BAB 2
DASAR TEORI

2.1 Kajian Pustaka

Penelitian [2] yang berjudul *Performance Evaluation of Container Runtimes* pada tahun 2020 meneliti tentang performansi dari 2 (dua) *runtime* yang umum digunakan yaitu *containerd* dan CRI-O pada 2 (dua) *Open Container Initiative* (OCI) yang berbeda yaitu *runc* dan *gVisor*. Beberapa aspek yang dievaluasi yaitu performa container yang berjalan (CPU, memory, Disk I/O), performa operasi *container runtime*, dan skalabilitas dengan dilakukan pengujian sebanyak 5-20 kali setiap aspek. Pada pengujiannya menggunakan jumlah container yang berbeda yaitu 5, 10, dan 50 container. Penelitian ini menggunakan tools yang disebut *Touch Stone* untuk mendapatkan hasil dari aspek yang dievaluasi. Hasil performa yang diperoleh menunjukkan performa *Containerd* yang lebih baik dalam hal penggunaan CPU time sebesar 23,86 s, latensi memori sebesar 0,11 ms, dan aspek skalabilitas sebesar 37,69 s pada 50 kontainer. Sedangkan operasi sistem file (khususnya operasi *write*) dilakukan lebih efisien oleh CRI-O sebesar 2,43 s.

Penelitian [3] yang berjudul *A Comparison Between Native and Secure Runtimes* pada tahun 2021 meneliti mengenai 2 (dua) *runtime* yang berbeda yaitu Crun dan Kata *Containers*. Pengujian dilakukan menggunakan Podman yang diinstal di Bare Metal dan dijalankan pada OS Ubuntu. Untuk menguji *runtime*, penelitian ini menggunakan *tools Python* yang disebut *pyperformance* dengan 20 kali percobaan untuk setiap *runtime*. Dengan tipe parameter pada *Python* yang diuji yaitu *Apps, Math, Logging, SciMark, Serialize, SQL, SymPy, Regex, Template*, dan *Various*. Hasil yang diperoleh bahwa *overhead* dalam rentang dari <1x hingga 44x membandingkan dua *runtime*. Penelitian ini menunjukkan modul dan pustaka di *Python* yang terpengaruh secara signifikan saat dieksekusi oleh Kata *Containers*, dan lebih baik dieksekusi oleh Crun. Hasil yang diperoleh pada masing-masing parameter yaitu parameter *Apps* Kata *Containers* 7,2 % lebih cepat dari Crun, parameter *Math* Crun selisih 0,48 % lebih rendah dari Kata *Containers,* parameter *Logging Crun* lebih cepat 1,59 % dari Kata *Containers,* parameter *SciMark* Crun lebih rendah 1,41 % dari Kata *Containers,* parameter *Serialize* Crun lebih rendah

2,86 % dari Kata *Containers,* parameter *SQL* Crun lebih rendah 0,96 % dari Kata *Containers,* parameter *SymPy* Crun lebih rendah 1,41 % dari Kata *Containers,* parameter *Regex* Crun lebih rendah 1,29 % dari Kata *Containers,* parameter *Template* Crun lebih rendah 1,57 % dari Kata *Containers*, dan parameter *Various* Crun lebih rendah 4,99 % dari Kata *Containers*. Hasil secara keseluruhan yaitu Crun memiliki rata-rata performansi 16 % lebih baik dari Kata *Containers.*

Penelitian [4] yang berjudul *Containers Runtimes War: A Comparative Study* pada tahun 2020 meneliti mengenai perbandingan kinerja OCI menggunakan beberapa *container engine* yaitu Docker 18, Docker 20, Rkt, WSL2, Podman, Firecracker, Kata Containers, dan gVisor. Penelitian ini menguji performansi parameter *benchmark* yaitu CPU, RAM, *disk* I/O, dan *network* I/O dengan menggunakan *tools* Y-cruncher, Bonnie++, STREAM, dan NetPerf. Pengujian dilakukan menggunakan VM *Azure*. Hasil yang diperoleh menunjukkan kinerja CPU pada Docker dan Podman memiliki efisiensi multi-core sebesar 108,739 % dan 99,043 %, kinerja disk I/O pada gVisor dan Kata Containers lebih unggul dengan rasio 0,124 % dan 0,136 %, kinerja *memory* pada Kata *Containers* memiliki *throughput* yang baik dan waktu yang singkat dengan rasio 0,15 %, serta kinerja *network* I/O pada Podman memiliki performa yang baik dibanding lainnya dengan rata-rata *throughput* TCP dan UDP sebesar 15579,165 MB/s dan *latency* TCP dan UDP sebesar 9,422 ms.

Pada penelitian sebelumnya telah membandingkan beberapa *container runtime* antara lain *containerd* dan CRI-O [2], Crun dan Kata *Containers* [3], serta Docker, Rkt, WSL2, Podman, Firecracker, Kata *Containers*, dan gVisor [4]. Dalam penelitian ini *container runtime* yang digunakan penulis untuk dibandingkan perfomansinya yaitu *Containerd*, CRI-O, dan Kata *Containers* menggunakan orkestrasi Kubernetes. *Container runtime* ini juga merupakan *improvement* dari penelitian sebelumnya karena penelitian ini menguji *container runtime* dari kategori *high-level* dan *low-level*. Pada penelitian ini menggunakan jenis komunikasi *intra-cluster*. Komunikasi *intra-cluster* merupakan komunikasi yang berada di dalam *cluster* Kubernetes antara *pod* (*container*) satu dengan *pod* (*container*) lainnya. Skenario pada penelitian [2] menggunakan jumlah container 5, 10, dan 50 *container*, sedangkan penelitian penulis menggunakan jumlah *container*

10, 20, dan 40 *container* dengan jumlah percobaan 30 kali. Parameter yang dibandingkan pada penelitian [2] dan [4] antara lain CPU, *memory*, *disk* I/O, operasi *container*, skalabilitas, dan *network* I/O, sedangkan penelitian penulis membandingkan parameter CPU dan *memory,* serta menambahkan parameter *throughput*, dan *latency*. Tabel 2.1 merupakan ringkasan dari penelitian sebelumnya.

