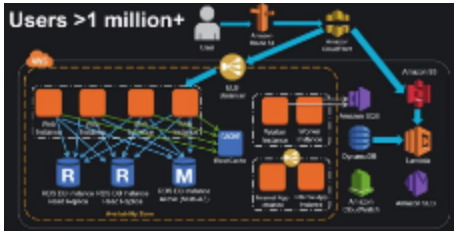
A Beginner's Guide to Scaling to 11 Million+ Users on Amazon's AWS

highscalability.com/blog/2016/1/11/a-beginners-guide-to-scaling-to-11-million-users-on-amazons.html

Monday, January 11, 2016 at 8:56AM

How do you scale a system from one user to more than 11 million users? Joel Williams, Amazon Web Services Solutions Architect, gives an excellent talk on just that subject: AWS re:Invent 2015 Scaling Up to Your First 10 Million Users.



If you are an advanced AWS user this talk is not for you, but it's a great way to get started if you are new to AWS, new to the cloud, or if you haven't kept up with with constant stream of new features Amazon keeps pumping out.

As you might expect since this is a talk by Amazon that Amazon services are always front and center as the solution to any problem. Their platform play is impressive and instructive. It's obvious by how the pieces all fit together Amazon has done a great job of mapping out what users need and then making sure they have a product in that space.

Some of the interesting takeaways:

- Start with SQL and only move to NoSQL when necessary.
- A consistent theme is take components and separate them out. This allows those components to scale and fail independently. It applies to breaking up tiers and creating microservices.
- Only invest in tasks that differentiate you as a business, don't reinvent the wheel.
- Scalability and redundancy are not two separate concepts, you can often do both at the same time.
- There's no mention of costs. That would be a good addition to the talk as that is one of the major criticisms of AWS solutions.

The Basics

- AWS is in 12 regions around the world.

    - A Region is a physical location in the world where Amazon has multiple Availability Zones. There are regions in: North America; South America; Europe; Middle East; Africa; Asia Pacific.
    - An Availability Zone (AZ) is generally a single datacenter, though they can be constructed out of multiple datacenters.
    - Each AZ is separate enough that they have separate power and Internet connectivity.
    - The only connection between AZs is a low latency network. AZs can be 5 or 15 miles apart, for example. The network is fast enough that your application can act like all AZs are in the same datacenter.
    - Each Region has at least two Availability Zones. There are 32 AZs total.
    - Using AZs it's possible to create a high availability architecture for your application.
    - At least 9 more Availability Zones and 4 more Regions are coming in 2016.
- AWS has 53 edge locations around the world.
    - Edge locations are used by CloudFront, Amazon's Content Distribution Network (CDN) and Route53, Amazon's managed DNS server.
    - Edge locations enable users to access content with a very low latency no matter where they are in the world.
- Building Block Services
    - AWS has created a number of services that use multiple AZs internally to be highly available and fault tolerant. Here is a list of what services are available where.
    - You can use these services in your application, for a fee, without having to worry about making them highly available yourself.
    - Some services that exist within an AZ: CloudFront, Route 53, S3, DynamoDB, Elastic Load Balancing, EFS, Lambda, SQS, SNS, SES, SWF.
    - A highly available architecture can be created using services even though they exist within a single AZ.

1 User

- In this scenario you are the only user and you want to get a website running.
- Your architecture will look something like:
    - Run on a single instance, maybe a type t2.micro. Instance types comprise varying combinations of CPU, memory, storage, and networking capacity and give you the flexibility to choose the appropriate mix of resources for your applications.
    - The one instance would run the entire web stack, for example: web app, database, management, etc.
    - Use Amazon Route 53 for the DNS.
    - Attach a single Elastic IP address to the instance.
    - Works great, for a while.

Vertical Scaling

- You need a bigger box. Simplest approach to scaling is choose a larger instance type. Maybe a c4.8xlarge or m3.2xlarge, for example.
- This approach is called vertical scaling.
- Just stop your instance and choose a new instance type and you're running with more power.
- There is a wide mix of different hardware configurations to choose from. You can have a system with 244 gigs of RAM (2TB of RAM types are coming soon). Or one with 40 cores. There are High I/O instances, High CPU Instances, High storage instances.
- Some Amazon services come with a Provisioned IOPS option to guarantee performance. The idea is you can perhaps use a smaller instance type for your service and make use of Amazon services like DynamoDB that can deliver scalable services so you don't have to.

