Nightmare infrastructure – episode 4. Just when you thought it couldn't get any worse...

29 October 2025 - 9 min. read

"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

Jack Skellington's declaration perfectly expresses an engineer's overconfidence, curiosity, and boundless desire to "improve" or re-invent a system, often with chaotic or disastrous results.

It captures the spirit of taking a simple, existing process and trying to make it better, only to discover that the original complexity was hiding just out of sight.

Many traditions are born without any serious intention, without planning, and as experiments. This series of articles is now something that we prepare and take notes on during the year, thinking about when it will be Halloween again to scare you.

So, without further ado, let's begin our journey through this year's horrors!

Schrodinger's container

"So we're in the box. We're the cat. We're both alive and dead. So, there are two separate realities, presumably until the comet passes."

Coherence, 2013

Schrodinger's cat is a paradox that explains quantum mechanics' indetermination principle: when you observe a system, you alter it, making it impossible to determine its initial state.

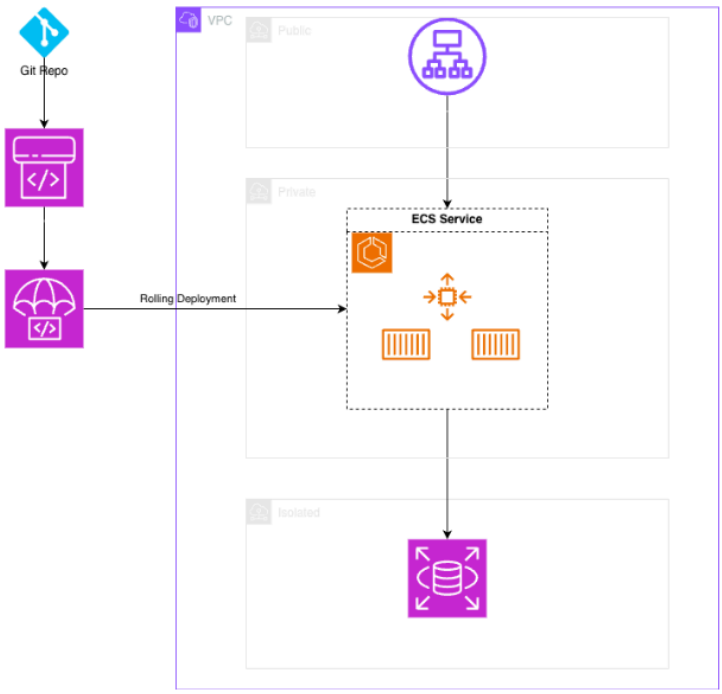
To make it clear to anyone, by using a metaphor, imagine that a cat and a vial of venom were put in a closed box, insulated from the external environment. Opening the box will break the vial, poisoning and killing the cat. But the cat has no food, nor water, so, at some time, he will surely be dead. The paradox is that, by opening the box, you will surely kill the cat, but if you don't open the box, you are not sure if the cat is alive or dead.

This can happen to a container, too.

We were involved in a refactoring project for a web application, the classical breaking down of the monolith from an EC2 instance to containers.

We started by analyzing the application, deploying a development environment in its dedicated AWS Account, and then proceeding with an initial containerization of the whole application to see how it interacted with other services. We then familiarized ourselves with CI/CD pipelines and adopted a cloud-native approach.

Here's the "boring" diagram of the infrastructure. Since the application was quite simple, we resorted to ECS Fargate to run the workload.



The application seemed to be fine, but after some initial testing the morning after the first deployment, we saw a 502 error. We thought about some temporary issues, since the application was working fine after a minute or two. In the afternoon, we noticed the same thing and started investigating.

We saw that the ECS task was restarted several times the night before, even without any traffic. The memory graph clearly indicated a memory leak. The strange issue was that there was no user traffic. The Application Load Balancer health check was enough to trigger the memory leak. So observing the system caused it to "die", like Schrodinger's paradox. The container was both running and dead: the old application's memory leak had been given a new life in the cloud.

The true tragedy wasn't the bug itself, but the fact that our architecture was now a memory leak delivery system. Simply containerizing a flawed application is not modernization; it is merely speeding up the failure rate.

Thank you, Mehmed, for the idea and the title!

Ariadne's thread

Ariadne whispers: "The Minotaur's den is in the very heart of the labyrinth. The sound of his breathing will show you in what direction you must go. Here is a sword, and here

is a clew of yarn, by means of which, after you have killed the monster, you can find your way back."

The Myth of Theseus and the Minotaur

When Theseus is in the Minotaur's maze, Arianna gives him a thread to guide him and help him find the way back to exit the maze.

Too bad for us that in this story, we had no one who gave us a guide.

It started with an experiment that became a big application used by many big customers. The company started with a single AWS account for all the environments to experiment. Suddenly, its application became a thing, and there was no time for governance. At some point, a Landing Zone could not be avoided anymore to govern AWS Cloud resources better, isolate resources and access, and comply with customers' requests.