Tabel 2.1 Kajian Penelitian Sebelumnya

Tahun Terbit	Penulis	Objektif	Container Runtime	Metode	Hasil
2020	Lennart Espe, Anshul Jindal, Vladimir Podolskiy, Michael Gerndt [2]	Membandingkan *Container runtime yaitu containerd* dan CRI-O pada 2 (dua) *Open Container Initiative* (OCI) yang berbeda (*runc* dan *gVisor)*	Containerd dan CRI-O	1) Parameter evaluasi : CPU, memory, disk I/O, operasi container, skalabilitas 2) Jumlah container yaitu 5, 10, dan 50 container 3) Menggunakan tools yaitu Touch Stone dengan 5-20 kali pengujian 4) Pengujian menggunakan virtual mesin dengan OS Debian Buster	Containerd lebih baik dalam penggunaan CPU, latensi memori, dan skalabilitas. Sedangkan CRI-O lebih efieisn pada operasi sistem file (khususnya operasi *write*)
2021	Fredrik Björklund [3]	Membandingkan *Container runtime* yaitu Crun dan Kata *Containers*	Crun dan Kata Containers	1) Pengujian dilakukan menggunakan Bare Metal pada OS Ubuntu 2) Pengujian menggunakan *tools Python* (*pyperformance)*	Kata *Containers* memiliki pengaruh secara signifikan pada modul dan library di *Python*

Tahun Terbit	Penulis	Objektif	Container Runtime	Metode	Hasil
				dengan 20 kali percobaan setiap *runtime* 3) Parameter yang diuji yaitu *Apps, Math, Logging, SciMark, Serialize, SQL, SymPy, Regex, Template*, dan *Various*.	
2020	Ramzi Debab, Walid Khaled Hidouci [4]	Membandingkan *container runtime* yaitu Docker 18, Docker 20, Rkt, WSL2, Podman, Firecracker, Kata *Containers*, dan gVisor	Docker 18, Docker 20, Rkt, WSL2, Podman, Firecracker, Kata *Containers*, dan gVisor	1) Parameter *benchmark* yang diuji yaitu CPU, RAM, *disk* I/O, dan *network* I/O 2) Tools pengujian menggunakan *tools* Y-cruncher, Bonnie++, STREAM, dan NetPerf 3) Pengujian dilakukan menggunakan VM Azure	1) Kinerja CPU pada Docker dan Podman lebih bagus 2) Kinerja disk I/O pada gVisor lebih unggul 3) Kinerja *memory* pada Kata Containers memiliki *throughput* yang baik dan waktu yang singkat 4) Kinerja *network* I/O pada Podman memiliki performa yang baik

Tahun Terbit	Penulis	Objektif	Container Runtime	Metode	Hasil
2023	Pratama Bagus Septianto	Membandingkan container runtime Containerd, CRI-O, dan Kata Containers dalam komunikasi intra-cluster Kubernetes.	Containerd, CRI-O, dan Kata Containers	1) Skenario penelitian menggunakan jumlah container yang berbeda yaitu 10, 20, dan 40 container 2) Pengujian pada komunikasi intra-cluster Kubernetes 3) Parameter yang dianalisis yaitu performansi *throughput, latency* CPU, dan *memory*, 4) *Tools* pengambilan data menggunakan ping, netperf, y-cruncher, dan STREAM	*Container runtime* Containerd memiliki kinerja yang unggul dan optimal berdasarkan hasil dari parameter *throughput*, *latency*, dan *memory* dengan hasil masing-masing parameter yaitu 2301,75 MB/s, 0,359 ms, dan 13001,8 MB/s.

2.2 Dasar Teori

2.2.1 *Container*

Container merupakan teknologi untuk pengemasan serta menjalankan aplikasi Windows dan Linux di berbagai lingkungan lokal maupun lingkungan *cloud*. Container menyediakan lingkungan yang ringan dan terisolasi yang membuat aplikasi lebih mudah dikembangkan, disebarkan, dan dikelola [5]. Container adalah paket kode perangkat lunak yang berisi kode aplikasi, *library* aplikasi, dan dependensi lainnya. *Container*isasi membuat aplikasi *user* menjadi portabel agar kode yang sama dapat berjalan di perangkat mana pun [6]. Pada container, aplikasi dipisahkan dari lingkungan asal aplikasi oleh container, yang mengurangi masalah antar organisasi yang menggunakan berbagai aplikasi atau

