

Zero Touch Rolling Upgrade Cloud-native Network Functions

Thi Lua Bui*, Hoang Phuong Thao Do*, Xuan Chinh Nguyen*, Van Hiep Hoang*,
Thanh Tung Bui*, Thi Phuong Thao Hoang**

*Viettel High Technology, Viettel Group, Hanoi, Vietnam

**Viettel Networks, Viettel Group, Hanoi, Vietnam

(luabt, thaodhp, chihnxx4, hiephv3, tungbt16, thaohtp2)@viettel.com.vn

Abstract—As Cloud-Native Functions (CNFs) become increasingly crucial in modern telecommunications architecture, the demand for efficient, reliable, and uninterrupted upgrade processes has grown more urgent. Traditional upgrade mechanisms often require manual intervention and can lead to service downtime, negatively impacting Quality of Service (QoS) and breaching strict Service Level Agreements (SLA).

To address these challenges, in this paper, we introduce the Zero Touch Upgrade solution for CNF — an automated upgrade solution designed to eliminate human intervention and minimize downtime. Our approach integrates intelligent orchestration systems, continuous monitoring, and rollback mechanisms to ensure seamless transitions between software versions without service disruption. By leveraging Kubernetes-based service mesh and container orchestration, we ensure high availability, scalability, and adherence to the Zero Downtime principle. Experimental results from our Viettel Online Charging System (vOCS) demonstrate significant improvements in upgrade performance, system reliability, and user experience, proving the solution's effectiveness in dynamic and critical telecommunications environments.

Keywords—CNF, Zero Touch Upgrade, Rolling Upgrade, Monitor, KPI, Online Charging System

I. INTRODUCTION

The combination of cloud computing [1] and containerization technology [2] not only enhances the performance and flexibility of telecommunications networks but also enables the rapid and efficient deployment of new services. In that trend, vOCS 4.0 is transforming from virtual machine technology to containerization technology, designed to serve up to 100 million profiles with about 100k concurrent users (CCU). With the characteristics of a telecommunications system, services provided to users must frequently be updated to align with business needs, and ensuring service upgrades without interruption is a top priority.

However, there are many challenges in maintaining application availability during upgrades. Some possible challenges include lack of resources, conflicts in infrastructure resources allocated for the service during upgrades, or compatibility issues with configuration versions, image versions, etc. In the past, we have witnessed numerous upgrade incidents that resulted in severe consequences. In their paper, Iulian Neamtui and Tudor Dumitras [3] provided some specific examples,

highlighting the challenges and opportunities for upgrades in cloud environments. It is evident that for each application, upgrading involves not only updating software versions and configurations but also requires compatibility checks to ensure the application's stability and performance during the upgrade process. Therefore, upgrade methods and actions must be carefully selected and coordinated according to dependencies and potential incompatibilities. Additionally, service upgrade actions may fail, and these upgrade errors must be handled properly to ensure system consistency.

Another major challenge that can arise during upgrades in cloud systems is flexibility. Cloud systems provide an auto-scaling mechanism [4] to adapt to workload changes by automatically adding or removing resources based on workload fluctuations. In practice, the auto-scaling feature can hinder the upgrade process in various ways. During an upgrade, the number of application instances decreases, which increases the workload on the remaining instances. Consequently, the system may misinterpret this as a need to auto-scaling to meet the increased workload, leading to resource contention. Additionally, auto-scaling can undo or obstruct the upgrade process by scaling in new instances with the upgraded version or scaling out instances using the old configuration/image version. Therefore, to avoid these issues, we have disabled the auto-scaling feature during the upgrade process.

There are three main methods to achieve zero-downtime upgrades, including Blue-Green Upgrade, Canary Upgrade, and Rolling Upgrade. Chaitanya K. Rudrabhatla [5] outlined the mechanisms of each strategy and compared these three strategies. From this, it can be seen that each strategy has its advantages and disadvantages, and the appropriate strategy should be applied based on the specific requirements of each system. By choosing the Rolling Upgrade strategy, Daniel Sun and his colleagues [6], focus on upgrading software versions within virtual machines in VNFs, and they also proposed a random model to represent the dynamics of the upgrade process. They then proposed the R2C method [7] with early error detection and predictability to enhance the robustness of Rolling Upgrade. This approach just focuses on services running on virtual machines, without addressing containers.

