BAB 2
KAJIAN PUSTAKA DAN DASAR TEORI

2.1 KAJIAN PUSTAKA

Penelitian terkait Analisa performa *cluster Kubernetes* dengan CNI (*Container Network Interface*) *calico* didukung oleh beberapa penelitian sebelumnya. Pada penelitian [1] dilakukan penelitian tentang analisis performa dari teknologi pada *container network* di *environtments cloud* menggunakan metode perbandingan *network* pada *cluster kubernetes* yang berbeda *host* dan *container* yang berbeda antar *host cluster Kubernetes* oleh , dengan menggunakan beberepa *plugin container*, *container network interface* di setiap *cluster kubernetes* , dan setelah melakukan pengujian performa dari beberapa *plugin container network interface* dapat di simpulkan bahwa *plugin* yang memiliki performa yang baik dari sisi koneksi antara *container* dan *host* pada *cluster Kubernetes* adalah *plugin container network interface calico*.

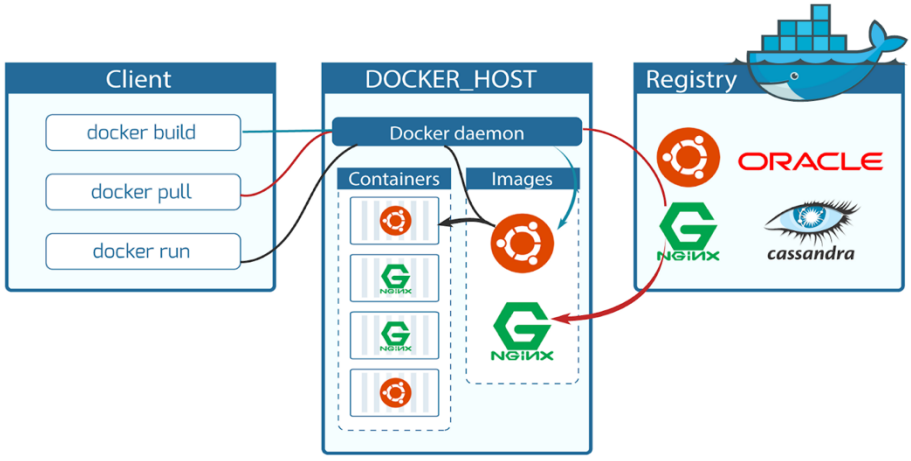
Selanjutnya [2] analisa dari penggunaan *container network interface plugin kuryr* dan *flannel* di implementasikan dengan *virtual private cloud* yaitu *open*stack, yang mana pada penelitian ini bahwa *plugin container network interface flannel* memiliki *overhead* lebih tinggi dari *kuryr*, dengan perbandingan nilai *throughput* dari *kuryr* lebih besar dari *flannel*.

Selanjutnya pada penelitian [3] dilakukan analisa dilakukan terhadap beberapa *plugin Container Network Interface* (CNI) yang umum digunakan dalam lingkungan *Kubernetes*, yaitu *Flannel*, *Calico*, dan *Weave*. Penelitian ini melibatkan konfigurasi tiga node dalam sebuah cluster *Kubernetes*. Evaluasi dilakukan dengan fokus pada kinerja jaringan, khususnya *latency* dan *throughput* TCP, dengan pengaturan *Maximum Transmission Unit* (MTU) yang berbeda pada setiap node. *Flanel*, *Calico*, dan *Weave* dipilih sebagai objek analisa karena ketiganya mewakili solusi jaringan CNI yang berbeda dalam pengelolaan komunikasi antar *container* di dalam *cluster Kubernetes*. Dalam eksperimen ini, setiap *node* akan dikonfigurasi dengan MTU yang berbeda-beda untuk mengobservasi pengaruh perubahan ukuran paket data terhadap kinerja jaringan.