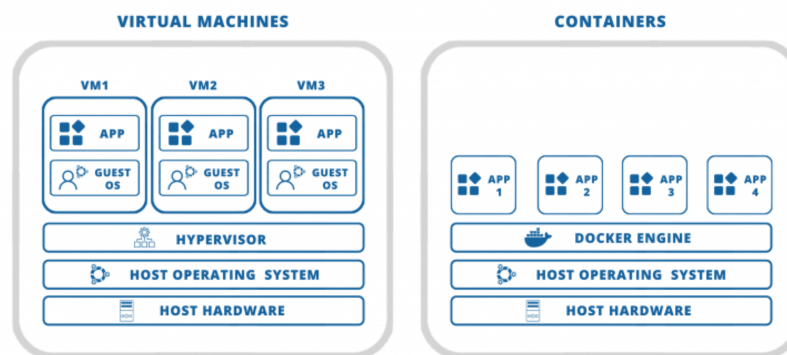
program di jaringan yang sama [7]. Secara umum container terbagi menjadi dua macam berdasarkan level isolasinya, yaitu:

1) *Container* berbasis sistem operasi

   Container yang mengisolasi level sistem operasi. Teknologi ini menawarkan fitur yang mirip dengan virtualisasi namun dengan peningkatan performa yang cukup signifikan. Sifatnya yang memiliki lingkungan terisolasi antara satu dengan yang lainnya serta *high performance* memberikan keunggulan terhadap container.

2) *Container* berbasis aplikasi

   Container yang mengisolasi level aplikasi yang dapat mempermudah para *user* untuk membuat dan memaksimalkan suatu aplikasi yang dikelola. Tidak hanya memanfaatkan *hardware* induk, dapat juga memanfaatkan layanan lain dari container-container yang saling berhubungan dan berjalan pada sistem [8].



Gambar 2.1 Perbedaan arsitektur container dan virtual mesin [9]

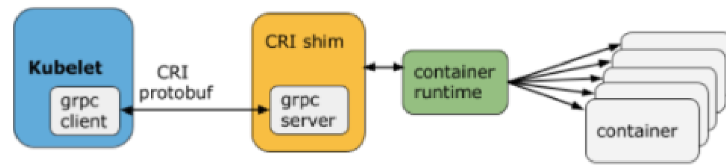
Pada gambar 2.1 menunjukkan perbedaan container dan virtual mesin berdasarkan arsitekturnya. Container diisolasi pada proses level OS, sedangkan virtual mesin diisolasi pada layer hardware. Perkembangan container tidak berarti bahwa mesin virtual akan hilang dalam waktu dekat. Mesin virtual memerlukan hypervisor untuk dijalankan seperti VMWare dan Hyper-V. Hypervisor dipasang di atas mesin fisik, dan kemudian mesin virtual dipasang di atas hypervisor. Hal ini memungkinkan industri TI mengemas banyak mesin virtual pada server fisik untuk meningkatkan kepadatan dan mendapatkan lebih banyak ROI dari hardware fisik. Mesin virtual meniru server fisik termasuk penyimpanan, jaringan, dan sistem operasi.

Container meningkatkan kepadatan dan pengoptimalan ke level berikutnya. *Container* masih merupakan bentuk virtualisasi tetapi hanya memvirtualisasikan inti untuk menjalankan aplikasi. Dengan container, tidak diperlukan hypervisor karena dijalankan langsung di kernel. Pengguna dapat mengemas lebih banyak container di server fisik. *Container* lebih ringan dan melakukan booting lebih cepat, serta pengelolaannya disederhanakan. Dengan container, beberapa komponen dasar dibagikan di semua *container* yang berjalan di host seperti penyimpanan dan jaringan [10].

2.2.2 *Container Runtime*

Container runtime adalah perangkat lunak yang bertanggungjawab menjalankan *container* dan mengelola *image container* pada *deployment node* [2]. Kini ruang lingkup *container runtime* telah berkembang pesat. *Container runtime* mempunyai 2 (dua) kategori yaitu *container runtime low-level* yang mematuhi spesifikasi *runtime Open Container Initiative* (OCI), dan *container runtime high-level* yang sesuai dengan *Container Runtime Interface* (CRI) spesifikasi *runtime*. OCI adalah proyek *Linux Foundation*, terutama berfokus pada penentuan panduan, standar, dan spesifikasi untuk *Linux container*. Spesifikasi OCI *Runtime* sebagian besar berurusan dengan pengelolaan siklus hidup *container* dan konfigurasi untuk berbagai *platform*, seperti *Linux, Windows*, dan *Solaris*. *Container runtime* yang sesuai dengan spesifikasi OCI dianggap sebagai runtime "*low-level*" [11]. *Image Docker* adalah *image* OCI dan *Docker runc* adalah *runtime* OCI [4].

Pada *node* Kubernetes, *container runtime* terdapat di lapisan terbawah yang berfungsi untuk memulai dan menghentikan *container*. Sebagai bagian dari upaya untuk membuat Kubernetes lebih dapat diperluas, Kubernetes telah mengerjakan API *plugin* baru untuk *container runtime* di Kubernetes, yang disebut *Container Runtime Interface* (CRI) [12]. *Container Runtime Interface* (CRI) adalah antarmuka standar untuk plugin Kubernetes yang menjalankan dan mengawasi *container*. CRI dibuat sebagai upaya untuk menstabilkan antarmuka komunikasi antara kubelet dan *container runtime host*. CRI didasarkan pada gRPC, perpustakaan lintas bahasa untuk panggilan prosedur jarak jauh menggunakan *Protocol Buffers* [2].



