Netflix

Netflix is an entertainment company, it has surpassed many top tech companies in terms of tech innovation. With its single video-streaming application, Netflix has significantly influenced the technology world with its world-class engineering efforts, culture, and product development over the years.[1]

It all began with their database corruption in 2008 when Netflix couldn't ship DVD's to their customer.
The decided to go native cloud and now they are using AWS as a Infrastructure as a Service.
As a significant part of their transformation, Netflix converted its monolithic, data center-based Java application into cloud-based Java micro services architecture. It brought about the following changes:

- Denormalized data model using NoSQL databases
- Enabled teams at Netflix to be loosely coupled
- Allowed teams to build and push changes at the speed that they were comfortable with
- Centralized release coordination
- Multi-week hardware provisioning cycles led to continuous delivery
- Engineering teams made independent decisions using self-service tools[2]

As a result it helped Netflix accelerate innovation and stumble about he DevOps culture.

Engineers at Netflix perceived that the best way to avoid failure was to fail constantly. And so they set out to make their cloud infrastructure more safe, secure, and available the DevOps way by automating failure and continuous testing.

Chaos Monkey

Netflix created Chaos Monkey, a tool to constantly test its ability to survive unexpected outages without impacting the consumers. Chaos Monkey is a script that runs continuously in all Netflix environments, randomly killing production instances and services in the architecture. It helped developers:

- Identify weaknesses in the system
- Build automatic recovery mechanisms to deal with the weaknesses
- Test their code in unexpected failure conditions

- Build fault-tolerant systems on day to day basis

The Netflix Simian Army

Chaos Monkey randomly disables our production instances to make sure we can survive this common type of failure without any customer impact. The name comes from the idea of unleashing a wild monkey with a weapon in your data center (or cloud region) to randomly shoot down instances and chew through cables – all the while we continue serving our customers without interruption. By running Chaos Monkey in the middle of a business day, in a carefully monitored environment with engineers standing by to address any problems, we can still learn the lessons about the weaknesses of our system, and build automatic recovery mechanisms to deal with them. So next time an instance fails at 3 am on a Sunday, we won't even notice.

Latency Monkey induces artificial delays in our RESTful client-server communication layer to simulate service degradation and measures if upstream services respond appropriately. In addition, by making very large delays, we can simulate a node or even an entire service downtime (and test our ability to survive it) without physically bringing these instances down. This can be particularly useful when testing the fault-tolerance of a new service by simulating the failure of its dependencies, without making these dependencies unavailable to the rest of the system.

Conformity Monkey finds instances that don't adhere to best-practices and shuts them down. For example, we know that if we find instances that don't belong to an auto-scaling group, that's trouble waiting to happen. We shut them down to give the service owner the opportunity to re-launch them properly.

Doctor Monkey taps into health checks that run on each instance as well as monitors other external signs of health (e.g. CPU load) to detect unhealthy instances. Once unhealthy instances are detected, they are removed from service and after giving the service owners time to root-cause the problem, are eventually terminated.

Janitor Monkey ensures that our cloud environment is running free of clutter and waste. It searches for unused resources and disposes of them.

Security Monkey is an extension of Conformity Monkey. It finds security violations or vulnerabilities, such as improperly configured AWS security groups, and terminates the offending instances. It also ensures that all our SSL and DRM certificates are valid and are not coming up for renewal.

10-18 Monkey (short for Localization-Internationalization, or l10n-i18n) detects configuration and run time problems in instances serving customers in multiple geographic regions, using different languages and character sets.

Chaos Gorilla is similar to Chaos Monkey, but simulates an outage of an entire Amazon availability zone. We want to verify that our services automatically re-balance to the functional availability zones without user-visible impact or manual intervention.

Netflix Container journey

Netflix designed a container management platform called **Titus** to meet its unique requirements.

Titus provided a scalable and reliable container execution solution to Netflix and seamlessly integrated with AWS. In addition, it enabled easy deployment of containerized batches and service applications.[2]

Google

Site Reliability Engineering (SRE) is a term (and associated job role) coined by Ben Treynor Sloss, a VP of engineering at Google. DevOps is a broad set of principles about whole-lifecycle collaboration between operations and product development. SRE is a job role, a set of practices we've found to work, and some beliefs that animate those practices. If you think of DevOps as a philosophy and an approach to working, you can argue that SRE implements some of the philosophy that DevOps describes, and is somewhat closer to a concrete definition of a job or role than, say, "DevOps engineer." So, in a way, ***class SRE implements interface DevOps*.**

Unlike the DevOps movement, which originated from collaborations between leaders and practitioners at multiple companies, SRE at Google inherited much of its culture from the surrounding company before the term *SRE* became widely popularized across the industry. Given that trajectory, the discipline as a whole currently does not foreground cultural change by default quite as much as DevOps. (That doesn't imply anything about whether cultural change is necessary to do SRE in an arbitrary organization, of course.)[4]

SRE vs DevOps: How do they differ?

Area	SRE	DevOps
Core philosophy and focus	An engineering-driven approach prioritizing system reliability and stability	A cultural shift emphasizing collaboration and efficiency in software development
Key practice	Implementing automated solutions for operational challenges and reliability engineering	Integrating continuous integration and delivery with infrastructure automation
Workflow and process	Focuses on proactive system monitoring, incident management, and post-incident analysis	Centers on streamlining the development, testing, and deployment cycle
Metrics and KPIs	Tracks SLOs, error budgets, and MTTR	Measures deployment frequency, change failure rate, and lead time for changes

[5]

What SRE and DevOps Have in Common

Ways of Working

●Adopting specific ways of working

●Solve communication problems break down silos between different organisational units.