2.2 DASAR TEORI

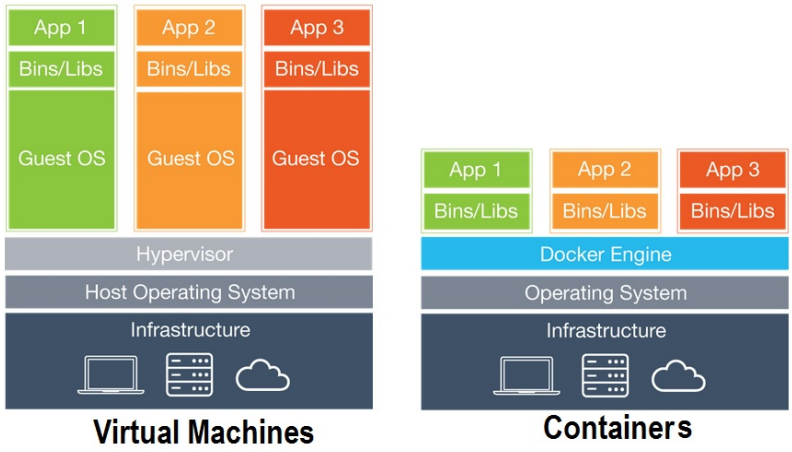
2.2.1 *Container*

Container merupakan aplikasi *open-source* yang dapat berdiri sendiri dan memiliki kemampuan untuk mengeksekusi kode, *runtime*, *tools sistem*, pustaka sistem, dan pengaturan yang diperlukan untuk menjalankan aplikasi. *Container* bekerja dengan cara membagikan sumber daya *kernel* dengan sistem operasi *host*, sehingga aplikasi atau layanan, dependensi, dan konfigurasi dapat dikemas menjadi sebuah gambar *container* yang dapat dipindahkan dengan mudah dan cepat. Seperti halnya sebuah wadah, *container* juga dapat mengisolasi aplikasi dari lingkungan sekitarnya dan memungkinkan penggunaan beberapa aplikasi pada satu mesin. *Container* dapat beroperasi pada semua distribusi *linux*, *microsoft windows*, dan infrastruktur lainnya, termasuk *Virtual machine, bare-metal virtualization, dan cloud.* Dalam hal ini, *container* dapat dianggap sebagai alternatif yang lebih efisien daripada *Virtual Machine* karena menggunakan lebih sedikit sumber daya komputasi dan RAM. Dalam hal manajemen *container,* terdapat alat yang digunakan untuk menggabungkan beberapa *host container* menjadi sebuah *cluster*, yang disebut dengan *container orchestrator.* Selain berfungsi sebagai alat manajemen *container.* Gambar 2.1 merupakan alur oprasional kerja *docker.*



Gambar 2. 1 Oprasional *Docker*

Container image adalah file yang berisi kode *executable*, pustaka sistem, *tools sistem,* dan dependensi yang dibutuhkan untuk menjalankan aplikasi. *Container image* merupakan suatu *snapshot* dari aplikasi dan lingkungannya, sehingga dapat dianggap sebagai sebuah paket lengkap yang terdiri dari semua yang diperlukan untuk menjalankan aplikasi secara terisolasi. Dalam hal ini, sebuah *container image* berbagi *kernel* dengan sistem operasi *host*, tetapi berjalan dalam lingkungan yang terisolasi dari *container image* lainnya. Hal ini memungkinkan *container image* untuk berjalan secara mandiri dan terpisah dari aplikasi atau lingkungan lainnya, sehingga dapat dianggap sebagai unit dasar dari sebuah *container*, *container image* dapat digunakan untuk menjalankan aplikasi pada berbagai platform seperti *Virtual Machine, Bare-metal Virtualization*, dan *Cloud*. Dengan menggunakan *container image*, pengguna dapat mempercepat proses deployment dan memudahkan proses migrasi aplikasi dari satu platform ke platform lainnya. Gambar. 2.2 merupakan perbedaan dari struktur *container* dan *virtual machine* .



