Designing Apps for Amazon Web Services

Mathias Meyer, GOTO Aarhus 2011
OLD MAN YELLS AT CLOUD
Me

❤️ infrastructure
❤️ code
❤️ databases

@roidrage
www.paperplanes.de
The Cloud

Unlimited resources, whenever you need them
Montag, 10. Oktober 11

10,000 feet
Amazon Web Services
EC2
On-demand computing
It’s not a cloud if it doesn’t have an API.
Pay as you go
Multiple regions

Geographically distributed, for all web services.
US East, West, EU, Singapore, Japan
Multiple datacenters

Called availability zones. At least two data centers in each region. Physically separated locations. API endpoint for a region is unspecific for availability zones.
Different instance types
High CPU
vs.
High memory

One CPU core is about the power of a 2007 Xeon processor.
1.7 GB
-
68 GB
Elastic Block Store

Storage on instances is ephemeral. Goes away when the instance goes away. EBS allows persisting data for longer than an instance’s lifetime. Bound to a data center.
Mount to any instance

A big number of block volumes can be mounted to a single instance.
Snapshots

A point in time, atomic snapshot of an EBS volume. Not a reliable means of backup, but one means for backups.
More AWS Products

S3
CloudFront
CloudFormation
CloudWatch
RDS
Auto Scaling

SimpleDB
Route 53
Load Balancing
Queue Service
Notification Service
Elastic MapReduce
What’s Scalararium?
Automates:

Setup

Configuration

One-Click Deploy

Montag, 10. Oktober 11
Scalarium uses a couple of Amazon's Web services
EC2 is the most interesting one
Own monitoring, because customers like saving money, CloudWatch costs money
Scalarium automates so that customers don’t have to. Most important part about deploying in the cloud. Every manual change is lost when an instance goes down.
The Dream:

Configure a cluster
Push a button
Boom!
Two ways...
Create image, boot it.
Build once, use forever

Point in time snapshot of a system, fully configured.
App server, database, web server, etc.
How do you handle updates?

Build a new image, install updates, discard old image.
Lather, rinse, repeat, with every update.
Configure from scratch
Start with a blank slate.
A clean operating system installation.
Use a configuration management tool. Abstracts installation of packages, writing of configuration files, handling file systems, etc.
Configuration
+ Cookbooks/Manifests
+ Chef/Puppet/ etc.
= Configured cluster

Configuration describes the final result of what the system should look like.
Configuration:
Chef Server
RightScale
JSON
Scalarium

Montag, 10. Oktober 11
There’s an infinite numbers of infrastructure service providers.
Just as many ways to store cluster configuration
A Web UI to automate and simplify the Amazon APIs and setting up servers/clusters.
In the beginning...
Lasted for a few months on just one instance. Instance ran RabbitMQ, CouchDB, Redis, background workers, web and application servers. Bootstrapped startup = start small, iterate quickly.
Eventually became overloaded.
Wrong Way
Automate, automate, automate!

Chef for everything.
Yes, our first instance was not fully automated. Hypocritic.
Vagrant is an excellent tool to test locally.
Scalarium customers can test their own and our cookbooks locally.
Automate setup, configuration, re-configuration of services, everything!
EC2 is not a traditional datacenter
It doesn’t look like this.
Still looks like this. But it’s transparent to you.
No operational access to you.
Multi-tenant

Shared resources (CPU, memory, network).
You instance likely shares resources with several other EC2 customers on the same physical host.
High likelihood of failure

It’s still hardware that’s running your servers. Hardware fails.
Faulty instances

They fail, not all the time, but if you have high turnover scaling up and down, they’ll fail. Discard, boot new instance, done.
Datacenter outage
Network partition
More instances = Higher chance of failure
Mean Time Between Failures
On EC2 as a whole it’s pretty small. Not an important metric.
Just because something fails doesn’t mean you have to be affected.
The big chance for nay-sayers and cloud haters.
The big chance for nay-sayers and cloud haters.
The big chance for nay-sayers and cloud haters.
Montag, 10. Oktober 11

The big chance for nay-sayers and cloud haters.
The big chance for nay-sayers and cloud haters.

Amazon’s Trouble Raises Cloud Computing Doubts

Amazon's Web Services outage: End of cloud innocence?

