CMPE 282 Cloud Services
Cloud-Native Application Design Patterns

Instructor: Kong Li

Content

- What and why
- Coffee Shop
  - Decomposition
  - Workload
  - Data/state
  - Component refinement
  - Elasticity and Resiliency
- Composite Cloud Applications

1

2

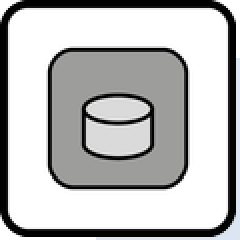
3

Coffee Shop App Design (3)

- Decomposition: How to distribute Application Functionality?
  - Distributed App
- Work load: What workload do components experience?
  - Static
  - Periodic
  - Once-in-a-lifetime
  - Unpredictable
  - Continuously Changing
- **Data (State): Where does the application handle state?**
  - **Stateful**
  - **Stateless**
  - **Strict consistency**
  - **Eventual consistency**
  - **Data Abstractor**
- Component Refinements: How are components implemented?
  - Message-oriented Middleware
  - User Interface Component
  - Processing Component
  - Batch Processing Component
  - Multi-component Image
- Elasticity and Resiliency
  - Elastic Load Balancer
  - Elastic Queue
  - Node-based Availability
  - Environment-based Availability
  - Watchdog

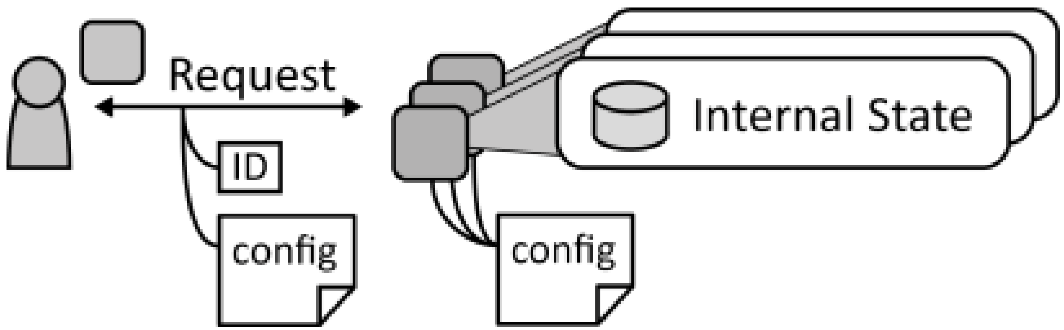
State = session state + app state

- **Session State** - *State of a client's interaction with an app*
  - Ex: customer's shopping card of an online store
- **App State -** *Data handled by an app*
  - Ex: customers shipping information stored by an app



Stateful Component

- Intent: Multiple instances of a scaled-out app component synchronize their internal state to provide a unified behavior

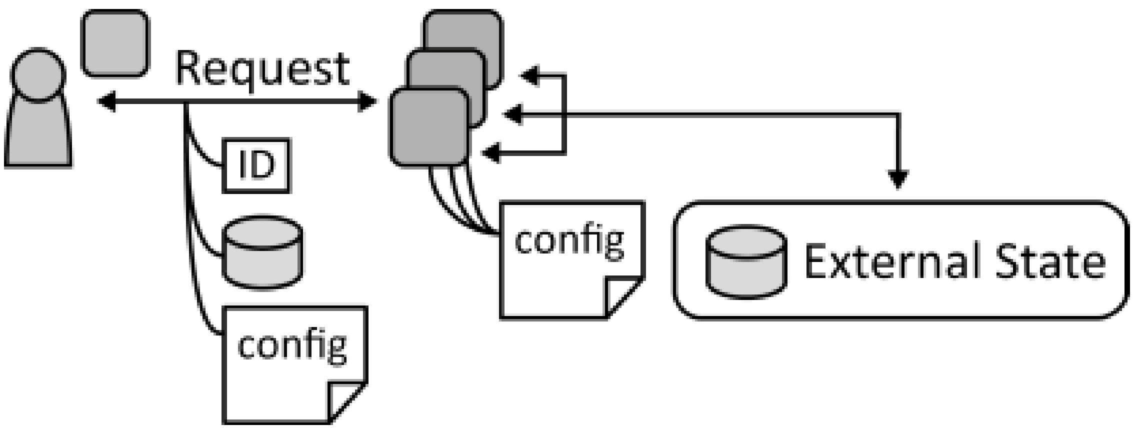


- Any storage offerings: block/blob storage, RDBMS, NoSQL, etc.

- Solution: The internal state maintained by app component instances is replicated among all instances
  - A config file stored centrally or config send by clients w/ every request
  - Internal state replication: strict consistency vs eventual consistency
    - Tradeoffs: consistency, availability, performance, durability

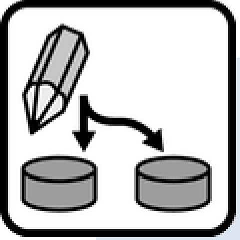
Stateless Component

- Intent: State is handled external of app components to ease their scaling-out and to make the app more tolerant to component failures
  - Isolation of component/resource failure



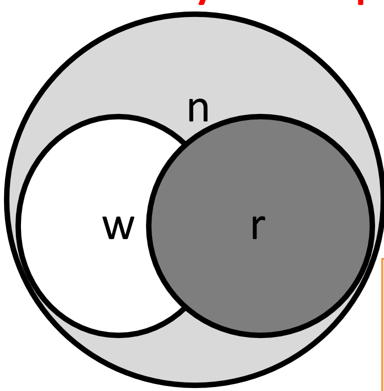
- Web services
- REST
- Shopping baskets in Amazon

- Solution: no internal state for app components. App state and config is stored externally in Storage Offerings or provided to the component with each request
  - Ex: Keep state on client-side in Web app, REST
  - Each request can be handled by arbitrary server

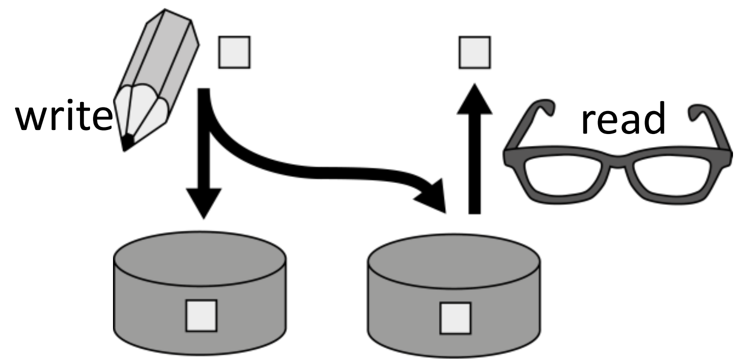


Strict Consistency

- Intent: Data is stored at different locations to improve response time and to avoid data loss in case of failures while consistency of replicas is ensured at **all** times



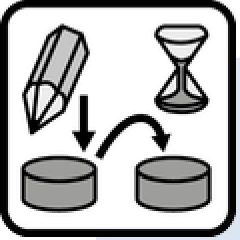
r + w > n
2*w > n



number of replicas (n) = 2
replicas accessed to write (w) = 2
replicas accessed to read (r) = 1

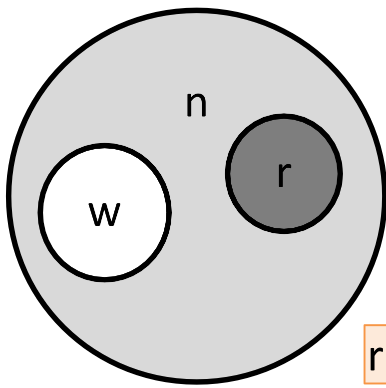
- Solution: Data replication
  - At least 1 replica w/ the latest data version is accessed during each op
    - Each read op: read >= r replicas
    - Each write op: write >= w replicas
    - **Quorum consensus protocol**: r + w > n and 2*w > n
  - Tradeoffs: consistency, availability, performance, durability
  - ACID (**A**tomicity, **C**onsistency, **I**solation, **D**urability)

- Many RDBMS, e.g., MySQL, MS SQL, DB2, etc.