Gambar 2. 2 Perbedaan *Virtual Machine* Dan *Contianer*

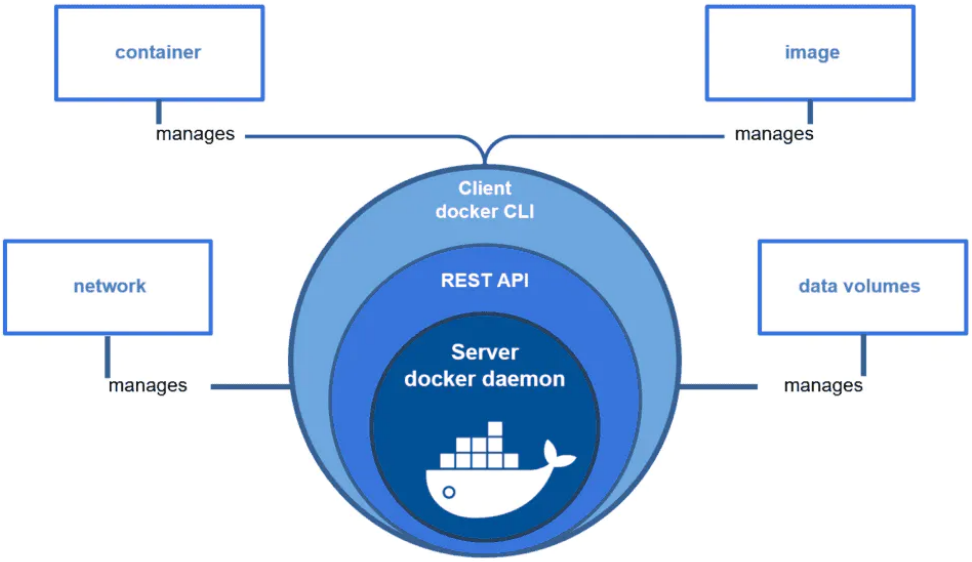
Container dapat menjalankan paket kode dan dependensi pada satu *layer* aplikasi, yang memungkinkan beberapa *container* untuk berjalan pada satu mesin dan berbagi *kernel* sistem operasi. Setiap *container* berjalan secara terisolasi satu sama lain, sehingga memungkinkan pengguna untuk menjalankan beberapa aplikasi secara bersamaan. Dalam hal ini, *container* juga dapat mengoptimalkan penggunaan ruang penyimpanan karena hanya perlu

menyimpan dependensi yang dibutuhkan untuk menjalankan aplikasi pada *layer* yang sama *container* adalah sebuah teknologi *virtual*isasi yang memungkinkan sebuah aplikasi dijalankan secara terisolasi di atas sistem operasi *host* tanpa harus menggunakan *virtual machine*. Setiap *container* memiliki lingkungan yang terisolasi sendiri, sehingga masing-masing *container* dapat memiliki aplikasi atau versi sistem operasi yang berbeda. Teknologi *container* menjadi populer seiring dengan perkembangan metode pengembangan aplikasi modern yang membutuhkan *deployment* yang cepat dan konsisten. Pada dasarnya, *container* menggunakan sistem operasi *host* sebagai basisnya dan berbagi *kernel* sistem operasi tersebut. Hal ini berbeda dengan teknologi *virtual*isasi lainnya seperti *hypervisor* yang mengizinkan *multiple* OS berjalan di atas *physical hardware*. Dengan demikian, *container* dapat dijalankan dengan lebih efisien dan ringan karena tidak perlu menambah beban *kernel virtual* .

Di sisi lain, *virtual machine* adalah sebuah abstraksi perangkat keras yang memungkinkan satu *server* diubah menjadi beberapa *server*. Satu *virtual machine* memiliki satu sistem operasi dan beberapa aplikasi yang berjalan di dalamnya, serta menggunakan banyak ruang penyimpanan untuk menyimpan *library* dan konfigurasi yang dibutuhkan. Dalam hal ini, *Virtual Machine* memungkinkan pengguna untuk menjalankan beberapa sistem operasi yang berbeda pada satu mesin fisik, tetapi memerlukan ruang penyimpanan yang lebih besar karena setiap *virtual machine* memiliki sistem operasi, *library*, dan konfigurasi yang terpisah. Oleh karena itu, *virtual machine* umumnya memerlukan sumber daya komputasi yang lebih besar dan lebih mahal dibandingkan dengan *container* [4]

2.2.2 *Docker*

Docker adalah platform *container* yang digunakan untuk membangun dan *mendeploy* aplikasi. Sebagai *application container runtime, docker* berjalan secara *native* pada sistem operasi *linux,* tetapi juga tersedia untuk dijalankan pada sistem operasi *mac os* dan *microsoft windows*. Dengan *docker,* pengembang dapat membangun dan mengirimkan aplikasi dengan mudah dan konsisten, tanpa khawatir tentang perbedaan lingkungan antara mesin pengembangan dan produksi. Gambar 2.3 merupakan kompenen penting *docker* .