Additionally, each time an upgrade is performed, the oper-

ator will usually upgrade a group of related applications. If done manually, the process will need to be repeated multiple times, which also introduces risks such as missing upgrades or upgrading the wrong version, making rollback difficult, and causing inconsistency and lack of synchronization among the applications. Therefore, automation is essential due to the large scale of the system, as well as to support operations that do not require human intervention.

Zero-touch Management (ZTM) is a prominent approach to fully automating management and operational processes, especially widely applied in fifth-generation (5G) networks and future networks (Beyond 5G). ZTM is understood as a fully automated network management solution with human oversight, enabling the system to self-optimize and self-heal, addressing the increasing complexity of modern networks. The trend of adopting ZTM-based solutions has also been promoted by the ETSI, which established a new Industry Specification Group (ISG) in 2017 called Zero-touch Network and Service Management (ZSM), aimed at defining the requirements and architecture for a network automation framework based on ZTM concepts [8]. A survey by Coronado et al. [9] identified ZTM - enabling technologies, including programmable control, virtualization, closed-loop control, and AI/ML techniques, which enable automated deployment from the Radio Access Network (RAN) layer to the Core Network (CN), creating an end-to-end management solution. Meanwhile, Mirna et al. [10] provides a comprehensive view of ZSM applications, discussing ZSM challenges and issues related to ML, and highlighting the need to explore diverse network automation solutions. Our zero-touch solution for upgrading CNF is only a small part of the ZTM approach; however, it has already brought certain efficiencies.

In this paper, we present a zero-touch solution for upgrading CNF applications on cloud platforms. In this, we develop a scenario model to automate the rolling upgrade process and monitor the rolling upgrade process through KPI metrics to minimize downtime to the greatest extent during upgrades. If, during the upgrade process, any KPI metrics exceed the allowed thresholds or upgrade failure, the rollback operation is handled automatically to roll back the application to its previous versions before the upgrade. In addition, we also provide recovery handling actions through retry and rollback operations if a scenario fails to execute.

The solution has been implemented in our vOCS system, significantly reducing operational costs, enhancing continuity, and improving both the performance and reliability of the system.

The paper's content is organized as follows: In Section II, we provide the preliminaries and the definitions of concepts used throughout the paper. Next, in Section III, we present our zero-touch upgrade solution. In Section IV, we describe our measurement and results. Finally, Section IV concludes the paper.

II. PRELIMINARIES AND CONCEPT DEFINITIONS

A. vOCS System

The Online charging system (vOCS) subsystem receives input from external service systems via Diameter message protocol [11]. OCS is located in the middle of the service chain and performs billing and plays a decisive role in whether users are allowed to use the service or not.

In our vOCS, we use the network functions virtualization (NFV) architecture according to the standards of the European Telecommunications Standards Institute (ETSI) [12]. The containerized network functions (CNFs) (see Figure 1) are built on NFV infrastructure (NFVI), designed, and deployed to run inside containers supported by Kubernetes ([13], [14]). The containerization of network architecture components allows for running various services on the same cluster and easily deploying already decomposed applications, while dynamically directing network traffic to the correct pods. The management framework for NFVI and the provisioning of new CNFs take place within the management, automation, and network orchestration (MANO) elements defined by NFV. The CNFs, once deployed on MANO, consist of a group of sub-functions, each of which is considered an application. Then MANO will store application information, the number of instances, and other details to support the management, scaling, upgrading, and monitoring of applications.

