Analisis Perbandingan Performa Horizontal Scaling pada Container Orchestration Tool K3s dan Nomad

Razoub Ramadhan[1], Ronggo Alit[2]
[1,2] Teknik Informatika, Universitas Negeri Surabaya
[1]razoub.20072@mhs.unesa.ac.id
[2]ronggoalit@unesa.ac.id

***Abstrak*— Salah satu aspek penting dalam orkestrasi kontainer adalah skalabilitas horizontal, yaitu kemampuan sistem dalam menambah maupun mengurangi kapasitasnya sesuai beban kerja. Beberapa contoh alat orkestrasi kontainer yang dapat melakukan penskalaan horizontal adalah Kubernetes, Docker Swarm, dan Nomad. Terdapat beberapa penelitian terdahulu yang membandingkan performa Kubernetes dan Docker Swarm, namun penelitian terhadap Nomad masih terbilang kurang, khususnya dalam aspek skalabilitas horizontal. Oleh karena itu, penelitian ini ingin menguji kemampuan skalabilitas horizontal dari Nomad dengan membandingkannya kepada salah satu alat orkestrasi kontainer populer, yaitu K3s. Metode penelitian yang digunakan adalah metode eksperimen komparatif dengan membandingkan performa *throughput*, penggunaan CPU dan memori saat melayani *request*, dan kecepatan *scaling up* dan *scaling down* dari kedua alat orkestrasi kontainer.**

Hasil pengujian menunjukkan bahwa K3s memiliki *throughput* yang lebih tinggi daripada Nomad, namun Nomad memiliki penggunaan CPU dan memori yang lebih rendah dari K3s. Nomad juga membutuhkan waktu *scaling up* dan *scaling down* yang lebih singkat dari K3s.

***Kata Kunci*— Alat Orkestrasi Kontainer, Skalabilitas Horizontal, K3s, Nomad**

I. PENDAHULUAN

Perkembangan teknologi *containerization* (kontainerisasi) telah mengubah cara aplikasi atau layanan dioperasikan [1]. Kontainerisasi adalah metode untuk menjalankan dan mendistribusikan aplikasi atau layanan dengan cara mengemas semua dependensinya ke dalam suatu unit yang disebut kontainer [2]. Sebagai bagian dari manajemen kontainer, *container orchestration* atau orkestrasi kontainer dapat mempermudah proses pengelolaan aplikasi dalam skala besar [3]. Untuk memastikan pengelolaan berjalan dengan baik, alat orkestrasi kontainer harus mampu menghadapi berbagai tantangan penting seperti skalabilitas, efisiensi penggunaan sumber daya, dan memaksimalkan *throughput* layanan [3].

Salah satu aspek penting yang disebutkan sebelumnya, yaitu skalabilitas, merupakan kemampuan alat orkestrasi kontainer dalam menambah maupun mengurangi kapasitas sistem guna memenuhi permintaan sesuai beban kerja yang dinamis [4]. Tujuan dari *scaling* adalah memastikan aplikasi atau layanan tetap berjalan dengan baik dan efisien sesuai kebutuhan sistem [4]. Terdapat dua jenis utama *scaling*, yaitu *vertical scaling* dan *horizontal scaling* [5].

Horizontal scaling adalah proses penambahan atau pengurangan jumlah *instance* (*cluster, node, pod*) dalam suatu *cluster* [6]. Dalam arsitektur *cloud-native* dan *microservices*, *horizontal scaling* menjadi strategi yang dominan dalam menjaga ketersediaan tinggi (*high availability*) tanpa adanya *single point of failure* (SPOF) [6]. Kecepatan dan efisiensi *horizontal scaling* inilah yang dapat menentukan kualitas layanan dari sebuah aplikasi [7]. Contoh alat orkestrasi kontainer yang mempunyai fitur horizontal *scaling* secara otomatis adalah Kubernetes.

Kubernetes adalah alat orkestrasi kontainer yang paling populer [8]. Namun, kompleksitasnya membuat Kubernetes membutuhkan sumber daya yang besar [9]. K3s adalah Kubernetes versi ringan yang dirancang untuk penggunaan di lingkungan dengan sumber daya terbatas [10]. K3s bertujuan untuk menyederhanakan pemasangan dan pengoperasian Kubernetes namun dengan tetap menyediakan fitur inti Kubernetes seperti *Horizontal Pod Autoscaler* (HPA) [10].

Selain Kubernetes, terdapat alat orkestrasi kontainer lain seperti Docker Swarm, Apache Mesos dan HashiCorp Nomad. Dengan adanya berbagai pilihan alat orkestrasi kontainer, diperlukan sebuah perbandingan terhadap performa mereka. Pada saat ini sudah ada beberapa penelitian yang membandingkan performa antar alat orkestrasi kontainer, salah satunya adalah penelitian yang melakukan analisis perbandingan skalabilitas terhadap dua alat orkestrasi populer, yaitu Kubernetes dan Docker Swarm [11]. Ada juga penelitian yang membandingkan skalabilitas dan manajemen sumber daya dari Kubernetes, Docker Swarm, dan Apache Mesos [12]. Hal ini membuat diperlukannya penelitian terhadap alat orkestrasi lain, khususnya Nomad.

