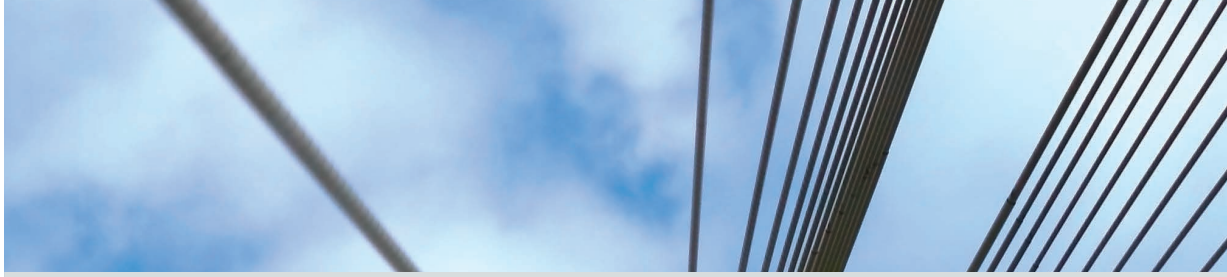

RADical Strategies for Engineering Web-Scale Cloud Solutions

Rohit Ranchal, Ajay Mohindra, and Justin G. Manweiler
IBM

Bharat Bhargava
Purdue University

No clear guidelines exist for designing and deploying cloud solutions that can handle billions of requests per day. This article reviews industry best practices and identifies principles for operating applications at Web scale.

Properly leveraged, the cloud allows organizations to deploy applications at unprecedented scale, robustness, and cost effectiveness, without the upfront investment in permanent IT infrastructure. Proponents cite well-known services such as Netflix, Reddit, Pinterest, and Zynga as examples of technological and economic success. However, the number of such wild successes attributable to cloud enablement is dwarfed by the many legacy applications as well as new applications built to legacy designs. Because these aren't fully cloud aware, they can't wholly exploit the potential advantages of cloud deployment.

Traditional applications run on the cloud much like they ran in the datacenter, with the primary enhancement being that billing for IT accrues uniformly across time. The most effective designs, however, leverage the conceptually infinite scalability and elasticity enabled by infrastructure as a service (IaaS) to satiate the inherent dynamism of the application workload. Multiple forms of dynamism exist, such as in the application's design, development, testing, and enhancement; in the interplay between scale and responsiveness across peaks in demand; in availability across unplanned network latencies and hardware outages; and in unforeseen external interactions across applications. The most complete cloud architectures leverage the programmability of cloud environments together with fundamentally scalable and ro-

 2325-6095/15/$31.00 © 2015 IEEE

bust software architectures. Ultimately, such strategies can enable a best-in-class user experience at a dramatically reduced IT cost point relative to traditional approaches. Any expectation of achieving true Web scale (that is, handling billions of requests per day) demands such an architectural rethink.

Netflix, Reddit, Pinterest, Zynga, and others have validated the premise that cloud computing, when properly leveraged, can be an effective strategy to achieve Web scale.[1] Netflix, in particular, has achieved 50 percent or more cost savings through dynamic load optimization—that is, predicting load and scaling to meet demand just in time.[2] However, not all cloud migration stories are positive.

Although major cloud environments are typically robust, occasional outages have made the hosted services unavailable, yielding substantial financial and customer losses.[3] To ensure continuous service availability, services hosted in the cloud must build robust architectures that can accommodate failures. Furthermore, a Web-scale design for the cloud requires new architectural thinking to exploit cloud capabilities. For instance, Web-scale solutions must be highly distributed and partitionable. Accordingly, given partitions, the architect can choose either availability or consistency (CAP theorem constraints[4]). Typical Web-scale solutions must prefer high availability and partition tolerance, sacrificing consistency, so new designs are required to relax consistency requirements (eventual consistency) when possible. Legacy applications can't be trivially migrated and scaled.

Despite the successes of Netflix and others, guidance on building and operating Web-scale services in the cloud that meet individual business requirements is lacking (see the related work sidebar). We seek to ease this burden, particularly for the traditional application engineer. We address the challenge of designing high-quality cloud solutions using preestablished building block cloud capabilities by defining a suite of best practices for their composition. We consider successful architectures of cloud-deployed Web-scale solutions to understand what has worked at the grandest scale. We use Netflix as our prime study candidate because its architecture is well-documented; the company has open sourced many of its architectural components under Netflix OSS; and many well-known companies (such as Yelp, Yahoo, and Coursera) have adopted these components in their solutions.[2] From our study, we identify important Web-scale principles and derive key strategies that enable each principle. The real challenge is in composition: we've found limited guidance for how to build and operate a new Web-scale cloud service or retrofit an existing one. Our primary contribution is in applying each established principle and related techniques to a new business service (a seemingly straightforward consumer ratings service). Through example, we expect that other cloud architects can apply similar methods to deliver Web-scale services. Our intention isn't to provide a comprehensive list of cloud patterns but to identify recipes that enable Web-scale operation in cloud-hosted solutions.

