International Journal of Core Engineering & Management
Volume-6, Issue-12, 2021 ISSN No: 2348-9510

OPTIMIZING MICROSERVICES DEVELOPMENT: THE ROLE OF MICRO CONFIGURATION FRAMEWORKS AND MONOREPO STRATEGIES

Akash Rakesh Sinha
Software Engineer 2

Abstract

Microservices architecture is a modern architectural style that brings scalability and maintainably in application development. In this paper, we explore how micro configuration frameworks and monorepo strategies improve microservices development. From monolithic architectures to microservices: benefits and challenges of this shift. We talk about architectural patterns and development methodologies, tools, and technical considerations for implementing microservices effectively. We're taking inspiration from real world examples where these concepts have been successfully applied. Finally, we discuss evolving trends and provide suggestive directions for future research, stating how microservices are still developing and will likely inform the future of software engineering.

Keywords: Microservices, Monorepo, Micro Configuration Frameworks, Software Architecture, Continuous Integration, Continuous Deployment, Scalability, Security, DevOps, API Gateway, Containerization, Service Mesh, Mumbai Software Development

I. INTRODUCTION

1.1. Background and Motivation

The world of software development has transformed over the past decade. As demands grow for systems to be more adaptable and scalable with faster deployment cycles, a shift is taking place from monoliths to microservices. Monolithic applications have the disadvantage of how tightly coupled all their components are, which makes them difficult to be maintained or scaled, because the applications are interwoven into a single codebase. In contrast, microservices splits the application into loosely coupled services, which can then be developed, deployed, and scaled independently.

The need for applications to be responsive and resilient is the primary force behind this evolution. When businesses adopt microservices, they frequently see significant gains in performance, agility and scalability. With a microservices architecture in place, innovation cycles in organizations can happen at a faster pace, allowing businesses to change direction quickly to meet up with new market conditions, as customers expect products and features to be delivered at in record time.

International Journal of Core Engineering & Management
Volume-6, Issue-12, 2021 ISSN No: 2348-9510

1.2. Purpose and Scope

This paper will reveal how micro configuration frameworks and monorepo strategy can further optimize microservices development. We will start with giving a definition of those concepts, assess their pros and cons and deliver some practical insight on how to use them in real life. We hope by closing this theory–practice gap, we can provide valuable lessons for practitioners as well as researchers who want to promote software innovation and contribute to industrial gains.

We are in an era of growth like never before, in the software development industry where speed and agility are everything. To cope with the ever-shifting landscape in modern development, microservices have emerged as a leading solution to this trend. As Sundar Pichai, the Google CEO, so aptly said, "This is an incredibly exciting time in technology. Microservices, I think, are central to this change." In this paper we will explore how micro configuration frameworks play together with monorepo strategies to bring even more possibilities for micro services.

II. MICROSERVICES ARCHITECTURE AND DEVELOPMENT APPROACHES

2.1. Overview of Microservices Architecture

Definition and Core Principles

Microservices architecture is a software development technique / approach in which each application is developed with a set of small, autonomous services where each service runs in its own process and communicate with each other using lightweight techniques, mainly through an API. Several guiding principles define this architectural style(Thönes, 2015).

- One rule is decentralization, encouraging data to be managed by individuals within each service and governance to be distributed instead of combining everything in one system.
- The other one is autonomy, where every microservice can be developed, deployed, and scaled independently, allowing teams to work in parallel and reducing cross-team dependencies.
- Scalability is an ability to scale an important aspect of microservices, as they can be scaled horizontally or vertically to accommodate higher volumes of traffic or workloads.
- Finally, the architecture is developed with resilience to ensure that if one of the services fails, the entire system is not compromised.

Microservices architecture promotes independent, agile development processes with continuous delivery and deployment. It fits the culture of speed prevalent in tech firms and startups, where pushing the frontiers via iterative development must be done to remain viable in the market.

International Journal of Core Engineering & Management
Volume-6, Issue-12, 2021 ISSN No: 2348-9510

2.2. Architectural Patterns

To implement microservices effectively you must understand the different architectural patterns with their pros and cons.

Event-driven architecture

In an event-driven architecture, services interact by emitting events and responding to them. Well, it is called popularly the key advantage of the event-based architecture — decoupled interaction that helps the services work in isolation and react to changes in an asynchronous manner. In an e-commerce example, for instance, when a customer places an order, the order service publishes an event that the inventory and shipping services subscribe to perform an action that decreases the stock levels and starts the delivery process respectively.