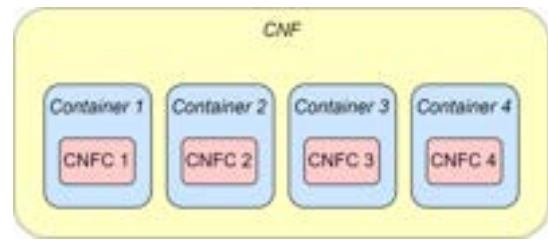


Figure 1. Cloud-native Network Function Basic Architecture

B. Template and Scenario

As mentioned above, each CNF consists of a group of related applications that require strict version compatibility within the CNF. Therefore, when performing upgrades, operators often upgrade multiple applications at once, with additional requirements on the order of the upgrades specifying which applications should be upgraded first and which should be followed to ensure service quality and optimal use of the system resources. To automate the upgrade of applications upon request while making the automated upgrade process easily reusable, we have developed a data model that includes Templates and Scenarios. In this model, a template contains a list of upgrade actions, where each action holds the upgrade information for a single application. This includes basic upgrade details such as the application's name, upgrade monitoring metrics, and the order in which the application should be upgraded. The Scenario inherits from the Template, with each action adding specific details for the current upgrade

time, such as upgrading the image/config from one version to another.

We define an upgrade action as an atomic action that can be executed at a single point in time. An upgrade action is always accompanied by a rollback action — a reverse operation that undoes the effects of the upgrade action applied to the application. For example, if application A has an upgrade action that updates the image from V1 to V2, the rollback action would revert the image from V2 back to V1. Each action also has different states, as described in Figure 2.

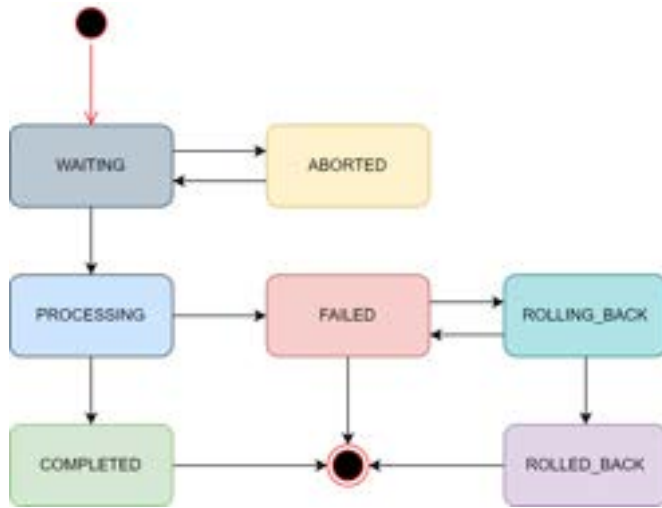


Figure 2. States of an action in scenario

After creating the template, we generate a corresponding scenario from it. At this point, the scenario is ready to be executed. We provide two options: first, the scenario can be executed immediately (on-demand), and second, it can be scheduled for a future time (scheduled). Regardless of which option is chosen, the execution process for the scenario remains the same. A scenario, from the start of execution to the end of its lifecycle, can go through multiple states as described in Figure 3, and depending on the specific state of the scenario, the operator can perform corresponding operations like:

- Impact scenario: Execute upgrade actions sequentially in the specified order.
- Re-impact scenario: The retry scenario starts from the upgrade action that failed in the previous attempt.
- Rollback scenario: Revert scenario, executes rollback actions in the reverse order of the impact sequence.
- Re-rollback scenario: The re-rollback scenario starts from the rollback action that failed in the previous attempt.
- Cancel scenario: Ending the scenario's lifecycle with a failed result.

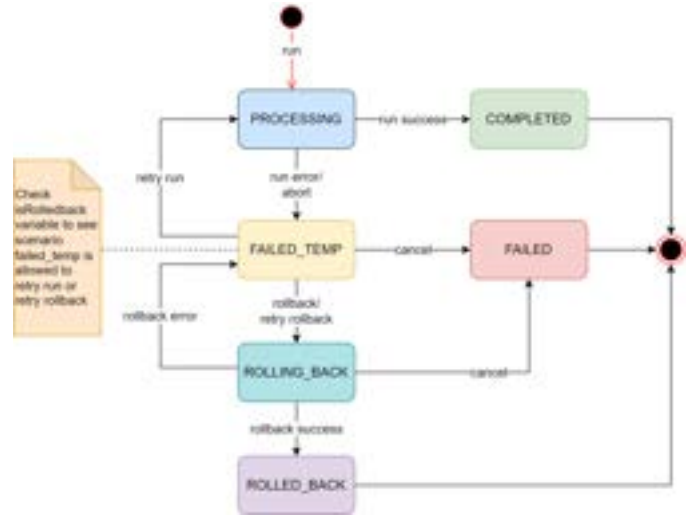


Figure 3. Lifecycle of a scenario

III. ZERO TOUCH ROLLING UPGRADE

A. Monitor Rolling Upgrade Process

The MANO system plays a crucial role in the application upgrade process by sending a trigger to the application when the upgrade is about to take place. This gives the application enough time to prepare necessary steps, such as completing ongoing processing sessions, saving data, and ensuring no processes are interrupted midway. As a result, the application's availability is maintained throughout the upgrade process, minimizing downtime and ensuring that the user experience remains unaffected.