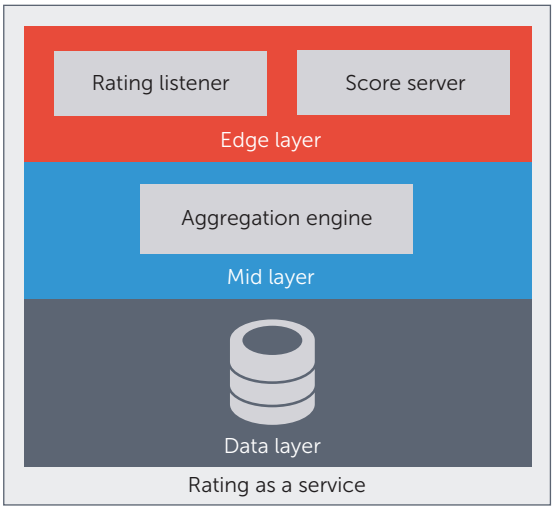

FIGURE 1. The multitier rating-as-a-service (RaaS) architecture. Online marketplaces can use RaaS to outsource the collection of customer feedback, aggregation of feedback data, and presentation of aggregated scores.

Design Principles and Patterns

Ratings are widely used in catalog-based services (Amazon, IMDB, Yelp, and so on) to collect consumer feedback. Consumer rating as a service (RaaS), which can be used to add consumer feedback functionality to an e-commerce platform, offers on-demand collection of consumer ratings, transparent aggregation, and presentation of aggregated scores for registered offerings[5]. Figure 1 shows the different tiers of the RaaS architecture:

RELATED WORK IN CLOUD DEVELOPMENT PATTERNS

As cloud technologies mature and practitioners gain cloud development experience, there are ongoing efforts to create patterns for development in the cloud. These patterns build on new cloud concepts and reinforce various traditional datacenter practices. Prior works by cloud providers (such as Amazon[1] and Microsoft[2]), cloud consumers (such as Netflix[3]), and academic researchers[4] have identified patterns for cloud design and architecture, described methods for addressing specific challenges in the cloud, and created guides on best practices for designing cloud-hosted applications. However, the guidance for Web-scale cloud solutions is scattered and vendor-specific as providers have different services, capabilities, and abstractions. We focus on provider-independent specific cloud patterns that are fundamental to engineering Web-scale cloud solutions. Contrary to the existing guidance, we take a top-down approach that allows a solution designer to map the selected high-level Web-scale principle to low-level patterns and cloud capabilities that enable the principle in the solution.

References

1. "Cloud Design Patterns," Amazon Web Service, Nov. 2013; http://en.clouddesignpattern.org/index.php/Main_Page.
2. A. Homer, *Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications*, Microsoft Patterns & Practices, 2014.
3. A. Cockcroft, "Cloud Native at Netflix: What Changed?" July 2013; www.slideshare.net/adrianco/netflix-what-changed-gartner-catalyst.
4. C. Fehling et al., *Cloud Computing Patterns*, Springer, 2014.

- an *edge layer* handles client requests for registration, incoming ratings, and aggregated scores;
- a *middle layer* contains the aggregation logic to process incoming consumer ratings and produce aggregated scores; and
- a *data layer* consists of the database management system that's used to store consumer ratings and aggregated scores.

A cloud-based deployment doesn't automatically enable a solution to handle Web-scale workloads. To achieve Web scale, cloud solutions need to follow RADS principles—that is, resiliency (R), high availability (A), continuous delivery (D), and elastic scalability (S). A solution can realize these principles using strategies such as redundancy, replication, and isolation, as well as cloud capabilities such as autoscaling, load balancing, and monitoring. Although most cloud providers presently offer some capabilities, there's a significant variation in support of the identified architectural components across cloud offerings. Furthermore, components alone aren't sufficient. Guidelines are needed for composing components to exploit the cloud and enable RADS principles. These are identified as emerging best practices suitable for Web-scale cloud solutions regardless of their structure and intent. We describe the step-by-step application of these patterns to RaaS and show the respective changes in its architecture, categorizing architectural changes as

- *components*, which utilize an "is-a" relationship (for example, the edge layer is-a microservice);
- *services*, which utilize a "use" relationship (for example, edge services use autoscaling); and
- *designs*, which utilize a "has" relationship (for example, an edge service has immutable deployment).