API-Driven Architecture

The services expose their functionalities using APIs—usually RESTful or GraphQL– which defines the explicit contracts for interaction. This allows loose coupling to be maintained across services while still allowing communication to be synchronous if required. The simplicity and ubiquity of HTTP protocols is one of the key reasons API-driven architecture is very popular and widely used.

Service Mesh

A service mesh adds a dedicated layer to streamline service-to-service communication that takes care of things such as load balancing, encryption, and authentication. Tools like Istio and Linkerd are abstractions over these capabilities and let developers focus on business logic by managing them behind the scenes. This becomes particularly helpful in sophisticated microservices landscapes, where communication between services can get highly complicated.(Chandramouli& Butcher, 2020)

2.3. Monorepo vs Polyrepo

How organizations organize their code repositories has an important impact on their development workflows in a microservices-based architecture.

Monorepo (Monolithic Repository)

A monorepo strategy merges the code for several services into a single repository. There is a clear advantage to this strategy. One example is that teams have streamlined access to shared libraries and components, which allows for collaborating and ensuring consistent coding standards across the organization. This also simplifies CI/CD pipelines, as a single repository can house build and deployment scripts. But as the codebase scales, monorepos can become unwieldy to manage, resulting in increased build times and a higher chance of merge conflicts. Fast-growing teams need solid tooling and strict processes to be able to operate in this kind of setup effectively.

International Journal of Core Engineering & Management
Volume-6, Issue-12, 2021 ISSN No: 2348-9510

Polyrepo (Multiple Repositories)
In a polyrepo approach, each service lives in a separate repository. This allows teams to own their respective services independently from one another. It also helps keep repositories smaller and easier to manage, which could lead to faster build times. However, dependencies can just as well be split in multiple repos so explicit version management and an explicit communication strategy is needed to keep them in sync and avoid duplication.

Suitability in Microservices Environments
While monorepo and polyrepo both have their advantages, the decision to use one over the other is determined based on factors such as project size, team flow, and organizational culture. For smaller, early-stage startups, monorepos are often the choice to facilitate collaboration and speed. For example, larger enterprises with distributed teams may prefer polyrepos to cater to different workflows and specialized needs.

2.4. Micro Configuration Frameworks
Definition and Purpose
Micro configuration frameworks are designed to streamline and centralize overall configuration data in one place across many services. They offer consistency in settings, limit the complexity of managing environment-specific variables, and keep sensitive information better secured. These frameworks mitigate "configuration drift," the risk that multiple services slowly fall out of alignment when those services are updated inconsistently.

Key Features
A reliable micro configuration framework use a centralized configuration store as the single source of truth for settings of the application. Supports environment-specific configurations for development, testing, and production environments, which empowers teams to adapt quickly to new releases or infrastructure changes. Version control for configuration data up on top also makes auditing and rollback simple in the event of mistakes.

Integration Strategies with Microservices
Integration typically includes having a central configuration server —for example, Spring Cloud Config or HashiCorp Consul — from which services will fetch configurations during startup or at runtime. Or Use a distributed approaches to manage configurations using systems such as: Apache Zookeeper, etcd. In both cases, micro configuration frameworks enormously reduce downtime and improve agility in the emerging world of rapid change.

International Journal of Core Engineering & Management
Volume-6, Issue-12, 2021 ISSN No: 2348-9510

III. IMPLEMENTATION BENEFITS AND CHALLENGES

3.1. Benefits of Microservices and Monorepo Development

Enhanced Collaboration

Monorepo approaches provide a common codebase for different codebases, enabling improved collaboration, making it easier for developers to contribute to multiple services when necessary and reducing the risk of knowledge barriers between teams. In tight, cohesive development teams where ideas cross-pollinate in a manner that will inevitably lead to faster innovations, this collaborative approach can be especially effective.

Better Deployment Pipelines

Teams have more fine-grained control over release cycles because microservices can be deployed independently. This allows the CI/CD pipelines to be standardized, which reduces the operational burden of having several different builds when integrated together with a monorepo. Because of this tighter integration, new features and bug fixes can be released faster. In fiercely competitive markets, the ability to deliver improvements faster can translate into significant competitive advantage.

