BAB 2
DASAR TEORI

2.1 KAJIAN PUSTAKA

Penelitian terkait *Container Network Interface* (CNI) pada Kubernetes telah dilakukan oleh beberapa peneliti. Penggunaan CNI sangat diperlukan di mana pengaplikasiaannya pengguna tidak perlu secara langsung membuat tautan antara *Pod* sehingga tidak perlu berurusan dengan memetakan *port* kontainer ke *port* pada *host*. Pada penelitian [7] meneliti mengenai kinerja penggunaan berbagai CNI yaitu *Flannel*, *Calico*, *Weave Net*, *Contiv*, *Cilium*, *Weave Net*, dan *Romana* pada jaringan 10Gbit/s. Menggunakan skenario komunikasi antara *pod* ke *pod* dan juga komunikasi *ingress* and *egress*. Parameter yang diambil pada penelitian tersebut adalah *performance comparison* meliputi uji *bandwidth* untuk protokol jaringan yang berbeda (TCP, UDP, HTTP, FTP and SCP) dan tes pemanfaatan sumber daya perangkat. Hasil yang diperoleh *Flannel* mudah diatur, otomatis mendeteksi MTU (*Maximum Transmission Unit*), menawarkan (rata-rata) *bandwidth* yang bagus untuk mentransmisikan paket di bawah semua protokol dan memiliki CPU dan pemanfaatan RAM. Namun, *Flannel* tidak mendukung *Network Policy* dan tidak tidak menyediakan penyeimbang beban. *Calico* dan *Weave Net* rata-rata menawarkan *bandwidth* yang baik untuk mentransmisikan paket, mendukung *Network Policy*, menyediakan penyeimbang beban dan memiliki sumber daya yang rendah pemanfaatan. Pada saat yang sama, kedua jaringan tidak mendeteksi MTU secara otomatis. Di antara jaringan terenkripsi, *Weave Net* menonjol lebih baik daripada *Cilium* di sebagian besar aspek termasuk *bandwidth* yang ditawarkan ke protokol jaringan dan sumber daya penggunaan (CPU dan RAM).

Pada penelitian [8] meneliti mengenai performansi perbandingan berbagai CNI pada Kubernetes *cluster*. CNI yang diuji antara lain *Flannel*, *Antrea*, *Calico*, *Contiv*, *Cilium*, OVN-Kubernetes, *Kube-Router*, *Polycube*-K8s, *WeaveNet*, dan *Amazon* CNI. Penelitian ini menggunakan skenario *pod to pod* dan *pod to services to pod* pada jaringan 10Gbits/s. Parameter yang diuji yaitu *throughput*, persentase penggunaan CPU pada *node*, dan *latency*. Penelitian menunjukkan hasil CNI yang memiliki kinerja terbaik dalam skenario yang ada dalam pengujian, didapatkan

bahwa *Calico* dan *Cilium* memiliki performansi terbaik, menggabungkan kinerja yang diperoleh dalam berbagai pengujian dengan keandalan dan keamanan yang baik. CNI lain yang gagal dalam salah satu dari dua pengujian yaitu *Polycube* di mana tidak selalu menggunakan keandalan total dengan performansi yang sangat baik. Pada CNI *Weave Net*, meskipun dapat diandalkan, tetapi tidak memiliki *throughput* yang baik.

Pada penelitian [9] meneliti mengenai performansi perbandingan CNI pada Kubernetes *cluster*. CNI yang diuji adalah *Flannel, Weave Net, Calico, Cilium*, dan *Kube-router*. Penelitian ini menggunakan skenario komunikasi *intra-host* (komunikasi dalam *host server* tunggal), komunikasi *inter-host* (berkomunikasi antara dua *pods* pada *host* yang berbeda), serta penambahan jumlah koneksi. Skenario jumlah koneksi yang digunakan yaitu koneksi TCP dari 1, 100, 200, 500, 800, 1000, 1300, dan 1500 koneksi. Parameter yang diuji yaitu *throughput* dan *latency* dengan waktu simulasi 30 detik. Berdasarkan hasil yang didapatkan, pada komunikasi *intra-host* CNI *Cilium* memiliki kinerja paling baik, dengan EBPF dioptimalkan untuk perutean dalam *host*. Pada komunikasi *inter-host*, CNI *Kube-Router* dan *Calico* memiliki kinerja lebih baik karena mode *routing* IP yang lebih ringan dibandingkan dengan rekan *overlay* mereka.