Gambar 2. 3 Komponen *Docker*

2.2.2.1 *Docker File*

Dockerfile adalah sebuah file teks yang berisi perintah-perintah untuk membangun sebuah *docker image*. Fungsinya adalah untuk mengotomatisasi proses pembuatan *docker image*. Setelah selesai dibangun, *docker image* tersebut dapat diunggah ke *docker* Hub, yaitu sebuah *repository registry* resmi untuk mempublikasikan *docker image* yang telah dibangun .

2.2.2.2 *Docker Networking*

Dalam *docker*, *container* menggunakan jaringan internal untuk berkomunikasi satu sama lain. Ada beberapa jenis jaringan pada *docker* yang digunakan untuk menghubungkan antar-*container* dengan karakteristik masing-masing, di antaranya :

2.2.2.3 *Bridge*

Bridge adalah *driver* jaringan *default* yang ada pada *docker*. Jenis jaringan ini umumnya digunakan untuk *container* standalone yang terisolasi dari jaringan *host* dan *container* lain. Ketika *container* dibuat, jaringan *Bridge* akan dibuat dan *container* akan terhubung ke jaringan tersebut secara otomatis.

2.2.2.4 *Host*

Host Networking memungkinkan *container* menggunakan jaringan *host* langsung tanpa isolasi. Ketika *container* dijalankan, *Host Networking* akan digunakan sehingga *container* tidak terisolasi dari jaringan *host* dan bisa berkomunikasi langsung dengan perangkat jaringan yang sama dengan *host*. Namun, *container* ini juga terkena risiko keamanan karena tidak terisolasi dari *host*.

2.2.2.5 *Overlay*

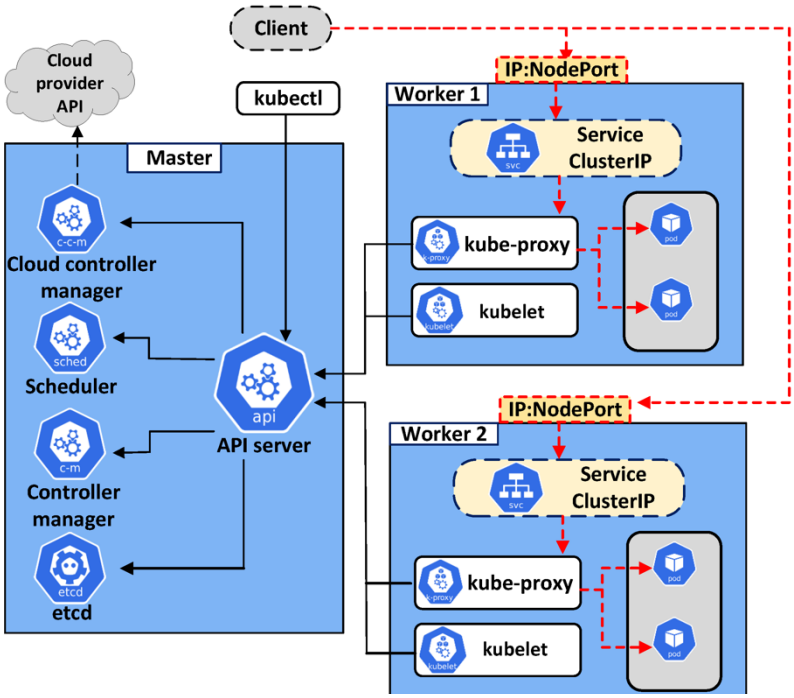
Overlay memungkinkan beberapa *daemon docker* terhubung dan berkomunikasi satu sama lain, yang membuat layanan di dalam *container* lebih mudah diatur dan dikelola. Jaringan *overlay* dapat menyelesaikan masalah kompleksitas pada jaringan yang ada pada *container* karena memungkinkan pengaturan dan manajemen jaringan secara fleksibel. Jaringan ini digunakan untuk mengaktifkan layanan *docker swarm* dan memungkinkan komunikasi antar-layanan dengan aman dan cepat. Jaringan *overlay* dapat menghubungkan beberapa layanan dan menjadikan layanan tersebut dapat saling berkomunikasi [5].