Similar relationships have been used to represent cloud patterns for interoperability.[6]

Continuous Delivery

Web-scale solutions follow the principle of continuous improvement. They use quick-release cycles as new features are added, technologies are updated, and components grow. This requires automated, reliable, and quick deployment of solution updates to production. A continuous-delivery pipeline allows frequent updates with fewer changes per deployment, which reduces the deployment dependencies and risk of deployment problems and makes it easier to prioritize and resolve deployment issues. Netflix, for instance, follows certain patterns to ensure reliable continuous delivery.[2]

Microservice architecture. Traditional Web solutions have a tightly coupled monolithic architecture that's difficult to maintain due to dependencies in development and deployment.[7] This architecture is unsuitable for large, complex, highly evolving Web-scale cloud solutions because the components have different update cycles and resource requirements. In addition, dependencies interlock with selected technologies, causing obstacles for continuous delivery. Therefore, Web-scale solutions must be modeled using a microservice architecture that decomposes the solution into smaller components based on separate business logic and independent implementation. Each component is modeled as a microservice with a standard API. This allows services to be closely matched with individual requirements, independently developed, and transparently deployed while maintaining a stable API for consumers. A microservice architecture eases maintenance by reducing dependency and complexity in the delivery pipeline (also known as DevOps). Well-known Web-scale solutions such as Amazon have documented the transition from a monolithic to a microservice architecture.[8]

Deployment automation. Because Web-scale solutions have a large and growing infrastructure, they use automation in every step of the delivery pipeline to make deployment seamless and ensure consistency in managing the updates and pushing them to production. Automation enables repeatability; it allows commits that pass testing to be automatically deployed to production and thus helps in quickly releasing new features as they're developed.

Selective delivery. The goal here is to identify and evaluate the changes that increase an outcome of interest (for example, banner click rate or shopping cart drop-off rate). This task is performed in a controlled group, with updates deployed incrementally for a selected set of users to understand the impact and usefulness of changes (also known as A/B testing). After evaluation, the changes are universally deployed or reversed.

Immutable deployment. The goal of this pattern is to deploy changes as rolling upgrades and maintain multiple versions of an implementation. The newer version is deployed on new servers (called canary clusters) without any modification to the current stable version. Both the current and the new version are monitored in parallel through peak load (that is, canary testing) with a weighted traffic distribution that allocates only a small portion of traffic to the new version. If successful, the updates are automatically deployed across the entire solution, and the deployment undergoes red/black testing, where all the traffic is switched to the new version and the current version is disabled. If successful, the current version is unprovisioned and the new version becomes the current stable version. If a failure occurs, the traffic is redirected to the current version. Thus, immutable deployment helps achieve zero downtime and is an easy way to roll forward and roll back.

Centralized configuration management. Web-scale solutions have multiple running instances of components (for example, servers and databases) that share several configuration properties (for example, URLs of dependencies or database connections), but individual configuration files are limited to a single instance. Specifying and managing local configurations for each instance separately leads to administrative overhead and could result in configuration inconsistencies during updates. Local configuration changes might also require redeployment of the instances, resulting in downtime. Centralized configuration management (CCM) provides the ability to dynamically store, retrieve, and enable configuration properties across different components and their instances without needing to redeploy. It also supports multiple versions of configuration properties for different contexts (such as development, testing, and production).

RaaS architecture with continuous delivery. Figure 2a shows the updated RaaS architecture to enable continuous delivery. RaaS components are decomposed into three microservices based on their functionality:

- The *edge* service receives registration requests and consumer ratings, and serves queries for rating scores.

