

Troubleshooting evolution from VM to Containers

Bassam Gamal
Eric Söderberg

May 17, 2022

1 Introduction

A core part of software engineering is troubleshooting issues, and it is central for DevOps in particular. Engineers usually spend a significant portion of their daily time trying to understand problems, reproduce scenarios, debug, and fix issues. Some of these issues might cost the company millions of dollars, so finding and fixing them should be a very fast process.

But, to find them engineers need to know technical details about the issue, the environment, the time, and many other things. Each minor detail helps to shed light on those nasty bugs. Sometimes engineers will have to debug the code to reproduce the issue and see what is happening that caused the issue.

Increasingly commonly in later years, the relevant pieces of information may span a complex cloud-based architecture from which information has to be consolidated appropriately for troubleshooting to possibly take into account all the variables potentially affecting the problem. The troubleshooting process went through an evolution process from blindly trying to understand to taking snapshots of information to replicate the issue.

2 History

In the old days, when everything was just running on a VM.

A VM is an isolated operating system version whose state could be saved to create a snapshot and easily duplicated. Engineers had a simple life. Simple logging was the main weapon in the engineer's arsenal. Apps wrote logs that captured the state, what was happening, and all technical information that the engineer needed. Also, the apps followed a monolithic architecture. Apps had multiple jobs, and functionalities, but they consisted of layers in the same place. So tracing a problem happened in place, and engineers could identify which layer was the issue.

Logs were stored in the file system, so if something happened, engineers could just examine the log file to look for relevant clues. Later on, files stopped being reliable as they started to grow huge.

To tackle this issue, transactional databases were used to store logs from different apps. When the size of the database grew, some maintenance was needed to make sure the database remained in good shape.

When high loads started to be a need in apps, databases weren't sufficient anymore. Instead, streaming databases such as Kafka[12] or ELK[6] stack were used. All logs were sent from apps to Kafka or ELK, and then the engineer would just go and check the logs in order to troubleshoot the issue.

Horizontal scaling wasn't that easy, so a typical app might have had an instance or two behind a load balancer if availability or heavy load was a need, and engineers relied mostly on vertical scaling of servers if needed.

This meant that logs were coming from few sources and servers, and were easy to trace. The cycle started when an engineer would try to troubleshoot an issue. If there was no id to trace, or other property that makes the operation unique, then the solution was to filter by a specific time frame of logs.

- Best-case scenarios: the issue is easily clear from the logs, and the fix is done.
- Worst-case scenarios: The engineers didn't have enough information, so more logs were added to get this info. CI/CD processes weren't in place, so manual deployments were highly error-prone. This meant that tracing and debugging were easy, but deploying wasn't.

VMs were huge in size, and they didn't provide the right level of isolation for engineers. Many times issues happened between similar VMs, as syncing them wasn't easy. Maintaining them was a burden which was handled by system engineers. It was common for issues to occur where engineers had to spend significant time comparing VMs to figure out their differences - a process which would often have to be repeated for each new deployment.

3 Current State of the Art

Everything has changed a great deal since the usage of Docker containers went viral. Now, horizontal scaling has become the norm in releasing software.