Penelitian terkait membandingkan kinerja antara CNI *Flannel*, *Calico*, *Weave Net*, *Contiv*, *Cilium*, *Antrea*, OVN-Kubernetes, *Kube-Router*, *Polycube*-K8s, *Amazon* CNI dan *Romana* telah dilakukan oleh penelitian [7] dan [8], dalam penelitian ini CNI *Calico*, *Cilium*, *Flannel*, dan *Weave Net* digunakan penulis untuk pembanding CNI Kubernetes untuk trafik *web server Nginx*. Skenario yang dilakukan penelitian [8] dan [9] yaitu *pod to pod* dan *pod to service to pod*, sedangkan skenario yang digunakan pada penelitian penulis yaitu *pod to pod*, *pod to service*, dan *client to service*. Skenario penelitian [9] menggunakan variasi jumlah koneksi, sedangkan penelitian penulis menggunakan variasi jumlah *simulated user* yaitu 10, 100, dan 200 user dengan waktu simulasi yaitu 60 detik. Pada penelitian [8] dan [9] membandingkan parameter pengujian *throughput*, CPU *usage*, *latency*, dan *packet loss*, penelitian penulis membandingkan parameter pengujian CPU *usage* dan *throughput* serta menambahkan parameter *response time* dan *transaction rate*. Tabel 2.1 merupakan ringkasan dari penelitian sebelumnya.

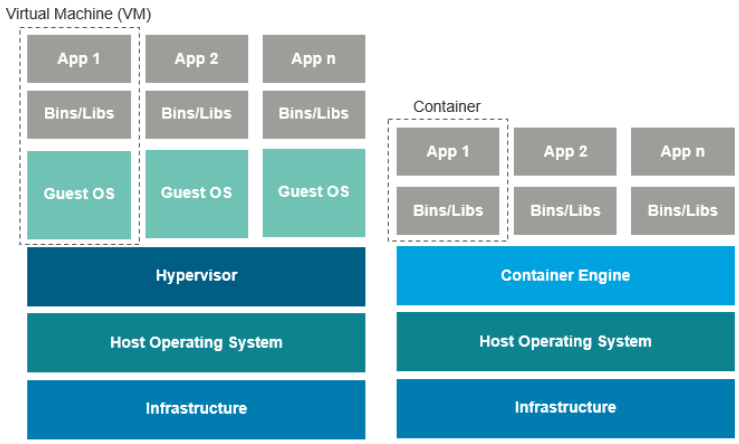
Tabel 2.1 Ringkasan Penelitian Sebelumnya

Penulis	CNI	Skenario	Kelebihan	Kekurangan	Hasil
Ritik Kumar, Munesh Chandra Trivedi [7]	*Flannel*, *Calico*, *Weave Net*, *Contiv*, *Cilium*, *Weave Net*, dan *Romana*	Skenario komunikasi *pod* ke *pod*, komunikasi *ingress* dan *egress*	Menggunakan skenario komunikasi *ingress* dan *egress*, serta CNI lebih banyak	Beberapa CNI unggul dalam parameter tertentu	Weave Net memiliki kinerja yang lebih baik dalam aspek bandwidth dan penggunaan CPU dan RAM
David Liffredo [8]	*Flannel*, *Antrea*, *Calico*, *Contiv*, *Cilium*, OVN-K8s, *Kube-Router*, *Polycube*-K8s, *WeaveNet*, dan *Amazon* CNI	Skenario *pod to pod* dan *pod to services to pod*	Menguji lebih banyak CNI	Memiliki lebih dari satu CNI untuk performansi yang baik, menguji hanya dengan dua skenario	*Calico* dan *Cilium* memiliki performansi terbaik, menggabungkan kinerja yang diperoleh dalam berbagai pengujian dengan keandalan dan keamanan yang baik.
Shixiong Qi, Sameer G Kulkarni, K. K. Ramakrishnan [9]	*Flannel*, *Weave Net*, *Calico*, *Cilium*, dan *Kube-router*.	Skenario komunikasi *intra-host*, komunikasi *inter-host*, serta penambahan jumlah koneksi.	Menggunakan lebih banyak jumlah koneksi dan CNI yang diuji	Hanya menguji dengan parameter *throughput* dan *latency*	Pada komunikasi *intra-host*, CNI *Cilium* memiliki kinerja paling baik. Pada komunikasi *inter-host*, CNI *Kube-Router* dan *Calico* memiliki kinerja lebih baik.