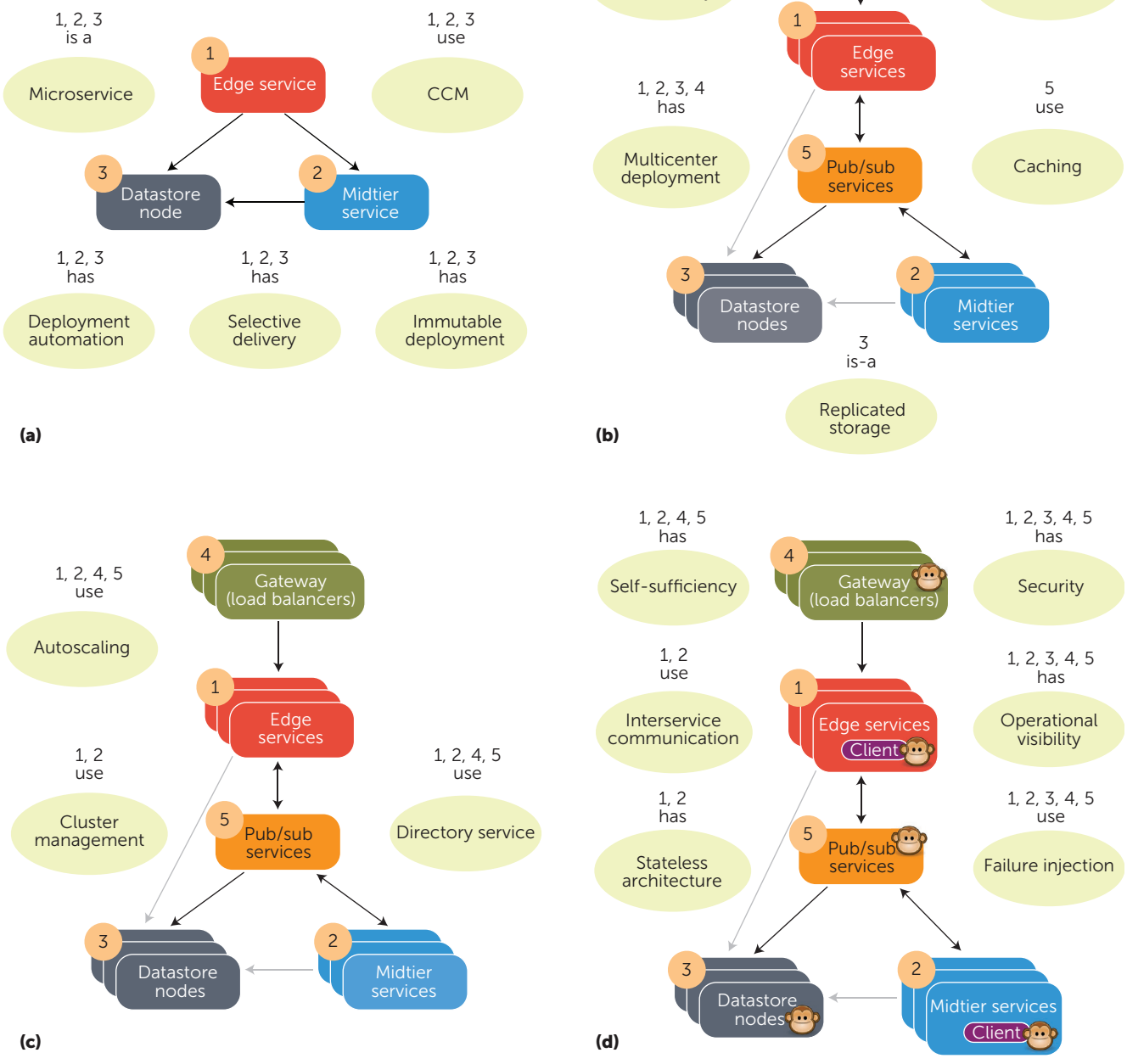

FIGURE 2. Application of RADS-based patterns to the RaaS architecture: (a) continuous delivery, (b) high availability, (c) elastic scalability, and (d) resiliency. (CCM: Centralized configuration management)

- The *midtier* service aggregates the incoming ratings and produces the aggregated scores.
- The *datastore* service handles the reads and writes for consumer ratings and rating scores into the persistent storage.

This architecture removes dependencies among layers and allows independent implementation and transparent deployment of the services while maintaining stable APIs. Further, this principle reduces automation complexity and allows selective delivery

and immutable deployment. CCM is used across services to support efficient deployment.

High Availability

High availability implies continuous uninterrupted availability of high-quality service to users in all locations. Web-scale solutions typically provide clients with service-level agreements (SLAs) and therefore strive to maximize availability (uptime of 99.99 percent with no scheduled downtime) and ensure that their SLA obligations are met. The following strategies enable high availability.

Component redundancy. Redundancy helps to avoid a single point of failure, decreasing the risk of unavailability. Each system component is at least triple replicated, so if a component failure occurs, the redundant instances of the component can ensure continuous service availability. The redundancy level should be greater than two. Otherwise, the only remaining component might not scale for Web traffic, resulting in slow recovery or even failure. Redundant instances can be actively used to share the load or as standby backup in case of failure.

