defer processing to guaranteed async message
- 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