A Multi-Layer Continuous-Time Markov Model for Availability Analysis of Kubernetes Systems

Volodymyr Mankivskyi[1], Oleksandr Romanov[1], Mikola Nesterenko[1], Mika Romanova [2] and Anton Marinov[1]
[1]Institute of Telecommunication Systems, National Technical University of Ukraine “Igor Sikorsky Kyiv Polytechnic Institute”, Beresteiskyi Avenue 37, 03056 Kyiv, Ukraine
[2]Faculty of Aeronautics, Electronics and Telecommunications, State University “Kyiv Aviation Institute”, 03058 Kyiv, Ukraine
v.b.mankivskiy@gmail.com, nikolaiy.nesterenko@gmail.com, a_i_romanov@ukr.net, 4639678@stud.kai.edu.ua, marinov.anton99@gmail.com

Keywords: Availability, Markov Chain, CTMC, Kubernetes, Pod, Container, CNI, MTTR, MTTB.

Abstract: This paper proposes a three-tiered analytical availability model for distributed systems running on top of a Kubernetes platform, where container reliability, pod redundancy, and CNI network component reliability are unified under a single Continuous Time Markov Chain model. Unlike typical methodologies based on end-to-end MTTF/MTTR metrics only, this work introduces a model where failure/recovery behaviors can be directly analyzed across multiple tiers and steady-state end-to-end availability between pods can be evaluated directly. Evaluation results show that the effect of the system’s recovery time is almost linear when MTTR ≪ MTBF and the availability benefits of pod redundancy increase exponentially but diminish their returns when the number of replicas is greater than three. A numerical evaluation of the proposed model is conducted using representative parameters of Kubernetes environments. The results demonstrate the influence of recovery time, Pod redundancy, and network reliability on end-to-end service availability. While redundancy significantly improves baseline availability, network recovery time may become the limiting factor for meeting strict SLA requirements.

1 INTRODUCTION

Kubernetes is currently one of the most well-known and convenient tools for deploying and orchestrating systems from various components. Deploying a service platform is accomplished by sequentially configuring each service and requires close coordination of internal and external resources. This approach makes enterprise systems inflexible and hinders the introduction of new services and features.

Kubernetes (k8s) is primarily focused on automating the deployment, scaling, and management of containerized applications. For Kubernetes to solve problems using containerized applications, manage containers, and service incoming requests, these containers must be pre-designed, operational, and publicly available.

Current research into the availability of Kubernetes is focused on analyzing the availability of nodes or pods as independent elements. Our advanced work [1] formalized the availability of boundary lines between Pods, but there is a new model that describes the degradation processes on:

1) Container levels;
2) Level’s Pod;
3) Border level (CNI + veth + bridge).

Kubernetes is a system where the performance of one component can be compensated by others (Pod replication, update policies, overlay boundaries). Therefore, it is natural to model them based on Continuous Time Markov Chain model with updates (CTMC).

The rest of the paper is organized as follows. Section II shows related work and focus paper. Section III introduces the interaction between pods in Kubernetes and architecture components. In Section IV we present our experiments settings, the metrics and various. The experiments, the results, and the analysis with respect to the research questions are presented in Section V. Conclusions and future work placed on section VI and VII accordingly.

2 AVAILABILITIES IN K8S

2.1 Related works

The paper [2] examined the construction of applications and their performance evaluation based on microservices. This paper concluded that the development of Kubernetes is insufficient for ensuring availability, especially high availability. For example, the configuration of Kubernetes by ordering is likely to lead to significant failures in operations during node failure [3]. The primary focus was on their availability. The paper [4] analyzed initial setup in a private cloud and the default Kubernetes configuration. System recovery tasks were investigated for various architectures and configurations. A series of experiments with Kubernetes were conducted, measuring downtime parameters for different failure scenarios. The goal of both studies was to ensure high pod availability.

2.2 Formulation of the Problem

The goal of our work is to answer the following research questions: "How to measure the level of availability of pods in the network to assess the availability of the entire network built by Kubernetes?" The description of the internal process of providing communication between nodes can be assigned to both the Kubernetes controller or network subsystem, and to the control programs located in the control plane. Consider a Kubernetes cluster consisting of a finite set of Pods:

$$P = \{P_1, P_2, \dots, P_M\}. \tag{2}$$

Each Pod Pi contains finite number of containers:

$$P_i = \{C_{i1}, C_{i2}, \dots, C_{iN_i}\}. \tag{3}$$

Containers are deployed within worker nodes and communicate through virtualized network components provided by the Container Network Interface (CNI). Communication between two Pods P_A and P_Bis established through a network path composed of a finite ordered set of network elements:

$$E_{AB} = \{e_1, e_2, \dots, e_K\}. \tag{4}$$

The system operates in continuous time and evolves stochastically due to random failures, recoveries, and performance degradations. Each system layer is modeled as a stochastic process with a finite state.