Multicenter deployment. A single-center deployment, even with redundancy, is prone to unavailability because of failures that span datacenters and geographic regions. Therefore, Web-scale solutions are deployed to multiple datacenters to improve availability. Netflix, for instance, has an active-active deployment model, where the solution components are deployed in multiple regions and at least triple replicated across three zones in each region.[2] Regions, which are based on geographic locations and contain multiple zones, provide isolation from location-based failures and latency-locality for the traffic. Zones are datacenters that have separate buildings and individual power, and are connected using low-latency links. This structure prevents inter-zone failure propagation. Deployment redundancy aims to have enough capacity to handle traffic even during zone failures or regional outages, thus ensuring high availability.

Intelligent load balancing. Load balancing is required to distribute traffic across replicated instances of the solution's components. Load balancers are placed at each level to interact with replicated components. Cloud services such as elastic load balancing can handle changes in request rate, and policy-based traffic distribution can ensure latency locality to improve performance and ensure high availability.

Replicated storage. Data storage needs to be distributed, scalable, and highly available to handle large volumes of data. The cloud provides limitless and on-demand storage that can be easily scaled. Data partitioning and replication are necessary to provide data localization and handle increases in storage and accesses. Highly replicated storage enables separation of development and continuous data availability even during network partitioning events. However, replication requires maintaining data consistency to ensure that each interaction views the same data. Maintaining strong consistency negatively impacts solution availability and scalability. Strongly consistent data operations aren't tolerant of common cloud failures (that is, network latency, partitioning, or outages) because of CAP theorem constraints. Web-scale solutions prefer availability over consistency and thus straddle the availability/partition tolerance (AP) side of the CAP theorem.[4] Components are modeled based on eventual consistency, and strong consistency is used only when absolutely necessary.

Caching. Data stores can be relatively slow for Web-scale needs and subject to the network latency of a cloud's shared environment. Therefore, caching is used to provide low-latency data access and improve performance. Local caching isn't adequate for Web-scale solutions because of scalability limitations and data consistency concerns. A shared-memory-based caching mechanism is scalable, ensures data consistency, and provides optimized data access. Cache misses can be transparently retrieved from the datastore and updates can be added to the cache and automatically written to the datastore. A caching layer can also be used to share state across decoupled layers.

RaaS architecture with high availability. Figure 2b shows the updated architecture of RaaS with high availability patterns. With the assumption of US-only service, the RaaS can be deployed in two geographic regions, US-East and US-West, with three service instances per region across three datacenters. A global load balancer acts as a gateway to two regions, and a set of internal load balancers route requests across edge services in different regions. The midtier consists of standby backup services for use during failures. A memory-based publish/subscribe caching service is used to decouple edge and midtier services and enable asynchronous aggregation of ratings. It provides low-latency access to consumer ratings (published by edge services, subscribed by midtier service) and aggregated scores (published by midtier service, subscribed by edge services).

Services switch to the datastore if failures occur in the caching service. The datastore cluster stores at least three copies of data per region and one copy per datacenter. It performs asynchronous transparent replication across datacenters and ensures continuous data availability. Edge services are designed with eventual consistency in mind, but the latest aggregated scores might not reflect the latest incoming ratings. These patterns work together to optimize throughput, avoid single points of failure, and enable high availability for RaaS even during failures.

Elastic Scalability

Elastic scalability is a solution's ability to adapt to workload fluctuations by automatically increasing or decreasing capacity to closely match resources with the current demand. Web-scale cloud solutions, capable of supporting large numbers of users and high volumes of data, require elastic autoscaling to maintain adequate performance, meet SLAs, and minimize operational cost. Solutions can be scaled either vertically, by changing the existing instance's capacity, or horizontally, by changing the number of instances dedicated to the workload.[9] Vertical scaling is unsuitable for a highly elastic workload due to maximum capacity limitations (for compute, storage, and network) and the downtime required to adjust resources. Horizontal scaling is seamless and preferred even though it requires more management. Elastic scalability is built on top of several patterns.

Vertical scaling is unsuitable for a highly elastic workload due to maximum capacity limitations

Autoscaling. This pattern is responsible for automated provisioning and unprovisioning of resources in the cloud. Cloud providers offer autoscaling as a fundamental capability.

