Nightmare Cloud Infrastructures: episode 3

29 October 2024 - 13 min. read

AWS Well-Architected | Cloud adoption | DevOps

"You've seen horrible things. An army of nightmare creatures. But they are nothing compared to what came before. What lies below. It's our task to placate the Ancient Ones. As it's yours to be offered up to them. Forgive us. And let us get it over with."

The Cabin in the Woods

Hello, dear horror lovers, and welcome back to our little scary corner!

As many of you already know, at this time of year, we tell you stories about errors and design problems that turned out to be monsters to fight.

In this article, we will see what went wrong, why, and how to avoid certain things in the future...

As always, no finger is pointed at anyone; sometimes, bad decisions depend on context and time.

So, sit back, relax, and keep reading!

If you missed them, here are the first and second episodes.

Complex complexity

"Sometimes dead is better."

Pet Sematary

What can you do when you find a potential problem that feeds itself and grows without anyone trying to stop this behavior?

It started on a sunny summer day during an interview for a Well-Architected review. We stumbled upon a critical workload running on EKS on EC2 and a second EKS cluster running on Fargate. As always, we are looking for improvements in manageability and elasticity, so we asked if there was a plan to adopt Fargate for the EKS workload running on EC2 instances.

There was no plan: the workload could run on Fargate, but Karpenter managed the elasticity of the cluster. Fair enough, Karpenter is a great tool, and it just got out of beta status. We were pretty happy about this finding because automatic resource decommissioning and assignment were considered, even if the manageability could be increased. We briefly discussed the elasticity and the R&D done to evaluate solutions because the application's load was pretty unpredictable; good job!

So we had a look at the EKS cluster running on Fargate and found out that it was used.. to run Karpenter!

We had to ask: did they migrate the EC2 Cluster from Autoscaler to Karpenter? The answer was no: they built a second Cluster to migrate the workload.

We were puzzled: why didn't they use the Fargate platform (or, even better, add a Fargate profile and start running pods on it) if a migration was needed? The answer was simple: "Because we use Karpenter!" We were puzzled but had to move on; we noted this for the report (and this article!).

The complexity of running a tool and maintaining its configuration to scale resources that can scale without any particular intervention will lead to errors and problems (especially during upgrades and changes). Instead of managing one component, now you have to take care of an additional one and evaluate their interaction.

In this case, we saw a pattern: what if someone adds a GUI to Karpenter in the future? We can bet we will find the GUI running while doing the next Well-Architected review! It's a neverending loop of adding complexity to a system to manage its complexity.

When we add a component, we have to evaluate its impact. Even if it seems to ease our job, we always have to ask ourselves: Is there a managed service designed for that?

CI/CR (Continuous Integration, Continuous Rollback)

"Diabolical forces are formidable. These forces are eternal, and they exist today. The fairy tale is true. The devil exists. God exists. And for us, as people, our very destiny hinges upon which one we elect to follow."

The Conjuring

For our project, we need three different environments and, of course, CI/CD pipelines for our front end and back end." Nothing makes us happier than hearing this requirement in a project's kick-off call! Finally, the "Developmenttestproduction" scenario was a distant memory.

Since there was no particular technology stack, we used CodeBuild and CodePipeline and agreed on using three different branches for different environments: dev, test, and

production, each associated with a pipeline running in the specific environment.

To "promote" code between environments, a merge has to be made from dev to test, and so on. Someone can argue that there are better flows, but this seemed to adapt well to the development workflow already in place.

After some IaC and CDK development, the dev and test environments were ready for both the front end and the back end. We were asked to identify releases on ECR images and assets on S3 buckets with the commit ID. It was easy and done, but we didn't have an idea of what was about to happen.

Production was ready after some time (and development), and the team requested a manual approval step to trigger a git tag on the repository and better identify releases. Only after a week or two did the situation become clear: we received a request to modify the pipelines for the test and production repository to select from which tag to deploy artifacts.

This was to accommodate a request to cherrypick which version of frontend and backend to deploy in each environment to ease rollbacks "just in case".

Needless to say, this is a recipe to cook the perfect disaster: rolling back arbitrary versions of different components can lead to unknown and unexpected interactions. What if you have a bug in the testing environment (or, for worse, in production) that you need to investigate? You will have to roll back the entire environment chain, even in local development environments.

There are plenty of articles on the cost of rolling back and having diverging environments. In some cases, strict corporate policies forbid a deployment that is not currently running and validated in the testing environment.

We even didn't take into consideration human error: manually selecting the wrong tag is easier than a pipeline that deploys the latest head from a branch...