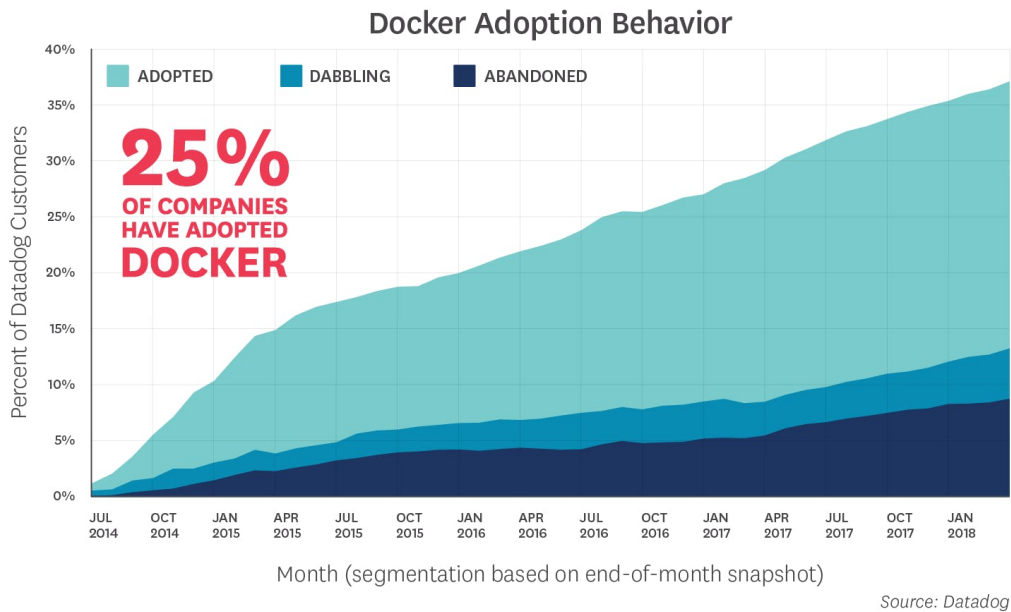


Figure 1: Docker adoption over time, as of 2018 [18].

Let's define some terms:

- A docker image is a recipe which describes the needed steps to run an app with a very specific environment.
- A container is a running image.
- An orchestrator (such as Kubernetes) is responsible for running containers and provides many benefits like scalability, high availability, service discovery, and many other things that are provided out of the box.
- A microservice/micro-frontend is a small self-contained app/service with a focused set of either domain or business functionalities.

These tools enabled engineers to write better software in a more reliable fashion. Ideas like microservices/apps became feasible and horizontal scaling turned effortless. Any app could have several replicas running with a simple setup. Engineers easily shipped software to production.

Containers if you compare them to VMs:

- Much smaller in size, leading to more efficient use of resources
- Provide a better isolation level.
- Are described well in Dockerfiles, so that any engineer can recreate them.
- Provide more granular control of what is running inside them.

However, these advances didn't come without any downsides; troubleshooting problems in a heterogeneous distributed system is far more complicated than in a monolithic system. Logging on the filesystem isn't an option. Many replicas are running, and containers together with

their filesystems are usually ephemeral by design.

Also storing logs in the transactional database isn't feasible, as the volume of logs increases drastically, and thus using one for logging would incur too great of a latency cost.

Loosely coupled solutions were used like the Sidecar pattern, and streaming databases like Kafka, Kinesis, and many others - where all the logs were centralized in one place.

The troubleshooting process itself could start, but this time far more technical details are needed.

Unlike before, it's no longer only App1 that is failing, but also Replica 9 of App1 that is failing specifically in the GMT+10 timezone, in containers deployed only on Australian servers. The temporary nature of many of the containers means that by the time an engineer is reviewing the issue, the original node and its accompanying program may no longer even exist. It is clear that many new dimensions have been added. Shipping the fixes is however much faster after having automated CI/CD processes and new deployment strategies like Zero downtime, Canary, and Blue-Green.

The move to containers has also meant that reproducibility can be ensured on a level not previously possible. The term "well, it works on my computer" has become far less frequent when the software's deployment environment can be ensured to have core characteristics and underlying structure identical to that of the developer.

While in the past one might have to ensure that the bug does not stem from for instance a difference in the operating system or package versions, containers largely eliminate this issue.

4 Comparison

Troubleshooting in monolithic apps was easier as horizontal scaling wasn't an option, and apps consisted of many layers in the same place. But, shipping the fix was a slow operation. In addition, engineers usually spent considerable time navigating issues stemming from the environment and configuration of the VMs instead of focusing on the real bugs.