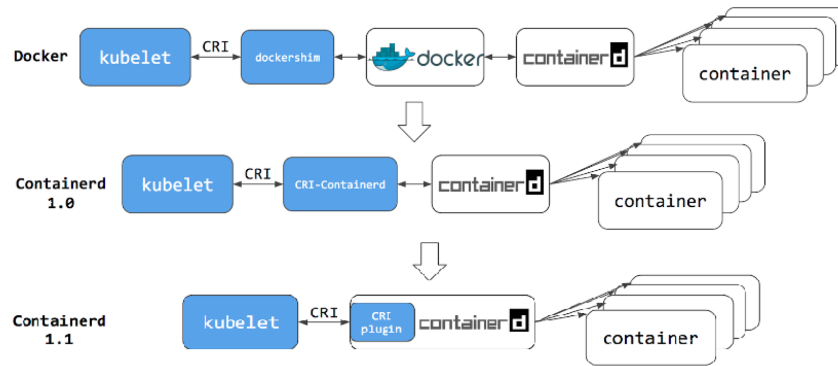
Gambar 2.2 Arsitektur container runtime CRI pada Kubernetes [12]

Pada gambar 2.2 menunjukkan Kubelet berkomunikasi dengan *container runtime* (atau CRI shim untuk *runtime*) melalui soket *Unix* menggunakan kerangka kerja (*framework*) gRPC, di mana kubelet bertindak sebagai klien dan CRI shim sebagai *server*. *Protokol buffer* API termasuk dua layanan gRPC, yaitu *ImageService* dan *RuntimeService*. *ImageService* menyediakan RPC untuk menarik *image* dari repositori, memeriksa, dan menghapus *image*. *RuntimeService* berisi RPC untuk mengelola siklus hidup *pod* dan *container*, serta panggilan untuk berinteraksi dengan *container* [12].

A. *Containerd*

Containerd adalah *runtime default* yang digunakan oleh *Docker Engine*. Ini sering disebut sebagai standar industri karena adopsi yang luas. Di bawahnya, *runtime* ini menggunakan *runc*, implementasi referensi dari spesifikasi *runtime Open Container Initiative* (OCI). *Containerd* menyimpan dan mengelola gambar dan *snapshot*; itu memulai dan menghentikan *container* dengan mendelegasikan tugas eksekusi ke *runtime* OCI. Untuk memulai proses *container*isasi baru, *containerd* harus melakukan tindakan berikut: 1) membuat *Container* baru dari OCI *image* yang diberikan; 2) buat *Task* (tugas) baru dalam konteks *Container*; 3) mulai Tugas (pada titik ini *runc* mengambil alih dan mulai menjalankan bundel OCI yang disediakan oleh *containerd*).

Containerd menyediakan *endpoint* gRPC yang kompatibel dengan CRI yang diaktifkan secara *default*. Saat menggunakan titik akhir ini, abstraksi khusus untuk *containerd* disembunyikan dari klien sehingga pengguna dapat beroperasi pada abstraksi level *Container Runtime Interface* (CRI) [2].



Gambar 2.3 Perubahan Arsitektur Containerd pada Kubernetes [13]

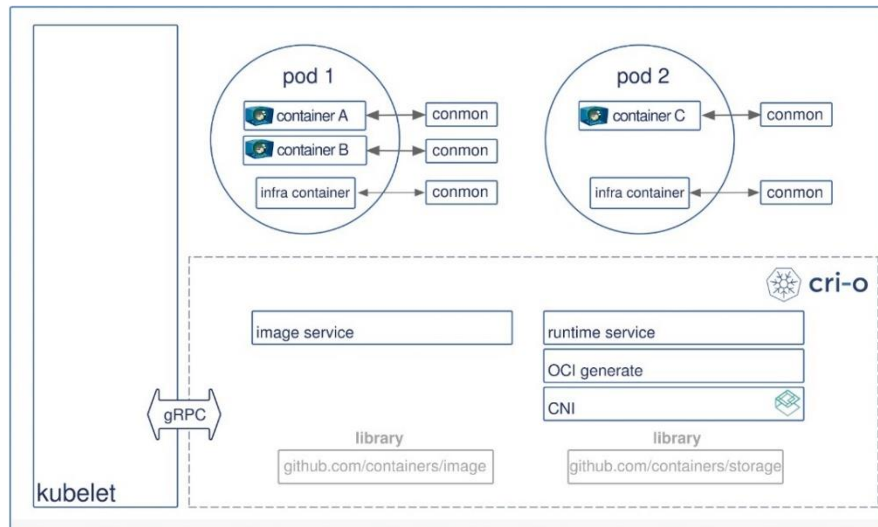
Berdasarkan Gambar 2.3, arsitektur integrasi containerd Kubernetes telah berkembang dua kali. Setiap evolusi telah membuat tumpukan lebih stabil dan efisien. Untuk containerd 1.0, daemon cri-containerd diperlukan untuk beroperasi antara Kubelet dan containerd. Cri-containerd menangani permintaan layanan *Container Runtime Interface* (CRI) dari Kubelet dan menggunakan containerd untuk mengelola container dan *image container* secara bersamaan. Dibandingkan dengan implementasi Docker CRI (dockershim), ini menghilangkan satu lompatan tambahan di tumpukan. Namun, cri-containerd dan containerd 1.0 masih merupakan 2 daemon berbeda yang berinteraksi melalui grpc. Daemon ekstra dalam loop membuatnya lebih kompleks bagi pengguna untuk memahami dan menyebarkan, dan memperkenalkan overhead komunikasi yang tidak perlu.