If something goes wrong in a deployment, it's easier to fix and move on than to elaborate on a complicated workflow "just in case"!

After some time and explanations, we continued to use the "normal" workflow, and guess what? No rollback was ever made!

Bad Luck for the bad

"Despite my ghoulish reputation, I really have the heart of a small boy. I keep it in a jar on my desk."

Robert Bloch

Sometimes, bad luck also happens to villains.

A company contacted us in a hurry after finding an unknown EC2 instance after a weekend. Since the IT team was small, they made an internal check, and no one started that huge instance.

They were worried about some administrative credential leak that could have happened, so we fired up CloudTrail to find that an unknown IP address from a foreign country used an access key to first describe account quotas, limits, and regions and then started the instance.

The region chosen was eu-west-3 (Paris), an uncommon region. Lucky for the customer, that region was used as the main region for the workload, and with few instances, spotting that kind of anomaly was easy and was done in a matter of minutes.

After more investigation, it was found that the "usual" bots found a vulnerability in a website's PHP framework, obtained the access key (which unfortunately had broad permissions), and used the credentials to spawn a crypto-miner in a region that seemed a good idea for them because it was not common.

In the end, being a villain is not an easy task; bad luck can happen to anyone.

Regarding our newly acquired customer, after an emergency review, we made a plan to fix many things. We are still working on all the issues, but relying on good practices is better than being lucky!

By the way, having temporary credentials handled by IAM roles can avoid many headaches, trust us!

Home disautomation

> *"You know, I think this Christmas thing is not as tricky as it seems! But why should they have all the fun? It should belong to anyone! Not anyone, in fact, but me! Why, I could make a Christmas tree! And there's not a reason I can find, I couldn't have a Christmastime! I bet I could improve it, too! And that's exactly what I'll do!"*
>
> **Jack Skellington, the Nightmare before Christmas**

Ehi, this is me! Damiano from the past, do you remember? You are making fun of a lot of disasters, but you are not perfect. As a nerd, you are vulnerable to the lure of new technologies, eager to try them at every opportunity, even at home. So, be brave, and tell them about your home automation project that turned the lights in your house into rebels.

Ok, fair enough. This story is about how my laziness and passion for new technologies turned their back on me.

It was 2018, and I purchased a SonOff smart switch to use in combination with Alexa to turn on a lamp in my living room because a light switch was unavailable. It wasn't long before I learned about home automation with ESP8266 Wi-Fi chips, Home Assistant, Tasmota, and

MQTT protocol. Everything was in the early years, and the best way to learn is always to experiment, even with what you use daily.

And, as always, before learning the big picture and how things work, you have to make mistakes.

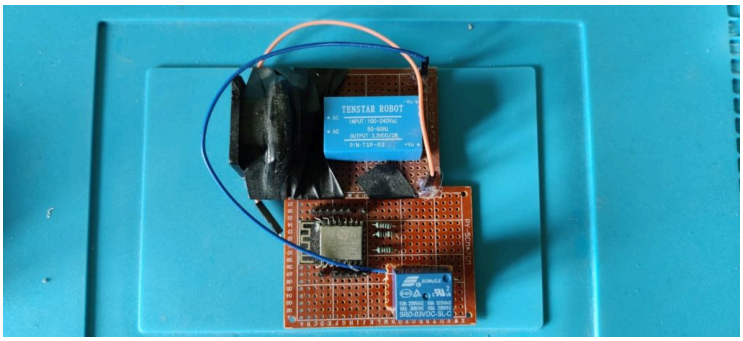
In July, I was on holiday in Sardinia, and my parents phoned me because they saw a light turned on in my kitchen. They entered my home and were not able to turn it off. Knowing that I was tinkering with lights at home, they already knew that maybe something wasn't working as expected.

It was obviously my fault, and this is why.

Having discovered ESP8266 chips and their capabilities, I started prototyping a "Wi-Fi remote switch extender" to control multiple light bulbs with a single switch, triggering different events for different actions, like double clicks, long presses, etc. The idea was to extend the capabilities of a simple light switch by embedding an ESP8266 chip and triggering webhooks to other smart devices (even a custom-made one).

With this kind of thing, I could turn everything on or off, and control even lights that weren't in the same room!

Here is an (ugly) prototype from that year:



The problem(s) related to the light that didn't turn off was that:

- Using webhooks, I had to make DHCP reservations on my Wi-Fi router and hard-code IP address in the Arduino code (or implement DNS auto-registration on a local run server)
- I forgot to make the reservation for the light switch
- A power outage occurred while I was on vacation, and the IP address for the light changed
- The switch I used as an "extender" was the one that controlled the light.