Cluster management. Homogeneous services are grouped together into autoscale groups based on their scaling requirements. The autoscale groups—that is, a collection of assets (such as member templates, load balancers, and virtual LANS) and policies (such as minimum and maximum number of group members and events that trigger scaling)—are managed by the cluster management component based on defined policies. It uses member templates to provision new resources and policies to trigger scaling events based on anticipated traffic changes or factors such as request arrival rate, CPU utilization, memory consumption, response time, and queue length.

Directory service. Elastic scalability creates a highly dynamic environment in which services are ephemeral. For instance, the average lifetime of a Netflix instance is 36 hours.[10] A directory service provides a registry for locating different services and the number of instances available for each service in the system. Elastic scalability requires services to be registered, discovered, and removed automatically. The directory service keeps track of registered services and automatically removes unhealthy services.

RaaS architecture with elastic scalability. Figure 2c shows the updated RaaS architecture with elastic scalability patterns. Edge services are autoscaled in real time based on the load. They're organized in a homogeneous pool using an image template, and an autoscale group is created in each region. Scaling policies are defined to set the minimum number of members (for example, three), maximum number of members (for example, based on number of subscribers and peak demand), and events that trigger scaling, such as changes in metrics (request arrival rate, CPU usage, memory consumption, response time, and so on). A scale-up event provisions a new instance of the edge service, configures it to be part of the pool, and registers it with the directory service. A scale-down event unprovisions an instance of the edge service, and the directory service removes it from its registry. Multiple instances of the directory service are deployed in each region. Midtier services are also autoscaled to create backup services when a primary service fails and the existing backup service is in use. These patterns ensure quality of service during peak and support resource optimization and cost reduction during off-peak hours.

Resiliency

Resiliency aims to provide failure tolerance and continuous service availability. Web-scale solutions hosted on the cloud have complex distributed architectures with many dependencies. In addition, the incoming request rate is very high. Incoming requests are further translated into multiple internal interactions—for example, Netflix has an aver-

age ratio of 1:6 for incoming requests to internal interactions.[11] Malfunctions and failures are inevitable in such high-activity dynamic environments. Failures occur randomly and nondeterministically for various reasons, such as latencies in cloud-shared environments, anomalies in components, or a partially reliable cloud infrastructure. Even a single dependency failure can saturate resources and bring the entire solution down. Therefore, Web-scale solutions require resiliency to be built into their architectures. Several patterns support this principle.

Self-sufficiency. This pattern aims for a solution design that expects its dependencies to fail and can handle these failures. Such a design prevents cascading failures by allowing dependencies to fail fast and recover rapidly or switch to fallback and fail gracefully. Netflix uses various strategies to support self-sufficiency patterns in its design.[11]

Dependency isolation aims to provide separation from dependencies to prevent failure propagation (a bulkhead pattern in the Netflix architecture). The idea is based on the bulkhead design used in ship and plane structures for containing water and fire during a hull breach. In cloud solutions, this design prevents the cascade of failures due to a failed dependency by using separate thread pools for communication with different dependent components. The failure is isolated in the thread pools without any effect on parallel and consecutive interactions.

The *Retry and fallback* strategy aims to achieve maximum solution uptime and deliver a degraded service experience during failures to prevent downtime. The components are designed to transparently retry interactions that fail due to transient faults in the cloud environment, such as overloads, timeouts, network delays, and temporarily unavailable resources. Retry policies are specified based on the type of fault to optimize delay before the subsequent attempt. Unusual faults are handled using a fallback logic that reduces the failure's impact. Each call to a dependent component defines a fallback mechanism, which is executed in case of a failure. Fallback mechanisms are simple compared to standard services and provide a default, cached, or coarse-grain service.

The *fail fast and circuit breaker* strategy aims to minimize the impact of failures on system performance. The idea is based on the electrical circuit breakers that prevent damage caused by an overload or short circuit by interrupting the current flow. It prevents the components from conducting (or repeating) the interactions that are likely to fail by disabling the interactions with faulty dependencies. Failing immediately prevents interactions with faulty components that will be blocked and continue to hold resources. It also helps reduce the load on overloaded dependencies. Circuit breaking isolates the possible failures and protects dependencies from excessive demand in a faulty state and the solution from the failed components.

Autorecovery aims to allow solution components to recover from temporary faults that correct themselves after a short period. For instance, overloaded dependencies with degraded performance that are temporarily disabled by the circuit breaker pattern are checked periodically and made active when the fault is resolved. A recovering component is made active slowly (for example, by limiting the volume of requests) to prevent overloads. It triggers automatic recovery of other components that interact with such dependencies.