On the other hand, troubleshooting problems in the container world is much harder, as systems are distributed, and scaled horizontally. Even though apps are microservices/frontend, the amount of logs is enormous. Yet, shipping and testing the change in production is much easier with CI/CD and new deployment strategies - so the total time from a problem occurring to a fix being deployed can very well still be considerably shorter.

This paradigm change also requires a higher level of knowledge and time to implement sufficient troubleshooting support. Any programmer can trivially learn how to write to a file on a disk and later read it, as was the case in the earliest days. The task of reading it could even fall on someone with no significant technical background since simply opening a text file and finding something that looks out of place requires only very basic computer skills.

In contrast, the more advanced state-of-the-art tools fall far outside the scope of traditional programming and computer knowledge. The underlying concepts and important theory may be hard to grasp, such as important performance characteristics and picking the correct product.

Merely operating the products requires a more specific skill set as well, unless specifically developed to cater to someone less familiar. The development and extension of the tools themselves also usually constitute large complex projects on their own and is typically not something that can easily be developed in-house by a small inexperienced team in a short period, hence the common use of some of the ready-made products mentioned in this essay.

It is not only developmental complexity but also a time in which requirements differ greatly. Even having the required prerequisite knowledge, The developer has to weigh the increased overhead in terms of time required, versus its flexibility once implemented.

Indeed, setting up a complex full-fledged cloud monitoring system with an accompanying streaming database for a small hobby project with only one singular VM, where the code itself may be minuscule, may well be a completely inappropriate use of resources compared to simpler solutions.

On the other hand, for large projects spanning several (possibly varying) services across different types of infrastructures, the more advanced modern solutions prove to be essential, and well worth both the initial and continued investment of time and knowledge.

The use of containers still carries with it some overhead similar to that explained above, but their use is far more widespread because their upsides are evident and significant even to the smallest project. Any project which needs to be deployed elsewhere than where it is locally developed (and often even there), stands to benefit from the reproducibility it bestows. Projects become portable by default rather than requiring adaptation specifically for a change in environment, and troubleshooting issues become more straightforward since many variables are eliminated as the potential culprit of error. The overhead in time is in most cases rather small, once the developer has learned about how containers work and how to develop them.

Going through these complications, engineers realized that they must be more proactive, and see issues before they happen. The need for more advanced tools arose, and many techniques/tools like Static code scanning, AI, Machine Learning, and observability started to be available.

So now engineers must filter many of the issues before they happen, predict what might happen, and make use of the huge amount of data they have.

5 Tools & Patterns

In the monolithic world, VMs, file systems, transactional databases, and sometimes streaming databases like Kafka (In case of heavy load) were used to store logs.

In the world of microservices, highly scalable streaming databases like Kinesis, and Kafka were used to store all logs. Cloud monitoring tools like: AWS CloudWatch, and Azure AppInsights are used for instrumentation. Patterns like Side Car Container and TransactionId are used. Also, deployment strategies like Canary, and Green/Blue are used.

5.1 Cloud & Managed Solutions

These days cloud providers are a cornerstone of most companies, not even exclusively within IT, as can be seen in fig. 2