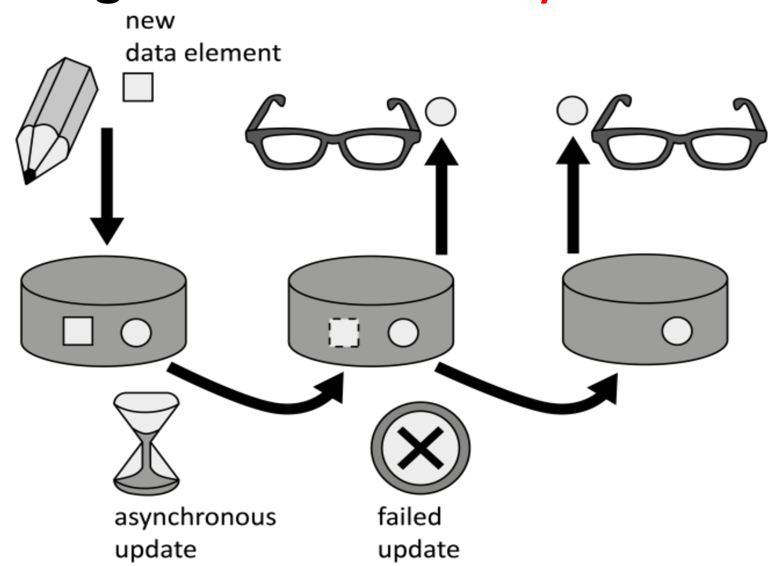


Eventual Consistency

- Intent: Performance and availability of data in case of network partitioning are enabled by ensuring data consistency eventually and not at all times

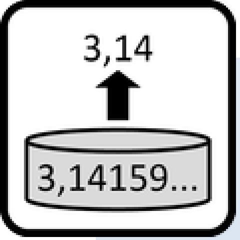


r + w ≤ n



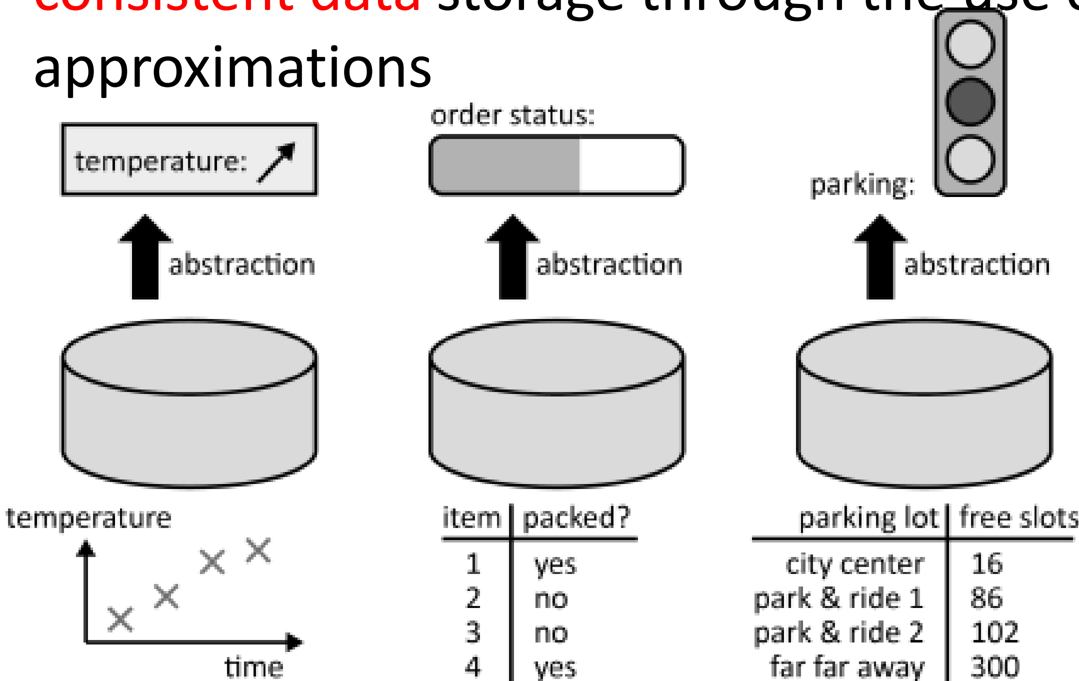
- Solution: trade consistency for availability (CAP)
  - reduces # of replicas to be accessed during read and write ops
  - modification eventually replicated to all replicas asynchronously (msg Q)
  - r + w ≤ n
  - BASE (**B**asically **A**vailable, **S**oft state, **E**ventual consistency)

- NoSQL: AWS DynamoDB, Cassandra, Riak, etc.



Data Abstractor

- Intent: Data is abstracted to inherently support eventually consistent data storage through the use of abstractions and approximations

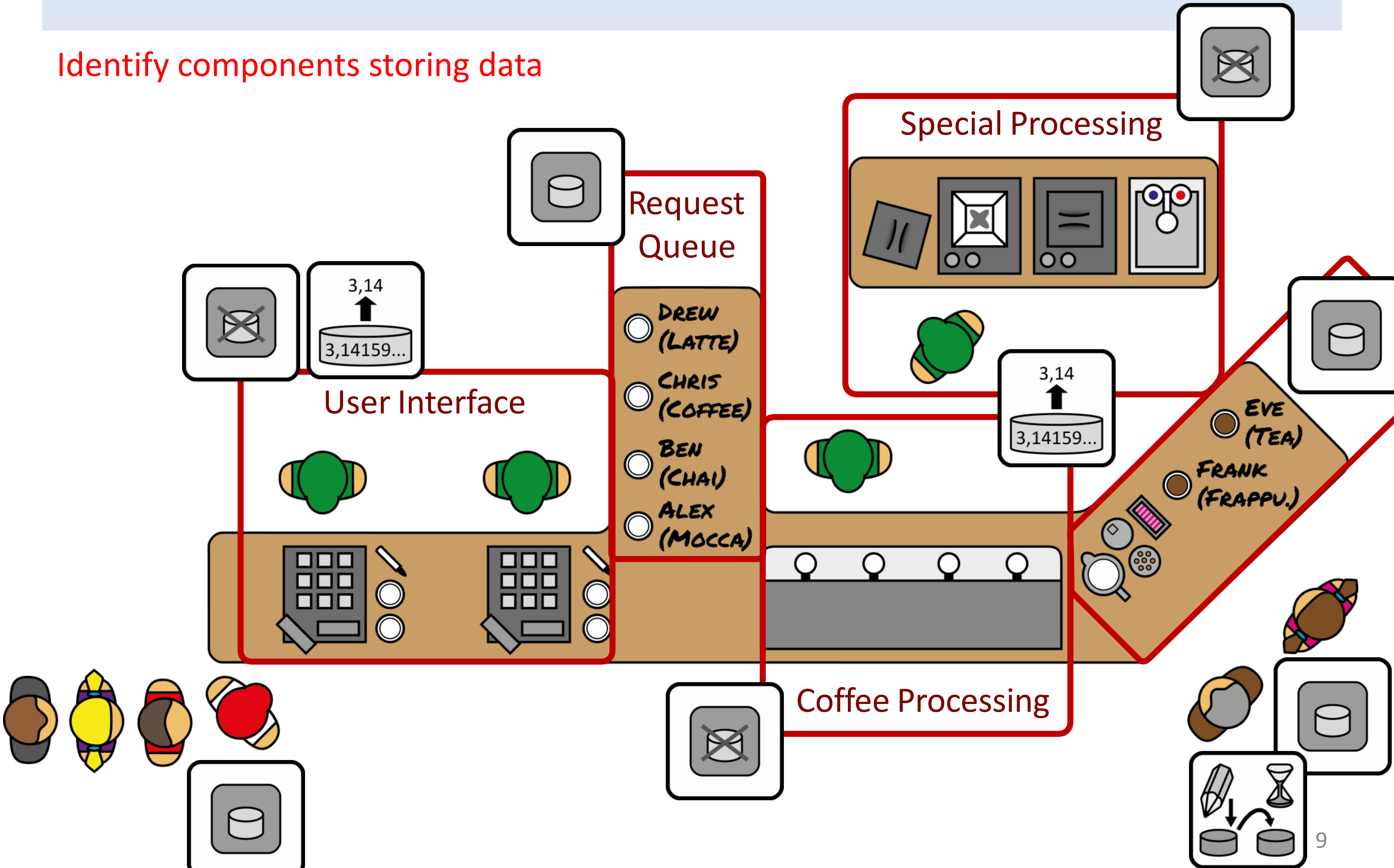


- SFPark.org
- Online stores: "in stock", "limited quantity", and "out of stock"

- Solution: data representation reflects that the consistent state is unknown by approximating values, or abstracting them into more general ones, such as progress bars, traffic lights, or change tendencies (increase / decrease)

Coffee Shop – Data (State)

Identify components storing data

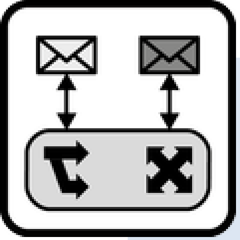


Lessons – Data (State)