Di containerd 1.1, daemon cri-containerd sekarang di-refactor menjadi plugin CRI containerd. Plugin CRI dibangun ke dalam containerd 1.1, dan diaktifkan secara default. Tidak seperti cri-containerd, plugin CRI berinteraksi dengan containerd melalui pemanggilan fungsi langsung. Arsitektur baru ini membuat integrasi lebih stabil dan efisien, serta menghilangkan lompatan grpc lainnya di stack. Pengguna kini dapat menggunakan Kubernetes dengan containerd 1.1 secara langsung. Daemon cri-containerd tidak lagi diperlukan [13].

B. CRI-O

CRI-O adalah *container runtime* yang dibangun untuk menjembatani antara *runtime* OCI dan CRI Kubernetes tingkat tinggi. Ini didasarkan pada versi arsitektur Docker yang lebih lama yang dibangun di sekitar *driver* grafik. Ini terutama dikembangkan oleh RedHat, dan berfungsi sebagai *runtime default* untuk

OpenShift, distribusi Kubernetes yang populer untuk perusahaan. Siklus hidup tipikal dari interaksi dengan CRI-O serupa dengan *containerd* pada titik akhir CRI. Perbedaan utama dalam penanganan *container* internal dengan *containerd* tidak ada karena runc adalah *runtime* OCI *default* saat menjalankan CRI-O [2].



Gambar 2.4 Arsitektur Container Runtime CRI-O [14]

Pada gambar 2.4 merupakan arsitektur dari container runtime CRI-O. Runtime CRI-O memanfaatkan *Open Container Initiative* (OCI), yang menyediakan spesifikasi untuk konfigurasi, lingkungan eksekusi, dan siklus hidup sebuah container serta spesifikasi untuk konfigurasi, sistem file, indeks, dan manifes gambar. Untuk mengelola image container dan penyimpanan, CRI-O menggunakan pustaka *containers/image* dan *containers/storage* dari proyek *Containers open source*, yang menyertakan beberapa alat container. Dua pustaka *container* digunakan untuk menarik image dari registri image dan menyimpan konten image dalam sistem berkas *container*. Jaringan CRI-O diimplementasikan dengan *Container Networking Interface* (CNI), proyek CNCF lain yang menyediakan spesifikasi dan pustaka untuk mengonfigurasi antarmuka jaringan dalam container Linux. *Container* dipantau dengan alat internal conmon, yang mengumpulkan log *container* dan mencatat kode keluar [14].

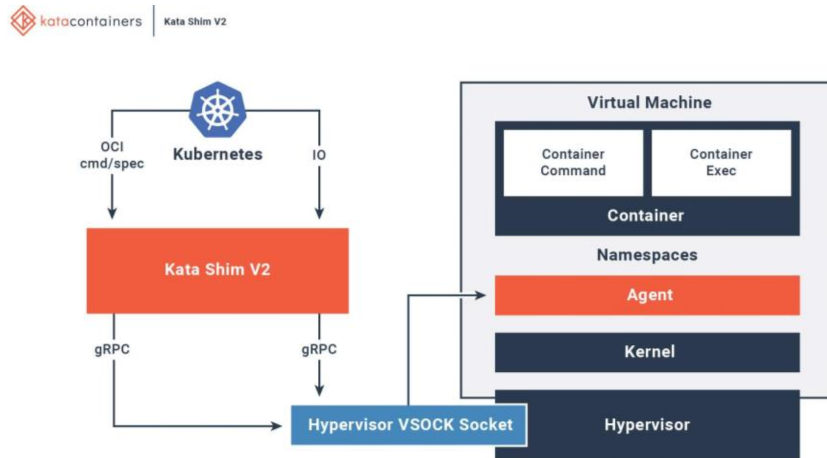
C. Kata *Containers*

Kata *Containers* adalah komunitas *open source* yang bekerja untuk membangun *container runtime* yang aman dengan mesin virtual ringan yang terasa

dan bekerja seperti *container*, tetapi memberikan isolasi beban kerja yang lebih kuat menggunakan teknologi virtualisasi perangkat keras sebagai lapisan pertahanan kedua [15]. Kata *Containers* adalah teknologi lain yang menarik berdasarkan prinsip *micro*-VMs. Ini berasal dari proyek *Clear Containers Intel* yang diluncurkan pada tahun 2015. *Clear Containers* didasarkan pada teknologi virtualisasi perangkat keras *Intel* VT dan *hypervisor* KVM/QEMU yang disesuaikan. Pada tahun 2017, *Hyper Runv*, *hypervisor* OCI lainnya hadir untuk menggantikan KVM/QEMU khusus di *Clear Containers* untuk memulai proyek Kata *Containers* baru yang dikelola oleh *OpenStack Foundation* (OSF) [4].

Runtime Kata *Containers* kompatibel dengan spesifikasi *runtime* OCI dan oleh karena itu bekerja mulus dengan *Container Runtime Interface* (CRI) Kubernetes melalui implementasi CRI-O dan *containerd* [16]. Gambar 2.5 merupakan arsitektur Kata Containers, adapun Kata *Containers* menawarkan beberapa fitur yang menjadi keunggulannya, antara lain :

1) *Security* : Berjalan dalam kernel khusus, menyediakan isolasi jaringan, I/O dan memori dan dapat memanfaatkan isolasi yang didukung perangkat keras dengan ekstensi VT virtualisasi seperti *Direct Device Assignment*, SRIOV, dan lainnya.
2) *Compatibility* : Mendukung standar industri termasuk format *container* OCI, *interface* CRI Kubernetes, serta teknologi virtualisasi lama (KVM,/QEMU, *Hyper Runv*, NEMU, dan lainnya).
3) *Performance* : Memberikan kinerja yang konsisten sebagai standar *container Linux*, peningkatan isolasi tanpa kinerja pajak (*tax*) standar mesin virtual.
4) *Simplycity* : Menghilangkan persyaratan untuk *container* bersarang di dalam mesin virtual yang penuh sesak, standar antarmuka memudahkan untuk memasang dan memulai. *Container* dijalankan di dalam VM ringan menggunakan *runtime* Kata, yang sesuai dengan OCI dan CRI [4] [15].