Scalability and Flexibility

One of the key benefits of microservices is the ability to scale horizontally in a targeted manner. For example, an e-commerce application can scale its payment service independently during high-traffic events like Diwali or Thanksgiving sales, but the resources do not need to be scaled up on services that are not under similar load. This selective scaling saves costs by implementing resources only when and where needed.

3.2.Challenges in Implementing Microservices.

Complexity

Running multiple services is, by nature, a new level of operational complexity. You lose visibility over your microservices, and routing between them can become very complicated and inefficient. Without rigorous planning and some solid tools, this complexity may cancel out most of the supposed advantages of microservices.

Data Consistency

Distributed services make it difficult to manage data. If there are multiple services in a business process and consistency across them must be maintained, then coordination of different components requires specific patterns, such as the Saga. Not doing so can result in various data discrepancies or user-facing errors.

International Journal of Core Engineering & Management
Volume-6, Issue-12, 2021 ISSN No: 2348-9510

Service Coordination
Orchestrating synchronous and asynchronous interactions across services is frequently difficult to do effectively. The retry, timeout, and failure mechanisms should be designed carefully to avoid cascading failures on local failures.

3.3. Mitigation Strategies

Embrace DevOps Culture
A strong DevOps culture supported by practices like infrastructure as code (IaC), CI, and CD can put microservices complexity in check. Automation reduces human error, increasing reliability and reducing lead time.

Use API Gateways
API gateway reduces the complexity of the client with a single entry point for requests. gateways can also centralize tasks such as authentication and rate limiting and caching which offloads these responsibilities from the individual services.

Implement Service Meshes
A service mesh acts at the network layer, abstracting the communication, and providing features like load balancing, encryption, retries, circuit breaking, again at the infrastructure level. This abstraction allows developers to operate on business logic and not concern about network traffic management in a microservices environment.

3.4. Case Studies

Case Study: Flipkart's Transition to Microservices
Flipkart, an e-retail giant in India, moved away from a monolithic architecture to a microservices-based system to accommodate a fast-growing user base. They also adopted a monorepo for shared libraries that improved collaboration across teams.

Lessons Learned
Their experience highlights the importance of migration in stages, gradually breaking down the monolith while minimizing potential disruptions. And also critical was the investment in specialized tooling — often done in-house. A supporting culture that encourages shared ownership, continuous learning, also helped propel their microservices efforts.

Case Study: Uber Microservices in Action
Uber was able to leverage microservices to achieve large transaction volumes with increased reliability They faced challenges in terms of service orchestration and data consistency, mainly because of the real-time aspect of ride-sharing.

Key Takeaways
The need for robust messaging solutions for reliable communication between the services. Comprehensive monitoring and observability was also key to detecting bottlenecks. To address

configuration errors that induced downtime, Uber employed micro configuration frameworks to centralize settings.

IV. DEVELOPMENT PRACTICES AND TOOLING

4.1. Best Practices for Management

Code Organization

It is a good practice to organize code matching the service boundaries, this helps with maintainability and clarity. Working with modular design and consistency in naming conventions and directory structures allows teams to get oriented in large codebases relatively quickly. In monorepo cases, proper segmentation of services ensures the repo does not get too large.

Dependency Management

Tools like npm, Maven, Gradle, etc simplify the dependency resolution. For monorepos — tools like Lerna or Bazel can be used to manage shared components across services. Ensuring that libraries are versioned, and changes are documented well to avoid integration headaches when multiple teams share common building blocks.

Agile and Collaboration

Having agile processes, and using collaboration tools (Jira, Trello, etc.) makes work visible and ensures accountability. Implementing regular stand-ups, code reviews, and pair programming sessions facilitate open communication and help avoid potential integration challenges early on in the development process. Having cross-functional teams comprised of developers, operations, and QA professionals can accelerate feedback loops and help reduces bottlenecks.

4.2. Tooling and Technologies

Development Tools

Most IDEs support microservices development features, with the most current and advanced ones including IntelliJ IDEA, Visual Studio Code, and Eclipse. Plugins like ESLint for JavaScript linting, Checkstyle for Java, and Prettier for code formatting improve code quality as a whole. Additionally, two static code analysis tools SonarQube, which is used to further reduce bugs, vulnerabilities, audit quality, and code smells, along with alerting developers at early stages of the development cycle for potential issues.