2.2.3 *Kubernetes*

Kubernetes merupakan platform *cluster open-source* yang digunakan untuk otomatisasi *deployment, scaling*, dan manajemen *container*. Dalam lingkungan *cloud*, *kubernetes* memungkinkan para pengembang dan operator IT untuk dengan mudah mengatur dan mengelola *container* aplikasi. *Kubernetes* memungkinkan pengguna untuk mengelola banyak aplikasi dan layanan dengan mudah, efisien, dan secara otomatis.

Salah satu keunggulan *Kubernetes* adalah portabilitasnya. *Kubernetes* dapat diaplikasikan pada *private* atau *public cloud*, bahkan dapat diaplikasikan pada *bare metal server*. Ini memungkinkan penggunauntuk mengelola lingkungan *cloud* mereka dengan lebih fleksibel, karena tidak terikat pada satu platform *cloud* saja. *Kubernetes* juga dapat diaplikasikan pada *multi-cloud*, yang memungkinkan pengguna untuk mengelola beberapa lingkungan *cloud* secara bersamaan dengan lebih mudah .

Selain itu, *kubernetes* menggunakan *resource hardware* yang dibutuhkan saja, sehingga penggunaan *resource* menjadi lebih optimal. *Kubernetes* memungkinkan pengguna untuk memanfaatkan *resource* secara efisien, karena hanya akan menggunakan sumber daya perangkat keras yang diperlukan. Hal ini akan meminimalkan pemborosan re*source* dan meningkatkan efisiensi penggunaan sumber daya. Gambar 2.4 merupakan arsitektur yang dimiliki *kubernetes* .



Gambar 2. 4 Arsitektur Kubernetes

Kubernetes terdiri dari beberapa komponen, yaitu komponen *master* dan komponen *node*. Komponen *master* terdiri dari beberapa komponen utama, yaitu:

2.2.3.1 *Kube Api Server*

Kubernetes memiliki Kube API *server* yang merupakan REST API yang digunakan untuk menangani permintaan dari pengguna. Dengan kube-api*server*, pengguna dapat berinteraksi dengan k*ubernetes* melalui API dan melakukan tindakan seperti membuat, memperbarui, dan menghapus objek di dalam *cluster kubernetes*.

2.2.3.2 *Etcd*

Etcd adalah penyimpanan *key-value* yang digunakan oleh *Kubernetes* untuk menyimpan data konfigurasi dan status dari objek-objek di dalam *cluster*. Etcd digunakan untuk menyimpan informasi penting seperti informasi konfigurasi, metadata, keadaan aplikasi, dan data lainnya yang diperlukan untuk menjalankan dan mengelola aplikasi di dalam *cluster Kubernetes*. Dengan etcd, informasi di dalam *cluster* dapat disimpan secara aman dan terorganisir, dan dapat diakses oleh berbagai komponen *kubernetes* seperti *kube-apiserver* dan kubelet.

2.2.3.3 *Kube Scheduler*

Scheduler dalam *Kubernetes* memiliki fungsi utama untuk menentukan *node worker* mana yang paling cocok untuk menjalankan suatu *container* baru. Dalam proses *scheduling*, *scheduler* akan mempertimbangkan beberapa faktor seperti kebutuhan re*source* dari *container*, ketersediaan *resource* pada *node worker*, serta *constraint* atau batasan yang mungkin ada pada *node worker* atau *pod* itu sendiri. Dengan begitu, *scheduler* akan dapat menentukan *node worker* mana yang paling optimal untuk *deploy* sebuah *pod* baru pada *cluster Kubernetes*.