Nomad adalah alat orkestrasi kontainer yang dikembangkan oleh HashiCorp. Jika dibandingkan dengan Kubernetes dan Docker Swarm, penelitian terhadap Nomad khususnya dalam aspek performa skalabilitas horizontal masih terbilang kurang. Oleh karena itu, penelitian ini dilakukan untuk mengetahui performa skalabilitas Nomad dengan membandingkannya kepada K3s yang dikenal luas dalam industri dan komunitas sebagai salah satu *platform* orkestrasi kontainer yang umum digunakan.

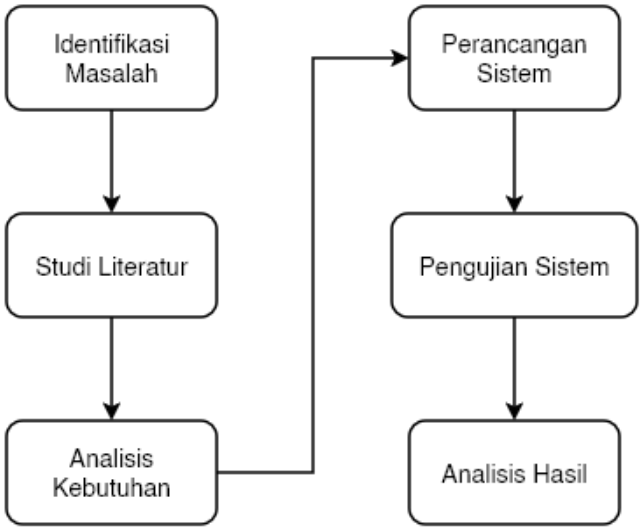
Penelitian ini bertujuan untuk menganalisis kelebihan dan kekurangan K3s dan Nomad dalam aspek kemampuan *scaling*, terutama penskalaan horizontal. Kemampuan *horizontal scaling* menentukan performa sistem dalam menangani *workload* yang dinamis, sehingga aspek ini mempunyai pengaruh yang besar terhadap adaptabilitas sistem. Hasil

penelitian ini diharapkan dapat memberikan pemahaman yang lebih baik mengenai performa masing-masing alat orkestrasi kontainer dalam mendukung kebutuhan skalabilitas.

II. METODE PENELITIAN

Metode yang digunakan dalam penelitian ini adalah metode eksperimen komparatif. Penggunaan metode ini bertujuan untuk mendapatkan analisis yang objektif dan pemahaman yang komprehensif terhadap perbandingan performa K3s dan Nomad dalam konteks skalabilitas horizontal.

Dalam penelitian ini terdapat beberapa tahapan pengerjaan yang dilakukan demi mendapatkan hasil penelitian. Berikut merupakan diagram alur penelitian:



Gbr. 1 Alur Penelitian

A. Analisis Kebutuhan

Tahapan analisis kebutuhan dilakukan guna mendukung proses eksperimen dan pengambilan data pada penelitian ini. Komponen yang dibutuhkan dalam penelitian ini meliputi kebutuhan perangkat keras (*hardware*), kebutuhan perangkat lunak (*software*), dan kebutuhan *virtual machine* (VM).

1) Kebutuhan Perangkat Keras

Perangkat keras yang digunakan pada penelitian ini berupa laptop sebagai sarana pengujian dengan spesifikasi sebagai berikut:

- Processor Intel Core i5-10210U (8 Core)
- RAM 12 GB
- SSD 512 GB

2) Kebutuhan Perangkat Lunak

Perangkat lunak yang digunakan pada penelitian ini berupa sistem operasi dan berbagai aplikasi pendukung penelitian. Berikut merupakan *software* yang digunakan:

- VirtualBox
- Docker
- K3s
- Nomad
- Consul
- Nginx
- Prometheus
- Grafana
- Grafana k6

3) Kebutuhan Virtual Machine

Virtual machine (VM) dibutuhkan sebagai sarana dalam membangun rancangan sistem pengujian dengan memanfaatkan VM sebagai virtual server. Berikut spesifikasi VM yang digunakan:

- Processor 4 Core
- RAM 5 GB
- Penyimpanan 30 GB
- OS Ubuntu Server 24.04.2 LTS

B. Perancangan Sistem

Penelitian ini menggunakan rancangan sistem yang berfokus pada kemampuan *cluster* untuk melakukan *scaling*. Arsitektur yang digunakan adalah 3 *node virtual server* yang terdiri dari satu *master node* dan dua *worker node* yang dijalankan pada VirtualBox.

1) Rancangan Spesifikasi Virtual Server

Untuk membuat lingkungan uji yang seragam, setiap *node* dalam *cluster* akan dijalankan sebagai *virtual server* menggunakan VirtualBox. Spesifikasi *virtual server* telah disesuaikan dengan kebutuhan layanan yang akan dijalankan dan kemampuan *hardware* yang digunakan. Spesifikasi *master node* dan kedua *worker node* ditunjukkan oleh Tabel I. II dan III.