Build and Deployment Tools

Containerization technologies such as Docker package each service along with its dependencies to ensure that it executes in the same manner in different environments. Orchestration platforms such as Kubernetes help avoid manual work by automating the deployment, scaling, and management of containerized applications — which is a key need as microservices architectures become increasingly complex. Just as CI/CD solutions (Jenkins, GitLab CI/CD,

CircleCI) help automate code integration and deployments so that smaller releases can happen more frequently with lower risk.

Monitoring and Logging Solutions
Monitoring solutions like Prometheus, along with visualization options such as Grafana, help teams monitor system performance and resource utilization. Centralized logging stacks such as ELK (Elasticsearch, Logstash, Kibana) enable quick debugging as well as historical data analysis. Jaeger and Zipkin are used for distributed tracing that identifies bottlenecks and latency problems that are common in microservices ecosystems.

4.3. Continuous Integration and Continuous Deployment (CI/CD)

Importance
CI/CD pipelines are essential in contemporary software engineering, automating the build, test and deployment processes. CI/CD not only speeds up releases but also lowers the risk of human error by minimizing manual intervention.

Pipeline Design
Automated test suites (unit, integration, and end-to-end tests) should be integrated in the pipeline to flag problems early. One can store their build artifacts securely in repositories such as Nexus or Artifactory. For advanced deployment strategies that minimize downtime, blue-green or canary releases can be utilized, promoting new tracking in parallel and maintaining production continuity.

Monorepo considerations
With a monorepo, build and test processes can be selective, doing work only on services changed by a particular commit. This optimization is important to make pipeline runtimes manageable. As the application scales up, concurrent builds for multiple services must also be supported by the CI/CD system.

4.4. Testing Strategies

Types of Testing
A robust testing suite is essential to ensure individual components work as expected, and the system as a whole operates as intended(Homès, 2012). This includes:

- **Unit Tests** for verifying the functionality of specific modules.
- **Integration Tests** for checking how services interact.
- **Contract Tests** for maintaining service interface consistency.
- **End-to-End Tests** that simulate user workflows to ensure the entire application lifecycle works as intended.

Testing Tools and Frameworks

JUnit by default in Java-based applications, Mocha and Jest for JavaScript and Node. js. Selenium and Cypress are used for end-to-end testing of web applications, while Pact is used for contract testing. By rigorously testing, you can confirm that the changes you made are not breaking anything or causing regressions, which is how you can finally ensure a reliable system.

V. TECHNICAL CONSIDERATIONS IN MICROSERVICES

5.1. Scalability and Performance Optimization

Scaling Strategies

Teams use horizontal scaling, where more instances of the service are added to accommodate increasing traffic, managed by Kubernetes through replication controllers or Horizontal Pod Autoscalers. Vertical scaling, on the other hand, has you add power through more CPU and memory to the same instance. These strategies are a balancing act to optimize performance and cost.

Performance Tuning

Monitoring and profiling with JProfiler or YourKit, you can build it and your performance with you able to see where the code hot paths are and start optimizing. Caching solutions (Redis, Memcached, etc.) can help eliminate duplicate computations or database queries. At the network layer, load balancers (NGINX, HAProxy, etc.) serve to distribute load across service instances and ensure that no single instance ever becomes a bottleneck.

5.2. Security Considerations

Security Challenges

Since microservices communicate via the network, increased attack surface requires more robust security measures. All services need to be secured uniformly so that malicious users won't exploit weaker links in the conceptual chain. (Chandramouli 2019)

Authentication and Authorization

Standards such as OAuth 2.0 and OpenID Connect are commonly used to secure APIs and implement authentication flows. JSON Web Tokens (JWT) provide a compact way to store and transmit user information securely between services, while Role-Based Access Control (RBAC) refines permission models for finer-grained security.

Best Practices

You should encrypt all service-to-service traffic with TLS. Another exception is input validation, which must also be utilized to thwart injection attacks. Updating third-party dependencies and frameworks helps to remove vulnerabilities once found.

A notable security incident can occur when a single microservice is misconfigured, allowing unauthorized access to sensitive data. This further highlights the need to implement standardized security practices across each component in the microservices environment.

5.3. Communication and Data Management

Service DiscoveryMechanisms

Consul, Eureka, or Kubernetes DNS enable services to find each other during runtime, thus avoiding the need for hardcoded IP addresses. This is particularly useful in dynamic environments such as containers where different instances of services are created and destroyed over time.