- Vertical scaling has a big problem: there's no failover, no redundancy. If the instance has a problem your website will die. All your eggs are in one basket.
- Eventually a single instances can only get so big. You need to do something else.

Users > 10

- Separate out a single host into multiple hosts
    - One host for the web site.
    - One host for the database. Run any database you want, but you are on the hook for the database administration.
    - Using separate hosts allows the web site and the database to be scaled independently of each other. Perhaps your database will need a bigger machine than your web site, for example.
- Or instead of running your own database you could use a database service.
    - Are you a database admin? Do your really want to worry about backups? High availability? Patches? Operating systems?
    - A big advantage of using a service is you can have a multi Availability Zone database setup with a single click. You won't have to worry about replication or any of that sort of thing. Your database will be highly available and reliable.
- As you might imagine Amazon has several fully managed database services to sell you:
    - Amazon RDS (Relational Database Service). There are many options: Microsoft SQL Server, Oracle, MySQL, PostgreSQL, MariaDB, Amazon Aurora.
    - Amazon DynamoDB. A NoSQL managed database.
    - Amazon Redshift. A petabyte scale data warehouse system.
- More Amazon Aurora:
    - Automatic storage scaling up to 64TB. You no longer have to provision the storage for your data.

    - Up to 15 read read-replicas
    - Continuous (incremental) backups to S3.
    - 6-way replication across 3 AZs. This helps you handle failure.
    - MySQL compatible.
- Start with a SQL database instead of a NoSQL database.
    - The suggestion is to start with a SQL database.
    - The technology is established.
    - There's lots of existing code, communities, support groups, books, and tools.
    - You aren't going to break a SQL database with your first 10 million users. Not even close. (unless your data is huge).
    - Clear patterns to scalability.
- When might you need start with a NoSQL database?
    - If you need to store > 5 TB of data in year one or you have an incredibly data intensive workload.
    - Your application has super low-latency requirements.
    - You need really high throughput. You need to really tweak the IOs you are getting both on the reads and the writes.
    - You don't have any relational data.

Users > 100

- Use a separate host for the web tier.
- Store the database on Amazon RDS. It takes care of everything.
- That's all you have to do.

Users > 1000

- As architected your application has availability issues. If the host for your web service fails then your web site goes down.
- So you need another web instance in another Availability Zone. That's OK because the latency between the AZs is in the low single digit milliseconds, almost like they right next to each other.
- You also need to a slave database to RDS that runs in another AZ. If there's a problem with the master your application will automatically switch over to the slave. There are no application changes necessary on the failover because your application always uses the same endpoint.
- An Elastic Load Balancer (ELB) is added to the configuration to load balance users between your two web host instances in the two AZs.
- Elastic Load Balancer (ELB):
    - ELB is a highly available managed load balancer. The ELB exists in all AZs. It's a single DNS endpoint for your application. Just put it in Route 53 and it will load balance across your web host instances.
    - The ELB has Health Checks that make sure traffic doesn't flow to failed hosts.
    - It scales without your doing anything. If it sees additional traffic it scales behind the scenes both horizontally and vertically. You don't have to manage it. As your applications scales so is the ELB.

Users > 10,000s - 100,000s

- The previous configuration has 2 instances behind the ELB, in practice you can have 1000s of instances behind the ELB. This is horizontal scaling.
- You'll need to add more read replicas to the database, to RDS. This will take load off the write master.
- Consider performance and efficiency by lightening the load off your web tier servers by moving some of the traffic elsewhere. Move static content in your web app to Amazon S3 and Amazon CloudFront. CloudFront is the Amazon's CDN that stores your data in the 53 edge locations across the world.

- Amazon S3 is an object base store.
    - It’s not like EBS, it’s not storage that’s attached to an EC2 instance, it’s an object store, not a block store.
    - It’s a great place to store static content, like javascript, css, images, videos. This sort of content does not need to sit on an EC2 instance.
    - Highly durable, 11 9’s of reliability.
    - Infinitely scalable, throw as much data as it as you want. Customers store multiple petabytes of data in S3.
    - Objects of up to 5TB in size are supported.
    - Encryption is supported. You can use Amazon’s encryption, your encryption, or an encryption service.
- Amazon CloudFront is cache for your content.
    - It caches content at the edge locations to provide your users the lowest latency access possible.
    - Without a CDN your users will experience higher latency access to your content. Your servers will also be under higher load as they are serving the content as well as handling the web requests.
    - One customer needed to serve content at 60 Gbps. The web tier didn’t even know that was going on, CloudFront handled it all.
- You can also lighten the load by shifting session state off your web tier.
    - Store the session state in ElastiCache or DynamoDB.
    - This approach also sets your system up to support auto scaling in the future.
- You can also lighten the load by caching data from your database into ElastiCache.