Gambar 2.5 Arsitektur Kata Containers pada Kubernetes [15]

2.2.3 Kubernetes

Kubernetes merupakan orkestrasi *container* yang juga dikenal dengan K8s. Tanggung jawab manajemen *container* Kubernetes meliputi penyebaran *container*, penskalaan (*scaling* dan *descaling*) container, serta *load balancing container* [17]. Kubernetes adalah *framework open-source* yang dibuat di lab Google untuk mengawasi aplikasi dalam *container* berbagai macam situasi, misalnya fisik, virtual, dan *cloud framework* [18]. Beberapa fitur pada Kubernetes antara lain :

1) *Service discovery and load balancing*

   Kubernetes dapat mengekspos *container* menggunakan nama DNS atau menggunakan alamat IP mereka sendiri. Jika trafik ke container tinggi, Kubernetes dapat menyeimbangkan beban dan mendistribusikan trafik jaringan sehingga penerapannya stabil.

2) *Storage orchestration*

   Kubernetes memungkinkan *user* memasang sistem penyimpanan pilihan *user* secara otomatis, seperti penyimpanan lokal, penyedia *cloud* publik, dan lainnya.

3) *Automated rollouts and rollbacks*

   User dapat mendeskripsikan status yang diinginkan untuk *container* yang diterapkan menggunakan Kubernetes, dan ini dapat mengubah status aktual ke status yang diinginkan dengan kecepatan yang terkontrol. Misalnya, dapat mengotomatiskan Kubernetes untuk membuat *container* baru untuk

deployment user, menghapus *container* yang ada, dan mengadopsi semua sumber dayanya ke *container* baru.

4) *Automatic bin packing*

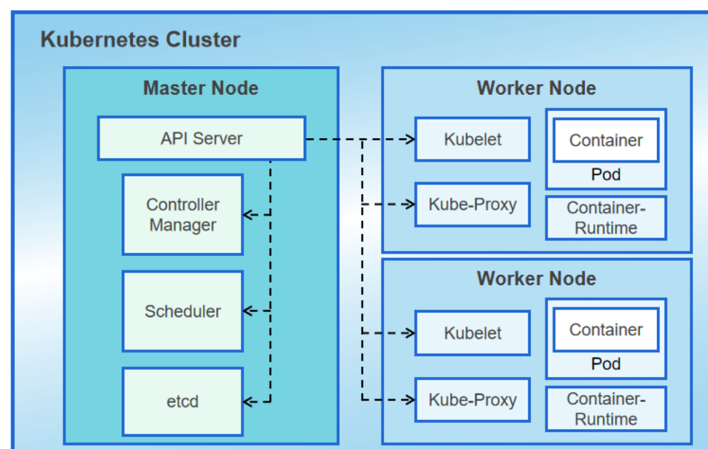
Kubernetes secara otomatis mengemas aplikasi dan menjadwalkan *container* berdasarkan persyaratan. Untuk memastikan pemanfaatan lengkap dan menghemat sumber daya yang tidak terpakai, Kubernetes menyeimbangkan antara beban kerja kritis dan upaya terbaik.

5) *Self-healing*

Kubernetes memulai ulang *container* yang gagal, mengganti *container*, mematikan *container* yang tidak merespons pemeriksaan kesehatan yang ditentukan pengguna, dan tidak mengiklankannya ke klien hingga siap untuk ditayangkan.

6) *Secret and configuration management*

Kubernetes memungkinkan *user* menyimpan dan mengelola informasi sensitif, seperti kata sandi, token *OAuth*, dan kunci SSH. *User* dapat menerapkan dan memperbarui rahasia dan konfigurasi aplikasi tanpa membuat ulang *image container*, dan tanpa membuka rahasia di konfigurasi *stack user* [19].



Gambar 2.6 Arsitektur Kubernetes [20]

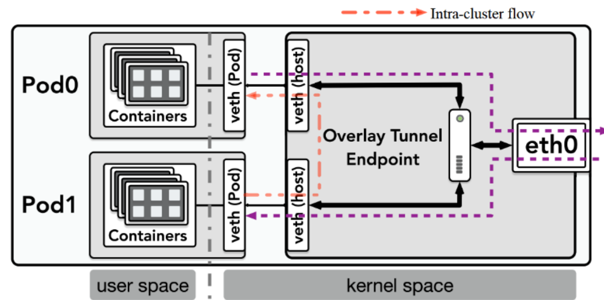
Gambar 2.6 merupakan arsitektur dari Kubernetes. Arsitektur kluster Kubernetes terdiri dari *Master node* dan sekumpulan *worker node*. *Master node* bertanggung jawab atas pemeliharaan dan pengelolaan *cluster*, sementara *worker node* menjalankan satu set *container* untuk aplikasi [20]. Komponen-komponen pada *Master node* Kubernetes antara lain :

1) API *Server* : API *server* memaparkan fungsionalitas kepada pengguna eksternal. API *server* mengelola Kubernetes menggunakan perintah kubectl.
2) *Controller Manager* : *Controller manager* yang mengelola proses dan layanan ini berjalan bersama di *master* Kubernetes.
3) *Scheduler* : *Scheduler* memastikan bahwa *pod* Kubernetes atau *deployment* dijadwalkan di suatu tempat di lingkungan.
4) *Etcd* : etcd yang merupakan *database backend* dengan pasangan *key value* yang disimpan di dalam *database*.