2.2 DASAR TEORI

2.2.1 *Container*

Kontainer adalah unit standar perangkat lunak yang mengemas kode dan semua dependensinya sehingga aplikasi berjalan dengan cepat dan andal dari satu lingkungan komputasi ke lingkungan komputasi lainnya [10]. Kontainer dapat diasumsikan sebagai sebuah wadah yang dapat menampung banyak data maupun aplikasi dengan tujuan menjadi lebih ringkas untuk dijalankan pada sistem.



Gambar 2.1 Perbandingan arsitektur *Virtual Machine* dan kontainer [11]

Perbandingan infrastruktur VM (*Virtual Machine*) dan kontainer pada Gambar 2.1 yaitu kontainer hanya mengisolasi *library* serta aplikasi yang digunakan tanpa mengisolasi keseluruhan perangkat keras, *kernel*, dan OS. Berbeda dengan VM yang mengisolasi keseluruhan sistem. Secara umum *container* terbagi menjadi dua jenis yaitu:

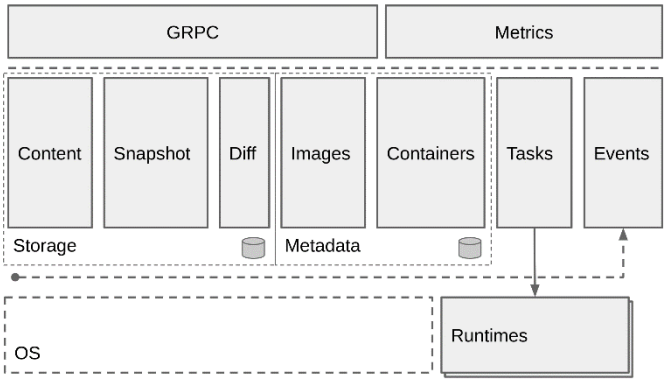
1. Kontainer berbasis sistem operasi, yaitu kontainer yang mengisolasi level sistem operasi. Teknologi ini menawarkan fitur yang mirip dengan virtualisasi namun dengan peningkatan performa yang cukup signifikan. Sifatnya yang memiliki lingkungan terisolasi antara satu dengan yang lainnya serta performansi tinggi memberikan keunggulan terhadap kontainer.
2. Kontainer berbasis aplikasi, yaitu kontainer yang mengisolasi level aplikasi. Di mana memudahkan para pengguna untuk membuat dan memaksimalkan suatu aplikasi yang dikelola. Selain memanfaatkan *hardware* sang induk, juga dapat memanfaatkan *service-service* lain dari kontainer-kontainer yang saling berhubungan dan berjalan pada sistem [12].

Kontainer dikenal juga sebagai sebuah sistem operasi virtualisasi yang ringan dan menjadi alternatif lain dari virtualisasi berbasis *hypervisor*. Kontainer membuat *instance* dengan *multiple userspace* yang terisolasi diatas OS *kernel* yang sama. Kontainer menyediakan sebuah abstraksi diatas suatu OS *kernel* yang memungkinkan untuk menjalankan *multiple* proses didalam sebuah kontainer yang terisolasi dari kontainer lainnya [13].

2.2.2 *Containerd*

Containerd adalah *runtime container* standar industri dengan penekanan pada kesederhanaan, ketahanan, dan portabilitas [14]. *Container runtime* bertanggung jawab untuk menjalankan *container*, dan abstrak manajemen *container* untuk Kubernetes [15]. *Containerd* tersedia sebagai *daemon* untuk *Linux* dan *Windows* [16]. Mulai 28 Februari 2019, *containerd* secara resmi merupakan proyek yang lulus dalam *Cloud Native Computing Foundation*, mengikuti Kubernetes, *Prometheus*, *Envoy*, dan *Core*DNS. *Containerd* mengelola siklus hidup *container* lengkap dari sistem *host*-nya, mulai dari transfer dan penyimpanan gambar hingga eksekusi, pengawasan *container*, penyimpanan tingkat rendah hingga lampiran jaringan dan seterusnya. *Containerd* dirancang untuk dipasang ke dalam sistem yang lebih besar, daripada digunakan langsung oleh pengembang atau pengguna akhir [14].