Interservice communication. ISC, which is performed using lightweight protocols, is either synchronous or asynchronous. Synchronous communication is done using standard HTTP-based REST mechanisms, and asynchronous communication is generally based on a publish–subscribe model. Synchronous communication relies on the directory service, whereas asynchronous communication relies on the message broker. Depending on the use case scenarios, the solutions use a mix of both communication types. ISC is handled using specialized clients that are created and configured to communicate with specific target dependencies. Clients obtain and cache the location of dependencies from the directory service. They're responsible for load balancing the interactions among the replicated service instances and support locality-aware routing to reduce latency. Clients are wrapped with the self-sufficiency patterns to handle latency and failures.

Stateless architecture. It's very difficult to maintain state association and prevent information loss when components are ephemeral and dynamically provisioned or unprovisioned. Furthermore, service interactions might not go to the same service instance again in a load-balanced environment, causing issues for availability and scalability. Therefore, Web-scale solutions avoid stateful cache and cookies and tend to have a loosely coupled stateless architecture where possible.

Failure injection. This pattern aims to simulate failures in the solution architecture. Failures can occur

randomly, but they aren't frequent enough to test the solution's resiliency. Cloud solutions, such as Netflix, have adopted failure injection mechanisms such as chaos (monkey) testing as part of their system.[12] Chaos testing introduces random failures in the operational production environment to proactively find and fix bugs at the earliest opportunity without impacting users.

Operational visibility. Web-scale cloud solutions consist of a large number of resources, most of which are hosted on the cloud provider's infrastructure. Netflix, for instance, has thousands of service instances and datastore nodes running in the cloud that are continuously created or removed.[2] Because of the system's enormous size and dynamic, distributed, and elastic nature, it's challenging to understand the state of the operational environment. Therefore, resource tracking is essential and is achieved through several strategies.

Rigorous *monitoring* allows Web-scale solutions to detect failures and determine SLA misses. In a cloud-hosted environment, solution components are subject to network latency and the availability of underlying computing and storage. Monitoring ensures that the components and resources are available and performing correctly. Components implement functional checks and expose metrics (such as request arrival rate, CPU usage, memory consumption, and response time) that the monitoring service periodically collects. The metrics are aggregated to create meaningful abstractions, establish causality of events, trigger alerts, and take action based on predefined rules.

A second strategy, *metering*, can be external or internal. Solutions with customer billing requirements implement external metering to record client usage and calculate charges. Internal metering is needed to track the usage of the whole solution and break down billing associated with individual components and specific resources since most cloud providers only produce standard billing details. Components are instrumented to measure operations and record resource usage. This helps to provide resource utilization information by region, component, and so on. Further analysis can predict future trends and help to optimize usage.

Logging is an essential part of any system for monitoring, debugging, and analysis. In Web-scale solutions, log messages are generated at a high rate, and they need to be quickly processed without affecting solution performance. Events must be captured in a nonblocking manner using separate threads to enable highly scalable logging.

Security. Cloud-hosted solutions must be secured in several ways.

First, solutions must use *secure communication*. The cloud has a shared network environment, so all interactions in cloud-hosted solutions should be conducted over a secure communication channel, for example using HTTPS.

Solutions must also ensure *federated authentication and authorization*. Incoming requests trigger several internal service interactions. Having individual components validate each interaction adds complexity and state maintenance. Separating authentication and authorization from the components and delegating these tasks to a centralized gateway simplifies development and minimizes administrative overhead. Furthermore, authentication and authorization are propagated throughout the request's lifecycle.

Proper *access control* must also be maintained. Direct access to solution's internal resources must be disabled and any access should be authorized through a centralized access point. Netflix, for instance, doesn't allow direct SSH login across instances to prevent unauthorized access.[10] The access point issues tokens to control the operation, duration, target resource, and access level. These tokens are revoked on completion of the operation or an access violation. Access management is based on individual rules for components' security groups, access control lists for resources, and roles/attributes for subjects to enable finer granularity.

RaaS architecture with resiliency. Figure 2d shows the updated RaaS architecture with resiliency patterns. Microservices are equipped with resiliency mechanisms to ensure continuous service availability during failures. Dependency failures are handled by ISC clients using the self-sufficiency patterns; these clients use short timeouts and quick retries to route around possibly failed services, monitoring latency and quickly disabling interactions with the failed dependencies. Service failures are handled by switching traffic to the replicated service instances. Datacenter failures are tolerated by switching traffic to other datacenters, and regional outages are handled using multiregion deployments. A failure injection service adds monkey-based testing to the solution components to test the solution's resiliency. The use of microservices and stateless architecture ensures that the unit of failure is a single component, so component failures can be easily isolated without bringing the entire system down. These patterns collectively enable solution resiliency and help support availability, scalability, and delivery.