Komponen-komponen pada *Worker node* Kubernetes antara lain :

A. Kubelet : Kubelet berfungsi untuk memastikan *container* beroperasi di dalam *Pod*.
B. Kube-*proxy* : Kube-*proxy* memastikan bahwa apapun yang terjadi di dalam *container* tersedia, serta memelihara *network rules* dan meneruskan koneksi ke suatu *host*.
C. *Pods* : Objek terkecil di dalam *cluster* kubernetes yang terletak di dalam *node*. *Pod* berfungsi untuk menjalankan *docker images* yang membentuk sebuah *container*. Kubernetes memastikan semua pod ini dipantau dan jika ada masalah muncul, *pod* baru akan dijadwalkan [17].
D. *Container Runtime* : perangkat lunak yang bertanggungjawab menjalankan *container* dan mengelola *image container* pada *deployment node* [2].

Pada Kubernetes terdapat komunikasi yang disebut komunikasi *intra-cluster* yang digambarkan Gambar 2.7, komunikasi yang juga digunakan dalam penelitian ini. Komunikasi *intra-cluster* merupakan komunikasi yang berada di dalam *cluster* Kubernetes antara *pod* (*container*) satu dengan *pod* (*container*) lainnya. Pada Gambar 2.7, jalur komunikasi *intra-cluster* menggunakan jalur berwarna *orange*, di mana jalur komunikasi kontainer masih berada antara dua kontainer (pod0 dan pod1) dalam satu *host* [21].



Gambar 2.7 Komunikasi intra-cluster [21]

2.2.4 *Netperf*

Netperf adalah *tools benchmark* yang dapat digunakan untuk mengukur berbagai aspek kinerja jaringan antara dua *host*. Fokus utamanya adalah *transfer* data massal (atau searah) dan kinerja *request/respons* menggunakan protokol TCP atau UDP dan *interface Berkeley Sockets*. *Netperf* menyediakan *test* untuk *unidirectional throughput* maupun *end-to-end latency* [22]. *Test default* pada *Netperf* yaitu TCP_STREAM dan UDP_STREAM yang mentransmisikan data TCP dan UDP antara klien *netperf* dan *server netperf* (*netserver*) [23].

2.2.5 *y-cruncher*

y-cruncher adalah program yang dapat menghitung Pi dan konstanta lainnya hingga triliunan digit. Ini adalah yang pertama dari jenisnya yang *multi-threaded* dan dapat diskalakan ke sistem *multi-core*. Sejak diluncurkan pada tahun 2009, *y-cruncher* telah menjadi aplikasi *benchmarking* dan *stress-testing* yang umum bagi para *overclocker* dan penggemar perangkat keras (*hardware*) [24]. Alat ini memberikan keluaran yang berbeda, yaitu *total computation time, start-to-end wall time,* CPU *utilization, multi-core efficiency, kernel overhead, Pi, topology (threads), usable memory, CPU base frequency*, dan lainnya. Total waktu digunakan untuk memverifikasi hasil, dan ini terdiri dari total waktu komputasi ditambah waktu yang diperlukan untuk mengelaborasi dan memproses hasilnya [23]. CPU *utilization* mengacu pada penggunaan sumber daya pemrosesan komputer, atau jumlah pekerjaan yang ditangani oleh CPU. CPU *utilization* sebenarnya bervariasi tergantung pada jumlah dan jenis tugas komputasi yang dikelola. Tugas-tugas tertentu membutuhkan waktu CPU yang berat, sementara yang lain membutuhkan lebih sedikit karena persyaratan sumber daya non-CPU

[25]. *Multi-core efficiency* adalah seberapa efisien penggunaan seluruh *core* pada CPU untuk menjalankan banyak proses pada saat yang sama. CPU yang menawarkan banyak inti (*multi-core*) dapat bekerja jauh lebih baik daripada CPU inti tunggal (*single-core*) dengan kecepatan yang sama. Pada y-cruncher, parameter CPU *utilization* dan *multi-core efficiency* memiliki kinerja yang bagus jika persentase-nya semakin tinggi. Berbeda dengan pengukuran CPU pada umumnya, y-cruncher menilai CPU dengan seberapa maksimum load persentase kinerja yang dihasilkan, bukan persentase terendah [24].

2.2.6 *STREAM*

Benchmark STREAM merupakan *tools benchmark* yang berfungsi untuk mengukur kinerja *bandwidth memory* (MB/s) menggunakan operasi kernel vektor yang sangat sederhana. Ini menghasilkan hasil untuk empat operasi berbeda, yaitu *Copy, Scale, Add (Sum),* dan *Triad* [23]. *Benchmark STREAM* dirancang khusus untuk bekerja dengan kumpulan data yang jauh lebih besar daripada *cache* yang tersedia pada sistem apa pun, sehingga hasilnya (mungkin) lebih menunjukkan kinerja aplikasi gaya vektor yang sangat besar [26]. Pada penelitian ini hanya mengambil hasil dari parameter *memory Copy* dan *Scale*. Parameter memory *copy* di aplikasi STREAM berarti mengukur tingkat transfer data tanpa adanya proses aritmatika. Paremeter *memory* s*cale* di aplikasi STREAM merupakan pengukuran transfer data dengan menambahkan operasi aritmatika.