Containerd runtime menyediakan abstraksi *layering* yang memungkinkan implementasi serangkaian fitur yang kaya seperti *gVisor* untuk memperluas fungsionalitas Kubernetes. *Runtime containerd* dianggap lebih hemat sumber daya dan aman daripada *runtime Docker* [15]. *Containerd* ini menawarkan kinerja yang unggul karena *overhead* komputasi yang lebih rendah, latensi memori minimum, dan batas sumber daya yang dapat dikonfigurasi [17].



Gambar 2.2 Arsitektur *Containerd* [14]

Pada Gambar 2.2 menunjukkan arsitektur *containerd*. Secara keseluruhan *containerd* dapat dibagi menjadi tiga blok besar yaitu *Storage*, *Metadata* dan *Runtime*. Seperti yang terlihat, *containerd* juga menggunakan arsitektur C/S, sisi *server* melalui soket *domain unix* untuk mengekspos antarmuka API gRPC tingkat rendah, klien melalui API ini untuk mengelola kontainer pada *node*, setiap *containerd* bertanggung jawab hanya untuk satu mesin, *Pull image* Setiap *containerd* bertanggung jawab hanya untuk satu mesin, *Pull image*, pengoperasian *container* (*start*, *stop*, dll.), jaringan, penyimpanan semua dilakukan oleh *containerd*. Kontainer khusus yang sedang berjalan adalah tanggung jawab *runc*, dan bahkan kontainer apa pun yang sesuai dengan spesifikasi OCI dapat didukung.

Plugin Content menyediakan akses ke konten yang dapat dialamatkan dalam *image*, tempat semua konten yang tidak dapat diubah disimpan. *Plugin Snapshot* digunakan untuk mengelola *snapshot* sistem *file* dari gambar kontainer, setiap lapisan *image* didekompresi menjadi *snapshot* sistem *file*, mirip dengan *graphdriver* di *Docker* [18].

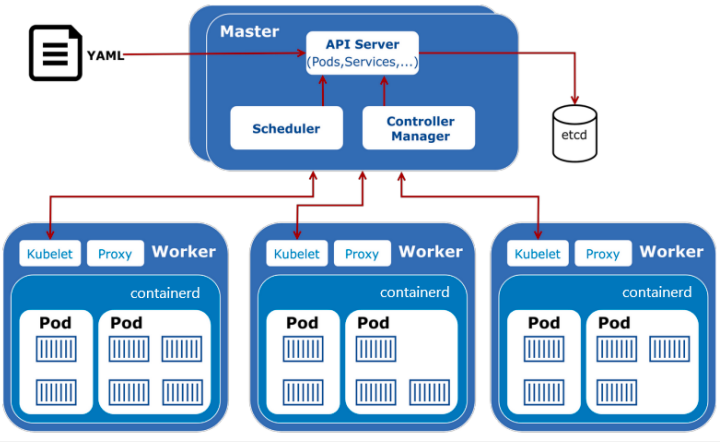
2.2.3 *Kubernetes*

Kubernetes adalah sebuah *container orchestration open source* untuk menjalankan aplikasi kontainer. Kubernetes secara resmi dikembangkan oleh *Google*, terinspirasi dari pengalaman selama satu dekade dalam proses menjalankan sistem yang *scalable*, *reliable* didalam kontainer melalui *application-oriented* APIs [13]. Kubernetes merupakan *platform cluster opensource* yang digunakan untuk otomatisasi dari segi *deployment*, *scaling* dan *manajemen container*. Kubernetes

merupakan *platform* yang *portable* karena dapat diaplikasikan pada *private* maupun *public cloud* [19].

Kubernetes sendiri berfungsi sebagai pengelola kontainer-kontainer serta menyediakan *platform* untuk mendukung fungsinya. Desain Kubernetes terdiri dari *Pods, Labels and Selectors*, *Controllers*, dan *Services*. Objek dasar yang dimiliki oleh Kubernetes antara lain:

1. *Pod*: Unit terkecil dan paling sederhana dalam model objek Kubernetes. *Pod* mewakili unit penyebaran di mana satu *instance* aplikasi di Kubernetes dapat terdiri dari satu kontainer atau sejumlah kontainer yang berbagi sumber daya.
2. *Service*: Abstraksi yang mendefinisikan serangkaian *Pods* sejenis, yang menjadi jalan masuk bagi *request* oleh *pod-pod* tersebut. *Service* mengawasi dan mencatat adanya perubahan *state* dari *pod-pod* yang merupakan bagiannya
3. *Volume*: Bagian dari penyimpanan di *cluster* yang telah disediakan. Data yang disimpan tidak akan hilang ketika dilakukan *destroy* terhadap *pod*.
4. *Namespace*: Cara untuk membagi sumber daya *cluster* diantara beberapa pengguna (berdasar kuota sumber daya).
5. *Node*: Mesin pekerja di Kubernetes yang dapat berupa mesin virtual atau fisik, tergantung pada *cluster*. *Node* dapat memiliki banyak *pod* dan *master* [12].