Each container C_ij is modeled as a continuous-time stochastic process. Let N_i denote the number of containers in Pod P_i.

Thus, the Pod state space is:

$$S_{pod} = \{s_0, s_1, \dots, s_{N_i}\}. \tag{1}$$

This threshold depends on the Kubernetes configuration (restart policy, readiness/liveness probes, application semantics). Each network element e_k in path E_AB has the state space: Operational, Degraded, Failed.

So, a container is a system with three states, between which it transitions randomly. With this approach, the best mathematical apparatus that describes such a system is a continuous Markov process (CTMC). In this work, this mathematical apparatus acts as the central driver.

2.3 Proposed Approach

In this section, we propose considering the three levels of the Kubernetes model. There is a main container in which the application is placed. In parallel with it, another container is placed, which works simultaneously with the main one and provides 100% resource redundancy. In this approach, resource availability is considered. In our work, we proposed changing this paradigm to three levels of availability. The model consists of three levels:

1) Level L1 - Container Level;
2) L2 - Pod Level;
3) L3 - CNI Network Components.

It is important to note that pods in Kubernetes are created in such a way that they have a short lifespan. This means that when a Pod completes its task or becomes inoperable, it can be destroyed and created (recreated) again. Kubernetes uses the concept of replication to ensure that the appropriate number of pods are always running.

Problem Statement with Given:

1) Kubernetes cluster with Pods P;
2) Container parameters $\{\gamma_c\}$;
3) Network parameters $\{\gamma_n\}$;
4) Threshold parameter k_crit.

Determine A_{AB} for arbitrary Pods P_A and P_B in steady-state conditions. The objectives of work are:

1) Construct a three-layer CTMC-based model of Kubernetes availability;
2) Derive analytical expressions for steady-state probabilities;
3) Compute end-to-end communication availability.

3 THREE-LAYER ARCHITECTURE OF KUBERNETES AVAILABILITY

These three levels are quite difficult to depict on a single classic Kubernetes cluster, but they can be distinguished as shown below on Figure 1.

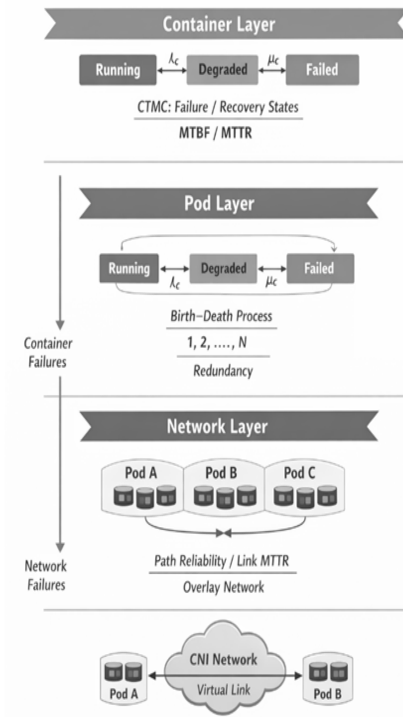

Figure 1: Three-layer arch of k8s availability.

Figure 1 illustrates the proposed three-layer availability model for Kubernetes-based systems. The container layer represents failure and recovery dynamics modeled using Continuous-Time Markov Chains with MTBF/MTTR parameters. The Pod layer captures redundancy effects through a birth-death process describing the number of failed containers. The network layer models end-to-end communication reliability across virtual links and CNI components. The overall service availability is obtained by composing availability metrics across all three layers.

3.1 Level L1 - Container

The container behavior is modeled using a Continuous-Time Markov Chain, as illustrated in Figure 2. Container states describe following states:

$$S_c = \{Run, Degrad, Failed, Recovering\},$$

transient intensities:

1) $\lambda_c = MTBF^{-1}$- failure intensity;
2) $\mu_c = MTTR^{-1}$- recovery intensity;
3) δ_c- intensity of the Running → Degraded transition (network degradations);
4) γ_c- intensity of exit from degradation.

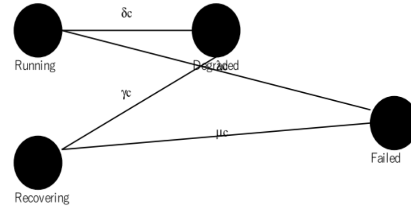

Figure 2: CTMC model container.

3.2 Level L2 - Pod

The basic concept and basic unit in the Kubernetes system is the Pod. A Pod is the basic unit of computing resources in Kubernetes, a container orchestration system. A Pod is a group of one or more containers that together perform a specific task.

A Pod of N containers has states:

$$\text{Spod}=\{s0,s1,\ldots,sN,sF\}, \quad (5)$$

where:

1) s_0- all containers are working;
2) s_k- k containers are unavailable;
3) s_F- pod unavailable (RestartPolicy threshold exceeded).