TABEL I
SPESIFIKASI MASTER NODE

Komponen	Spesifikasi
CPU	2 Core
RAM	2 GB
Storage	10 GB
OS	Ubuntu Server 24.04.2 LTS
Software	Docker, K3s, Nomad, Consul, Nginx
Container	Nomad *Autoscaler*

TABEL II
SPESIFIKASI WORKER NODE 01

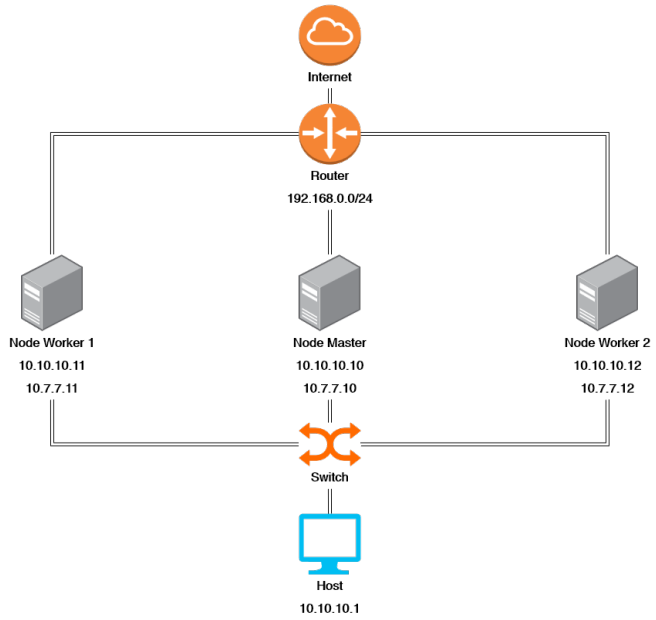
Komponen	Spesifikasi
CPU	1 Core
RAM	1,5 GB
Storage	10 GB
OS	Ubuntu Server 24.04.2 LTS
Software	Docker, K3s, Nomad, Consul
Container	razoub77/currency-converter:1.1

TABEL III
SPESIFIKASI WORKER NODE 02

Komponen	Spesifikasi
CPU	1 Core

RAM	1,5 GB
Storage	10 GB
OS	Ubuntu Server 24.04.2 LTS
Software	Docker, K3s, Nomad, Consul
Container	razoub77/currency-converter:1.1

2) Rancangan Topologi Jaringan



Gbr. 2 Diagram Topologi Jaringan

Untuk membuat sebuah *cluster multi-server* dibutuhkan semua *node* untuk bisa saling berkomunikasi dan dapat terhubung ke internet. Pada Gbr. 2 ditunjukkan topologi jaringan yang digunakan. Masing-masing *node* menggunakan 3 *adapter*, yaitu NAT *Network*, *Host-Only Network*, dan *Internal Network*. Jaringan NAT digunakan sebagai penghubung ke internet, jaringan *Host-Only* digunakan untuk melakukan koneksi SSH dari komputer *host* ke *virtual server*, dan jaringan *Internal* digunakan untuk komunikasi antar *node*. Detail konfigurasi alamat IP statik ditunjukkan oleh Tabel IV.

TABEL IV
KONFIGURASI STATIC IP ADDRESS

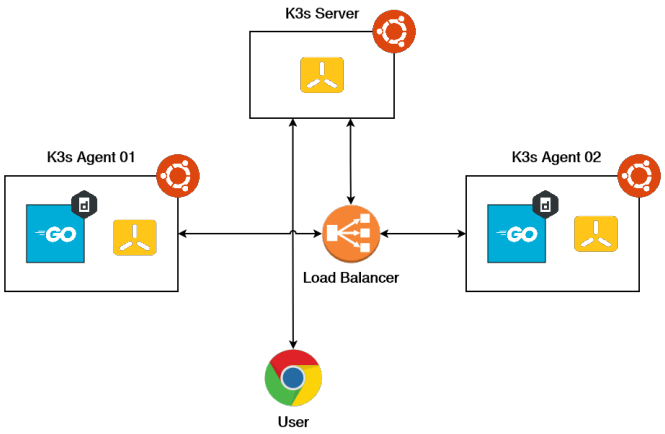
Node	**NAT**	***Host-Only***	***Internal***
Host	-	10.10.10.1	-
Master Node	192.168.0.10	10.10.10.10	10.7.7.10
Worker Node 01	192.168.0.11	10.10.10.11	10.7.7.11
Worker Node 02	192.168.0.12	10.10.10.12	10.7.7.12

3) Alur Kerja Sistem

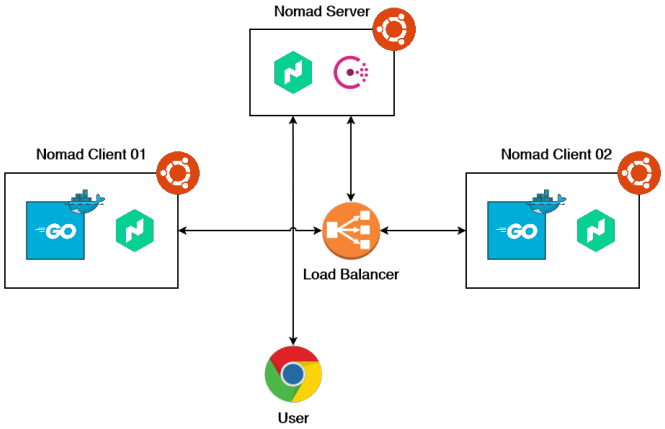
Lingkungan *cluster* K3s dan Nomad yang dibangun akan memiliki pola konfigurasi yang hampir sama. Kedua *cluster* akan menggunakan OS Ubuntu Server 24.04.2 LTS pada setiap *node*-nya. Setelah mengatur koneksi antar *node*, dilakukan pemasangan K3s dan Nomad pada masing-masing *node*. Pemasangan Docker dan Consul dilakukan sebagai *container runtime* dan *service discovery* untuk Nomad.