Gambar 2.3 Arsitektur Kubernetes [20]

Pada Gambar 2.3, arsitektur Kubernertes dibagi menjadi dua bagian yaitu *Master Node* dan *Worker Node*. *Master Node* sebagai tempat mengatur *workload* dan komunikasi antar sistem serta mengontrol keseluruhan unit dan *cluster* pada *container*. *Worker Node* atau yang dikenal sebagai pekerja atau *minion*, merupakan mesin tempat *container* (beban kerja) digunakan atau sebagai *host* dari *container*

dijalankan [20] [12]. Unit dasar di Kubernetes ialah *Pod* yang merupakan unit terkecil di dalam objek model Kubernetes. *Pod* dapat dibuat dan di-*deploy* yang memiliki fungsi sebagai wadah dari satu atau beberapa *container*. Kubernetes tidak mengelola *container* secara langsung, namun mengelola *pod* tersebut [21].

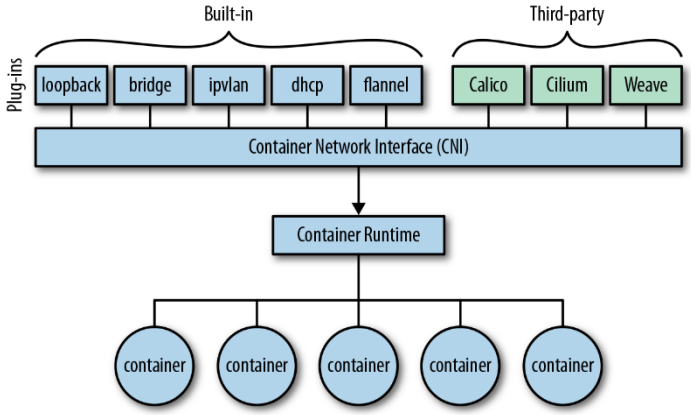
Beberapa komponen yang ada pada arsitektur Kubernetes antara lain:

1. Komponen *Master Node*, menyediakan fungsi *control-plane* pada *master* Kubernetes.
    a. API *server*: bertugas untuk membuka API pada Kubernetes dan sebagai *front-end* dari *control plane* Kubernetes yang didesain agar dapat di-*scale* secara horizontal. Komponen Etcl untuk penyimpanan *key value* konsisten sebagai penyimpanan data *cluster* Kubernetes.
    b. *Scheduler*: bertugas untuk mengatur penjadwalan dan pembagian *container* pada Kubernetes pada setiap *node*, serta mengamati *pod* yang baru dibuat dan belum di-*assign* ke suatu *node* dan akan memilih sebuah *node* untuk menjalankan *pod* baru tersebut.
    c. *Controller Manager*: bertugas untuk menjalankan *controller*. Di dalamnya memiliki beberapa *controller*, yaitu *node controller*, *replication controller*, *end-point controller*, *service account* dan *token controller*, serta *cloud controller manager*.
2. Komponen *Worker Node*
    a. *Kubelet*: sebagai agen pengecekan *container* setiap *pod* di *cluster* dan memastikan *container* dijalankan di dalam *pod*.
    b. *Proxy*: bertugas untuk mengatur komunikasi dari *container* atau *pod* ke *service* dan melakukan *port forwarding*.
    c. *Container Runtime*: aplikasi yang bertugas menjalankan *container*. Kubernetes mendukung beberapa *container runtime*, antara lain *containerd*, *Docker*, cri-o, rktlet, serta semua implementasi Kubernetes CRI (*Container Runtime Interface*) [21] [22].

2.2.4 *Container Network Interface (CNI)*

Container Network Interface (CNI) merupakan spesifikasi jaringan kontainer yang diusulkan oleh *CoreOS*. *Plugin* jaringan pada CNI hanya

menyediakan dua perintah. Pertama untuk menambahkan antarmuka pada *Container* dan kedua untuk menghapusnya [20]. Kunci CNI terletak pada hubungan sederhana antara *container runtime* dan *plugin* jaringan. Selain itu, CNI mendefinisikan *input* dan *output* dibutuhkan oleh *plugin* CNI melalui sintaks JSON (*JavaScript Object Notation*) yang membangun seluruh arsitektur CNI. CNI menyediakan model jaringan kontainer *universal*, kontainer dapat ditambahkan ke beberapa jaringan yang digerakkan oleh berbagai *plugin*, dan setiap jaringan memiliki *plugin* yang sesuai dan nama yang unik. Oleh karena itu, CNI dapat kompatibel dengan teknologi kontainer lainnya dan sistem atas [23].