Communication Patterns

Microservices frequently use synchronous RESTful or gRPC calls for real-time interactions. While traditional systems follow synchronous messaging and generally request and response from similar HTTP services, event-driven architecture using asynchronous messaging systems (such as RabbitMQ or Apache Kafka events) allows huge flexibility in that services publish or subscribe to events in real time as they happen. Neither way is wrong or right, it depends on application requirements and expected traffic so it may vary per case.

Data Management and Database Patterns

A critical decision is whether to use a database-per-service model or a shared database for services. While having a separate database for each service enables greater autonomy, it makes ensuring data consistency much more challenging. On the other hand, shared databases make managing transactions easier, but allow coupling between services. Distributed transaction patterns (like the Saga pattern) provide something in between, as they orchestrate long running transactions that avoid the need for a single source of truth.

5.4. API Gateway Implementation

Role

An API Gateway is a single entry point that forwards the request to the appropriate microservices. It will reduce round-trips by collecting or transforming data from many services, and then the client gets a single response. And gateways typically handle cross-cutting concerns such as rate limiting, caching, and authentication.

Implementation Approaches

Popular solutions such as Kong (NGINX based) and NGINX Plus offer plug-and-play functionality for authentication, logging, and throttling. A few teams build custom gateways with Node js (Express) or Java (Spring Cloud Gateway) and address specialized needs. Regardless of the approach, good design prevents the gateway from becoming a bottleneck or single point of failure.

5.5. Deployment and Orchestration Strategies

Benefits of Containerization

Containerization encapsulates applications with their dependency, so deployments are more consistent. They also have less resource overhead than full virtual machines.

Orchestration Tools

Kubernetes is the de facto orchestrator for containerized workloads, with automated rollouts and rollbacks and self-healing. Another option is Docker Swarm, which is easier to set up, but has fewer features. An immutable infrastructure approach, in which containerized services are replaced—as opposed to installed or updated in place—reduces the risk of configuration drift across any platform.

Deployment Strategies

Teams can use blue-green or canary deployment techniques to validate new releases alongside stable ones. This technique minimizes user damage in case of failure. Systems maintain consistency across staging and production as infrastructure as code (IaC) tools (like Terraform or Ansible) that automate environment provisioning.

5.6. Service Monitoring & Observability

Importance

Observability is crucial in microservices due to the several moving parts. Application metrics and logs can alert us about performance issues or service outages before they happen.

Monitoring Tools

In Prometheus, we have metric collection and alerting, and in Grafana, we have persuasive dashboards to visualize key performance indicators (KPIs). Meaningful alerts can help teams prevent alert fatigue. Centralizing logs with the ELK stack (Elasticsearch, Logstash, Kibana) makes root-cause analysis easier.

Logging and Tracing

Compatible distributed tracing supports such as Jaeger or Zipkin enables tracing of requests across services, allowing teams to pinpoint which services or calls cause bottlenecks. Best practices thus far incorporate correlating logs and metrics for a top-to-bottom view of system's health and custom dashboards serving individual teams' specific needs.

5.7. Circuit Breaking and Fault Tolerance

Ensuring Reliability

Systems should be designed to avoid cascading full-scale outages from small failures. Built-in structural redundancy and isolation at every layer is key to fault tolerance design.

International Journal of Core Engineering & Management
Volume-6, Issue-12, 2021 ISSN No: 2348-9510

Circuit Breaker Implementation
Libraries such as Resilience4j or infrastructure-wide solutions via Istio can provide robust circuit-breaking, bulkhead isolation, retries, and rate-limiting functionality. Netflix's Hystrix was an early adopter in this field, but it is now deprecated, developers are encouraged to migrate to more updated alternatives.

Illustrative example:
A payment service that integrates with external payment gateways. If a gateway becomes unresponsive, any request calls to the affected service are redirected to a fallback mechanism (if defined) to allow the rest of the system to function while the gateway comes back up.

VI. FUTURE DIRECTIONS AND FURTHER RESEARCH

6.1. Future Trends

Emerging Technologies
Serverless architectures such as AWS Lambda or Azure Functions are likely to drive microservices into the next abstraction level, allowing developers to focus on application logic overtaking the infrastructure management. Similarly, service meshes are changing at breakneck speed, with advanced features that accelerate traffic management, amplify security, and simplify service introspection.