- State should not be handled by components whenever possible
- Handle session state in
  - requests (has to be provided with every request)
  - provider-supplied storage and communication offerings
- “Lie” about state whenever possible / acceptable

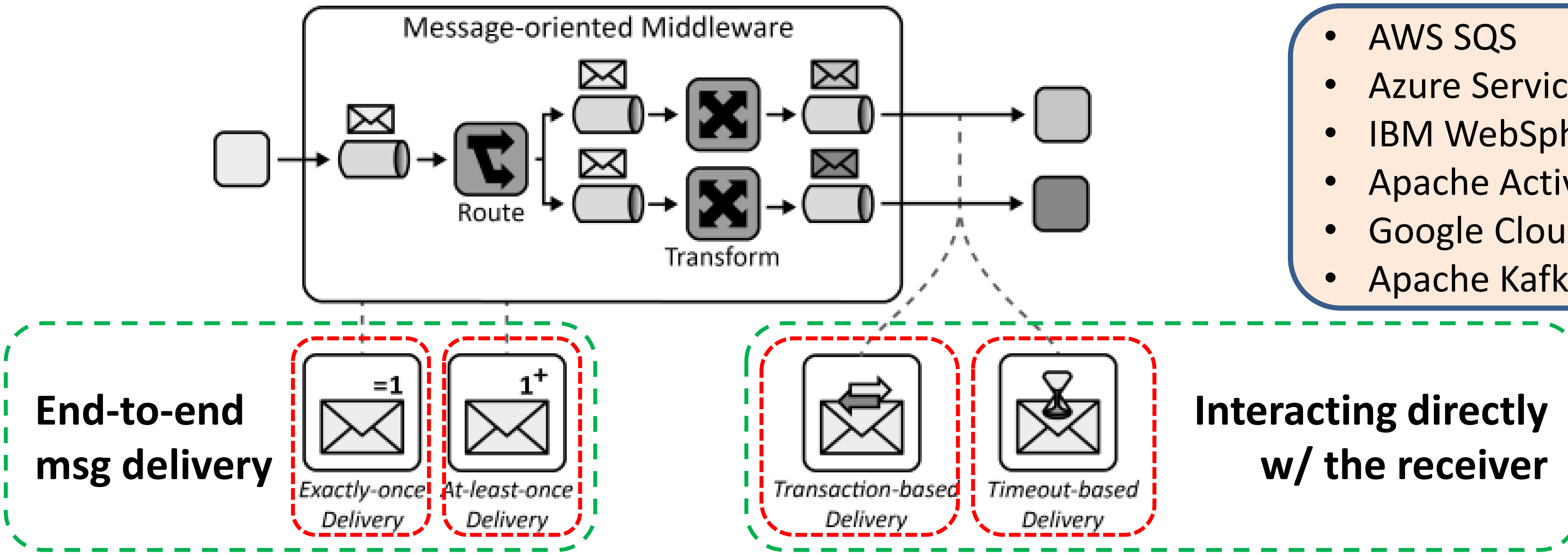
Coffee Shop App Design (4)

- Decomposition: How to distribute Application Functionality?
  - Distributed App
- Work load: What workload do components experience?
  - Static
  - Periodic
  - Once-in-a-lifetime
  - Unpredictable
  - Continuously Changing
- Data (State): Where does the application handle state?
  - Stateful
  - Stateless
  - Strict consistency
  - Eventual consistency
  - Data Abstractor
- **Component Refinements: How are components implemented?**
  - **Message-oriented Middleware**
  - **User Interface Component**
  - **Processing Component**
  - **Batch Processing Component**
  - **Multi-component Image**
- Elasticity and Resiliency
  - Elastic Load Balancer
  - Elastic Queue
  - Node-based Availability
  - Environment-based Availability
  - Watchdog



Message-oriented Middleware

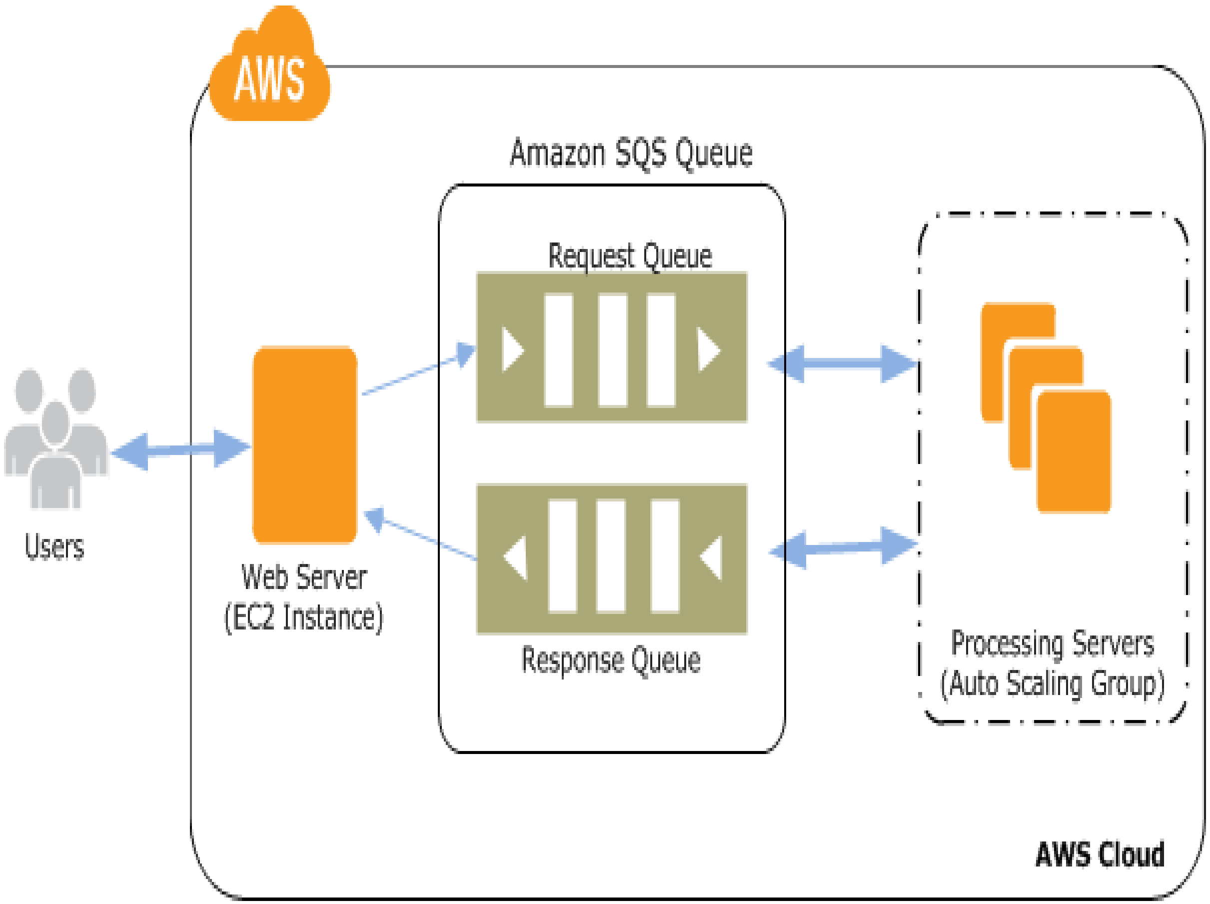
- Intent: Asynchronous communication is provided while hiding complexity of addressing, routing, or data formats to make interaction robust and flexible



- AWS SQS
- Azure Service Bus
- IBM WebSphere MQ
- Apache ActiveMQ
- Google Cloud Pub/Sub
- Apache Kafka

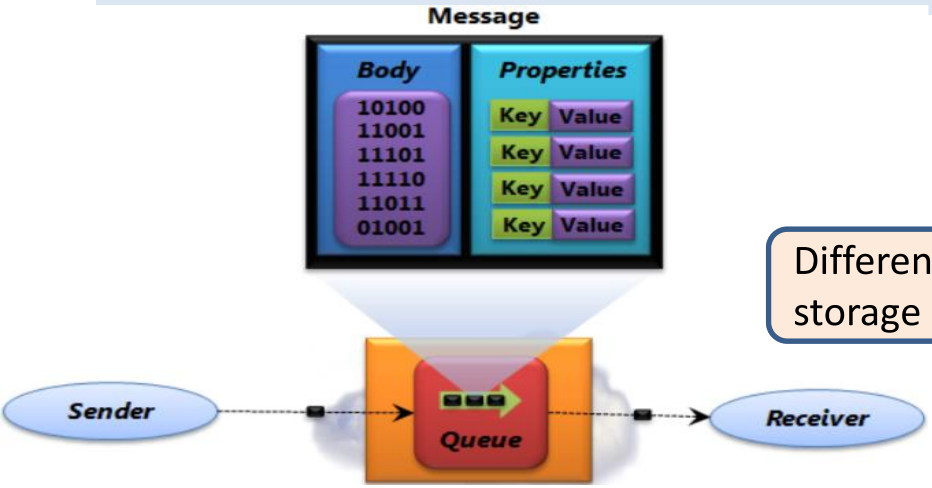
- Solution: message queue and pub-sub channel
  - Assumptions of communication partners reduced (***Loose Coupling***)
    - **Platform**: implementation language used
    - **Reference**: location of the communication partner (routing)
    - **Time**: communication partners are active at different time / speed
    - **Format**: message formats can change (transformation)