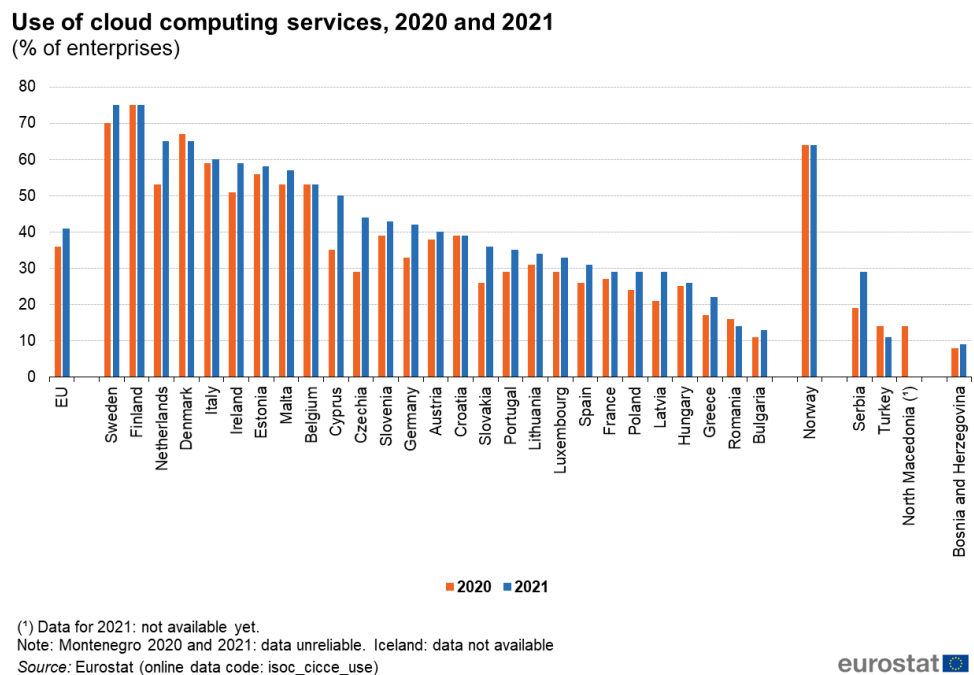


Figure 2: Use of cloud computing services within the European Union [4].

While there are often suitable and highly customizable open-source tools, such as the aforementioned Apache Kafka, self-hosting these on-premises carries with it some significant tradeoffs. For starters, the high customizability often leads to it having a significant startup cost in terms of setup before it can meet the businesses' specific needs. It also requires physical infrastructure to run on, and the technical knowledge as well as time of one or more engineers to not only set it up but also keep it running smoothly.

Furthermore, as business grow and expand into other products provided by cloud providers, with it comes an increased need for integration between said services, and this naturally includes its monitoring and logging solutions.

It is for these reason that managed solutions may be a more suitable option for many businesses. While they generally offer fewer options in terms of customizability, their appeal come from an ability to work with a large number of products and usecases out of the box. A popular previously mentioned example is the by AWS managed Kinesis, which provides functionality similar to the self-hosted Kafka. These products usually integrate very well with other products within their respective ecosystems, including everything from controlling clusters of nodes, alerting, and setting up monitoring charts, and do not require the same kind of skillset or time to setup and maintain. This frees up developers to instead focus on fixing the bugs, rather than setting up the tools to find them.

These enterprise level solutions are robust enough that the companies tend to provide SLAs, guaranteeing uptime on their services. Later advancements in the field have even started to pivot toward utilizing the massive datacenters' capabilities to use AI to analyze and troubleshoot metrics in real-time.

6 Reflections

When I reflect first on the old way of troubleshooting from my first job more than a decade ago. I was developing a huge desktop application written in MFC C++. Sometimes, I used pop-up windows to see debug information, which isn't ideal for production environments.

On production systems, I used to log information and errors to a file. So when something happened or I wanted to trace something, I went and checked the file. Sometimes when errors happened, I could even know which line of code caused the issue, and then the rest is history.

But, after a while, if no one takes care of the log files, they become huge, so a technique we used to do was to use a new log file every day. Eventually that caused disk space issues that either required someone to manage them or use a cron job to delete old files.

Then, years later in a web app or cron jobs, we started to use SQL databases for logging, which suffered from similar issues of Disk Space, System latency. So workarounds like data archiving/cleaning, and replicas were used. Still, the solution wasn't ideal, especially when database servers went down.

Then when I reflect on my recent experience working as an architect for one of the biggest Swedish operators. Hundreds of microservices were running as containers on Kubernetes, which makes sure services are up and running.

A sidecar container collects logs and sends them to our managed Kafka cluster on the cloud. It in turn would scale up to absorb an enormous amount of logs. We had even different availability zones for some resources on the cloud so when a complete cloud region goes down, your solutions don't. That happened a couple of times, as the service provided has a SLA.