Probabilities of pod states in steady state. Stationary probabilities:

$$\pi_k = \pi_0 \prod_{i=1}^{k} \frac{(N-i+1)\lambda_c}{i\mu_c}, \quad (6)$$

normalization:

$$\pi_0 = \left(1 + \sum_{k=1}^{N} \prod_{i=1}^{k} \frac{(N-i+1)\lambda_c}{i\mu_c}\right)^{-1}. \quad (7)$$

This is the standard formula for birth-death processes. All containers in a pod are located on a single worker node (the node that performs the work) and share a common network space, allowing them to communicate with each other and with other pods within the cluster. Considering this, moving to a tiered model allows us to generalize the availability:

$$k_{availability} = \frac{MTBF}{MTBF + MTTR}. \quad (8)$$

Normalization to the type of multi-container:

$$A_{pod} = 1 - \sum_{k=k_{crit}}^{N} \pi_k \ , \quad (9)$$

where: k_{crit}- the minimum number of container failures after which the pod is "dead". In this case, all the listed steps and elements of Kubernetes are aimed at creating the minimum unit of deployment - pod.

Figure 3 shows all the listed elements that take part in creating a pod. At the same time, the elements that take part in forming the network stack between pods are not marked.

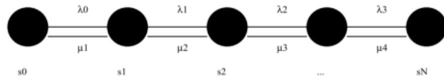

Figure 3: CTMC model of Pod availability with N containers.

As mentioned above, a pod is the fundamental logical unit of Kubernetes. A pod is an abstraction of a virtual machine within a Kubernetes cluster: it has its own private IP address, hostname, and shared disk data. A pod is a deployment unit, and within it, one or more containers run, all related by a common purpose and representing a logical application (consisting of one or more processes/containers).

3.3 Level L3 - CNI

If there is only one container in the pod and it fails, then states are entered for each element: veth, bridge, CNI-overlay:

$$S_{net} = \{Operational, Degraded, Failed\}. \quad (10)$$

In classical reliability theory, transition rates are not loads, but frequencies of events that occur randomly. In our model Figure 4 describes (like container) life circle:

1) λ_n - failure rate;
2) μ_n - repair rate;
3) δ_n- intensity if degraded;
4) γ_n - intensity of recovery.

Components availability:

$$A_{net} = 1 - \pi_{failed}. \quad (11)$$

The failure rate of a component - failure rate (λn) is the mathematical equivalent of "the probability that an element will fail per unit of time". For a network component see on Table 1.

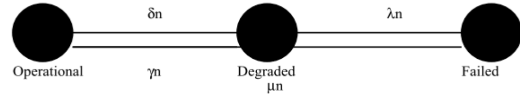

Figure 4: CTMC model of CNI network component.

Table 1: Network elements failure list.

Component	Degradation Cause	Estimate
veth	wrong mtu, kernel bug	dmesg / dropping
bridge	overloaded forwarding table	drop/sec
OVS	kernel datapath crash	ovs-vswitchd
CNI-plugin	deadlock / race condition	kubelet, calico/cilium agent

The intensity of entering the Degraded state (δn) is related to the load.

This is the moment when the component:

1) Does not completely fail;
2) Perform worse;
3) Does not meet the SLO (latency, jitter, packet loss).

These degradation modes and their underlying causes are summarized in Table 2.

Table 2: Network degradation reason list.

Component	Degradation Cause
veth	qdisc drop > threshold
bridge	MAC recycle/drop
OVS	dpif overload
CNI	vxlan latency > 10 ms

The probability that a network component "sags" under high load (δn).

The rate of recovery from degradation (γn), when the load drops, the system returns to normal operation.

A good example is OVS or Calico [5]: during peak traffic, delay increases → degraded, and when traffic drops → operational

Kubernetes can distribute traffic between different containers in a pod using load balancers, which allows you to evenly distribute traffic between available containers in a pod. This ensures high availability of the Kubernetes network, even if some containers in a pod stop or fail.

4 MATHEMATICAL MODEL

Contemporary Kubernetes-based infrastructure operates in dynamic, distributed, and unpredictable environments. For instance, the containers, Pods, and network components experience stochastic failures, degradation, and healing mechanisms. Hence, deterministic modeling approaches cannot be used to properly model the time-related behavior of these infrastructures.

All key elements of Kubernetes exhibit probabilistic behavior:

1) Container failures may occur due to software bugs, resource exhaustion, kernel-level issues, or external dependencies;
2) Pod re-creation and rescheduling depend on control-plane reaction time and system load;
3) Network components (CNI plugins, overlay networks, virtual interfaces) may experience packet loss, latency spikes, or temporary degradation.