Inside Amazon's Cloud Disaster
7/8/2011

Downtime in EU data centers. Lightning strike caused power outage. Again, cascading failure in the EBS storage layer. More than three days 'til full recovery.
I'M
AWESOME
SAUCE
Cause I am

Montag, 10. Oktober 11
failure is cool
Failure is a good thing
You can ignore it
Learn from it
Design for it
Don’t fear failure
Plan for failure

Failure becomes a part of your apps’ lifecycle.
Deploying in the cloud has a bigger effect on culture than it does on your application.
In case of failures, your app should handle them gracefully, not breaking along the way entirely. Serve statics instead of failure notices to the user.
Plan for recovery

Can you re-deploy your app into a different region quickly?
If not, why not?
MTTR
Disaster recovery plan

What do you do when your site goes down?
What do you do when you need to restore data?
Plan, verify, one click.
Multi-datacenter deployments

Deploy App across multiple data centers/availability zones. Make deploying to different data centers part of the deployment process. Staged deployments: new set of instance, flip load balancer.
Storage becomes a key part in handling failure. Everything else is usually much easier to scale and distribute. Replicate data across availability zones, across regions.
Availability zones are geographically distributed. Reading from in between them means increased latency. Replication ensures data is in multiple geographic locations. Replication allows to recover quickly by moving to different data centers. Not all databases do this well, but they do it.
Multi-region deployments

If you need to be very highly available.
Deploying highly distributed is expensive.
How distributed is up to your budget.
And to how much your availability is worth to your business.
Relax consistency requirements

Strong consistency increases need for full availability
Distribute and partition data, on different instances and different datacenters
Immediately became an issue when we scaled out. Network latency adds to EBS latency and made for higher response times from the database. All network traffic on EC2 is firewalled, even internal traffic.
Keep data local
Keep data in memory
Cache is king

Disk is expensive because it touches the network.
Increased performance, better network utilization
EBS performance is okay, but not great. Don’t expect SATA or SAS like performance.
RAID 0, 5 or 10.
EBS is redundant, but extra redundancy with striping doesn’t hurt. More likely recovery when one EBS volume fails.
RAIDs won’t save you from EBS unavailability.
Use local storage

Don’t use EBS at all.
Local storage requires redundancy.
Instance storage is lost when the instance is lost.
RAID across local storage.
More reliable in terms of I/O than EBS.
Services that uses local storage where mostly unaffected by the EC2 outages.
Use bigger instances

The bigger your instance the less shared it is on the host. Bigger instances have higher I/O throughput.
What would I do differently?
Small services can run independent of each other. Small services are easy to deploy, easy to reconfigure (Chef). Don’t have to know about all the other services upfront, leave that to CM tools. Layered system with small services allows failure handling on every layer. Failure in one layer doesn’t have to drag down the rest.
Frontend vs. Small APIs
When components fail, don’t block waiting for them. Timeout quickly.
Circuit breaker: track failures and fail operations immediately if you know they’re likely to fail. recover when it’s safe again.
Montag, 10. Oktober 11

Retry with an exponential backoff. Assume failure always.
Don’t just assume failure
Test failure

Montag, 10. Oktober 11

Shut off instances randomly, see what happens.
Turn on the firewall, adds network timeouts, see what happens.
The cloud makes it so easy to bring up test environments, and to move resources quickly when necessary.
Netflix’ Chaos Monkey randomly kills instances.
“Think about your software running.”

Theo Schlossnagle, OmniTI
Understand your code’s breaking points

Use patterns like circuit breakers and bulkheads to reduce failure points.
Think about outcome and implications, not just features.
Understand your code’s breaking points and how they handle unavailability, timeouts, and the like.

All these are so much more likely in a cloud environment.
Isn’t all that what you do at large scale?

It’s what you do at any scale where availability is a factor.
Cloud

==

Large scale
You’re a part of it

Prepare for the worst, plan for the worst.
The cloud made failure at a large scale obvious even when you’re working at a small scale.
Scalarium today
Scalarium runs on Scalarium
Easy to boot up somewhere else, switch over DNS, done.
Lack of visibility

Montag, 10. Oktober 11

Takes up to an hour still to acknowledge problems. Amazon is not good at admitting failure happens a lot on EC2. Not enough education on how to build apps for EC2 and their web services, especially how to deal with failure.
Don’t fall for SLAs
Amazon only handles infrastructure
How you build on it is up to you
Fun fact
amazon.com is served off EC2

Montag, 10. Oktober 11
Since 21/10/2010.
Yes, they were down too, at least during the EU outage.
It’s not the cloud that matters, it’s how you use it.

EC2 doesn’t make everything harder, quite the opposite, it makes things easier:
Adding capacity, automation, responding to failures.
Thank you!