●Improving the team or the organisation rather than the individual

●Leverage a collaborative culture with shared ideas, processes, practices, and technologies with the goal to streamline product development to maximise business value.

●Embrace feedback loops, a blameless culture, and psychological safety.

●Leverage different team topologies (e.g., central team, a coach squad model).

Production System

●Improve efficiency, productivity and improve customer (end-user) satisfaction.

●Improving the speed and quality of applications and services.

●Cost savings through improved process and automation work eliminating effort and duplication.

Tooling

●The toolboxes of both are similar, or even overlap, but used for different automation purposes. [6]

AWS

Amazon is one of the most prolific tech companies today. Amazon transformed itself in 2006 from an online retailer to a tech giant and pioneer in the cloud space with the release of **Amazon Web Services (AWS)**, a widely used on-demand **Infrastructure as a Service (IaaS)** offering. [9]

AWS provides a set of flexible services designed to enable companies to more rapidly and reliably build and deliver products using AWS and DevOps practices. These services simplify provisioning and managing infrastructure, deploying application code, automating software release processes, and monitoring your application and infrastructure performance.[7]

Continuous Integration and Continuous Delivery

The AWS Developer Tools help you securely store and version your application's source code and automatically build, test, and deploy your application to AWS or your on-premises environment.

Start with AWS CodePipeline to build a continuous integration or continuous delivery workflow that uses AWS CodeBuild, AWS CodeDeploy, and other tools, or use each service separately.

Microservices

Build and deploy a microservices architecture using containers or serverless computing.

Infrastructure as Code

Provision, configure, and manage your AWS infrastructure resources using code and templates.

Monitor and enforce infrastructure compliance.

Monitoring and Logging

Record logs and monitor application and infrastructure performance in near real-time.

Platform as a Service

Deploy web applications without needing to provision and manage the infrastructure and application stack.

Version Control

Host secure, highly scalable Git repositories in the cloud.

Horizontal scaling

Horizontal scaling, commonly referred to as scale-out, is the capability to automatically add systems/instances in a distributed manner in order to handle an increase in load. Examples of this increase in load could be the increase of number of sessions to a web application. With horizontal scaling, the load is distributed across multiple instances. By distributing these instances across Availability Zones, horizontal scaling not only increases performance, but also improves the overall reliability.

In order for the application to work seamlessly in a scale-out distributed manner, the application needs to be designed to support a stateless scaling model, where the application's state information is stored and requested independently from the application's instances. This makes the on-demand horizontal scaling easier to achieve and manage.

This principle can be complemented with a modularity design principle, in which the scaling model can be applied to certain component(s) or microservice(s) of the application stack. For example, only scale-out Amazon Elastic Compute Cloud (EC2) front-end web instances that reside behind an Elastic Load Balancing (ELB) layer with auto-scaling groups. In contrast, this elastic horizontal scalability might be very difficult to achieve for a monolithic type of application.

Leverage the content delivery network

Leveraging Amazon CloudFront and its edge locations as part of the solution architecture can enable your application or service to scale rapidly and reliably at a global level, without adding any complexity to the solution. The integration of a CDN can take different forms depending on the solution use case.

For example, CloudFront played an important role to enable the scale required throughout Amazon Prime Day 2020 by serving up web and streamed content to a worldwide audience, which handled over 280 million HTTP requests per minute.

Go serverless where possible

As discussed earlier in this post, modular architectures based on microservices reduce the complexity of the individual component or microservice. At scale it may introduce a different type of complexity related to the number of these independent components (microservices). This is where serverless services can help to reduce such complexity reliably and at scale. With this design model you no longer have to provision, manually scale, maintain servers, operating systems, or runtimes to run your applications.

For example, you may consider using a microservices architecture to modernize an application at the same time to simplify the architecture at scale using Amazon Elastic Kubernetes Service (EKS) with AWS Fargate.[8]

Conclusion

As we have seen for three different leading IT companies implementing DevOps culture and tools has became a successful story.

Netflix uses AWS for CI/CD but have implemented his own Simian Army when AWS has attack or network issues Netflix services are still available.

As for the small organization in my opinion they must accept small steps leading them into dev ops culture and strategy for start it will be hard but the outcomes will be more efficiency, fast deploying but more money.

References:
1.https://medium.com/@maeydhaw/case-study-how-netflix-became-a-master-of-devops-7f6f6fa8ad86
2.https://www.simform.com/blog/netflix-devops-case-study/
3.https://insights.sei.cmu.edu/blog/devops-case-study-netflix-and-the-chaos-monkey/
4. https://sre.google/workbook/how-sre-relates/
5.https://www.squadcast.com/devops-best-practices/site-reliability-engineering-vs-devops
6.https://www.devopsinstitute.com/sre-and-devops/
7.https://aws.amazon.com/devops/
8.https://aws.amazon.com/blogs/architecture/architecting-for-reliable-scalability/
9.https://insights.sei.cmu.edu/blog/devops-case-study-amazon-aws/