Empirical studies of large-scale distributed systems have shown that time to failure, as well as time to recover, often follows an exponential distribution or a hyper-exponential distribution, especially when the environment has independent and memoryless failures [6]. In such a case, the probability of a failure occurring in the next infinitesimal period is independent of the length of the preceding states, which makes the use of Continuous Time Markov Chain feasible.

4.1 Why Does Apply CTMC?

A CTMC is defined by:

1) A finite set of states;
2) Transition rates between states;
3) Memoryless (Markov) property.

Kubernetes components naturally satisfy conditions.

Finite state space:

1) Container: Running, Degraded, Failed, Recovering;
2) Pod: {0,1…N failed containers};
3) Network component: {Operational, Degraded, Failed}.

Random state transitions:

1) Failure events occur randomly over time;
2) Recovery is triggered automatically with measurable average duration.

Thus, CTMC provides a mathematically rigorous and tractable framework for modeling availability in Kubernetes.

4.2 Justification of the Birth-Death Model for Pods

A Pod consisting of Ncontainers can be described as a stochastic process where:

1) Failures increase the number of unavailable containers;
2) Recovery decreases that number.

This exactly corresponds to a birth-death process:

$$\lambda_k = (N - k)\lambda_c, \tag{12}$$

represents the rate of new failures (birth of failed containers), with:

$$\mu_k = k\mu_c, \tag{13}$$

represents the recovery rate (death of failed containers).

The birth-death process is appropriate because:

1) Containers fail independently;
2) Each failed container recovers independently;
3) Transition probabilities depend only on the current number of failed containers.

This model enables closed-form stationary solutions and analytical availability computation.

In addition to binary states (working/failed), real Kubernetes systems experience intermediate degraded states. The parameters:

1) δ- transition rate from operational to degraded;
2) γ- recovery rate from degraded to operational.

Unlike failure rates λ, which describe hard crashes, degradation rates represent threshold-based performance violations such as:

1) Latency exceeding SLO limits;
2) Packet loss exceeding the predefined percentage;
3) CPU throttling or resource contention;
4) Overlay encapsulation delays in CNI networks.

Formally, degradation can be defined as:

$$\delta = f(P_{loss}), \tag{14}$$

where:

1) $\lambda_{traffic}$- incoming traffic rate;
2) C- capacity;
3) D- delay;
4) P_{loss}- packet loss probability.

Thus, degradation rates capture load-induced instability while remaining compatible with Markovian modeling.

4.3 Suitability of CTMC for Availability Composition

4.3.1 Suitability of CTMC for Availability Composition

Compose multi-layer availability models in the proposed three-layer Kubernetes model:
1) L1 - container-level CTMC;
2) L2 - Pod birth-death CTMC;
3) L3 - network component CTMC.

In the proposed model, failures of containers, Pods, and network components are assumed to be statistically independent. This assumption simplifies the analytical formulation and allows the use of Continuous-Time Markov Chains to derive steady-state availability metrics.

The overall service availability is computed as a composition of independent stochastic subsystems, making CTMC a natural and consistent modeling tool. This generalizes our previous Shannon-Moore graph transformation method in previous work [1].

Markov transition matrix:

$$Q_c = \begin{bmatrix} -(\lambda_c + \delta_c) & \delta_c & \lambda_c \\ \gamma_c & -(\gamma_c + \lambda_c) & \lambda_c \\ \mu_c & 0 & -\mu_c \end{bmatrix}, \quad (15)$$

where:
1) λc (lambda) container drop rate;
2) μc (mu) recovery rate (inverse of MTTR);
3) δc (delta) frequency of degradation (network problems, resource starvation);
4) γc (gamma) degradation recovery rate.

The first row of this matrix:

$$Running \rightarrow \left\{\frac{Degraded}{Failed}, \frac{\delta_c}{\lambda_c}\right\}. \quad (16)$$

The sum on the left: - $(\lambda c + \delta c)$ shows that the container leaves the Running state.

Stationary solution:

$$\pi_c = \pi_c Q_c = 0, \sum_i^k \pi_i = 1. \quad (17)$$

After solving the system, we learn:
1) What fraction of the time the container is working normally;
2) How long does it take to degrade?
3) What is the chance that it has crashed?

4.3.2 Pod Model CTMC

It is considered "alive", depending on the parameters of the system itself in k8s:
1) Restart Policy: Pod is considered available if at least one container is alive;
2) Strict Liveness Probe: pod may be unavailable with just one container hanging.

