Security Aspects in Cloud Transitioning

Mr. Vinay Singh[1]
Mr. Amarjeet[2]

ABSTRACT

Microservices architecture has emerged as a preferred development model for enterprise applications due to its ability to address the inherent complexity, scalability demands, and low-latency requirements of such systems. This architecture enables enterprises to build API-driven, loosely coupled, highly scalable, and adaptable platform solutions, particularly in containerized cloud environments. Despite its growing popularity, there is a noticeable lack of comprehensive research in this domain, particularly regarding early defect identification.

Key words: *Version Control System, Git, Subversion, micro-service, Kubernetes, container.*

1. Introduction

Currently, some studies focus on microservices, while others address issues and challenges related to Kubernetes. However, a significant gap exists in exploring the interplay between the two. Addressing this gap motivated our proposed work, which aims to identify and mitigate defects at the initial stages of development for microservices deployed on Kubernetes. To achieve this, we propose a set of foundational guidelines for microservices architecture. These include best practices in naming conventions, automation, monitoring and alerting, fault tolerance, and design principles, all of which contribute to building robust and efficient systems.

In a microservices architecture, each service operates with its own data model and manages its own data independently. Communication between micro-services is facilitated through "mute pipes," which rely on lightweight protocols such as Event Brokers or REST APIs. Each microservice is designed with a narrow focus, encapsulating specific business functionality, and its internal operations remain hidden, accessible only via its exposed APIs [1]-[2].. A common challenge in this ecosystem is the degradation of container image quality, often caused by using custom images without proper registry configuration. This issue becomes particularly pronounced when working with Kubernetes or implementing CI/CD pipelines for the first time [3]. Kubernetes documentation recommends deploying configurations via setup manifests or securely transmitting them as secrets during application initialization to address these challenges effectively each environment; Our development cluster includes a proprietary software system designed as an isolated application, incorporating various technologies such as standalone applications, microservices, and containerized environments like Kubernetes. Currently, our cluster manages approximately ten deployments per minute and thousands of module configurations. However, as the

[1]*Engineer, 6032 Blue Ridge Dr, Apt#A, Highlands Ranch, Colorado, 80130, USA, vsbuild7@gmail.com*
[2]*Engineer, 710 Duncan Ave Apt#1409 Pittsburgh, Pennsylvania, 15237, USA, amar.jeet@gmail.com*

number of containers within the cluster increases, we are encountering issues related to slow scheduling [5]. This problem arises when more than 70 pods are deployed on a single node, leading to delays of several minutes during sequential operations. While slow scheduling is manageable in production environments handling heavy workloads and high CPU utilization, it is essential to maintain a limit of 70 pods per node to avoid bottlenecks. Kubernetes provides two critical monitoring functions to ensure system reliability: liveness probes and readiness probes. Liveness probes periodically perform actions—such as sending HTTP requests, opening TCP connections, or executing commands within the container—to verify the application's health and trigger container restarts if necessary. Readiness probes, on the other hand, ensure that services do not route traffic to a module until it is fully initialized and ready to process requests.

In transitioning from LXC containers to Kubernetes, we observed that certain services performed worse on Kubernetes despite using the same CPU. Even major IT departments noted lower performance without altering the codebase. Continuous monitoring and fine-tuning are crucial to mitigate such regressions.

Furthermore, advanced Nginx servers in the current setup automatically point to Kubernetes endpoints. The ingress controller dynamically handles event-driven updates and resets configurations without service interruptions. However, a deeper examination of the rolling update process is necessary to ensure a seamless and efficient deployment pipeline that the status test comes up short, this module cannot be gotten to as a benefit endpoint, which implies that activity will not be sent to this module until it is prepared. Within the current conveyance, Progressed Nginx servers point to Kubernetes endpoints by default. The logon controller listens to occasions and resets the grouping without interference. However, Let's take a closer look at the process of rolling updates. During this process, the startup sequence for containers associated with Kubernetes pods is initiated and reported to the Kubernetes API. As these processes run concurrently, some services might temporarily go offline. However, traffic can still be routed to the pods transitioning from the current upstream, which ensures minimal disruption [6]-[8].Few works focus on Micro-services while others on Kubernetes issues and challenges but there is no relation between the two has been found. That motivated us to go for the proposed work, which depicts to identify the defects in early stage for Micro-services deployed on Kubernetes. Few standard basic guidelines for the micro-service architecture, in terms of naming convention, automation, monitoring & warning, fault design, and design philosophy, are proposed.

Rest of the paper is organized as follows. Section 2 gives the proposed methodology. Results and discussion are given in section 3.

II Methodology

The term "micro" in the context of microservices has often been a source of confusion and varying interpretations depending on project requirements. Determining the appropriate granularity for a microservice component remains a challenge, necessitating a more precise focus on improving the reusability of components within their functional domains. Software project teams developing micro-services face numerous hurdles, with approximately

70% requiring significant source code refactoring and analysis during the development process [9]-[12]. The complexity of securing Kubernetes for microservice-based applications surpasses that of monolithic applications due to the numerous interdependent components involved.