AWS - Simple Queue Service (SQS)



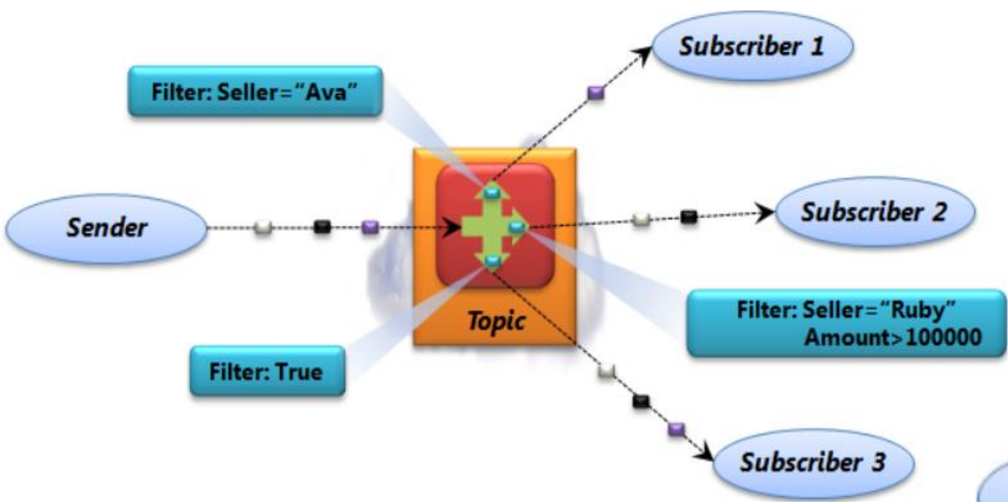
- Message queuing service - reliable msg delivery
- Msg redundantly distributed across SQS servers
- While a msg is received and being processed, it remains in Q and is not returned to subsequent receive reqs for the duration of the *visibility timeout*
- CloudWatch: dynamic scaling based on queue length

Azure - App Services - Service Bus

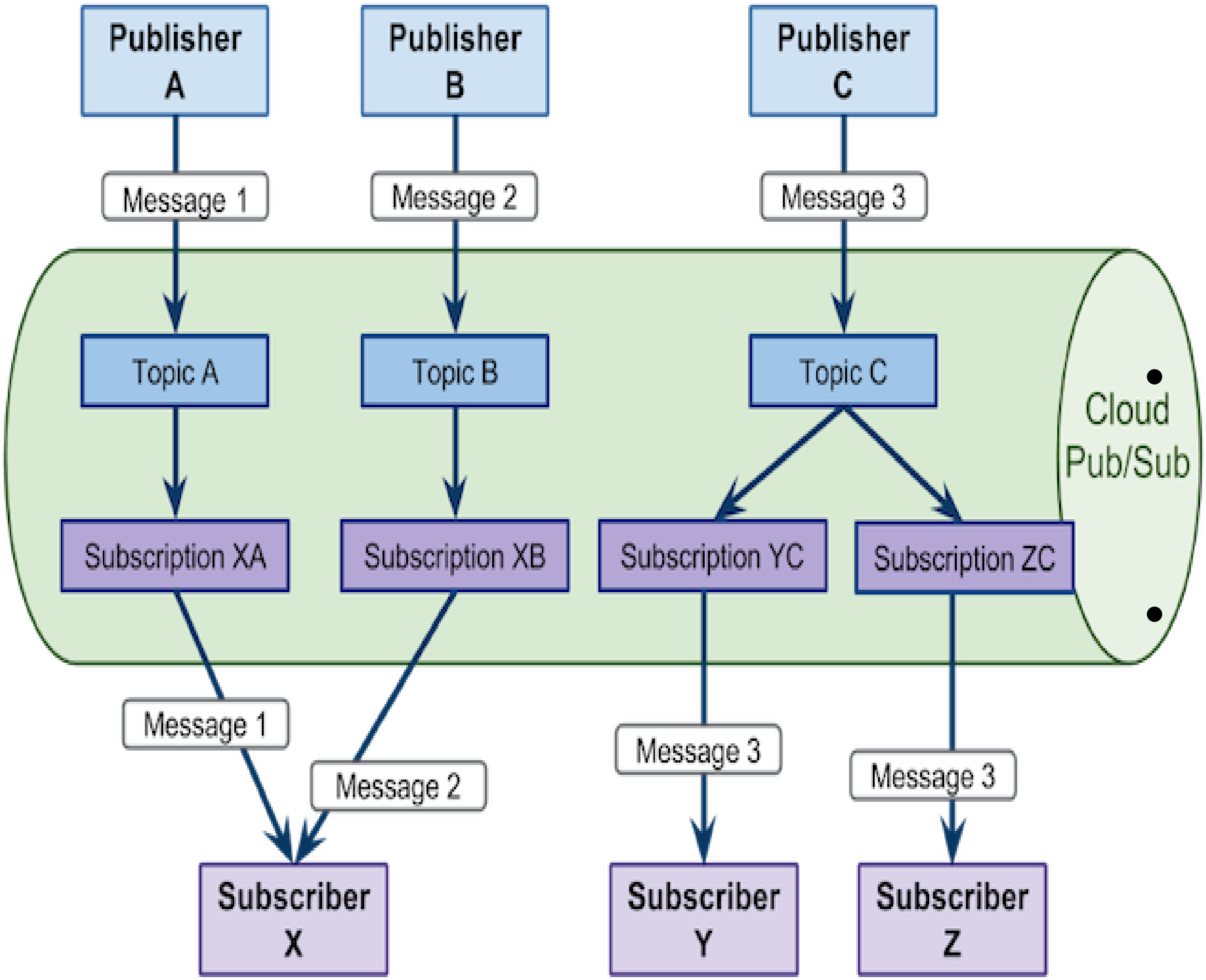


Different from Azure storage Queues

- Keep apps and devices connected across private and public clouds
- Communication among apps
  - Queues: 1 way, FIFO
  - Topics: 1 way, publish/subscription
  - Relays: 2 way
- Multi-tenant
- Apps can be any where



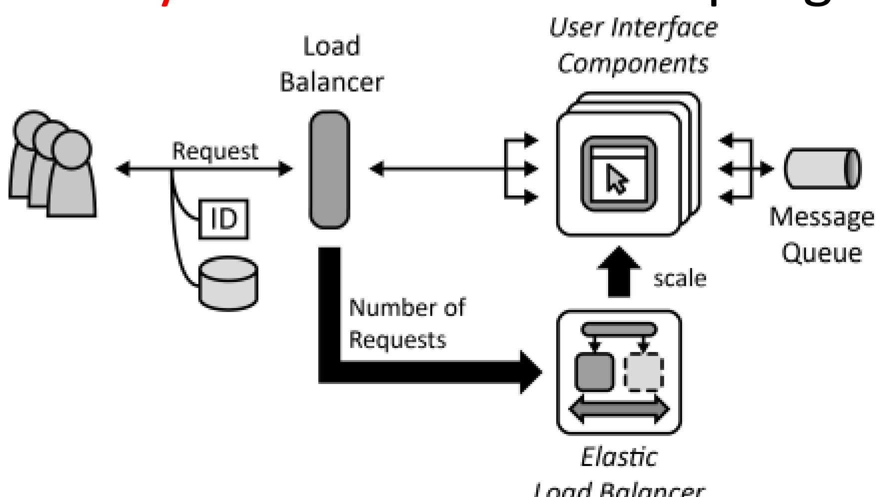
Google - Cloud Pub/Sub



- Message queue
  - Reliable, HA
  - scalability
  - Many to many
  - Async
- Publisher
  - From any apps via https
  - Topic
- Subscriber
  - Any apps
  - modes
    - Pull: https
    - Push: POST over https

User Interface Component

- Intent: Synchronous user interfaces are accessed by humans, while application-internal interaction is realized asynchronously to ensure loose coupling