    Your database doesn’t need to handle all the gets for data. A cache can handle a lot of that work and leaves the database to handle more important traffic.
- Amazon DynamoDB - A managed NoSQL database

    - You provision the throughput you want. You dial up the read and write performance you want to pay for.
    - Supports fast, predictable performance.
    - Fully distributed and fault tolerant. It exists in multiple Availability Zones.
    - It's a key-value store. JSON is supported.
    - Documents up to 400KB in size are supported.
- Amazon Elasticache - a managed Memcached or Redis
    - Managing a memcached cluster isn't making you more money so let Amazon do that for you. That's the pitch.
    - The clusters are automatically scaled for you. It's a self-healing infrastructure, if nodes fail new nodes are started automatically.
- You can also lighten the load by shifting dynamic content to CloudFront.

    A lot of people know CloudFront can handle static content, like files, but it can also handle some dynamic content. This topic is not discussed further in the talk, but here's a link.

Auto Scaling

- If you provision enough capacity to always handle your peak traffic load, Black Friday, for example, you are wasting money. It would be better to match compute power with demand. That's what Auto Scaling let's you do, the automatic resizing of compute clusters.
- You can define the minimum and maximum size of your pools. As a user you decide what's the smallest number of instances in your cluster and the largest number of instances.
- CloudWatch is a management service that's embedded into all applications.
    - CloudWatch events drive scaling.
    - Are you going to scale on CPU utilization? Are you going to scale on latency? On network traffic?

    - You can also push your own custom metrics into CloudWatch. If you want to scale on something application specific you can push that metric into CloudWatch and then tell Auto Scaling you want to scale on that metric.

Users > 500,000+

- The addition from the previous configuration is auto scaling groups are added to the web tier. The auto scaling group includes the two AZs, but can expand to 3 AZs, up to as many as are in the same region. Instances can pop up in multiple AZs not just for scalability, but for availability.
- The example has 3 web tier instances in each AZ, but it could be thousands of instances. You could say you want a minimum of 10 instances and a maximum of a 1000.
- ElastiCache is used to offload popular reads from the database.
- DynamoDB is used to offload Session data.
- You need to add monitoring, metrics and logging.
    - Host level metrics. Look at a single CPU instance within an autoscaling group and figure out what's going wrong.
    - Aggregate level metrics. Look at metrics on the Elastic Load Balancer to get feel for performance of the entire set of instances.
    - Log analysis. Look at what the application is telling you using CloudWatch logs. CloudTrail helps you analyze and manage logs.
    - External Site Performance. Know what your customers are seeing as end users. Use a service like New Relic or Pingdom.
- You need to know what your customers are saying. Is their latency bad? Are they getting an error when they go to your web page?
- Squeeze as much performance as you can from your configuration. Auto Scaling can help with that. You don't want systems that are at 20% CPU utilization.

Automation

- The infrastructure is getting big, it can scale to 1000s of instances. We have read replicas, we have horizontal scaling, but we need some automation to help manage it all, we don't want to manage each individual instance.
- There's a hierarchy of automation tools.
    - Do it yourself: Amazon EC2, AWS CloudFormation.
    - Higher-level services: AWS Elastic Beanstalk, AWS OpsWorks
- AWS Elastic Beanstalk: manages the infrastructure for your application automatically. It's convenient but there's not a lot of control.
- AWS OpsWorks: an environment where you build your application in layers, you use Chef recipes to manage the layers of your application.

    Also enables the ability to do Continuous Integration and deployment.
- AWS CloudFormation: been around the longest.
    - Offers the most flexibility because it offers a templatized view of your stack. It can be used to build your entire stack or just components of the stack.
    - If you want to update your stack you update the Cloud Formation template it will update just that one piece of your application.
    - Lots of control, but less convenient.
- AWS CodeDeploy: Deploys your code to a fleet of EC2 instances.
    - Can deploy to one or thousands of instances.
    - Code Deploy can point to an auto scaling configuration so code is deployed to a group of instances.
    - Can also be used in conjunction with Chef and Puppet.