With the rolling upgrade method, new versions of the application are gradually deployed, while the old versions continue to run and handle requests. At any given time, the rolling upgrade process removes a small number of instances running the old version of the software and replaces them with new version instances. This process is repeated in batches, gradually spreading across the entire system until all old instances are replaced. This approach causes less disruption compared to a Blue-Green Upgrade, as the new version is introduced progressively.

We offer two options: if the number of nodes in the cluster exceeds the number of application instances, the rolling out strategy can be used—meaning new instances are activated first, followed by deactivating the old ones. If the number of nodes in the cluster is equal to the number of application instances, the rolling-in strategy can be used—meaning the old instances are deactivated first, and the new ones are activated afterward.

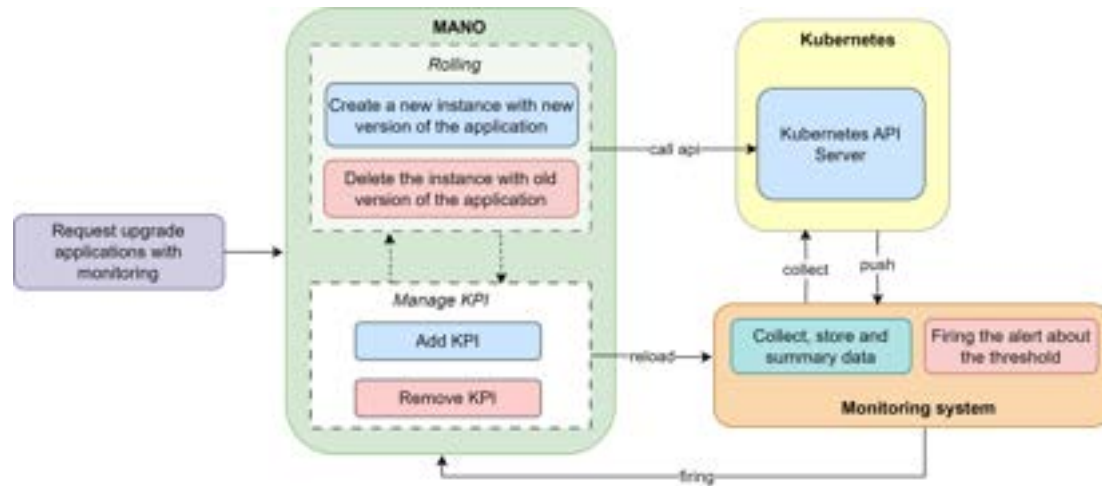


Figure 4. Monitor Rolling Upgrade Process in MANO

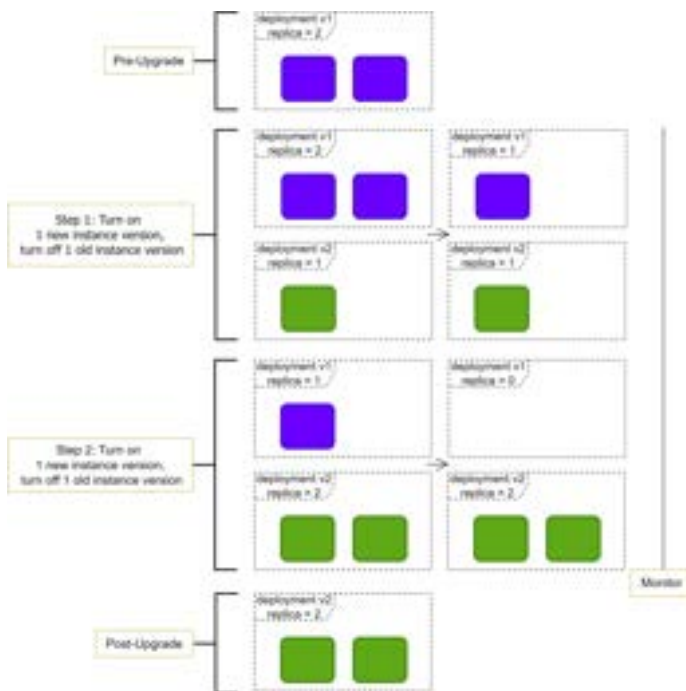


Figure 5. A typical Rolling Upgrade with monitoring diagram

Monitoring during the upgrade process involves overseeing the performance and operational metrics of the application during the upgrade. This monitoring can be applied to the application being upgraded or to other related applications. Each application can be monitored by one or more KPIs, and if any of these metrics exceed their thresholds, it is considered a failure and will be handled by rollback. At the start of the upgrade, the KPI metrics will be added to the monitoring system to track the newly activated application. When the upgrade process is completed, whether successful or unsuccessful, these monitored KPIs will be removed from the monitoring system. During the upgrade process, if the new instances fail the health check or any anomalies are

detected by the monitoring system, the upgrade will be halted, and the system will automatically revert the application to its original state, maintaining the previous software version and configuration before the upgrade. The rollback process will also be carried out using the rolling strategy, which means sequentially turning on the old version instances and then turning off the new version instances. Conversely, if no anomalies are detected and all metrics are within acceptable limits, the software and configurations of the application will be upgraded to the new version. See Figure 4, Figure 5, and Figure 6.



Figure 6. Rolling Upgrade with monitoring activity diagram

B. Scenario Upgrade