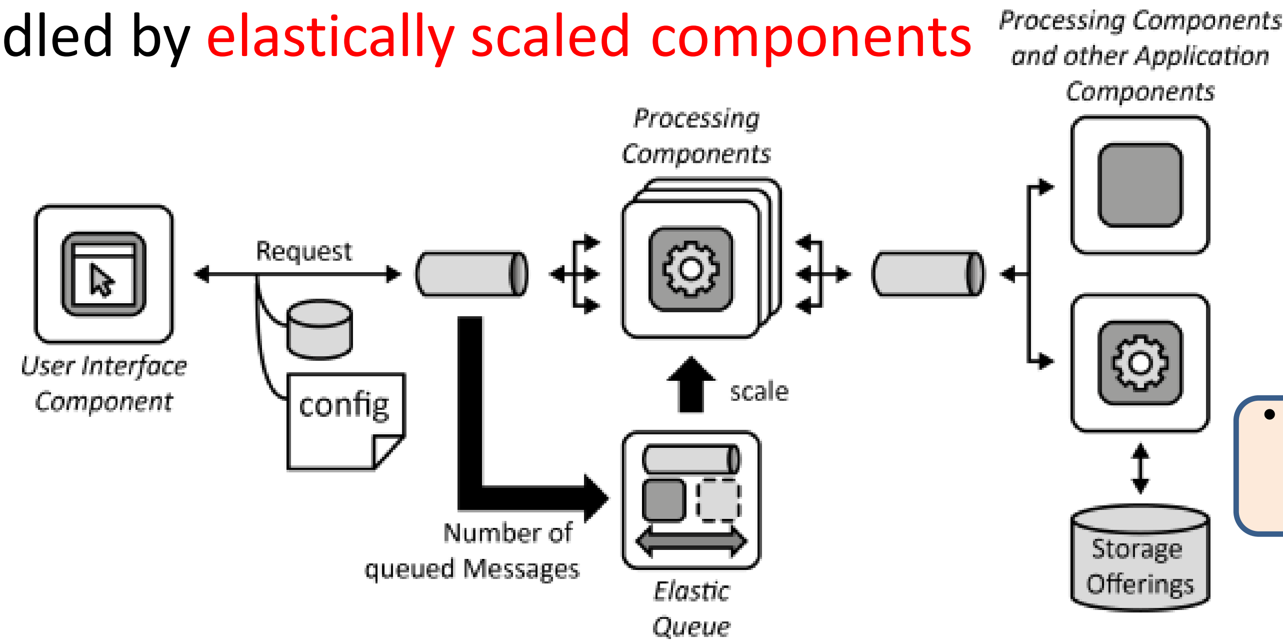


- Google Web Toolkit
- AJAX

- Solution: as a bridge b/w the sync access of the human user and the async communication used w/ other app components
  - State info held externally (Stateless Component pattern) - attached to requests, on user's device, or obtained from external storage
  - instances of UI Components scaled based on # of synchronous requests (Elastic Load Balancer pattern)

Processing Component

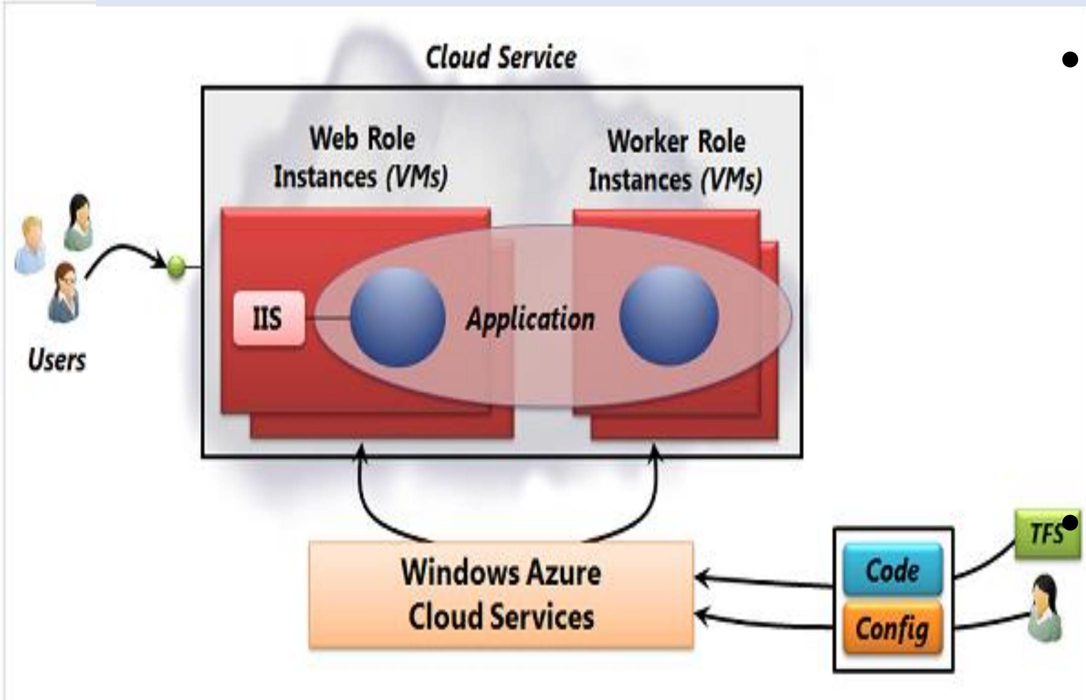
- Intent: Possibly long running processing functionality is handled by elastically scaled components

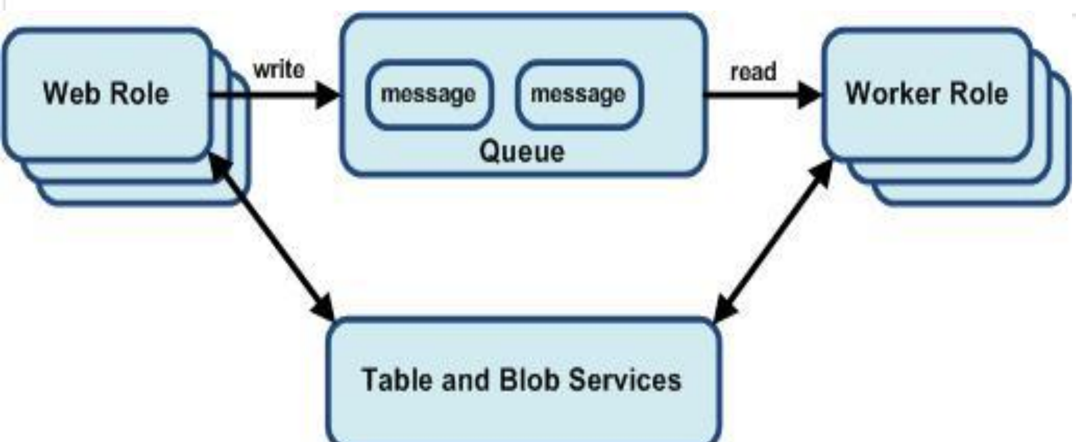


- Azure: worker role of "Web Services"

- Solution: Processing functionality split into separate function blocks and assigned to independent Processing Components
  - Each processing component: scaled out independently and stateless
  - Scaling is handled by an Elastic Queue
  - Data required for processing provided w/ reqs or by Storage Offerings

Azure - Compute - Cloud Services

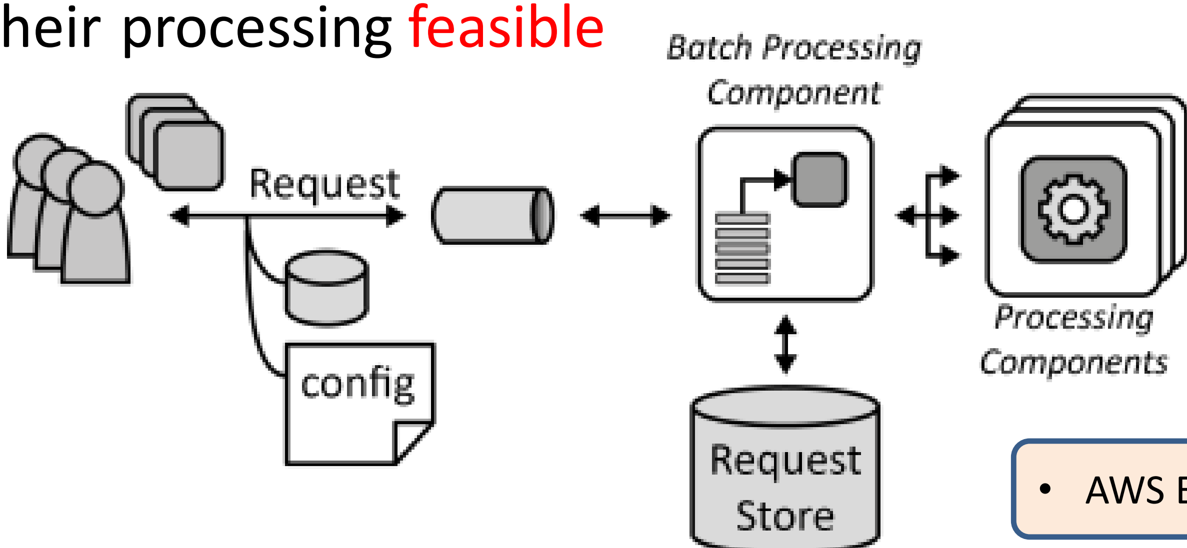




- PaaS for apps
  - VM roles: web role, worker role
  - Monitor: failure detection (apps, VM, host) & restart
  - Auto scale, load balancing
  - HA & FT: VM pool
- Cloud service apps
  - Do not create VM directly (PaaS)
  - User deploy apps and specify # of web/worker roles
  - apps state: only in SQL DB, blobs, tables, etc. Not in file systems of VM (why?)