2.2.3.4 *Kube Control Manager*

Control manager dalam *kubernetes* bertanggung jawab untuk mengelola setiap proses *controller* dalam sistem. Proses *controller* dijalankan secara terpisah untuk mengurangi kompleksitas, dan *container manager* dikompilasi menjadi satu *binary* yang berjalan sebagai satu proses. Dengan adanya *control manager*, pengguna dapat memonitor dan mengontrol setiap *controller* dengan lebih mudah dan efisien.

2.2.3.5 *Kubelet*

Kubelet merupakan komponen yang berjalan pada setiap *node* dalam *cluster kubernetes*. Tugas utama dari *kubelet* adalah untuk mengatur dan memantau *pods*, *container*, *image*, dan *volume* yang berjalan pada *node* tersebut. *Kubelet* juga bertanggung jawab untuk memastikan bahwa *container* berjalan sesuai dengan keinginan yang diatur oleh pengguna.

2.2.3.6 *Kube Proxy*

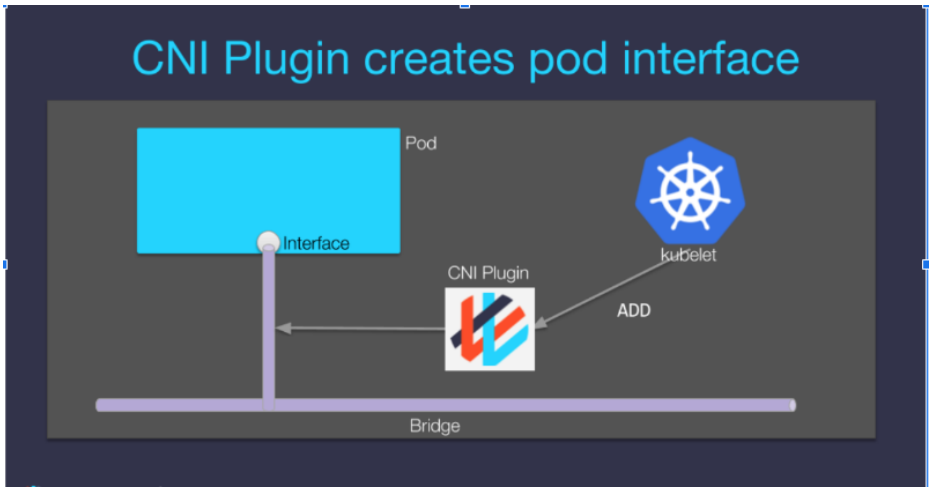
Kube-proxy berfungsi sebagai komponen jaringan pada setiap *node* dalam sebuah *cluster Kubernetes*. Tujuan dari *kube-proxy* adalah untuk menyediakan layanan jaringan yang memungkinkan *host* untuk menjalankan koneksi *forwarding*. *Kube-proxy* berjalan pada setiap *node* dalam *cluster Kubernetes*, sehingga setiap *node* dapat terhubung ke layanan jaringan dengan mudah.

2.2.3.7 *Container Runtime*

Container runtime adalah komponen penting dari sistem manajemen kontainer dan bertanggung jawab untuk mengelola dan menjalankan *container*. Dalam penelitian ini, *docker* dipilih sebagai perangkat lunak *container runtime* karena popularitasnya dan kemudahan penggunaannya [6].

2.2.4 *Container Network Interface Calico*

Calico adalah salah satu solusi *open-source* untuk *networking* pada *kubernetes* yang memungkinkan *container* dalam suatu *cluster* untuk berkomunikasi satu sama lain. *Calico* menggunakan teknologi *routing* dan pengalamatan IP (IP *routing*) yang memungkinkan *container* dapat terhubung dan berkomunikasi dengan *container* lainnya yang ada dalam jaringan.