- 1) **Impact Scenario Process** (flow is illustrated in Figure 7 and impact algorithm on Algorithm 1): If after the impact the scenario has a FAILED_TEMP status, the operator can either re-impact or rollback the scenario,

depending on the specific situation. For example, if the issue is simple and can be quickly resolved, the operator can re-impact the scenario to continue upgrading the applications that were not upgraded in the previous attempt. On the other hand, if the issue is complex and cannot be immediately fixed, and there is a strict requirement for version synchronization across the applications, the operator may choose to rollback. This will automatically revert the applications that were successfully upgraded in the previous attempt to their pre-upgrade versions. If the application's upgrade process was monitored, then its rollback process will also be monitored with the same KPI metrics as during the upgrade.

Algorithm 1 Impact Scenario algorithm

Input: Scenario: The scenario will be impacted, including a list of actions that will be upgraded.

Output: Scenario after impact.

```

1: scenario.state = "PROCESSING"
2: for index = 1; index <= scenario.getActions().size(); index++ do
3:   action = getActionByIndex(scenario, index)
4:   action.state = "PROCESSING"
5:   upgradeRequest(action)
6:   if isUpgradeSuccessful then
7:     action.state = "COMPLETED"
8:   else
9:     action.state = "FAILED"
10:    for futureIndex = index + 1; futureIndex <= scenario.getActions().size(); futureIndex++ do
11:      futureAction = getActionByIndex(scenario, futureIndex)
12:      futureAction.state = "ABORTED"
13:    end for
14:    scenario.state = "FAILED_TEMP"
15:    return scenario
16:  end if
17: end for
18: scenario.state = "COMPLETED"
19: return scenario

```

- 2) **Re-impact Scenario Process:** If the scenario is in a FAILED_TEMP state and has not undergone a rollback, the operator can retry (re-impact) the scenario. When re-impact the scenario, its state will be updated to the PROCESSING state, and the applications that were not upgraded in the previous attempt will be set to the WAITING (ready-to-upgrade) state. The upgrade process will restart from the application that failed to upgrade previously. The process is carried out similarly to the scenario impactation described above.
- 3) **Rollback Scenario Process** (flow is illustrated in Figure 8 and rollback algorithm on Algorithm 2): If the scenario is in a FAILED_TEMP state and has not been rolled back previously, the operator can rollback the scenario.

The rollback process will be the reverse of the scenario impactation, including both the order of execution and the versions being rolled back.

Algorithm 2 Rollback Scenario algorithm

Input: Scenario: The scenario will be rolled back, including a list of actions that will be rolled back.