2.2.7 Parameter *QoS* dan *Kinerja Hardware*

Quality Of Service merupakan kemampuan untuk menyediakan kualitas layanan yang baik dari suatu jaringan dengan mengatasi *jitter* dan *delay*, serta menyediakan *bandwidth*. Parameter dari QoS adalah *jitter, latency, packet loss, throughput*, *echo cancellation,* MOS, dan PDD.

2.2.7.1 *Throughput*

Throughput mengacu pada jumlah data yang dapat dikirim dan diterima selama periode waktu tertentu. *Throughput* mengukur tingkat rata-rata di mana pesan berhasil tiba di tujuan yang relevan. Alih-alih mengukur pengiriman teoretis

paket, *throughput* memberikan pengukuran praktis pengiriman aktual. Rata-rata data *throughput* jaringan memberi pengguna wawasan tentang jumlah paket yang berhasil tiba di tujuan yang benar. *Throughput* diukur dalam *byte per second* (Bps). Saat mengukur *throughput* jaringan, dapat mengambil rata-rata dan itu dianggap sebagai representasi akurat dari *throughput* kinerja jaringan secara keseluruhan. Jadi, jika administrator jaringan menemukan bahwa *throughput* rendah, itu mungkin berarti ada masalah kehilangan paket (*packet loss*) [28]. *Throughput* dapat dihitung menggunakan formula pada persamaan 2.1 [29].

$$Throughput = \frac{paket\ data\ yang\ diterima\ (Bytes)}{waktu\ pengiriman\ data\ (s)} \quad (2.1)$$

Tabel 2.2 Standarisasi *Throughput* versi TIPHON [27]

Kategori	***Throughput***	**Indeks**
Sangat Baik	> 2,1 MBps	4
Baik	1200 KBps – 2,1 MBps	3
Cukup	700 KBps – 1200 KBps	2
Jelek	338 KBps – 700 KBps	1
Sangat Jelek	0 KBps – 338 KBps	0

2.2.7.2 *Latency*

Latency adalah jumlah waktu yang diperlukan paket data untuk berpindah dari satu titik di jaringan ke titik lainnya. Menurunkan latensi adalah bagian penting dalam membangun *user experience* yang baik [30]. Dalam kasus pengujian permintaan/*respons*, latensi akan menjadi latensi transaksi. Dalam kasus tes *receive-only*, mereka akan menghabiskan waktu dalam menerima panggilan. Dalam kasus tes *send-only*, mereka akan menghabiskan waktu dalam panggilan kirim. Satuannya yaitu *milisecond* (ms) [22]. Jenis *latency* yang diukur pada penelitian ini adalah *latency* pada jaringan. Latensi jaringan adalah jumlah waktu yang diperlukan paket data untuk berpindah dari satu tempat ke tempat lain [31]. *Latency* dapat dihitung menggunakan rumus pada persamaan 2.2 [29].

$$Latency\ \text{(rata-rata per paket)} = \frac{total\ delay\ (sec)}{total\ packet\ yang\ diterima} \quad (2.2)$$

Tabel 2.3 Standarisasi *Latency* versi TIPHON [27]

Kategori	*Latency*	Indeks
Sangat Baik	< 150 ms	4
Baik	150 ms – 300 ms	3
Cukup	300 ms – 450 ms	2
Jelek	> 450 ms	1

2.2.7.3 *CPU*

CPU mewakili pemrosesan komputasi dan ditentukan dalam satuan CPU Kubernetes. Batasan dan permintaan sumber daya CPU diukur dalam unit cpu. Di Kubernetes, 1 unit CPU setara dengan 1 inti CPU fisik, atau 1 inti virtual, bergantung pada apakah *node* tersebut adalah *host* fisik atau mesin virtual yang berjalan di dalam mesin fisik [32]. CPU diukur menggunakan *tools* y-cruncher. Y-cruncher merupakan program CPU *benchmark* yang dihitung dalam satuan Pi (banyaknya konstanta atau perintah yang dijalankan). Satuan Pi ini akan dikonversi ke satuan persentase (%) agar mempermudah proses analisis [24]. CPU dapat dihitung menggunakan persamaan 2.3.

$$\text{CPU (\%)} = \frac{jumlah\ Pi\ yang\ dapat\ diproses}{total\ Pi} x\ 100\% \quad (2.3)$$

2.2.7.4 *Memory*

CPU komputer semakin cepat jauh lebih cepat daripada sistem memori komputer. Seiring perkembangan ini, semakin banyak program akan dibatasi kinerjanya oleh *bandwidth* memori sistem, bukan oleh kinerja komputasi CPU [26]. Pada penelitian ini *tools* STREAM digunakan untuk mengambil data, dimana performa yang diukur oleh *tools* tersebut memiliki ketergantungan yang kuat pada ukuran *cache* CPU [23]. *Memory* yang diukur menggunakan STREAM memiliki satuan (MB/s) [26]. *Memory* dapat dihitung menggunakan persamaan 2.4.

$$\text{Memory} = \frac{banyaknya\ memori\ yang\ digunakan\ (MB)}{total\ waktu\ (sec)} \quad (2.4)$$