State probabilities:

$$\pi_k = \pi_0 \prod_{i=1}^{k} \frac{\lambda_{i-1}}{\mu_i}. \quad (18)$$

The availability of component *i* is calculated using the steady-state probability that the component is in operational state. In the proposed model, the component behavior is represented by a two-state Continuous-Time Markov Chain consisting of an operational state and a failure state. The transition rate from the operational state to the failed state is denoted by λi, while μi represents the recovery rate. Under these assumptions, the steady state availability of the component is given by:

$$A_i = 1 - \frac{\lambda_i}{\lambda_i + \mu_i}, \quad (19)$$

which corresponds to the stationary probability of the operational state in the Markov reliability model. The intensity data describes the following physical meaning:
1) If there are 5 containers available $\lambda c = 0.001$;
2) Then the total fall rate = $5 \times 0.001 = 0.005$.

More containers → more chances that one will fall.

5 CASE STUDY

The objective of this case study is to compute the steady-state availability of communication between two Pods in a Kubernetes cluster using the proposed three-layer CTMC-based model. Unlike classical network availability analysis, this study incorporates:
1) Container-level failures;
2) Pod-level redundancy;
3) Network component reliability;
4) Performance-related availability constraints.

5.1 Task Definition

Determine the steady-state availability of communication between two Pods (P_A) and (P_B) deployed in a Kubernetes cluster.

Communication is considered available if:
1) Both Pods are operational;
2) All network components along the virtual path are operational;
3) The end-to-end latency does not exceed a predefined threshold.

The availability of a Kubernetes network is defined as the probability that all required components operate within acceptable performance limits over continuous time [3].

The following assumptions are introduced for analytical tractability:
1) All components follow Continuous-Time Markov processes;
2) Failures and recoveries are statistically independent;
3) A Pod becomes unavailable when the number of failed containers exceeds the threshold (k_crit);
4) Network components are modeled as three-state systems (Operational, Degraded, Failed);
5) Latency violation probability is incorporated into degradation state transitions.

A key modeling simplification is that the final virtual branch connecting a container is treated as an indivisible reliability unit.

Thus, container and branch availability are composed multiplicatively. Let:
1) D_{th}be the latency threshold;
2) p be the probability that request latency exceeds $D_{th.}$.

In this model:

$$p = Pr\ (D > D_{th}). \quad (20)$$

This probability is obtained from network capacity calculations or monitoring data from production clusters.

The degradation rate δn of network components reflects this threshold violation probability. Thus, availability accounts not only for hard failures but also for Quality-of-Service violations.

The following input parameters are used in the numerical evaluation:
1) t- continuous operational time;
2) $MTBFpod$mean between Pod failures;
3) $MTTRpod$- mean time to repair Pod;
4) $MTBFnet$Network failures;
5) $MTTRnet$- mean time to repair network component;
6) Rm- reliability of individual network branch;
7) p- probability of exceeding latency threshold;
8) k_{crit}- maximum tolerable number of failed containers.

The resulting availability metric represents the probability that:
1) Both Pods remain operational;
2) All intermediate network elements function correctly;
3) Latency remains below the predefined threshold.

This formulation integrates reliability theory and Quality-of-Service constraints within a unified CTMC-based framework.

5.2 Container Availability with MTTR

First, let us attract your attention to availability of containers. Figure 5 illustrates container availability as a function of recovery time (MTTR) for a fixed mean time between failures $MTBF = 500$ hours.

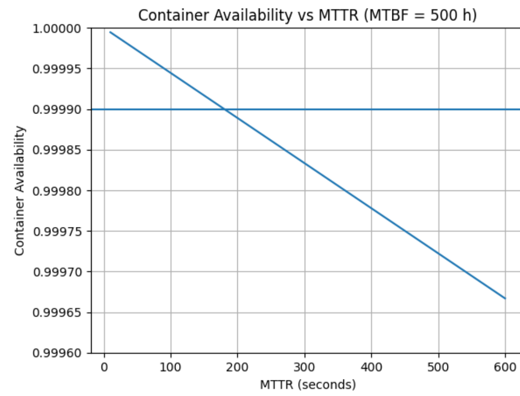

Figure 5: Container availability versus MTTR.

The availability is calculated analytically as:

$$A_c = \frac{MTBF}{MTBF+MTTR}. \quad (21)$$

The results demonstrate a monotonic decrease in availability as MTTR increases from 10 seconds to six hundred seconds. Although the absolute numerical change appears small (from approximately 0.999994 to 0.999667), its operational impact is significant.

Specifically:
1) At $MTTR = 10$seconds, container availability corresponds to approximately 3 minutes of downtime per year;
2) At $MTTR = 60$seconds, downtime increases to approximately 18 minutes per year;
3) At $MTTR = 600$seconds, annual downtime exceeds 175 minutes (nearly 3 hours).

This result highlights a key observation: even when container reliability (MTBF) is relatively high, recovery time has a disproportionate impact on long-term availability.

From an engineering perspective, the graph confirms that rapid self-healing mechanisms (fast restart, efficient liveness probes, minimal image initialization time) are critical for achieving strict service-level objectives such as 99.99% availability.