Batch Processing Component

- Intent: Requests are delayed until environmental conditions make their processing feasible



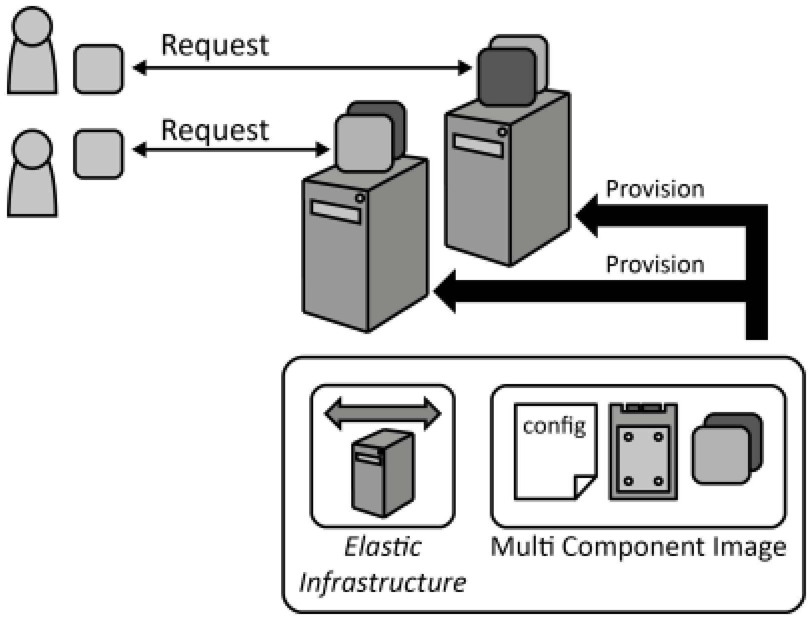
How to keep users happy?

- AWS EC2 Spot Instances

- Solution: Async processing reqs accepted at all times, but stored them until conditions are optimal for their processing
  - Based on # of stored requests, environmental conditions, and custom rules, components are instantiated to handle the requests
  - Requests are only processed under non-optimal conditions if they cannot be delayed any longer

Multi-Component Image

- Intent: Virtual servers host multiple application components that may not be active at all times to reduce provisioning and decommissioning operations



vs single image, tradeoffs?

- VM templates: VMware, AWS EC2, Azure, GCE
- Containers: Docker, AWS ECS, Google Container

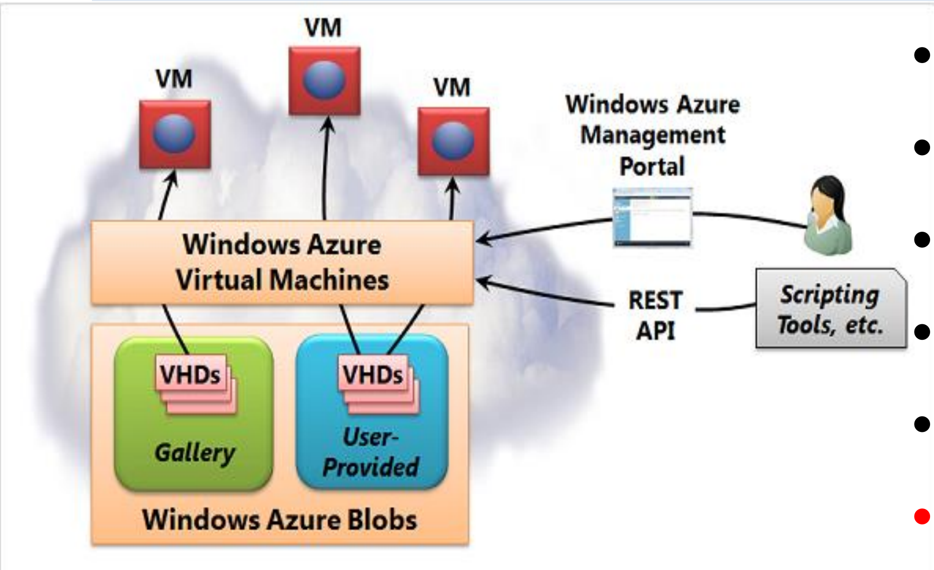
- Solution: Multiple app components (possibly including middleware) are hosted on virtual servers to ensure that running virtual servers may be used for different purposes w/o making provisioning or decommissioning operations necessary
  - Challenges: optimize distribution of app components among images

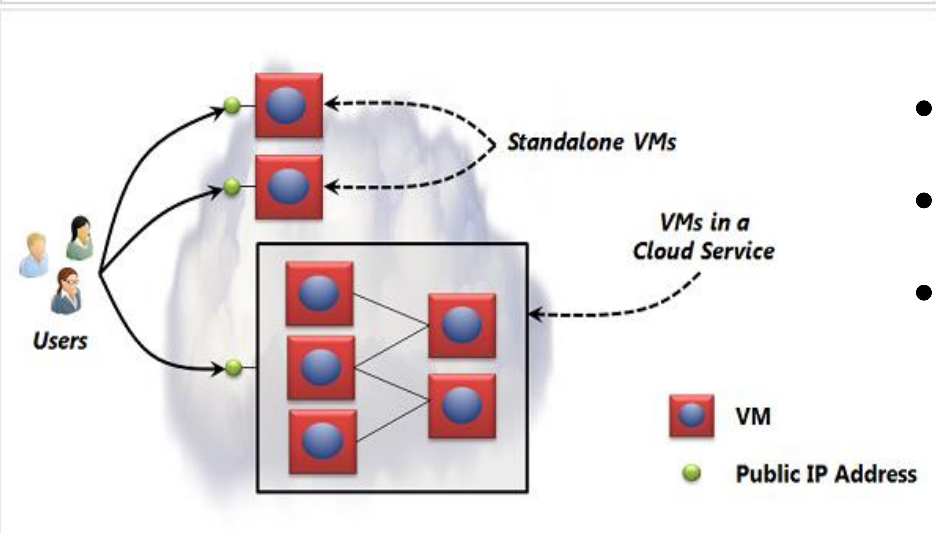
AWS - Elastic Computing (EC2)

- IaaS: HVM (Xen)-based VM
- Various instance types
- OS : Linux, Windows, etc.
- Create EC2 instance
  - Select or create an Amazon Machine Image (AMI)
  - Config security, network
  - Choose instance type
  - Decide locations, static IP, etc
- Access EC2 instance
  - DNS name
  - private IP: internal
  - public IP: external, dynamic
  - NAT: internal ←→ external

- Elastic IP addr: static, $
  - not released when instance is stopped or terminated
  - must be released when not needed
- EBS snapshot: FS volume only; no VM memory; VM up or down
- No live migration, no hot clone
- EC2 SLA (monthly uptime)
  - < 99.95%, > 99%: 10% service credit
  - < 99%: 30% service credit
- UI: AWS Management Console
- AWS SDK libraries and toolkits
  - Java, PHP, C#, etc
  - SOAP (WSDL), REST