Gambar 2. 5 Alur *Container Network Interface Calico*

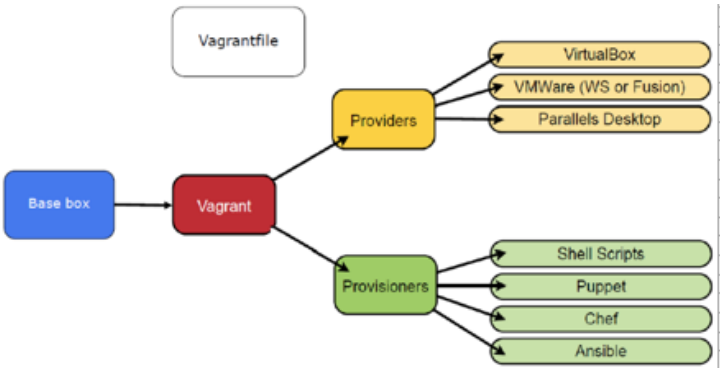
Gambar 2.5 merupakan alur kerja dari *container network interface calico, Calico* menggunakan arsitektur CNI (*Container Network Interface*) untuk mengelola jaringan antar *container* pada sebuah *cluster kubernetes*. CNI adalah standar *open-source* untuk menghubungkan antara *container* dan jaringan yang

lebih besar. *Calico* memanfaatkan CNI untuk mengelola *routing* dan pengalamatan IP[7].

Protocol untuk memungkinkan *container* dapat berkomunikasi dengan *container* lainnya, baik yang berada pada *node* yang sama atau berbeda dalam jaringan. Dengan demikian, *calico* dapat meningkatkan performa dan keamanan jaringan pada sebuah *cluster kubernetes* .

2.2.5 *Vagrant*

Vagrant adalah sebuah *software open-source* yang digunakan untuk membangun dan mengelola lingkungan pengembangan perangkat lunak secara *virtual*. *Vagrant* memudahkan pengembang dalam membuat lingkungan pengembangan yang dapat diulang dan mudah dipindahkan, serta mempercepat proses konfigurasi lingkungan pengembangan pada mesin local. Gambar 2.6 alur kerja dari *vagrant*.



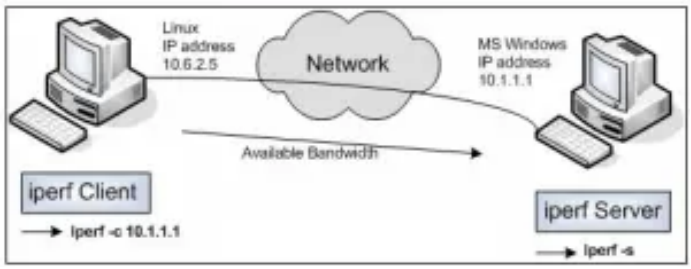
Gambar 2. 6 Alur Kerja *Vagrant*

Vagrant menggunakan konsep file konfigurasi yang dikenal sebagai "*Vagrantfile*" untuk menentukan lingkungan *virtual* yang dibutuhkan. *Vagrantfile* berisi informasi tentang konfigurasi lingkungan *virtual*, seperti jenis mesin *virtual* yang digunakan, memori dan CPU yang dialokasikan, serta cara menghubungkan mesin *virtual* ke jaringan

Vagrant mendukung beberapa platform mesin *virtual*, seperti *virtualbox*, *vmware*, dan *docker*, sehingga memungkinkan pengembang untuk memilih platform yang paling sesuai dengan kebutuhan mereka [8].

2.2.6 *Iperf*

Iperf adalah sebuah perangkat lunak *open source* yang digunakan untuk mengukur *throughput* atau *bandwidth* jaringan. *Iperf* dapat dijalankan di berbagai sistem operasi seperti *linux*, *windows*, dan mac os. *Iperf* mendukung berbagai protokol jaringan seperti TCP, UDP, dan SCTP, *bandwitdh*, *througput* serta dapat digunakan untuk menguji kinerja jaringan dalam berbagai skenario, seperti uji coba *transfer file,* uji coba streaming video, atau uji coba penggunaan VoIP. Gambar 2.7 merupakan alur kerja pada *iperf.*.



Gambar 2. 7 Alur Kerja *Iperf*

Iperf biasanya digunakan oleh para *administrator* jaringan untuk menguji kinerja jaringan dalam lingkungan yang kompleks, seperti jaringan yang terdiri dari banyak *server* dan *router*. Dengan menggunakan *Iperf*, *administrator* jaringan dapat mengetahui apakah jaringan berjalan dengan lancar dan dapat menentukan apakah terdapat masalah pada jaringan yang perlu diatasi[9].