Also, data archiving was automatically set up, so many of the issues engineers worried about, are automatically fixed.

Yet, sometimes troubleshooting a critical issue could take a few hours for teams. Imagine the complicated matrices of communication between these different microservices.

7 Conclusions

Troubleshooting, just like most other aspects of IT, went through a cycle of evolution. It was easy when the old technologies, paradigms, and tools were used, but engineers suffered from many other issues. This was both due to manual work, as well as primitive tools.

Nowadays, the tools are very advanced, managed even by providers with known SLAs, and automation is everywhere. But, troubleshooting became harder and more complex due to the increased complexity and scale of systems. More dimensions meant more time to troubleshoot issues and that led to the need for new advanced tools that utilize things like: AI, observability, and Machine Learning so engineers could be more proactive than reactive.

References

- [1] Amazon Kinesis - Process & Analyze Streaming Data - Amazon Web Services. <https://aws.amazon.com/kinesis/>.
- [2] AWS Kinesis vs Kafka comparison: Which is right for you? <https://www.softkraft.co/aws-kinesis-vs-kafka-comparison/>.
- [3] bliki: CanaryRelease. <https://martinfowler.com/bliki/CanaryRelease.html>.
- [4] Cloud computing - statistics on the use by enterprises. https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Cloud_computing_-_statistics_on_the_use_by_enterprises.
- [5] Dataflow. <https://cloud.google.com/dataflow>.
- [6] The ELK Stack: From the Creators of Elasticsearch. <https://www.elastic.co/what-is/elk-stack>.
- [7] Microservices Pattern: Distributed tracing. <http://microservices.io/patterns/observability/distributed-tracing.html>.
- [8] Microservices Pattern: Exception tracking. <http://microservices.io/patterns/observability/exception-tracking.html>.
- [9] Microservices Pattern: Log aggregation. <http://microservices.io/patterns/observability/application-logging.html>.
- [10] Microservices Pattern: Monolithic Architecture pattern. <http://microservices.io/patterns/monolithic.html>.
- [11] What are microservices? <http://microservices.io/index.html>.
- [12] What is Apache Kafka®? <https://www.confluent.io/lp/apache-kafka/>.
- [13] What is Container Orchestration? extbar VMware Glossary. <https://www.vmware.com/topics/glossary/content/container-orchestration.html>.
- [14] Blue-green deployment. https://en.wikipedia.org/w/index.php?title=Blue-green_deployment&oldid=1077231965, Mar. 2022. Page Version ID: 1077231965.
- [15] CI/CD. <https://en.wikipedia.org/w/index.php?title=CI/CD&oldid=1080944167>, Apr. 2022. Page Version ID: 1080944167.
- [16] Docker overview. <https://docs.docker.com/get-started/overview/>, May 2022.
- [17] Docker overview. <https://docs.docker.com/get-started/overview/>, May 2022.
- [18] DATADOG. 8 surprising facts about real Docker adoption. <https://www.datadoghq.com/docker-adoption/>, June 2018.

- [19] DAVOUDIAN, A., AND LIU, M. Big data systems: A software engineering perspective. *ACM Computing Surveys (CSUR)* 53, 5 (2020), 1–39.
- [20] DINH-TUAN, H., MORA-MARTINEZ, M., BEIERLE, F., AND GARZON, S. R. Development frameworks for microservice-based applications: Evaluation and comparison. In *Proceedings of the 2020 European Symposium on Software Engineering* (2020), pp. 12–20.
- [21] EDPRICE-MSFT. Sidecar pattern - Azure Architecture Center. <https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar>.
- [22] YUSSUPOV, V., SOLDANI, J., BREITENBÜCHER, U., BROGI, A., AND LEYMAN, F. Faasten your decisions: A classification framework and technology review of function-as-a-service platforms. *Journal of Systems and Software* 175 (2021), 110906.