Decouple Infrastructure

- Use SOA/microservices. Take components from your tiers and separate them out. Create separate services like when you separated the web tier from the database tier.
- The individual services can then be scaled independently. This gives you a lot of flexibility for scaling and high availability.
- SOA is a key component of the architectures built by Amazon.
- Loose coupling sets you free.
    - You can scale and fail components independently.
    - If a worker node fails in pulling work from SQS does it matter? No, just start another one. Things are going to fail, let's build an architecture that handles failure.
    - Design everything as a black box.
    - Decouple interactions.
    - Favor services with built-in redundancy and scalability rather than building your own.

Don't Reinvent the Wheel

- Only invest in tasks that differentiate you as a business.
- Amazon has a lot of services that are inherently fault tolerant because they span multiple AZs. For example: queuing, email, transcoding, search, databases, monitoring, metrics, logging, compute. You don't have to build these yourself.
- SQS: queueing service.
    - The first Amazon service offered.
    - It spans multiple AZs so it's fault tolerant.
    - It's scalable, secure, and simple.

    - Queuing can help your infrastructure by helping you pass messages between different components of your infrastructure.
    - Take for example a Photo CMS. The systems that collects the photos and processes them should be two different systems. They should be able to scale independently. They should be loosely coupled. Ingest a photo, put it in queue, and workers can pull photos off the queue and do something with them.
- AWS Lambda: lets you run code without provisioning or managing servers.
    - Great tool for allowing you to decouple your application.
    - In the Photo CMS example Lambda can respond to S3 events so when a S3 file is added the Lambda function to process is automatically triggered.
    - We've done away with EC2. It scales out for you and there's no OS to manage.

Users > 1,000,000+

- Reaching a million users and above requires bits of all the previous points:
    - Multi-AZ
    - Elastic Load Balancing between tiers. Not just on the web tier, but also on the application tier, data tier, and any other tier you have.
    - Auto Scaling
    - Service Oriented Architecture
    - Serve Content Smartly with S3 and CloudFront
    - Put caching in front of the DB
    - Move state off the web tier.
- Use Amazon SES to send email.

- Use CloudWatch for monitoring.

Users > 10,000,000+

- As we get bigger we'll hit issues in the data tier. You will potentially start to run into issues with your database around contention with the write master, which basically means you can only send so much write traffic to one server.
- How do you solve it?
    - Federation
    - Sharding
    - Moving some functionality to other types of DBs (NoSQL, graph, etc)
- Federation - splitting into multiple DBs based on function
    - For example, create a Forums Database, a User Database, a Products Database. You might have had these in a single database before, now spread them out.
    - The different databases can be scaled independently of each other.
    - The downsides: you can't do cross database queries; it delays getting to the next strategy, which is sharding.
- Sharding - splitting one dataset across multiple hosts
    - More complex at the application layer, but there's no practical limit on scalability.
    - For example, in a Users Database ⅓ of the users might be sent to one shard, and the last third to another shard, and another shard to another third.
- Moving some functionality to other types of DBs
    - Start thinking about a NoSQL database.

    - If you have data that doesn’t require complex joins, like say a leaderboard, rapid ingest of clickstream/log data, temporary data, hot tables, metadata/lookup tables, then consider moving it to a NoSQL database.
    - This means they can be scaled independently of each other.

Users > 11 Million

- Scaling is an iterative process. As you get bigger there's always more you can do.
- Fine tune your application.
- More SOA of features/functionality.
- Go from Multi-AZ to multi-region.
- Start to build custom solutions to solve your particular problem that nobody has ever done before. If you need to serve a billion customers you may need custom solutions.
- Deep analysis of your entire stack.

In Review

- Use a multi-AZ infrastructure for reliability.
- Make use of self-scaling services like ELB, S3, SQS, SNS, DynamoDB, etc.
- Build in redundancy at every level. Scalability and redundancy are not two separate concepts, you can often do both at the same time.
- Start with a traditional relational SQL database.
- Cache data both inside and outside your infrastructure.
- Use automation tools in your infrastructure.

- Make sure you have good metrics/monitoring/logging in place. Make sure you are finding out what your customers experience with your application.
- Split tiers into individual services (SOA) so they can scale and fail independently of each other.
- Use Auto Scaling once you're ready for it.
- Don't reinvent the wheel, use a managed service instead of coding your own, unless it's absolutely necessary.
- Move to NoSQL if and when it makes sense.

Further Reading