Output: Scenario after rolled back.

```

1: for each action ∈ scenario.getActions() do
2:   if action.getState = COMPLETED then
3:     action.state = WAITING
4:   end if
5: end for
6: scenario.state = "ROLLING_BACK"
7: for index = maxIndexWaitingAction(scenario); index > 0; index-- do
8:   action = getActionByIndex(scenario, index)
9:   action.state = "ROLLING_BACK"
10:  rollbackUpgradeRequest(action)
11:  if isRollbackSuccessful then
12:    action.state = "ROLLED_BACK"
13:  else
14:    action.state = "FAILED"
15:    for futureIndex = index - 1; futureIndex > 0; futureIndex-- do
16:      futureAction = getActionByIndex(scenario, futureIndex)
17:      futureAction.state = "ABORTED"
18:    end for
19:    scenario.state = "FAILED_TEMP"
20:    return scenario
21:  end if
22: end for
23: scenario.state = "ROLLED_BACK"
24: return scenario

```

- 4) **Re-rollback Scenario Process:** If the scenario is in FAILED_TEMP state and has been rolled back at least once (even if the rollback was unsuccessful), the operator can re-rollback the scenario.

When re-rolling back the scenario, the scenario's state is updated to ROLLING_BACK, and the applications that were not successfully rolled back in the previous attempt are set to WAITING. The rollback process begins with the application that failed during the previous rollback attempt. The process follows the same steps as the rollback procedure.

- 5) **Cancel Scenario Process:** If the scenario is in the PROCESSING state and the operator chooses to cancel, the scenario will be updated to the FAILED_TEMP state.

If the scenario is in the ROLLING_BACK or FAILED_TEMP state and the operator chooses to cancel, the scenario will be updated to a FAILED state, ending the scenario's lifecycle.

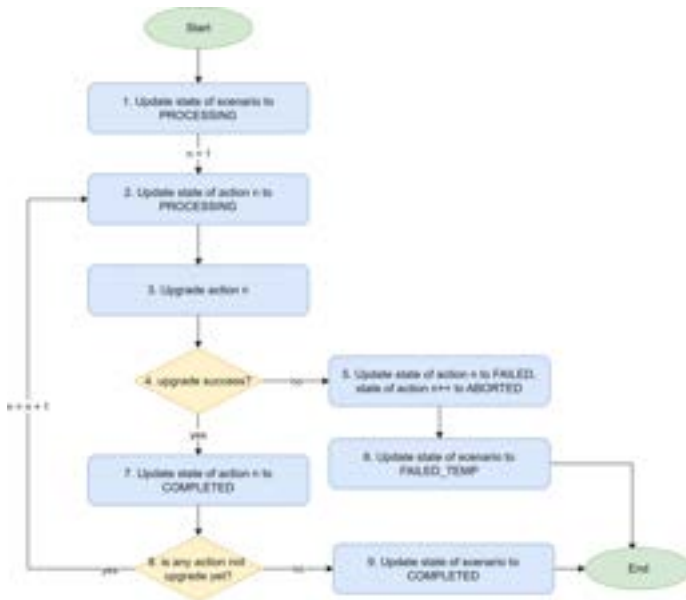


Figure 7. Impact scenario activity diagram



Figure 8. Rollback scenario activity diagram

IV. MEASUREMENTS

A. Methodology

1) *System informations:* Containers were run on seven physical nodes with Intel 6348, which have 112 vCPU and 256GB RAM. Each CNF has approximately ten applications, each application contains three CNFCs, and each CNFC (or pod) uses four vCPUs and 6GB RAM. The Kubernetes version was 1.21. We perform measurements on the data, voice as well

as messaging services. The metrics were considered include: throughput (in TPS - transaction per seconds), latency (in milliseconds), error rate (in %), meantime to detect (in MTTD - minutes), and meantime to recovery (in MTTR - minutes).

2) *Test cases:* We used two test cases to evaluate the effectiveness of the solution:

- Case 1: Successful Upgrade
- Case 2: Rollback

3) *Get results:* For every 1s, we log out the TPS, latency, and error results. To avoid bias in a performance evaluation, we take the average of 100 consecutive measurements.