Forecasts for the Microservices Landscape
AI as well as machine learning (ML) are gradually getting integrated into microservices supporting smart features like anomaly detection and predictive scaling. At the same time, the concept of DevSecOps is emerging, which emphasizes the idea that security is integrated into the entire development pipeline. Another new frontier is edge computing, which allows for localized processing that can greatly reduce latency and increase real-time responsiveness.

Impact on Tech-Startup Ecosystem
For fast-growing startups across local and global economies, such innovations need to be baked in. Organizations that start embedding AI-powered analytics, service meshes, and strong security features will have a huge advantage when it comes to winning and retaining customers.

6.2. Areas for Further Research

Dynamic Configuration Management in Microservices
Research is needed to improve real-time configuration changes without redeployments. This could involve advanced versioning, conflict resolution mechanisms, or even the emergence of self-healing configurations.

The Role of Monorepo Strategies in Enhancing Team Productivity
While monorepos can accelerate crisis speed, empirical studies quantifying their long-term impact on development speed, code quality, and team morale could open interesting avenues of research. The variation in organizational size and geographical distribution could also affect these results.

Micro Configuration Frameworks Can Have Security Vulnerabilities
As these frameworks centralize configuration authority, this also centralizes the risk. Higher education institutions have important missions and better serve their students and communities when they learn from the data generated on their campuses; future studies could expose potential security vulnerabilities and offer recommendations, from encryption best practices through to strong access control, that help mitigate threats.

Cultural Factors in Microservices Adoption
Organizational culture plays a crucial role in the success of microservices. Investigating how regional differences, leadership styles, and team dynamics influence microservices adoption can guide companies in shaping their development culture more effectively.

VII. CONCLUSION

7.1. Summary of Key Points

This paper examined ways micro configuration frameworks and monorepo strategies can optimize microservices development. We also explored the historical evolution from monolithic architectures to microservices and looked at different architectural patterns while analyzing how repository management approaches (monorepo vs. polyrepo) impact collaboration in teams and code maintainability. We outlined the advantages, such as improved scalability and simplified deployments, and we also tackled complex problems, like data consistency and service coordination. We covered microservices best practices for development, tooling and technical principles that matter for microservices to work successfully, including real-life case studies.

7.2. Final Thoughts

Microservices are a seismic shift in how software development works, offering greater speed and the ability to scale quickly. With rapid-fire tech ecosystems, organizations can use micro configuration frameworks and monorepo strategies to boost innovation for the organization while also managing complexity well. Microservices are an evolving topic and there are many new problems and opportunities to be found which needs constant research and experimentation.
With these ingredients, the path to fully optimized microservices development is one of collaboration, purposeful tooling investment, and a cultural emphasis on learning. By adopting the approaches and practices in this paper, we as individuals and organizations alike can help

mold the future of software architecture into one where innovation, speed, and resilience are the cornerstones of technological advancement.

REFERENCES

1. Fowler, M., & Lewis, J. (2014). Microservices: A definition of this new architectural term. MartinFowler.com. Retrieved from https://martinfowler.com/articles/microservices.html
2. J. Thönes, "Microservices," in IEEE Software, vol. 32, no. 1, pp. 116-116, Jan.-Feb. 2015, doi: 10.1109/MS.2015.11.
3. Ramaswamy Chandramouli&Zack Butcher(May 2020).Building Secure Microservices-based Applications Using Service-Mesh Architecture.https://doi.org/10.6028/NIST.SP.800-204A
4. Bernard Homes. (January 2012). Testing Throughout the Software Life Cycle.https://doi.org/10.1002/9781118602270.ch2
5. Ramaswamy Chandramouli (August 2019).Security Strategies for Microservices-based Application Systems. https://doi.org/10.6028/NIST.SP.800-204
6. Brousse, N. (2019, April). The issue of monorepo and polyrepo in large enterprises. In Companion proceedings of the 3rd international conference on the art, science, and engineering of programming (pp. 1-4).
7. Pichai, S. (2019). Keynote at Google Cloud Next. Google Cloud. Retrieved from https://cloud.withgoogle.com/
8. Singh, A. (2018). Scaling Flipkart's infrastructure for the billion days. Flipkart Engineering Blog. Retrieved from https://tech.flipkart.com/