I could turn off the light remotely, but they could not turn it on again if they needed something in my home, but they had to phone me first!

What I learned from this episode?

1. Don't change the behavior of commonly used components. A light switch should act as a light switch. You can extend this concept to lambdas, containers, and so on; use the technology for what it is designed to.
2. Some things, even if they are "smart", should retain their functionality even if there is no network communication or a system fails.
3. Having a way to fix things (even from Sardinia) is nice.
4. "Everything fails all the time" is a motto, but also a way of life :D

The little evil green dot

> *"I met this six-year-old child with this blank, pale, emotionless face, and the blackest eyes... the devil's eyes. I spent eight years trying to reach him, and then another seven trying to keep him locked up because I realized that what was living behind that boy's eyes was purely and simply evil."*
>
> **Halloween**

This is the story of an evil little green dot and how it managed to make a multi-region migration fail in a matter of minutes.

It was a quiet night. After months of refactoring, containerization, and re-platforming, we finally released the new shiny version of a distributed communication system handling millions of clients and components.

A lot was changed and optimized: from bare metal servers around the world to small containers running on Fargate in different regions with custom scaling algorithms, from a single MySQL cluster to MongoDB instances, and so on. A throttling mechanism was also implemented to ease the migration and avoid congestion collapse (this re:Invent talk is a must-see about this kind of issue).

We had a dashboard to monitor the load and see how things were going. It was late, but the throttling mechanism to mitigate the load worked as expected, and everything was returning to normal. After a few minutes, everything stopped working, and the load on the database increased.

Being in a maintenance window and having the freedom to watch, we observed that clients couldn't connect anymore, timing out on API calls, and for worse, the database was at its peak usage.

We decided to throttle aggressively to understand which components were misbehaving, but everything was fine except the database. After 20 minutes, we shut down traffic from external systems and rebooted the database instances.

Even this didn't work: after a few minutes, the load was 99%, and we were confused.

We started the analysis and were ready to open a support ticket for an anomaly when we saw queries running, which maxed out the engine resource utilization.

We saw that db.collection.find() queries were running like crazy, and we saw the light at the end of the tunnel. That query was associated with the last bit added in the days before the migration: a simple lambda behind an API Gateway that reported the status of components for a system that contained a simple query:

```
for component in componentCol.find({"id":{"$in": ids}},{"id":1}):

                onlineIdsList.append(int(component["id"]))

    return {"id": onlineIdsList}
```

That API, which had been available for ages, was made available for components to find each other's status. Its data storage was ported from MySQL to MongoDB. It didn't experience any performance problems in the past with the old deployment, but this time, everything was going wrong, even if it was tested. Ultimately, it was a simple query to display a green dot if everything was online!

As maybe some of you are already thinking, the problem wasn't in the lambda or the database. With the increase of components connecting to the new system, the number of entities increased exponentially, going from some hundreds used in tests to millions, making the query analyze a lot of records on a database engine that is not made for this kind of activity.

On MySql, the query was a perfect fit for a relational engine, but it was a disaster for NoSQL databases; even after some optimizations, we had to stop and revert to the old system, delaying the migration for at least two weeks.

We refactored the queries, but even with that, dedicated search nodes are required, and the work is still ongoing to port a part of the system to a relational database and save money.

What did we learn? Even a simple green dot can be evil when dealing with complex systems, especially if there are millions of them!

Conclusion

Some say that the phrase "**three is a charm**" comes from the story of John Babbacombe Lee, who survived three hangings and was then let live, influencing the Old English law.

"There are only 365 days left until next Halloween!" — The Mayor, The Nightmare Before Christmas

So, is this the last episode of our scary series?

We don't think so, as we are humans dealing with tech problems. We know we will find new material for this series, so feel free to share your scary stories with us in the comments below; maybe you will see them in the next episode!

About Proud2beCloud

Proud2beCloud is a blog by beSharp, an Italian APN Premier Consulting Partner expert in designing, implementing, and managing complex Cloud infrastructures and advanced services on AWS. Before being writers, we are Cloud Experts working daily with AWS services since 2007. We are hungry readers, innovative builders, and gem-seekers. On

Proud2beCloud, we regularly share our best AWS pro tips, configuration insights, in-depth news, tips&tricks, how-tos, and many other resources. Take part in the discussion!

Damiano Giorgi

Ex on-prem systems engineer, lazy and prone to automating boring tasks. In constant search of technological innovations and new exciting things to experience. And that's why I love Cloud Computing! At this moment, the only "hardware" I regularly dedicate myself to is that my bass; if you can't find me in the office or in the band room try at the pub or at some airport, then!