In the next Section, we present the test result and compare the relative outcomes of each case.

B. Test Results

1) *Case 1: Successful Upgrade:* The measurement results before and after using the Zero Touch solution in the successful case are listed in Table I and II.

TABLE I. SUCCESSFUL UPGRADE BEFORE USING ZERO TOUCH SOLUTION

Service	TPS	Latency	Error rate
Voice	501.07	3.67 ms	1.11%
Data	1421.12	3.52 ms	1.02%
SMS	102.31	2.19 ms	1.07%

TABLE II. SUCCESSFUL UPGRADE AFTER USING ZERO TOUCH SOLUTION

Service	TPS	Latency	Error rate
Voice	501.33	3.51 ms	0%
Data	1421.81	3.05 ms	0%
SMS	102.56	1.93 ms	0%

The results in Table I and Table II show that both latency and error rate during the upgrade process decreased when using zero-touch solution. Specifically, for the Data service, latency decreased from 3.52 ms to just 3.05 ms, and the error rate also dropped from 1.02% to 0%. Similar results can be observed for the other two services. (The TPS difference between data logging instances is negligible, so here we will consider them equal and focus only on the remaining metrics).

2) *Case 2: Rollback:* The measurement results before and after using the Zero Touch solution in the failed then rollback case are listed in table III and table IV.

TABLE III. ROLLBACK BEFORE USING ZERO TOUCH SOLUTION

Service	TPS	Latency	Error rate	MTTD	MTTR
Voice	501.41	5.61 ms	50.45%	30 m	30 m
Data	1420.99	3.54 ms	55.37%	30 m	30 m
SMS	102.68	2.11 ms	48.27%	30 m	30 m

The results in table III and table IV show a significant reduction in both latency and error rate across all services

TABLE IV. ROLLBACK AFTER USING ZERO TOUCH SOLUTION

Service	TPS	Latency	Error rate	MTTD	MTTR
Voice	501.79	3.51 ms	5.11%	2 m	2 m
Data	1421.65	3.44 ms	6.35%	2 m	2 m
SMS	102.60	2.01 ms	3.03%	2 m	2 m

during the upgrade process. Specifically, before using the solution, the latency during the rollback of the Voice service was 5.61 ms, and the error rate was 50.45% – a fairly high figure. However, after applying the solution, latency and error rate decreased to just 3.51 ms and 5.11%, respectively.

Additionally, in the rollback case, we also measured two more metrics: MTTD (Mean Time to Detection) and MTTR (Mean Time to Recovery). Before implementing the solution, detecting errors took a considerable amount of time, ranging from 10 minutes to 1 hour, sometimes even longer (an average of 30 minutes). Once the error was detected and rollback initiated, the process had to be done manually, which added significant time to return the system to its normal state (an average of 30 minutes). However, after applying solution, the time to detect errors and restore the system was greatly improved, averaging only 2 minutes. (The TPS difference between data logging instances is negligible, so here we will consider them equal and focus only on the remaining metrics).

V. CONCLUSION

In this paper, we introduced the Zero Touch solution for upgrading CNF applications in cloud computing environments. By eliminating manual intervention and minimizing downtime, our solution addresses the challenges associated with traditional upgrade methods that can disrupt services.

Through the integration of intelligent orchestration systems, continuous monitoring, and rollback mechanisms, Zero Touch Upgrade enables a seamless transition between software versions while maintaining operational continuity. The implementation of this solution in our vOCS system has demonstrated significant improvements in operational efficiency, system reliability, and user experience, validating the effectiveness of this automated upgrade solution in telecommunication network environments.

In the future, we plan to integrate more process-enhancing technologies to detect anomalies even after a successful upgrade and simplify the creation of scenarios with just a few commands, further optimizing the experience for system operators.

REFERENCES

- [1] E. Thomas, M. Zaigham, and P. Ricardo, "Cloud computing concepts, technology & architecture," 2013.
- [2] J. Watada, A. Roy, R. Kadikar, H. Pham, and B. Xu, "Emerging trends, techniques and open issues of containerization: A review," *IEEE Access*, vol. 7, pp. 152 443–152 472, 2019.
- [3] I. Neamtii and T. Dumitraş, "Cloud software upgrades: Challenges and opportunities," in *2011 International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*, 2011, pp. 1–10.