Azure - Compute - Virtual Machine



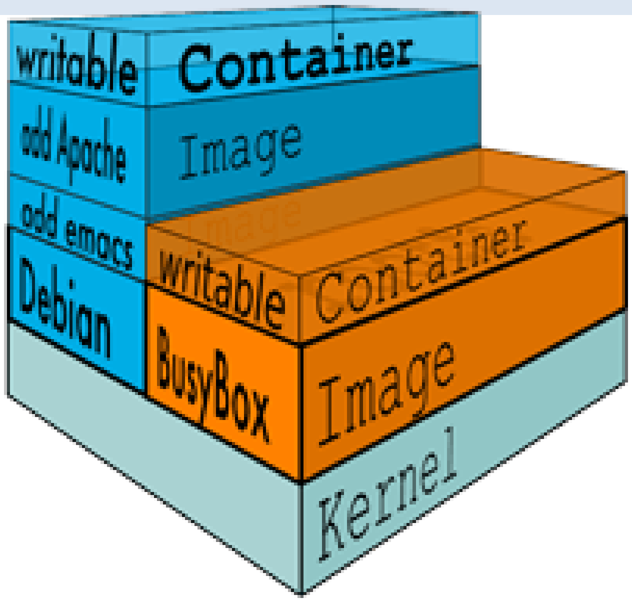


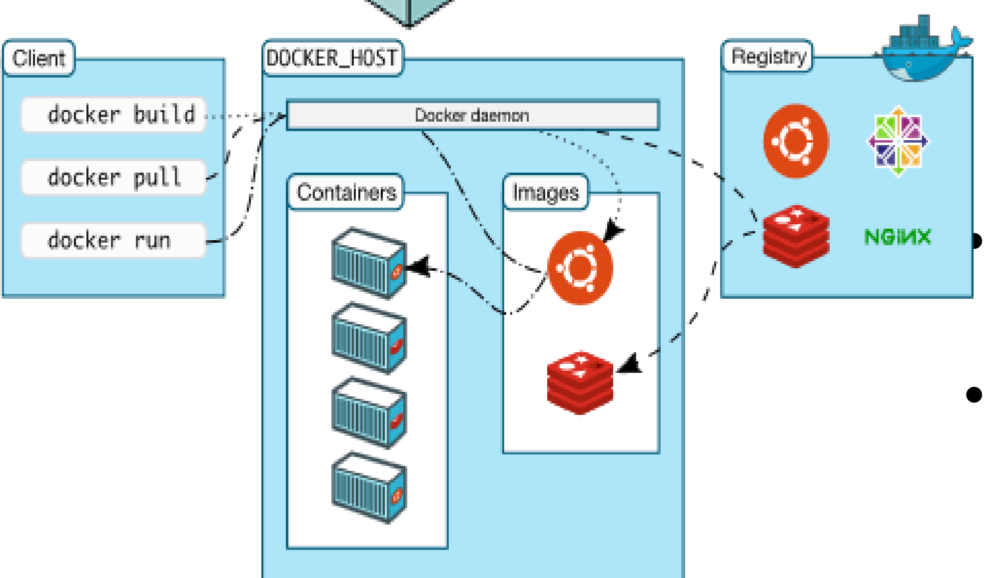
- IaaS
- OS types: Windows, Linux
- Image: portal image, custom
- File: .VHD
- Connect to VM: VPN, ExpressRoute
- Load balancing multiple VMs
  - DNS level, network level
- Scalability: Scale Sets - autoscale
- HA: Availability sets - redundancy
- Grouping VMs as a service (multi-tier apps): Single public IP
  - Load balancing + Scalability + HA

Google Compute Engine (GCE)

- IaaS: KVM-based VM
- OS: various Linux, Windows
- Machine types
  - Standard, small
  - High CPU, high memory
- Instance template
- Load balancing: Replica pool
  - Network, HTTP (cross-region)
- Instance group: pool of VMs
  - Size up/down
- Autoscaler: managed instance group
  - Network, HTTP (cross-region)
  - CPU util%, Cloud metrics
- Authorizing to/from other Google service: OAuth 2.0
- Resource: global / region / zone
  - Zone: isolated location in a region
  - Region-specific resources can be accessed by resources in the region
  - Isolation of failure, redundancy
- Disk snapshot: FS only; no VM memory; VM up or down
- no VM hot clone
- Host maintenance policies
  - [default] VM live migration
  - Terminate and (optionally) restart
- VM crash: [default] auto restart

Docker

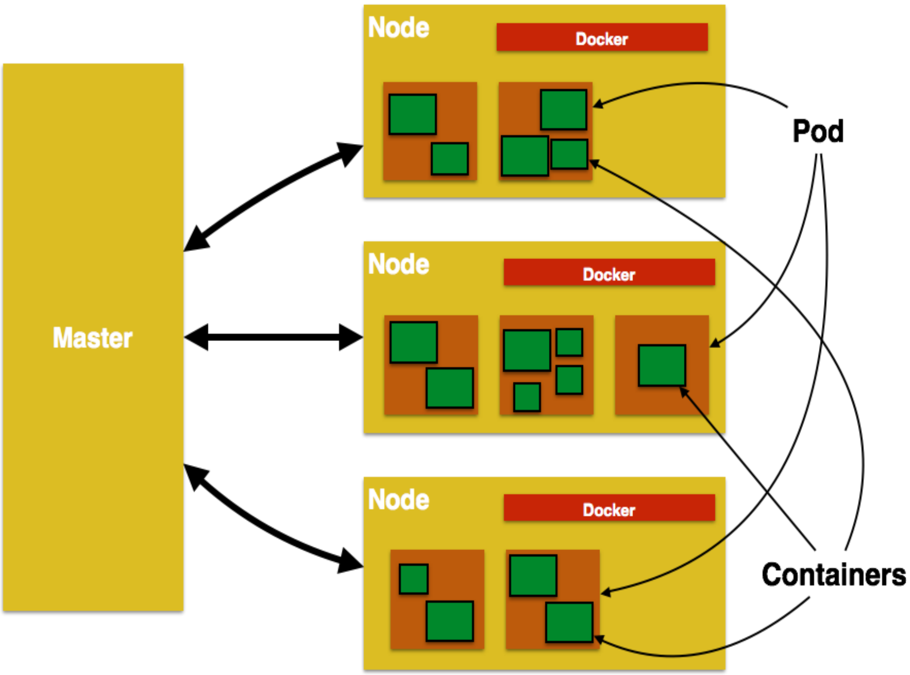




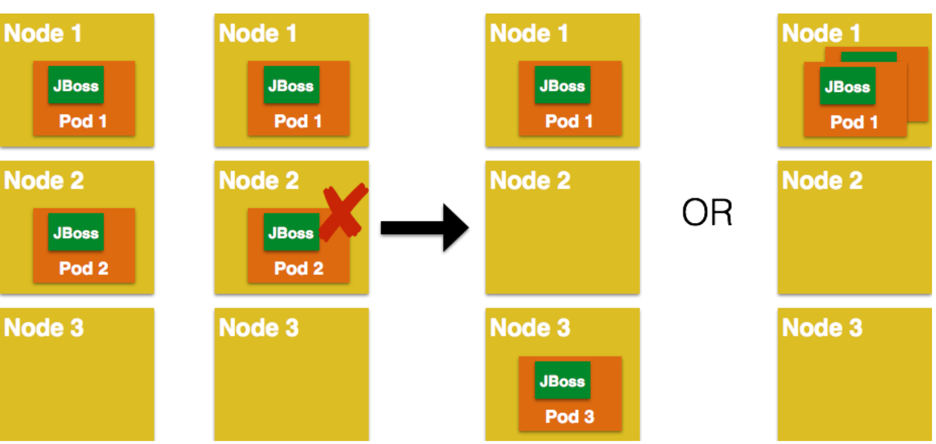
- Docker Engine
  - Based on Linux container LXC
  - Images (left)
    - Union File system: Multiple layers of FS + writable layer
  - Share/deploy faster, scale easily
  - Docker Hub, Docker Registry (left)
- Built-in orchestration – Swarm
  - cluster, scheduling, placement
  - Integrated or standalone: since v1.12
- Docker Machine: auto provision hosts across multi-platforms
- Docker Compose: define multi-container apps

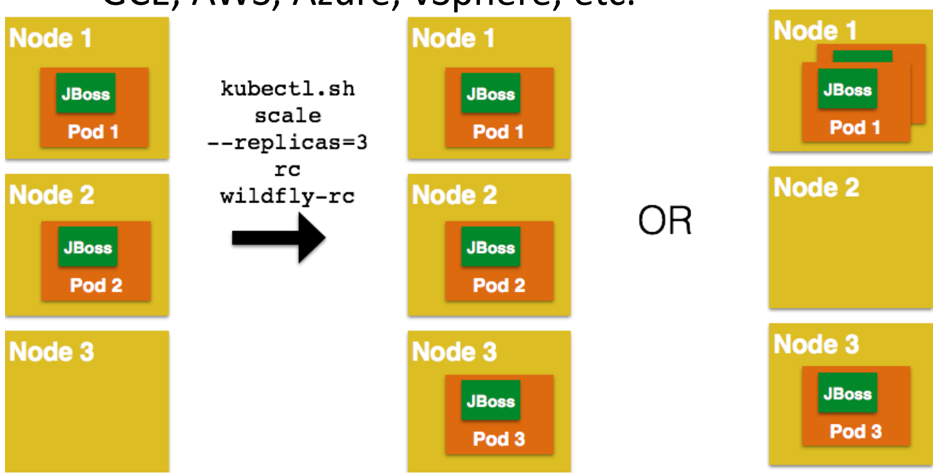
Container Engine

Google Container Engine (GKE)