Pada kedua *cluster* dipasang *load balancer* Nginx sebagai pengatur arus beban kerja agar merata. Selanjutnya dilakukan pemasangan sistem *autoscaling*. K3s mempunyai fitur *scaling* otomatis bawaan yang bernama *Horizontal Pod Autoscaling* (HPA). Untuk Nomad akan dilakukan pemasangan fitur *autoscaling* menggunakan Nomad *Autoscaler*.

Pada penelitian ini aplikasi web yang digunakan berasal dari sebuah Docker *registry* yang berisi *image* sebuah web. Docker *image* tersebut akan dijalankan sebagai kontainer dan di-*deploy* pada kedua w*orker node*, dan dengan menggunakan *load balancer* maka beban kerja bisa didistribusikan kepada masing-masing w*orker* secara merata.



Gbr. 3 Diagram *Cluster* K3s



Gbr. 4 Diagram *Cluster* Nomad

Pada Gbr. 3 dan 4 diperlihatkan diagram *cluster* K3s dan Nomad. *User* mengirim *request* kepada *server* melalui alamat IP *host-only adapter master node*, yaitu 10.10.10.10. *Load balancer* yang berjalan di dalam *master node* mendengar *request* yang masuk kemudian mengarahkannya kepada *worker node* secara merata. Setelah *request* diproses oleh kontainer aplikasi web di dalam *worker node*, konten aplikasi web kemudian dikembalikan kepada *user*.

Untuk melakukan *monitoring* penggunaan CPU dan memori pada setiap *node*, dilakukan pemasangan Prometheus dan Grafana pada komputer *host*. Setelah itu dilakukan pemasangan Grafana k6 sebagai alat pengujian beban.

C. Pengujian Sistem

Penelitian ini bertujuan untuk membandingkan performa *scaling* K3s dan Nomad sebagai alat orkestrasi kontainer.

Akan dilakukan dua jenis pengujian, yaitu pengujian performa dan pengujian skalabilitas.

1) Pengujian Performa

Parameter yang diukur dalam pengujian performa adalah *throughput* atau jumlah *request* yang berhasil dilayani. Metrik ini secara langsung mencerminkan kemampuan *cluster* dalam memaksimalkan kapasitas sistem saat melayani *request*. Selain *throughput*, dilakukan juga pengukuran penggunaan CPU dan memori saat *cluster* sedang melayani *request*. Efisiensi sumber daya ini krusial untuk menentukan kelayakan biaya operasional.

2) Pengujian Skalabilitas

Pengujian skalabilitas dilakukan dengan menghitung waktu yang diperlukan masing-masing alat orkestrasi kontainer untuk melakukan proses *scaling up* dan *scaling down*.

Scaling up dilakukan ketika penggunaan CPU kontainer telah mencapai 80% di mana ini merupakan batas optimal untuk melakukan *scaling up* [13]. Pengukuran dilakukan dengan menjumlahkan waktu untuk membuat kontainer baru dengan waktu untuk melakukan *load balancing*. Waktu *scaling up* yang singkat dapat menghindari penurunan performa saat terjadi lonjakan *traffic*.

Scaling down dilakukan ketika beban sudah menurun dan kontainer sudah tidak digunakan. Pengukuran dilakukan mulai saat *load testing* selesai hingga kontainer dihapus. Waktu *scaling down* yang singkat dapat menghindari penggunaan *resource* yang tidak digunakan.

III. Hasil dan Pembahasan

A. Implementasi Sistem

1) Konfigurasi K3s

Pada konfigurasi *deployment* K3s yang ditunjukkan oleh Gbr. 5, setiap *pod* diatur untuk memiliki 200 mCPU dan 256 MiB memori. Saat awal *deployment*, replika *pod* minimal berjumlah 2.

k3s-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cc-deploy
spec:
  replicas: 2
  selector:
    matchLabels:
      app: cc-web
  template:
    metadata:
      labels:
        app: cc-web
    spec:
      containers:
        - name: cc-web
          image: razoub77/currency-converter:1.1
          ports:
            - containerPort: 8080
          resources:
            limits:
              cpu: '200m'
              memory: '256Mi'
```

Gbr. 5 Konfigurasi Deployment pada Cluster K3s

Untuk membuat *cluster* K3s dapat melakukan *scaling* secara otomatis, dilakukan konfigurasi HPA. Konfigurasi pada Gbr. 6 membuat *cluster* melakukan *horizontal pod scaling* ketika penggunaan CPU dalam suatu *pod* sudah mencapai 80%. Properti "minReplicas" dan "maxReplicas" mengatur banyaknya jumlah minimal dan maksimal *pod* yang dialokasikan, yaitu 2 dan 10.

k3s-hpa.yaml

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: cc-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: cc-deploy
  minReplicas: 2
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 80
```

Gbr. 6 Konfigurasi HPA pada Cluster K3s

2) Konfigurasi Nomad

Konfigurasi *job* Nomad pada Gbr. 7, setiap *task* mendapat alokasi 200 mCPU dan 256 MiB memori. Saat awal *deployment*, *job* ini memiliki 2 *count* atau replika *task*.

nomad-deployment.hcl

```
job "cc-deploy" {
```

```
  datacenters = ["dc1"]
  type = "service"

  group "cc-group" {
    count = 2

    network {
      port "http" {
        to = 8080
      }
    }

    task "cc-web" {
      driver = "docker"

      config {
        image = "razoub77/currency-
converter:1.1"
        ports = ["http"]
        cpu_hard_limit = true
      }

      resources {
        cpu = 200
        memory = 258
      }
    }
  }
}
```