Gambar 2.4 Arsitektur CNI [24]

Seperti yang terlihat pada Gambar 2.4, CNI menyediakan solusi jaringan berorientasi *plugin* untuk *container* dan orkestra *container*. Ini terdiri dari spesifikasi dan *libraries* untuk menulis *plugin* untuk mengkonfigurasi antarmuka jaringan dalam *container Linux*. Spesifikasi CNI yaitu ringan, CNI hanya berurusan dengan konektivitas jaringan *container*, serta pengumpulan sampah sumber daya setelah *container* dihapus [24].

A. *Calico*

Calico adalah jaringan virtual yang skalabel dan efisien dikembangkan oleh Tigera. *Calico* menyediakan dua opsi untuk konektivitas kontainer di beberapa *host* yaitu *overlay* IPIP dan *routing Border Gateway Protocol* (BGP). Berbeda dengan yang dinyatakan jaringan *overlay*, yang menggunakan paket UDP atau VxLAN (*Virtual eXtensible Local Area Network*) untuk enkapsulasi paket, IPIP mengenkapsulasi paket di *layer* 3 (*network layer*), VxLAN mengenkapsulasi paket

pada *layer* 2. Sedangkan BGP menggunakan *Virtual Router* untuk meneruskan paket yang diterima. Paket IP asli dibungkus paket IP lain di sumber *tunnel* jaringan dan didekapsulasi di sisi tujuan *tunnel*. Sementara *Calico* berbagi kelemahan yang sama dari enkapsulasi dan dekapsulasi yang mahal, itu memberlakukan satu batasan tambahan. *Calico* juga membutuhkan infrastruktur yang mendasari untuk mendukung protokol IPIP [25].

B. *Cilium*

Cilium merupakan *plugin* jaringan L3 dan *policy* jaringan (*Network Policy*) yang dapat menjalankan *policy* HTTP/API/L7 secara transparan. Mendukung mode *routing* maupun *overlay*/*encapsulation* [26]. *Cilium* menyediakan dan secara transparan mengamankan konektivitas jaringan dan penyeimbangan beban antara beban kerja aplikasi seperti wadah (kontainer) atau proses aplikasi. *Cilium* beroperasi pada *Layer* 3/4 untuk menyediakan jaringan tradisional dan layanan keamanan serta *Layer* 7 untuk melindungi dan mengamankan penggunaan protokol aplikasi modern seperti HTTP, gRPC dan Kafka. *Cilium* terintegrasi ke dalam kerangka orkestrasi umum seperti Kubernetes [27].

Cilium adalah proyek *open source* untuk menyediakan jaringan, keamanan, dan observabilitas untuk lingkungan *native cloud* seperti *cluster* Kubernetes dan *platform orchestration container* lainnya. Dasar dari *Cilium* adalah teknologi *kernel Linux* baru yang disebut eBPF yang sangat efisien dan fleksibel, yang memungkinkan penyisipan *bytecode* eBPF dinamis dari keamanan yang kuat, visibilitas, dan logika kontrol jaringan ke dalam *kernel Linux*. eBPF digunakan untuk menyediakan jaringan berkinerja tinggi, kemampuan *multi-cluster* dan *multi-cloud*, penyeimbangan beban tingkat lanjut, enkripsi transparan, kemampuan keamanan jaringan yang luas, kemampuan pengamatan yang transparan, dan banyak lagi. *Cilium* terdiri dari agen yang berjalan di semua *node cluster* dan *server* di lingkungan *user*. *Cilium* menyediakan jaringan, keamanan, dan *observability* untuk beban kerja yang berjalan di *node* itu. Beban kerja dapat ditampung atau dijalankan secara *native* di sistem [28].