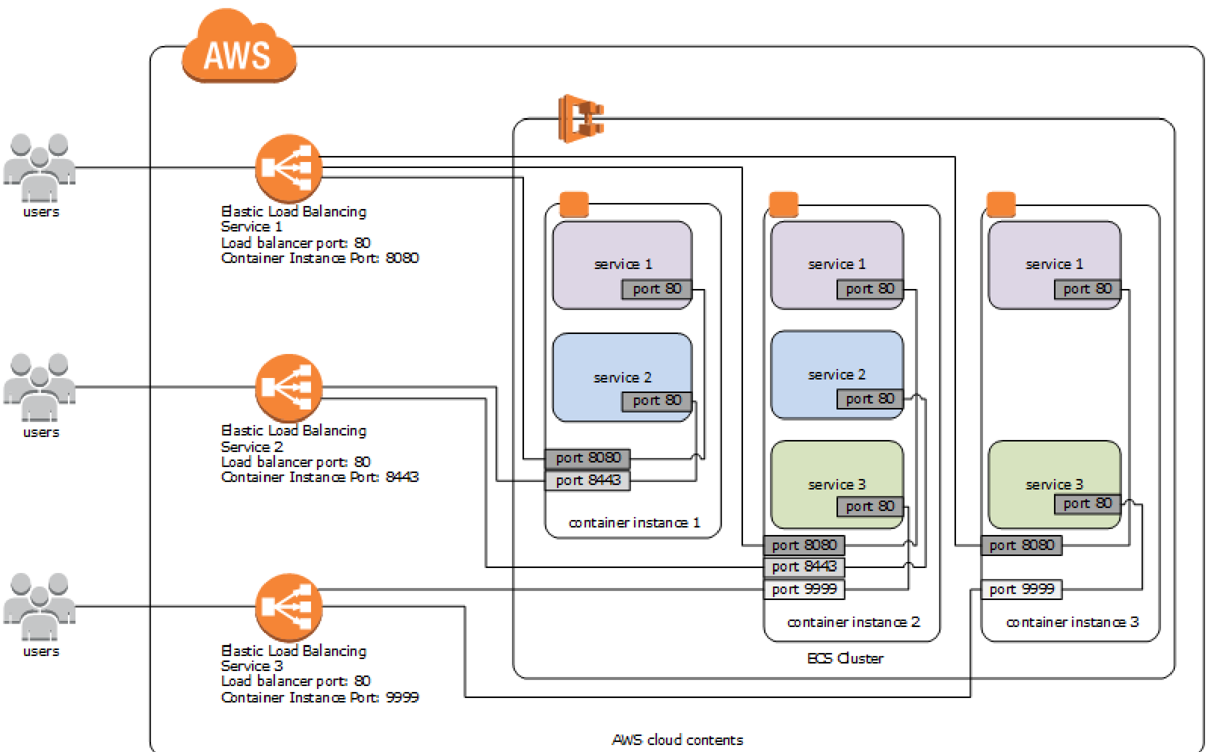


- Docker support
- Run on a cluster of GCE (VMs)
- Kubernetes: managing containerized apps among multi-hosts
  - Logically organize containers into groups
  - Replication: optimal # of pods
    - Rescheduling pods (lower left)
    - Scaling pods (lower right)
  - Cluster Auto-restarting
  - Docker & rkt support
  - GCE, AWS, Azure, vSphere, etc.





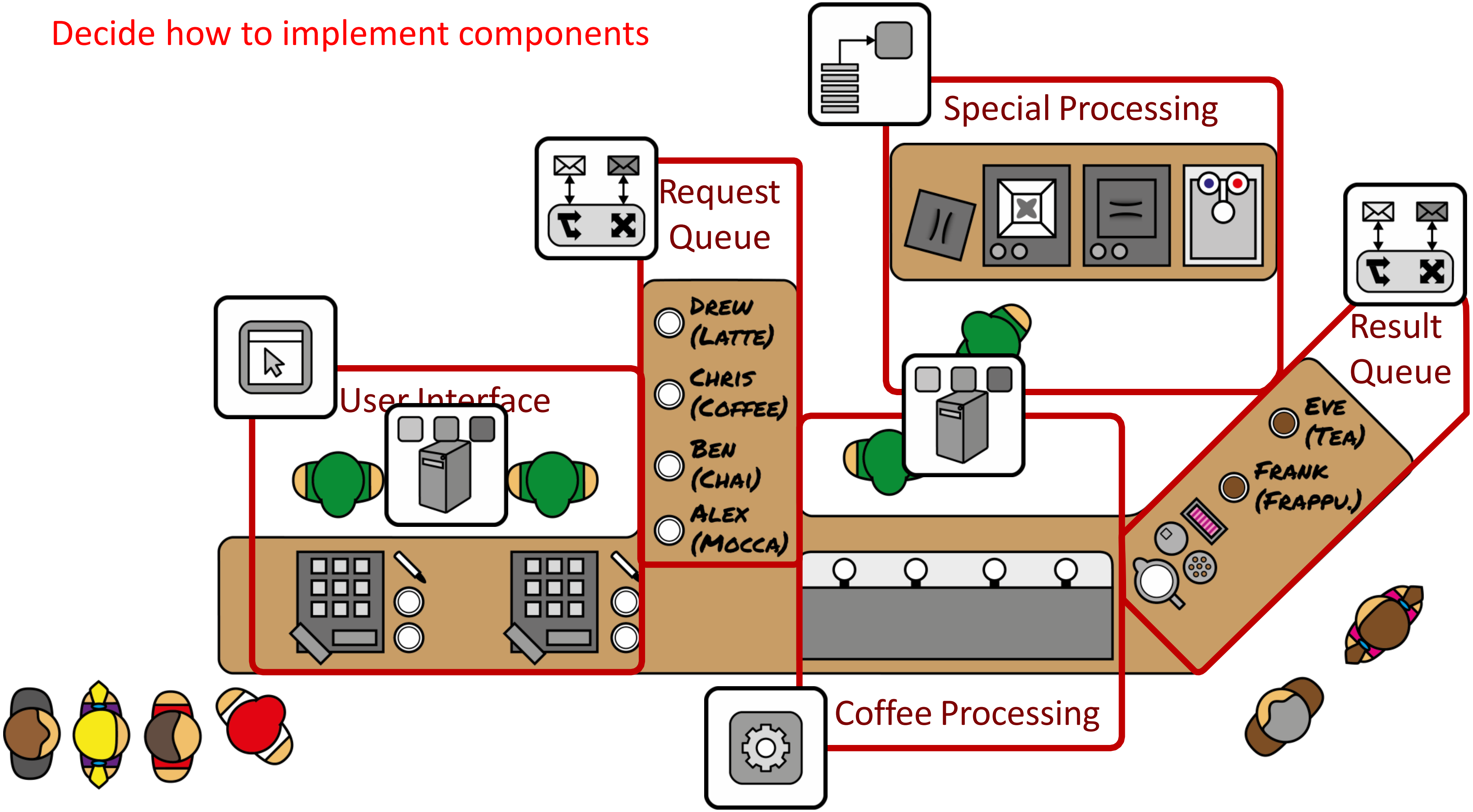
AWS EC2 Container Service



- Docker support
- Across a cluster of EC2 instances
  - App span multiple AZs
  - Placement mgmt: auto or manual control
  - Cluster mgmt and config mgmt
  - Scheduling: balance between resource needs and availability requirements
  - Integrated w/ ELB (left)
- Docker workload can be migrated to/from AWS

Coffee Shop – Refinement of Components

Decide how to implement components



Lessons – Component Refinement

- Components should be integrated with messaging to ensure loose coupling
- Resources supporting functional components should be flexible (multi-component image)

References

- http://www.cloudcomputingpatterns.org/
- Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter, *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer; 2014
  - http://www.springer.com/978-3-7091-1567-1
- The coffee shop example is adapted from
  - https://indico.scc.kit.edu/indico/event/26/session/1/contribution/12/material/slides/0.pdf
  - http://www.sei.cmu.edu/library/assets/presentations/retter-saturn2013.pdf