Gbr. 7 Konfigurasi Deployment pada Cluster Nomad

Nomad memiliki fitur *scaling* otomatis bernama Nomad *Autoscaler*. Blok "scaling" yang ditunjukkan pada Gbr. 8 ditambahkan dalam blok "group" (Gbr. 7). Pada blok "scaling" ini, terdapat properti "min" dan "max" yang mengatur jumlah minimal dan maksimal alokasi kontainer, yaitu 2 dan 10. Blok "policy" mengatur ketentuan *scaling* yang akan dilakukan ketika penggunaan CPU oleh kontainer mencapai 80%.

nomad-deployment.hcl

```
    scaling {
      enabled = true
      min = 2
      max = 10

      policy {
        check "avg_cpu" {
          source = "nomad-apm"
          query = "avg_cpu-allocated"

          strategy "target-value" {
            target = 80
          }
        }
      }
    }
```

Gbr. 8 Konfigurasi Nomad Autoscaler

B. *Implementasi Pengujian*

Pengujian dilakukan dengan mengirim sejumlah HTTP *request* menggunakan aplikasi Grafana k6 dengan metode *ramping arrival rate*. Metode ini mengirimkan *request* secara dinamis dan cocok untuk menyimulasikan perubahan beban kerja. Pengujian dilakukan selama 5 menit dan diulang sebanyak 10 kali.

Pada skenario uji yang ditampilkan oleh Gbr. 9, *request* awalnya dikirim sebanyak 10 *request per second* (RPS). Dalam 30 detik awal, jumlah *request* ditingkatkan menjadi 100 RPS, lalu 30 detik selanjutnya ditingkatkan menjadi 200 RPS, lalu satu menit selanjutnya ditingkatkan lagi menjadi 400 RPS. Pada tahap inilah alat orkestrasi kontainer biasanya melakukan *scaling up* karena banyaknya jumlah *request* per detik. Setelah itu, jumlah *request* diturunkan ke 200 RPS selama 2 menit, lalu diturunkan lagi menjadi 100 RPS selama 1 menit.

load-test.js

```
import http from 'k6/http';
export const options = {
  scenarios: {
    test: {
      executor: 'ramping-arrival-rate',
      startRate: 10,
      timeUnit: '1s',
      preAllocatedVUs: 10,
      maxVUs: 100,
      stages: [
        { target: 100, duration: '30s' },
        { target: 200, duration: '30s' },
        { target: 400, duration: '1m' },
        { target: 200, duration: '2m' },
        { target: 100, duration: '1m' },
      ]
    },
  },
```

```
};
export default function () {
  http.get('http://10.10.10.10:8081/');
};
```

Gbr. 9 Konfigurasi Skenario Pengujian

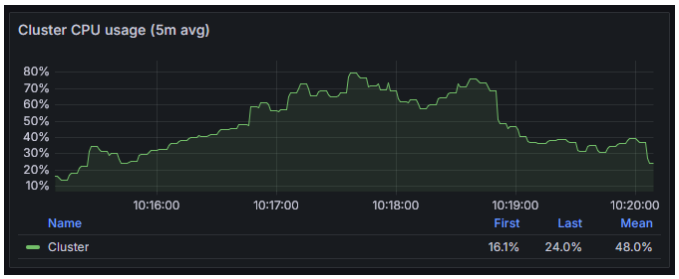
1) Pengujian Throughput

Data *throughput* atau jumlah *request* yang berhasil dilayani oleh *cluster* diambil dari statistik hasil pengujian yang dikeluarkan oleh Grafana k6. Pada Gbr. 10, performa *throughput* dapat dilihat pada properti "http_reqs".

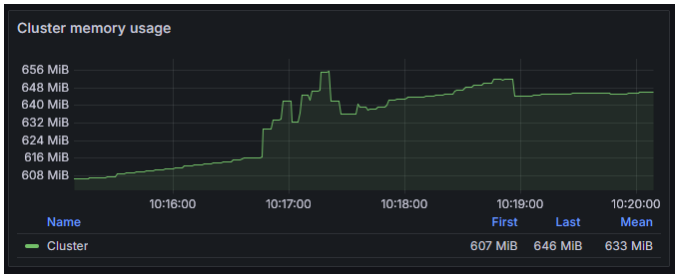
Gbr. 10 Jumlah Throughput pada Grafana k6

2) Pengujian Penggunaan CPU dan Memori

Pengambilan data dilakukan dengan cara menghitung rata-rata (mean) penggunaan CPU dan memori yang dilakukan oleh *cluster* selama pengujian berlangsung.