2.2.7 *Throughput*

Throughput adalah ukuran kecepatan efektif *transfer data*, yang diukur dalam bit per detik (bps). Ini mencerminkan jumlah total paket yang berhasil tiba dengan sukses pada tujuan tertentu selama interval waktu yang ditentukan, yang kemudian dibagi oleh durasi interval waktu tersebut. Dengan kata lain, *throughput* menggambarkan seberapa efisien data dapat di*transfer* dalam jaringan dengan mengukur jumlah paket yang berhasil dikirim dan diterima selama periode waktu tertentu. Tabel 2.1 merupakan standarisasi *throughtput* dari tiphon.

Tabel 2. 1 Standarisasi *Throughput*

Kategori	Nilai (bps)
Sangat bagus	100
Bagus	75
Sedang	50
Buruk	<25

2.2.8 *Packet Loss*

Istilah "*Packet Loss*" merujuk pada parameter yang mengindikasikan keadaan di mana sejumlah paket data hilang dalam suatu jaringan. Hal ini dapat disebabkan oleh konflik (*collision*) dan kepadatan (*congestion*) dalam jaringan, dan kondisi ini berdampak pada semua aplikasi yang menggunakan jaringan tersebut. Dalam situasi tersebut, retransmisi paket-paket yang hilang akan mengurangi efisiensi keseluruhan jaringan, bahkan jika lebar pita (*bandwidth*) yang cukup tersedia untuk aplikasi-aplikasi tersebut. Apabila terjadi kongesti yang berlangsung dalam waktu yang cukup lama, *buffer* dalam jaringan akan terisi penuh, dan data baru tidak akan diterima.

Pentingnya pemahaman terhadap "*Packet Loss"* adalah untuk mengidentifikasi dan mengatasi masalah dalam jaringan yang mungkin menyebabkan hilangnya paket-paket data. Hal ini akan membantu meningkatkan kinerja dan kehandalan jaringan secara keseluruhan, serta memastikan pengiriman data yang efisien dan terpercaya dalam berbagai aplikasi yang bergantung pada jaringan tersebut. Pada Tabel 2.2 merupakan standarisasi *packet loss* dari tiphon

Tabel 2. 2 Standarisasi *Packet Loss*

Kategori	Nilai (%)
Sangat bagus	0
Bagus	3
Sedang	15
Buruk	25

2.2.9 *Delay*

Istilah "*Delay*" merujuk pada waktu yang diperlukan bagi data untuk melakukan perjalanan dari sumber ke tujuan. *Delay* dapat dipengaruhi oleh beberapa faktor, seperti jarak geografis antara sumber dan tujuan, jenis media fisik yang digunakan untuk mentransmisikan data, tingkat kongesti dalam jaringan, serta waktu yang diperlukan untuk proses pengolahan data.

Perbedaan dalam jarak antara sumber dan tujuan dapat mempengaruhi *delay* karena data harus melewati jalur fisik yang panjang. Selain itu, karakteristik media fisik seperti kecepatan transmisi dan kapasitas juga dapat mempengaruhi *delay.* Jika jaringan mengalami kongesti atau kepadatan lalu lintas yang tinggi, *delay* dapat meningkat karena paket-paket data harus antri sebelum dapat ditransmisikan. Terakhir, waktu yang diperlukan untuk memproses data di setiap titik dalam jaringan juga dapat menyumbang pada *delay* keseluruhan. Tabel 2.3 merupakan standarisasi *delay* dari tiphon.

Tabel 2. 3 Standarisasi *Delay*

Kategori	Nilai (ms)
Sangat bagus	<150
Bagus	150 – 300
Sedang	300 – 450
Buruk	>450

Pemahaman tentang konsep "*Delay*" ini penting untuk mengoptimalkan kinerja jaringan. Dengan meminimalkan *delay*, dapat meningkatkan responsivitas jaringan dan memastikan pengiriman data yang lebih cepat dan efisien. Selain itu, pemahaman terhadap faktor-faktor yang mempengaruhi *delay* juga memungkinkan perencanaan yang lebih baik dalam pengembangan infrastruktur jaringan untuk memenuhi kebutuhan aplikasi dan layanan yang semakin kompleks [10] .