Furthermore, the nearly linear shape of the curve in the examined region is explained by the condition:

$$MTTR \ll MTBF.$$

Under this condition, availability can be approximated as:

$$A_c \approx 1 - \frac{MTTR}{MTBF}. \quad (22)$$

This explains why availability degrades almost linearly with increasing recovery time within the operational range.

In practical terms, even small increases in MTTR produce measurable changes in availability.

More importantly, the second derivative is positive, indicating convexity of the function:

$$\frac{d^2 A_c}{dx^2} = \frac{2MTBF}{(x)^3}. \quad (23)$$

Therefore, the system becomes progressively more sensitive to MTTR growth as recovery time increases. From an SLA perspective, the most relevant metric is annual downtime:

$$Downtime = (1 - A_c) \cdot 8760\ hours. \quad (24)$$

For a 99.99% SLA, the maximum allowed downtime per year is:

$$0.0001 \times 8760 \approx 52.6\ minutes.$$

This demonstrates that even when failure frequency (MTBF) remains constant, recovery performance directly determines SLA compliance [7], [8].

In practical Kubernetes environments, this emphasizes the importance of:

1) Fast container restart mechanisms;
2) Optimized readiness/liveness probes;
3) Reduced images pull and initialization times;
4) Efficient orchestration response.

Therefore, MTTR optimization often yields greater SLA impact than further improvements in intrinsic container reliability.

5.3 Pod Availability with N

Figure 6 presents Pod availability as a function of the number of containers N for three recovery times: 30, 60, and 120 seconds, assuming $MTBF = 500$hours.

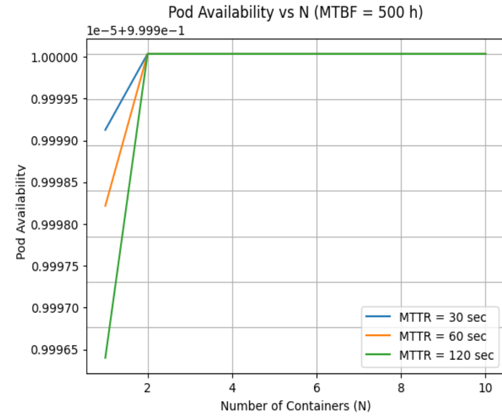

Figure 6: Pod availability as a function of container count.

The graph demonstrates three key effects:

1) Rapid improvement from N = 1 to N = 2;
2) The most significant availability gain occurs when moving from a single-container Pod to a two-container Pod;
3) Diminishing returns for N ≥ 3;
4) Beyond three containers, availability improvement becomes marginal. The curve approaches saturation because the probability of all containers failing simultaneously becomes extremely small;
5) Impact of MTTR.

Higher MTTR shifts the curves downward. Although redundancy mitigates failures, slow recovery reduces the overall benefit of scaling.

The Pod is considered unavailable only when all Ncontainers have failed (i.e., $k_{crit} = N$). The steady-state probability of complete failure is obtained from the birth-death process:

$$\pi_N = \frac{\prod_{k=1}^{N} \frac{(N-k+1)\lambda_c}{k\mu_c}}{\sum_{i=0}^{N} \prod_{k=1}^{i} \frac{(N-k+1)\lambda_c}{k\mu_c}}. \quad (25)$$

Pod availability is defined as:

$$A_{pod} = 1 - \pi_N \quad (26)$$

Let:

$$\rho = \frac{\lambda_c}{\mu_c} = \frac{MTTR}{MTBF} \quad (27)$$

When $\rho \ll 1$(which holds in our scenario), the steady-state probability of complete failure is approximately:

$$\pi_N \approx \rho^N. \quad (28)$$

Thus:

$$A_{pod} \approx 1 - \rho^N. \quad (29)$$

This approximation explains the exponential improvement in availability with increasing N.

For example, if:

$$\rho = \frac{60\ sec}{500\ h} \approx 3.3 \times 10^{-5}.$$

then:

$$\rho^2 \approx 10^{-9}\rho^3 \approx 10^{-14}.$$

Therefore, increasing N from 1 to 2 dramatically reduces failure probability, while further increases produce diminishing returns.

The marginal availability can be approximated:

$$\Delta A_{pod}(N) \approx \rho^{N-1}(1-\rho). \quad (30)$$

This shows that:

1) Marginal benefit decreases exponentially N;
2) Practical saturation occurs for N ≥ 3 under realistic Kubernetes parameters.

For 99.99% SLA (max 52.6 min/year downtime):

1) N = 1 with MTTR = 120 sec may violate SLA;
2) N = 2 satisfies SLA comfortably;
3) N ≥ 3 significantly exceeds SLA.

Adding a second replica is often sufficient to meet strict SLA targets, while further scaling should be justified by load considerations rather than availability alone.