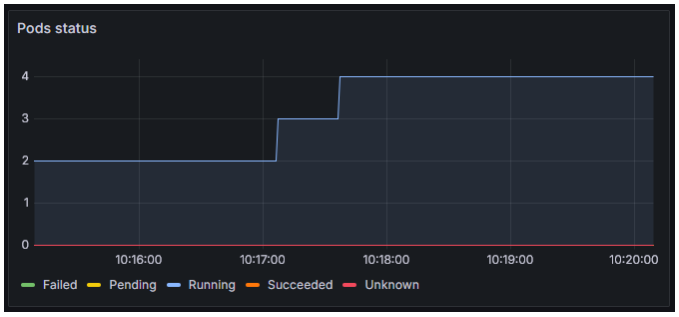
Gbr. 11 Grafik Penggunaan CPU oleh Cluster



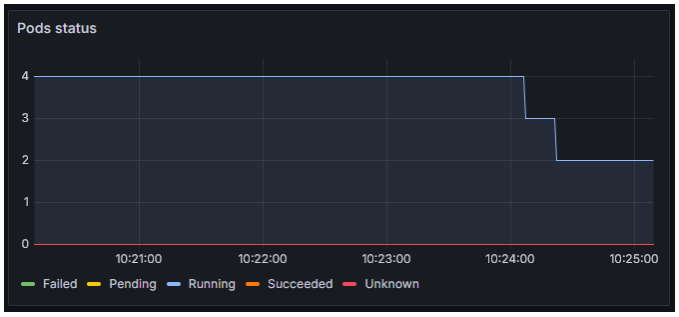
Gbr. 12 Grafik Penggunaan Memori oleh Cluster

3) Pengujian Kecepatan Scaling

Pengambilan data waktu *scaling up* dilakukan saat awal penjadwalan kontainer baru hingga selesainya proses *load balancing*. Sedangkan data waktu *scaling down* diambil saat *load test* dari Grafana k6 sudah selesai, hingga terhapusnya kontainer yang sudah tidak dipakai. Jika terdapat lebih dari satu kontainer tambahan dari hasil *scaling up*, maka data *scaling up* dan *scaling down* dirata-rata.



Gbr. 13 Grafik Jumlah Kontainer Berjalan (Scaling Up)

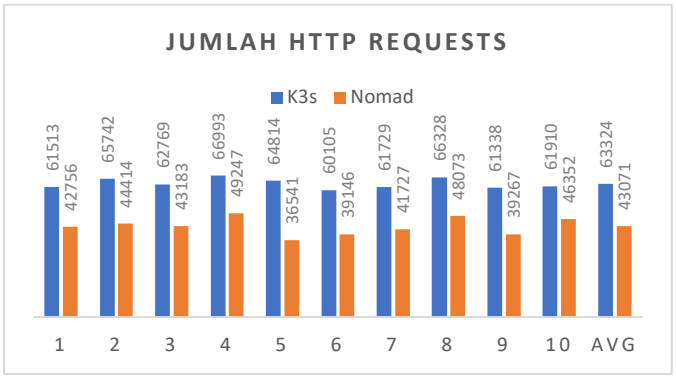


Gbr. 14 Grafik Jumlah Kontainer Berjalan (Scaling Down)

Dengan grafik yang ditunjukkan oleh Gbr. 13 dan 14 dapat diketahui kapan terjadinya proses *scaling up* dan *scaling down*.

C. Hasil Pengujian Performa

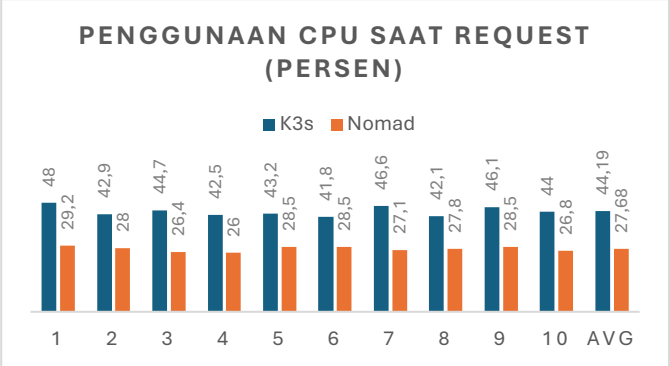
1) Jumlah Throughput



Gbr. 15 Bagan Jumlah Throughput

Pada Gbr. 15 dapat dilihat bahwa dalam hal jumlah *request* yang berhasil ditangani, K3s mendapat hasil yang lebih baik. Penerimaan *request* oleh K3s tertinggi mencapai 66993 dan terendah 60105, dengan rata-rata sebanyak 63324. Sementara penerimaan *request* oleh Nomad tertinggi adalah 49247 dan terendah 36541, dengan rata-rata sebanyak 43071.

2) Penggunaan CPU

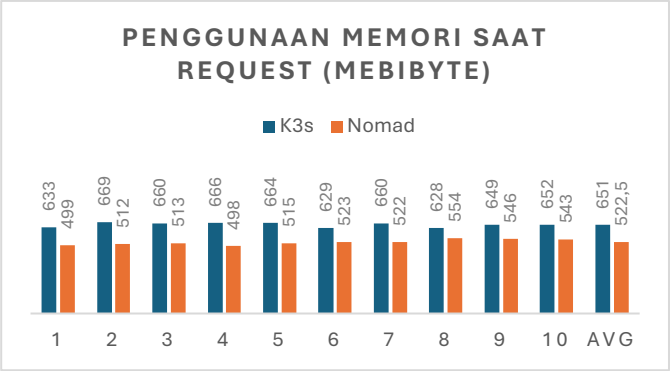


Gbr. 16 Bagan Penggunaan CPU

Pada Gbr. 16 dapat dilihat hasil penggunaan CPU *cluster* saat melayani *request*. Dari 10 pengujian, penggunaan CPU tertinggi K3s mencapai 48% dan terendah 41,8%, dengan rata-rata 44,19%. Pada Nomad penggunaan CPU tertinggi mencapai 29,2% dan terendah 26%, dengan rata-rata 27,68%.