- [4] H. Alipour, Y. Liu, and A. Hamou-Lhadj, "Analyzing auto-scaling issues in cloud environments," in *CASCON*, vol. 14, 2014, pp. 75–89.
- [5] C. K. Rudrabhatla, "Comparison of zero downtime based deployment techniques in public cloud infrastructure," in *2020 Fourth International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, 2020, pp. 1082–1086.
- [6] D. Sun, D. Guimarans, A. Fekete, V. Gramoli, and L. Zhu, "Multi-objective optimisation of rolling upgrade allowing for failures in clouds," in *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*, 2015, pp. 68–73.
- [7] D. Sun, A. Fekete, V. Gramoli, G. Li, X. Xu, and L. Zhu, "R2c: Robust rolling-upgrade in clouds," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 5, pp. 811–823, 2018.
- [8] ETSI. Zero-touch network and service management (zsm); reference architecture. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/ZSM/001_099/002/01.01.01_60/gs_ZSM002v010101p.pdf
- [9] E. Coronado, R. Behraves, T. Subramanya, A. Fernández-Fernández, M. S. Siddiqui, X. Costa-Pérez, and R. Riggio, "Zero touch management: A survey of network automation solutions for 5g and 6g networks," *IEEE Communications Surveys Tutorials*, vol. 24, no. 4, pp. 2535–2578, 2022.
- [10] M. El Rajab, L. Yang, and A. Shami, "Zero-touch networks: Towards next-generation network automation," *Computer Networks*, vol. 243, p. 110294, 2024. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1389128624001269>
- [11] 3GPP. Ts 32.296: Telecommunication management; charging management; online charging system (ocs): Applications and interfaces. [Online]. Available: <https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=1913>
- [12] Q. Duan, N. Ansari, and M. Toy, "Software-defined network virtualization: An architectural framework for integrating sdn and nfv for service provisioning in future networks," *IEEE Network*, vol. 30, no. 5, pp. 10–16, 2016.
- [13] J. Arundel and J. Domingus, *Cloud Native DevOps with Kubernetes: building, deploying, and scaling modern applications in the Cloud*. O'Reilly Media, 2019.
- [14] B. Ibryam and R. Huß, "Kubernetes patterns," 2019.



Thi Lua Bui received his B.Sc. degree in Software Engineering in 2021 from Posts and Telecommunications Institute of Technology, Vietnam. She is currently a software deployment engineer in Viettel High Technology, Viettel Group. She has many years of experience with cloud computing, cloud native platform. Her research focus on algorithm, operation system, cloud computing.



Hoang Phuong Do Thao is currently a software deployment engineer in Viettel High Technology - Viettel Group, graduating from the Posts and Telecommunications Institute of Technology in 2021. She has many years of experience with cloud computing and cloud native platforms, especially for telecommunications. Her research interests include networking, cloud computing.



Xuan Chinh Nguyen is currently a software deployment engineer at Viettel High Technology - Viettel Group, graduated from the Hanoi University of Industry in 2019. He has many years of experience with cloud computing and cloud native platforms, especially for telecommunications. His research interests include networking, cloud computing, network security, the Future Internet.



Van Hiep Hoang received his B.Sc. degree in Software Engineering in 2020 from Posts and Telecommunications Institute of Technology, Vietnam. He is currently a software deployment engineer at Viettel High Technology - Viettel Group. He is very experienced with many years working with cloud computing in both infrastructure and applications domain. Specially, he focuses on cloud native, high performance systems in Telecommunication Core Network.



Thanh Tung Bui was graduated in Information Technology in 2021 from Posts and Telecommunications Institute of Technology, Vietnam. Currently, he is a software deployment engineer in orchestration platform department at Viettel High Technology, Viettel Group. He has many years of experience with cloud computing and cloud native platforms, especially for telecommunications. His research focus on high performance computing, operating system, networking, cloud computing, network security.



Thi Phuong Thao Hoang is currently a planing engineer in Viettel Networks, Viettel Group. She has many years of experience in research and development of 5G core systems such as online charging system. Her research interests include telecommunications core networks, cloud computing.