The analysis demonstrates:

1) Horizontal redundancy is highly effective in Kubernetes;
2) Recovery time directly affects redundancy efficiency;
3) Over-scaling Pods for availability alone may be inefficient beyond N = 3;
4) Availability optimization must balance between MTTR reduction and Replica count.

While container availability improves linearly with MTTR reduction, Pod availability improves exponentially with redundancy. However, this exponential gain rapidly saturates.

5.4 End-to-End Availability with Network MTTR

Figure 7 presents the steady-state end-to-end availability between two Pods as a function of network recovery time ($MTTR_{net}$). The number of containers per Pod varies as $N = 1,2,3,5$. Container MTTR is fixed at 60 sec and $MTBF = 500$hours.

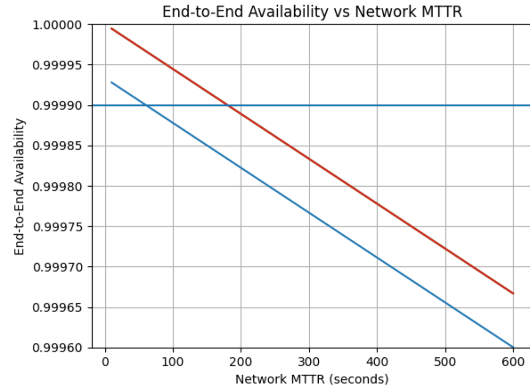

Figure 7: End-to-end availability versus network MTTR.

All curves exhibit identical slopes with respect to $MTTR_{net}$. This occurs because network availability is multiplicatively composed with Pod availability:

Thus, changing Naffects only the vertical scaling factor A_{pod}^2, while the dependence on $MTTR_{net}$ remains structurally identical.

Increasing the number of containers significantly increases baseline availability (vertical shift upward). However, the degradation caused by network MTTR remains proportional across all configurations [7], [9].

Pod-level redundancy cannot eliminate the impact of network recovery delays.

In other words, application-level replication does not compensate for CNI unreliability.

For a 99.99% SLA (52.6 minutes per year downtime), the graph shows:

1) For $N = 1$, SLA is violated at relatively low network MTTR.
2) For $N = 3$, SLA margin improves.
3) For $N = 5$, baseline availability increases further, but network sensitivity remains.

Thus, meeting strict SLA requirements requires not only Pod replication but also reliable network recovery mechanisms [10]. Degradation in end-to-end availability is approximately linear in the operational region. Even when Pod availability is extremely high ($N \geq 3$), network components directly determine whether SLA thresholds are crossed.

This demonstrates that end-to-end availability in Kubernetes is fundamentally constrained by the weakest reliability layer [11].
The findings provide several valuable practical insights:

1) Investing in faster container restart alone is insufficient;
2) CNI plugins and overlay networks must have optimized recovery procedures;
3) design must consider all layers simultaneously.

The end-to-end availability in Kubernetes clusters is limited in their reliability, despite the redundancy in the application layer.

Although the redundancy in Pods results in an exponential increase in availability, the linear multiplicative degradation in network reliability cannot be compensated.

6 CONCLUSIONS

This paper presented a three-layer convenience model for Kubernetes-based systems using CTMC. Unlike traditional network reliability models, the proposed approach integrates container-level failures, Pod-level redundancy, and network-layer reliability within analytical framework.

The main contributions of this work can be summarized as follows:

1) A formal problem formulation of end-to-end communication availability in Kubernetes;
2) A CTMC-based container reliability model incorporating failure and recovery;
3) A birth-death process model describing Pod-level redundancy;
4) A network-layer availability model accounting for virtualized CNI components;
5) An analytical composition framework for computing end-to-end availability;
6) A numerical evaluation demonstrates the sensitivity of availability to recovery time;
7) Several interesting results followed from the study. Firstly, the effect of the average time to repair the containers on the steady-state availability is linear, provided that the average time to repair the containers is significantly less than the average time between failures. Secondly, the effect of the pod-level redundancy on the steady-state availability is exponential, but the returns diminish after three replicas per pod, with the initial replica offering the largest compatibility of the increased availability. Lastly, the effect of the network layer time to repair on the end-to-end availability is multiplicative and cannot be fully compensated for through the application of redundancy mechanisms; this highlights the importance of the reliability of the container network interface (CNI) and network self-healing to achieve service-level agreement targets of 99.99% or higher;
8) The study shows that to optimize Kubernetes cluster availability. It is essential to consider network recovery performance-not just container reliability or scaling.

From an engineering perspective, the results emphasize:

1) Minimization of MTTR through rapid restart and orchestration mechanisms;
2) Careful selection and monitoring of CNI;
3) Balanced design between redundancy and operational efficiency.

7 FUTURE WORK

The proposed three-layer availability framework provides assorted opportunities for further research.