Penggunaan CPU oleh K3s yang lebih tinggi berkaitan dengan kompleksitas komponen yang dijalankan. Proses komunikasi antar komponen pada arsitektur K3s yang lebih rumit seperti API *server, kubelet, kube-proxy,* dan lain-lain menghasilkan peningkatan penggunaan CPU secara menyeluruh pada *cluster*. Sebaliknya, arsitektur yang lebih sederhana pada Nomad membuat penggunaan CPU pada *cluster* secara rata-rata lebih rendah. Walaupun demikian, tingginya konsumsi CPU pada K3s berbanding lurus dengan jumlah *request* yang berhasil dilayani.

3) Penggunaan Memori



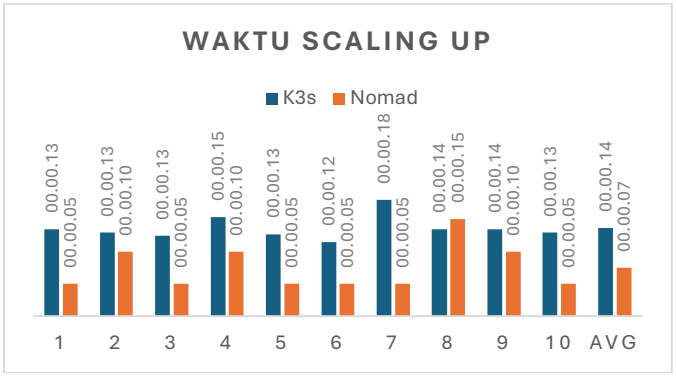
Gbr. 17 Bagan Penggunaan Memori

Pada Gbr. 17 terlihat perbedaan pada penggunaan memori oleh K3s dan Nomad. K3s menggunakan memori tertinggi mencapai 664 MiB dan terendah 628 MiB, dengan rata-rata 651 MiB. Sementara penggunaan memori pada Nomad tertinggi 554 MiB dan terendah 498 MiB, dengan rata-rata 522,5 MiB.

Sama seperti penggunaan CPU, K3s dengan komponen yang lebih lengkap cenderung membutuhkan memori yang lebih banyak baik pada *node master* maupun *worker*. Sebaliknya, Nomad dengan arsitekturnya yang minimalis membutuhkan memori yang lebih sedikit, namun dengan konsekuensi *throughput* layanan yang lebih rendah.

D. Hasil Pengujian Skalabilitas

1) Waktu Scaling Up

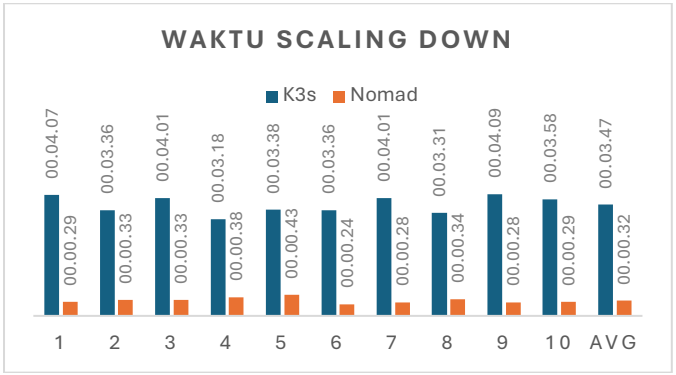


Gbr. 18 Bagan Waktu Scaling Up

Pada Gbr. 18 dapat dilihat perbedaan waktu yang dibutuhkan oleh kedua alat orkestrasi kontainer untuk menambah kontainer baru. K3s melakukan *scaling up* paling cepat 12 detik, paling lambat 18 detik, dengan rata-rata 14 detik. Sementara Nomad paling cepat 5 detik, paling lambat 15 detik, dengan rata-rata 7 detik.

Data hasil pengukuran menunjukkan bahwa Nomad memiliki rata-rata waktu *scaling up* dua kali lipat lebih cepat daripada K3s (7 detik dan 14 detik). Hal ini dapat dipengaruhi oleh faktor arsitektural. K3s memiliki arsitektur yang lebih modular, dalam setiap permintaan *scaling* terjadi proses komunikasi antara API *Server*, *Controller Manager*, dan *Scheduler* yang dapat membuat proses *scaling up* menjadi lebih lama dibandingkan Nomad yang memiliki arsitektur *single binary*.

2) Waktu Scaling Down



Gbr. 19 Bagan Waktu Scaling Down

Pada Gbr. 19 terlihat waktu yang dibutuhkan oleh masing-masing alat orkestrasi kontainer untuk menghapus kontainer yang sudah tidak diperlukan. K3s memiliki rata-rata waktu 3 menit 47 detik, dengan waktu tercepat 3 menit 18 detik, dan waktu terlama 4 menit 9 detik. Sementara Nomad memiliki rata-rata waktu 32 detik, dengan waktu tercepat 24 detik, dan waktu terlama 43 detik.

Terdapat perbedaan waktu yang sangat jauh antara K3s dan Nomad dalam kebutuhan waktu *scaling down*. Alasan

mengapa K3s memiliki waktu *scaling down* yang cukup lama dikarenakan HPA memiliki *downscale stabilization* yang mengatur seberapa lama HPA harus menunggu sebelum melakukan penghapusan kontainer. Secara *default* nilai *downscale stabilization* ini adalah 300 detik (5 menit). Hal ini untuk memastikan arus beban kerja sudah benar-benar berkurang dan bukan fluktuasi sesaat. Sementara pada Nomad terlihat pola *scaling down* yang lebih agresif, dengan penghapusan kontainer yang segera dilakukan ketika sistem mendeteksi arus beban kerja tidak melebihi kapasitas sistem.