After defining what was better for the use case, deploying AWS Accounts, integrating with the IdP, and implementing a centralized network, it was time to move the workloads from the single AWS account to their dedicated one.

We didn't dig much into the application, but everything was containerized and followed a microservice approach, so we started the interview process to better define the dependencies. Since time was a real constraint, we began deploying things in the new account.

The issue was that, after deploying the first databases and migrating data, every service seemed to be linked to something else: an EC2 autoscaling group to obtain specific configuration data, another microservice to be able to start up the frontend, and then a batch job that queried the service that we were deploying to populate a table for the first service.

Trying to trace a dependency diagram ended up in a Picasso drawing.

The experiment became an application, and the application became a curse. We quickly realized the true labyrinth wasn't the migration: it was the result of the history of dependencies, lack of documentation, and yesterday's decisions.

We already saw the difference between containers and microservices here.

Too much monitoring

“Where is the knowledge we have lost in information?”

T.S. Eliot - Choruses from the Rock

Knowing what’s going on with your customer is always a challenge; if you have an e-commerce site, every little detail can impact your user experience, driving sales up or down.

Even a small delay when a customer visits your website can result in a loss of money, so it's crucial to find a way to monitor the user experience.

Meet MegaCorp, which has e-commerce made with every no-code platform and service available. When something went wrong, troubleshooting was nearly impossible, and knowing which component made the experience worse was daunting.

To worsen things, deployments were scheduled every 15 days for the dev and test environments and every month for the production environment.

Lucky for us, Amazon RUM makes this activity easy, giving you insights into every little event in the browser. You can monitor your users' experience with different components, such as resource download and page render times, and even monitor JavaScript exceptions in the console.

Amazon RUM inserts a little JavaScript that reports user events to an endpoint. It is good practice to test for unwanted interactions in a test environment. Also, since its billing uses the pay-as-you-go model billed by the generated events, you have to find the right balance between monitoring every user or enabling sampling and selecting the kind of user experience events to report and record.

So, we started with a simple testing environment to find the sweet spot and keep everything under control.

Our tests reported no issues with the integration. However, we had to exclude events for image loading since a fast CDN was in use, and the number of events would be too large.

There were some JavaScript errors in the console, but we were told that they were due to developers testing a new feature.

So, production it is! After waiting for the deployment for 20 days, RUM went live, and it was working as expected, but...

After half a day of data collection, we received a budget alert: our threshold of 80% of the monthly budget ($6000) had already been reached.

We checked the numbers: visitors and page views were aligned with our forecast, but something was wrong with the number of events. We discovered that every visit was throwing at least 5 to 6 JavaScript exceptions in the console, which RUM recorded. Errors were always the same due to a bug that was fixed but not put into production due to the release cycle.

We couldn't deactivate RUM because the deploy window was gone. Lucky for us, RUM uses a Cognito Identity pool to create temporary credentials to push data. Deactivating Cognito stopped the billing. In the end, a perfectly monitored bug is just

a perfectly expensive bug, and the only thing more crucial than a monitoring solution is a process to control bugs and deployments in production! When the deployment cycle can't keep up with the technical debt, the tools designed to give you clarity can very quickly turn into your greatest expense that doesn't add any value.

The usual suspects

"In a world where nothing is what it seems you've got to look beyond..."

Roger Kint (Kevin Spacey) - The Usual Suspects

“Our infrastructure is cloud-native”.

When we heard this, we were happy but a little suspicious.

After a split between companies, we were engaged to move an AWS account out of an organization and implement a new, shiny Landing Zone. This meant a new set of accounts, centralized networking, service delegations, and a new IAM Identity Center federation.

After the initial assessment and interviews, we were asked to troubleshoot some latency and unavailability issues in the front-end user-facing web application, hosted in an Amazon EKS cluster.

During some peak hours, latency seemed to increase until the web application was unavailable for minutes. Then, without any action, the issue solved itself.

Even if it was a microservice-based application, no tracing was available. After implementing a solution, the picture was clear: It wasn't so cloud-native: only the user-facing application was running natively on Kubernetes. All the other components were

running in an on-premise datacenter, with a VPN connection configured, a single active tunnel, and no BGP. During peak hours, traffic exceeded the 1.25 Gbit quota for the tunnel, adding latencies, delays, and packet loss, with an avalanche effect.

We had built a shiny new cloud kingdom, but the crown jewels were still sitting in a dusty server that was quietly throttling our entire modern infrastructure.

The light at the end of the tunnel

And so, it's time to close the coffin lid on another year's collection of infrastructural nightmares with a stark reminder: simply adopting the latest technology does not grant you immunity from the sins of the past.

But fear not! The horrors will return next Halloween season. Until then, may your pipelines be green and your memory graphs flat.

We've spun our yarns, now we want to hear yours! What unspeakable horrors are lurking in your production environment right now?

Drop us a comment below, let's swap tales of terror before the lights go out for good.

Happy Halloween, and may your infrastructure rest in peace... for now.

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!