C. *Flannel*

Flannel adalah layanan perencanaan jaringan yang dirancang untuk Kubernetes. Penyebaran dan konfigurasi untuk *Flannel* relatif sederhana, melalui alamat IP yang ditetapkan Etcd, dapat mengelola beberapa *container* lintas *host*. *Flannel* dirancang untuk menjadi ringan dan unggul dalam kinerja. *Flannel* adalah mode Jaringan *Overlay* yang memungkinkan data TCP untuk dienkapsulasi dalam paket jaringan lain untuk *routing forwarding* dan komunikasi. *Flannel* mendukung protokol UDP, Vxlan, AWS VPC dan GCE. Kubernetes hadir dengan *Flannel* untuk komunikasi jaringan, jadi *Flannel* kompatibel dengan Kubernetes, dan *Flannel* dapat digunakan untuk meningkatkan kemampuan Kubernetes mengatur dan menjadwalkan efisiensi di bawah Kubernetes jaringan klaster [23].

D. *Weave Net*

Weave Net adalah jaringan yang tangguh dan mudah digunakan untuk Kubernetes dan aplikasi yang di-*hosting*-nya. *Weave Net* berjalan sebagai *plugin* CNI atau berdiri sendiri. Di kedua versi, itu tidak memerlukan konfigurasi atau kode tambahan untuk dijalankan, dan dalam kedua kasus, jaringan menyediakan satu alamat IP per *Pod* - seperti standar untuk Kubernetes [29]. *Weave Net* menyediakan jaringan serta *Network Policy*, yang akan membawa kedua sisi dari partisi jaringan, serta tidak membutuhkan basis data eksternal [26].

2.2.5 *Web Server*

Web server merupakan *server* yang memberikan layanan data yang berfungsi menerima permintaan *Hypertext Transfer Protocol* (HTTP) dari *user* sebagai protokol pengiriman data (*browser web*) dan mengirimkan kembali hasilnya dalam bentuk halaman-halaman *website* yang umumnya menampilkan teks, gambar, animasi, dan video [30] [31]. *Web server* juga menyediakan layanan akses ke suatu berkas, berkas tersebut dapat berupa *Hypertext Markup Language* (HTML), berkas *Javascript*, dan berkas *Perl* [32]. Beberapa *web server* yang bersifat *open source* antara lain *Apache* HTTP *Server*, *Nginx*, *Lighttpd*, *OpenLiteSpeed*, *Apache Tomcat*, dan lainnya.

Web Service dapat diakses dan dipublikasi menggunakan *standard Internet* (TCP/IP, HTTP, *web*). *Web service* dapat diimplementasikan pada lingkungan internal (*intranet*) untuk kebutuhan integritas antar sistem aplikasi (EAI/*Enterprise Application Integration*) ataupun pada lingkungan eksternal (*internet*) untuk mendukung aplikasi *business-to-business* (*e-business*). *Web Server* selalu terhubung ke *internet* dan setiap *web server* akan dilengkapi dengan alamat unik yang disusun dengan serangkaian empat nomor antara 0 dan 255 yang dipisahkan oleh periode. Selain itu, *web server* memungkinkan penyedia *hosting* mengelola beberapa *domain* (pengguna) di *server* tunggal [33].

2.2.6 *Nginx*

Nginx atau biasa disebut "*Engine-x*" merupakan salah satu *web server* yang dapat digunakan untuk menjalankan sebuah situs. *Nginx* bersifat *open source* dan memiliki banyak fitur seperti HTTP *cache*, *load balancer*, *reverse proxy*, IMAP/POP3 *proxy server*. *Nginx* dapat menjalankan situs dengan mekanisme pertukaran data menggunakan protokol HTTP dan memiliki banyak dukungan pengguna [31].

Nginx dibuat oleh Igor Sysoev dan dirilis ke publik pada bulan Oktober 2004. Saat awal dirilis Igor meyakinkan publik bahwa *Nginx* dapat menjadi jawaban untuk mengatasi permasalahan yang ada pada saat itu yaitu performa *web server* jika memiliki koneksi aktif lebih dari 10.000 koneksi secara bersamaan. *Nginx* menawarkan penggunaan memori yang lebih rendah dibandingkan *web server* lainnya dan juga beberapa fitur seperti *reverse proxy*, IPv6, *load balancing*, *Fast*CGI *support*, *web sockets*, *handling static files*, TLS/SSL [34].