To address security concerns, developers sometimes leave shell access enabled in container images for troubleshooting in production. However, this approach exposes the system to potential threats, as attackers may exploit this access to inject malicious code. A recommended best practice is to use immutable containers. If defects or vulnerabilities are identified, developers can rebuild and redeploy the containers instead of patching them in place.

Unavailable services can be managed through runtime APIs or by establishing remote shell sessions on the host systems where the microservices are deployed, ensuring operational continuity while minimizing risks. Therefore, there are numerous open-source bundles for designers with promptly accessible holders, counting Node.js., Apache Web Server, and the Linux working framework. In any case, for security purposes, we would like to know where holders start, when they were upgraded, and in the event that they're free of any known vulnerabilities and malicious code. It's best to set up a trusted picture store and run pictures as it were from that trusted source.

III. Results And Discussion

The microservices architectural style has emerged as a prominent and impactful approach for developing cloud-native systems. We advocate for its adoption in distributed systems development to maximize its potential. However, pitfalls and challenges often lead to suboptimal implementations, fostering a perception among teams that microservices are merely another fleeting trend. The added complexity in managing and troubleshooting these systems exacerbates the issues.

When these challenges influence strategic decisions, it is crucial to implement robust technical debt management mechanisms to mitigate their impact. The extent to which organizations can address these challenges depends largely on the maturity of their systems and processes.

To improve the adoption and effectiveness of microservices, adhering to foundational guidelines is essential. Key areas include:

Naming Conventions: Microservices should use clear, descriptive URLs that reflect their purpose. For example, the endpoint GET api/v1/accounts clearly represents a service to list all accounts, ensuring clarity and functionality.

Automation: Automation streamlines processes, reduces manual errors, and enhances system reliability. By emphasizing these principles, teams can mitigate complexities and create robust, scalable microservices architectures.

References:

- Hou Q., Ma Y., Chen J., and Xu Y., "An Empirical Study on Inter-Commit Times in SVN," Int. Conf. on Software Eng. and Knowledge Eng.," pp. 132–137, 2014.
- O. Arafat, and D. Riehle, "The Commit Size Distribution of Open Source Software," Proc. the 42nd Hawaii Int'l Conf. Syst. Sci. (HICSS'09), USA, pp. 1-8, 2009.
- C. Kolassa, D. Riehle, and M. Salim, "A Model of the Commit Size Distribution of Open Source," Proc. the 39th Int'l Conf. Current Trends in Theory and Practice

of Comput. Sci. (SOFSEM'13), Czech Republic, pp. 52–66, 2013.

- L. Hattori and M. Lanza, "On the nature of commits," Proc. the 4th Int'l ERCIM Wksp. Softw. Evol. and Evolvability (EVOL'08), Italy, pp. 63–71, 2008.
- P. Hofmann, and D. Riehle, "Estimating Commit Sizes Efficiently," Proc. the 5th IFIP WG 2.13 Int'l Conf. Open Source Systems (OSS'09), Sweden, pp. 105–115, 2009.
- Kolassa C., Riehle, D., and Salim M., "A Model of the Commit Size Distribution of Open Source," Proceedings of the 39th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'13), Springer-Verlag, Heidelberg, Baden-Württemberg, p. 5266, Jan. 26-31, 2013.
- Arafat O., and Riehle D., "The Commit Size Distribution of Open Source Software," Proceedings of the 42nd Hawaii International Conference on Systems Science (HICSS'09)," IEEE Computer Society Press, New York, NY, pp. 1-8, Jan. 5-8, 2009.

R. Purushothaman, and D.E. Perry, "Toward Understanding the Rhetoric of Small Source Code Changes," IEEE Transactions on Software Engineering, vol. 31, no. 6, pp. 511–526, 2005.

- A. Alali, H. Kagdi, and J. Maletic, "What's a Typical Commit? A Characterization of Open Source Software Repositories," Proc. the 16th IEEE Int'l Conf. Program Comprehension (ICPC'08), Netherlands, pp. 182-191, 2008.
- A. Hindle, D. Germán, and R. Holt, "What do large commits tell us?: a taxonomical study of large commits," Proc. the 5th Int'l Working Conf. Mining Softw. Repos. (MSR'08), Germany, pp. 99-108, 2008.
- Alok Aggarwal, Vinay Singh, Narendra Kumar, "A Rapid Transition from Subversion to Git: Time, Space, Branching, Merging, Offline Commits & Offline builds and Repository Aspects," Recent Advances in Computer Science and Communications, vol. 14: e210621194190, 2021.
- V. Singh, M. Alshehri, A. Aggarwal, O. Alfarraj, P. Sharma et al., "A holistic, proactive and novel approach for pre, during and post migration validation from subversion to git," Computers, Materials & Continua, vol. 66, no.3, pp. 2359–2371, 20