IV. KESIMPULAN

Berdasarkan hasil penelitian dan pembahasan, maka dapat diperoleh kesimpulan bahwa dalam aspek performa melayani *request,* K3s unggul dengan jumlah *throughput* sebanyak 63324 dibandingkan dengan Nomad yang sebanyak 43071. Pada aspek penggunaan sumber daya CPU dan memori, *cluster* Nomad menggunakan sumber daya yang lebih rendah daripada K3s. Pada aspek kecepatan *scaling up* maupun *scaling down*, data penelitian menunjukkan Nomad memiliki kecepatan *scaling* yang lebih singkat dari K3s.

Kesimpulan akhir yang bisa didapat dari penelitian ini adalah K3s cocok digunakan untuk kebutuhan sistem yang berfokus pada kemampuan *cluster* dalam melayani *request*, dengan menukarkan kebutuhan penggunaan CPU dan memori yang lebih besar. Di sisi lain, Nomad cocok untuk sistem dengan kapasitas CPU dan memori yang lebih terbatas, karena penggunaan sumber daya CPU dan memorinya lebih rendah. Nomad juga menunjukkan kecepatan *scaling up* dan *scaling down* yang lebih singkat, membuat Nomad dapat diandalkan dalam skenario kebutuhan penambahan dan pengurangan kapasitas sistem secara cepat.

V. SARAN

Penelitian ini tentunya masih mempunyai kekurangan yang dapat diperbaiki dan dieksplorasi lebih lanjut. Beberapa saran untuk penelitian ini adalah tambahkan aspek perbandingan lain selain performa *horizontal scaling*, seperti performa *high availability*, *reliability*, dan *fault tolerance,* sehingga akan didapatkan gambaran yang lebih komprehensif mengenai keunggulan masing-masing alat orkestrasi kontainer.

Selanjutnya, skenario pengujian dapat diperluas dengan menambah jumlah *node cluster* dan konfigurasi jaringan yang lebih bervariasi, untuk melihat apakah hasil performa dan skalabilitas tetap konsisten pada skala yang lebih luas.

Selain itu, tambahkan alat *monitoring* yang lebih detail seperti pemantauan pada tingkat *node* dan *pod/allocation*, agar performa *horizontal scaling* dapat dianalisis secara lebih mendalam.

VI. REFERENSI

[1] Bentaleb, O., Belloum, A. S., Sebaa, A., & El-Maouhab, A. (2022). Containerization technologies: Taxonomies, applications and challenges. *The Journal of Supercomputing*, *78*(1), 1144-1181.

[2] Turnbull, J. (2014). *The Docker Book: Containerization is the new virtualization*. James Turnbull.

[3] Rodriguez, M. A., & Buyya, R. (2019). Container-based cluster orchestration systems: A taxonomy and future directions. *Software: Practice and Experience*, *49*(5), 698-719.

[4] Gupta, A., Christie, R., & Manjula, R. (2017). Scalability in internet of things: features, techniques and research challenges. *Int. J. Comput. Intell. Res*, *13*(7), 1617-1627.

[5] Pandiya, D. K. (2021). Scalability patterns for microservices architecture. *Educational Administration: Theory and Practice*, *27*(3), 1178-1183.

[6] Razavi, K., Salmani, M., Mühlhäuser, M., Koldehofe, B., & Wang, L. (2024). A tale of two scales: reconciling horizontal and vertical scaling for inference serving systems. *arXiv preprint arXiv:2407.14843*.

[7] Goli, A. (2021). Improving the Performance and Availability of Microservice-Based Cloud Applications.

[8] Conway, S. (2017, 28 Juni). *Survey shows Kubernetes leading as orchestration platform*. Diakses pada 19 September 2024. https://www.cncf.io/blog/2017/06/28/survey-shows-kubernetes-leading-orchestration-platform/

[9] Telenyk, S., Sopov, O., Zharikov, E., & Nowakowski, G. (2021, September). A comparison of kubernetes and kubernetes-compatible platforms. In *2021 11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)* (Vol. 1, pp. 313-317). IEEE.

[10] K3s. (n,d.). *K3s Official Documentation*. Diakses pada 18 Juni 2025. https://docs.k3s.io/

[11] Firdaus, B. A., Suryani, V., & Karimah, S. A. (2020). Analisis Performansi Proses Scaling pada Kubernetes dan Docker Swarm Menggunakan Metode Horizontal Scaler. *eProceedings of Engineering*, *7*(2).

[12] Yekollu, R. K., Haldikar, S. V., Ghuge, T. B., Kader, O. F. M. A., & Biradar, S. S. (2024, December). Resource Management and Scalability in Container Orchestration Platforms: A Comparative Study. In *2024 IEEE 16th International Conference on Computational Intelligence and Communication Networks (CICN)* (pp. 1146-1151). IEEE.

[13] Al-Haidari, F., Sqalli, M., & Salah, K. (2013). Impact of CPU Utilization Thresholds and Scaling Size on Autoscaling Cloud Resources. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science* (Vol. 2, pp. 256-261).