Salah satu yang membuat *Nginx* menjadi sangat cepat adalah jenis arsitektur *Nginx*. Jika di bandingkan dengan *Apache* yang *process based*, *Nginx* menjadi jauh lebih unggul karena *event-based* nya. Sehingga mampu memanfaatkan seminimal mungkin *thread* untuk memproses *request* dari *user*, sehingga akhirnya memori yang terpakai oleh *Nginx* menjadi minimal. Karena memori yang dipakai sangat kecil, maka hasilnya *server* menjadi ringan dan jauh lebih responsif (memiliki respon sangat cepat) [35].

2.2.7 Siege

Tools Siege merupakan tes regresi *open source* dan utilitas *benchmark* yang dikembangkan oleh Jeffrey Fulmer. Siege dapat menguji satu URL dengan jumlah pengguna yang ditentukan pengguna simulasi, atau dapat membaca banyak URL ke dalam memori dan menekankannya secara bersamaan. Program melaporkan total jumlah *bit* yang direkam, *byte* yang ditransfer, waktu respon, konkurensi, dan status pengembalian. Siege mendukung protokol HTTP/1.0 dan 1.1, HTTPS, FTP, GET dan POST *directives*, *cookie*, pencatatan transaksi, dan otentikasi dasar. Fitur-fiturnya dapat dikonfigurasi berdasarkan per pengguna [36].

Siege ditulis pada GNU/*Linux* dan telah berhasil di-*porting* ke AIX, BSD, HP-UX dan Solaris. Siege harus dikompilasi pada sebagian besar varian *System* V UNIX dan pada sebagian besar sistem BSD yang lebih baru. Karena Siege mengandalkan fitur POSIX.1b yang tidak didukung oleh *Microsoft*, maka Siege tidak akan berjalan di *Windows*. Tentu saja *user* dapat menggunakan Siege untuk menguji *server Windows* [37].

2.2.8 Parameter *Benchmark*

A. *Response Time*

Response time merupakan waktu rata-rata yang diperlukan untuk menanggapi setiap permintaan pengguna yang disimulasikan. Waktu respon didapatkan dari total waktu dibagi dengan jumlah respon dengan satuan *second* atau detik, seperti pada persamaan 2.1 [37].

$$Response\ Time\ = \frac{\text{total waktu (s)}}{\text{jumlah respon (n respon)}} = (second) \qquad (2.1)$$

B. *Transaction Rate*

Transaction rate adalah jumlah rata-rata transaksi yang dapat ditangani *server* per detik. *Transaction rate* didapatkan dari total transaksi dibagi dengan waktu yang telah berlalu dengan satuan *transaction/second* atau *trans/sec*, seperti pada persamaan 2.2 [37].

$$Transaction\ Rate\ = \frac{\text{total transaksi (n transaksi)}}{\text{jumlah waktu (s)}} = (trans/sec) \qquad (2.2)$$

C. *Throughput*

Throughput merupakan nilai rata-rata pengiriman yang sukses dalam interval waktu tertentu yang diukur dalam satuan *bit per second* (bps atau bit/s) [19]. *Throughput* yaitu jumlah rata-rata *byte* yang ditransfer setiap detik dari *server* ke semua pengguna yang disimulasikan [37]. Kecepatan transmisi data yang dapat dipertahankan *Nginx* saat memproses permintaan HTTP untuk konten statis selama periode waktu tertentu. Konten statis adalah *file* 1-MB, karena permintaan untuk *file* yang lebih besar meningkatkan *throughput* sistem secara keseluruhan [38]. Persamaan 2.3 merupakan cara menentukan *throughput*.

$$Throughput = \frac{\text{jumlah paket data yang dikirim (bit)}}{\text{waktu pengiriman paket (s)}} x10^6 = (Mbps) \qquad (2.3)$$

D. *CPU Usage*

CPU merupakan jenis sumber daya yang memiliki unit dasar. CPU mewakili pemrosesan komputasi dan ditentukan dalam unit CPU Kubernetes. Untuk beban kerja *Linux, user* dapat menentukan sumber daya halaman yang sangat besar. Halaman besar adalah fitur khusus *Linux* di mana *kernel node* mengalokasikan blok memori yang jauh lebih besar dari ukuran halaman *default* [39]. CPU *usage* dilakukan untuk mendapatkan informasi tentang beban proses yang diterima setiap *web server* [19].

Batas dan permintaan untuk sumber daya CPU diukur dalam CPU *units* dalam satuan % (persen). Di Kubernetes, 1 unit CPU setara dengan 1 CPU *core* fisik, atau 1 *virtual core*, tergantung pada apakah *node* tersebut adalah *host* fisik atau mesin *virtual* yang berjalan di dalam mesin fisik [39].