The cloud platform is a commodity to drive Web-scale solutions, but the cloud's promise of a scalable infrastructure can only be leveraged by a cloud-aware scalable architecture. Therefore, cloud migration necessitates a fundamentally different approach to engineering Web-scale solutions. Cloud providers need to offer a platform with the identified patterns as foundational services that enable the RADS principles, in addition to offering the basic compute, network, and storage capabilities. Such a platform will streamline the process of solution development and enable seamless Web-scale operation in the cloud.

There's still a need to explore the usage of cloud-agnostic multicloud architectures, which will add another layer of redundancy, provide wide coverage, offer competitive variable pricing, and prevent lock-in to a single cloud provider.

References

1. M. Solomonov, "Why Are So Many Large and Small Companies Transitioning to AWS?" Smart Data Collective, 25 Apr. 2014; www.smartdatacollective.com/connectriahosting/197171/who-are-some-heavy-users-amazon-web-services-aws.
2. A. Cockcroft, "Patterns for Continuous Delivery, High Availability, DevOps & Cloud Native Open Source with NetflixOSS," Dec. 2013; www.slideshare.net/adrianco.
3. L. Saccone, "The Most Recent Amazon Outage Exposes the Dark Side of the Cloud," Daily Lounge; http://dailylounge.com/the-daily/entry/the-recent-amazon-outage-exposes-the-dark-side-of-the-cloud.
4. E. Brewer, "CAP Twelve Years Later: How the 'Rules' Have Changed," *Computer*, vol. 45, no. 2, 2012, pp. 23–29.
5. R. Ranchal et al., "Hierarchical Aggregation of Consumer Ratings for Service Ecosystem," *Proc. 22nd IEEE Int'l Conf. Web Services*, 2015, pp. 575–582.
6. B. Di Martino, G. Cretella, and A. Esposito, "Semantic and Agnostic Representation of Cloud Patterns for Cloud Interoperability and Portability," *Proc. 5th IEEE Int'l Conf. Cloud Computing Technology and Science*, vol. 2, 2013, pp. 182–187.
7. D. Namiot and M. Sneps-Sneppe, "On Micro-Services Architecture," *Int'l J. Open Information Technologies*, vol. 2, no. 9, 2014, pp. 24–27.
8. "Amazon Architecture," High Scalability, 8 Sept. 2007; http://highscalability.com/amazon-architecture.
9. M.L. Abbott and M.T. Fisher, *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*, Pearson Education, 2009.
10. A. Cockcroft, "Cloud Native at Netflix: What Changed?" July 2013; www.slideshare.net/adrianco/netflix-what-changed-gartner-catalyst.
11. B. Christensen, "Fault Tolerance in a High Volume, Distributed System," blog, 29 Feb. 2012; http://techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html. Accessed: Nov 2014.
12. A. Tseitlin, "The Antifragile Organization," *Comm. ACM*, vol. 56, no. 8, 2013, pp. 40–44.

ROHIT RANCHAL *is a senior software engineer in the Watson Health Cloud at IBM. His research interests include cloud and service computing systems with a focus on healthcare applications, scalable architectures, secure data dissemination, and policy enforcement. Ranchal has a PhD in computer science from Purdue University. He's a member of IEEE. Contact him at ranchal@us.ibm.com.*

AJAY MOHINDRA *is the director of the Watson Health Cloud at IBM. His research interests include electronic healthcare systems, cloud computing, mobile computing, and outcome-based services. Mohindra has a PhD in computer science from Georgia Institute of Technology. He's a senior member of IEEE and a member of ACM. Contact him at ajaym@us.ibm.com.*

JUSTIN MANWEILER *is a researcher with the IBM Thomas J. Watson Research Center. His research interests include mobile and ubiquitous computing systems, especially those enhanced through unique cloud services. Manweiler has a PhD in computer science from Duke University. Contact him at jmanweiler@us.ibm.com.*

BHARAT BHARGAVA *is a professor of computer science at Purdue University. His research interests include security and privacy issues in cloud and service computing systems, vehicular networks, and electronic healthcare systems. Bhargava has a PhD in electrical engineering from Purdue University. He's a fellow of IEEE and the Institution of Electronics and Telecommunication Engineers. Contact him at bbshail@purdue.edu.*