In the present study, degradation states were introduced conceptually but not fully incorporated into the numerical evaluation. Future work should explicitly integrate degraded states into the CTMC structure, allowing the system to operate in intermediate performance regimes rather than strictly binary operational/failed modes:

1) Quantitative modeling of latency violations;
2) Integration of packet loss probabilities;
3) Representation of partial service availability;
4) Analysis of QoS degradation dynamics.

Such a model would more accurately reflect real Kubernetes environments where performance deterioration often precedes complete failure.

The current model assumes statistical independence between containers, Pods, and network components. While this assumption ensures analytical tractability, real-world distributed systems frequently exhibit correlated failures, including:

1) Node-level crashes affecting multiple Pods,
2) Shared resource exhaustion.
3) Network partition events.

A limitation of the current model is the assumption of statistical independence between component failures. In practical Kubernetes deployments, correlated failures may occur due to

shared infrastructure, such as node crashes, network partitioning, or storage subsystem failures. Such events may simultaneously impact multiple containers or Pods and therefore reduce the overall system availability beyond the predictions of the independent failure model.

Future extensions should incorporate correlated failure mechanisms, potentially using:

1) Multi-dimensional CTMC;
2) Copula-based reliability modeling;
3) Common-cause failure analysis.

This would significantly enhance realism and enable evaluation of resilience under large-scale fault scenarios. The current study focuses on steady-state availability. However, transient behavior following failure events is often critical in cloud-native systems.

Future research may address:

1) Time-dependent availability;
2) Recovery trajectory analysis;
3) Short-term SLA violation probability;
4) Burst failure modeling.

This approach also facilitates the modeling of incident scenarios and recoveries beyond the equilibrium assumption. Overall, the above-proposed framework outlines a mathematically sound basis for multi-layer availability modeling within a Kubernetes ecosystem.

Although this research work is focused on the steady-state analytical evaluation under the assumption of independence, the outlined research directions aim at taking this model a step further toward realism, verifiability, and applicability. With the inclusion of degradation modeling, correlated failures, control plane reliability, and optimization, this framework is bound to evolve into a comprehensive reliability engineering practice for cloud-native systems under severe service level agreement pressures.

REFERENCES

[1] O. Romanov, V. Mankivskyi, L. Globa, M. Skulysh and A. Romanov, “Enhancing resource availability: Indicators and strategies for optimizing the Kubernetes network,” in Information and Communication Technologies and Sustainable Development (ICT&SD 2022), A. Dovgyi et al., Eds., Lecture Notes in Networks and Systems, vol. 809. Cham, Switzerland: Springer, 2023, , doi: 10.1007/978-3-031-46880-3_2.

[2] The Kubernetes Authors, “Kubernetes API server - Security,” Kubernetes Documentation, 2020, [Online]. Available: https://kubernetes.io/docs/reference/access-authn-authz/. Accessed: May 16, 2020.

[3] L. A. Vayghan, M. A. Saied, M. Toeroe and F. Khendek, “Kubernetes as an availability manager for microservice applications,” arXiv preprint arXiv:1901.04946, 2019, doi: 10.48550/arXiv.1901.04946.

[4] The Kubernetes Authors, “Kubernetes official website,” Kubernetes Documentation, 2020, [Online]. Available: https://kubernetes.io/. Accessed: May 16, 2020.

[5] O. Romanov, M. Nesterenko, L. Veres, R. Kamarali and I. Saychenko, “Methods for calculating the performance indicators IP multimedia subsystem (IMS),” in Lecture Notes in Networks and Systems, vol. 152. Cham, Switzerland: Springer, 2021, pp. 229-256, doi: 10.1007/978-3-030-58359-0_13.

[6] K. Pentikousis, ONOS: Security and Performance Analysis, Technical Report No. 1, Sept. 19, 2017.

[7] C. O’Connor, T. Vachuska and B. Davie, Software Defined Networks: A Systems Approach. Cambridge, MA, USA: Morgan Kaufmann, 2021.

[8] K. Phemius, M. Bouet and J. Leguay, “Distributed multi domain SDN controllers,” Thales Communications & Security, 2013, pp. 198-209, doi: 10.1109/NOMS.2014.6838330.

[9] J. Lam, S. Lee and O. Yustus, “Securing SDN southbound and data plane communication with IBC,” Mobile Information Systems, vol. 2016, art. ID 2016, 2016, doi: 10.1155/2016/1708970.

[10] K. Phemius, M. Bouet and J. Leguay, “ONOS intent monitor and reroute service: Enabling plug and play routing logic,” Thales Communications & Security, 2013.

[11] A. Poniszewska Marańda and E. Czechowska, “Kubernetes cluster for automating software production environment,” Sensors, vol. 21, no. 5, art. 1910, 2021, doi: 10.3390/s21051910.