BAB 2
KAJIAN PUSTAKA DAN DASAR TEORI

2.1 KAJIAN PUSTAKA

Penelitian [6] pada tahun 2022 yang berjudul "Perbandingan Kinerja *Ingress Controller* Pada Kubernetes Menggunakan *Traefik* dan *Nginx*" meneliti perbandingan kinerja dari *Traefik* dan *Nginx ingress controller* pada Kubernetes. Penelitian ini membandingkan *latency*, *throughput*, kecepatan *server* menyelesaikan permintaan, penggunaan CPU, dan penggunaan memori. Pengujian *server* yaitu *kube ingress traefik, kube ingress nginx,* dan *kube* tanpa *ingress* dengan jumlah *klien* 100, 300, dan 500 *klien* pada teknologi virtualisasi KVM. Hasilnya menunjukkan bahwa *ingress controller Traefik* memiliki kinerja yang baik pada parameter *latency*, kecepatan menangani *request*, serta penggunaan CPU. Sedangkan *ingress controller Nginx* unggul pada parameter *throughput* dan penggunaan memori.

Penelitian [7] pada tahun 2018 yang berjudul "Performance Evaluation and Comparison of Ingress Controllers on Kubernetes Cluster" meneliti mengenai evaluasi performansi dan perbandingan dari ingress controller pada Kubernetes cluster. Penelitian ini menguji 4 controller yaitu ingress controller Nginx, ingress controller Traefik, ingress controller Voyager, dan GCE L7 load balancer controller (GLBC). Pengujian dengan membandingkan parameter throughput dan response time dari keempat controller yang datanya diambil menggunakan tools Wrk2 dan Jmeter. Hasil yang diperoleh yaitu GCE L7 Load Balancer (GLBC) unggul pada parameter throughput dan response time dengan waktu respons paling singkat, diikuti oleh ingress controller Traefik, ingress controller Voyager, dan ingress controller Nginx. GCE L7 Load Balancer (GLBC) adalah opsi terbaik untuk cloud publik (platform cloud Google) dalam hal kenyamanannya karena sudah terinstal di Google Kubernetes Engine tanpa pemasangan Ingress Controller tambahan.

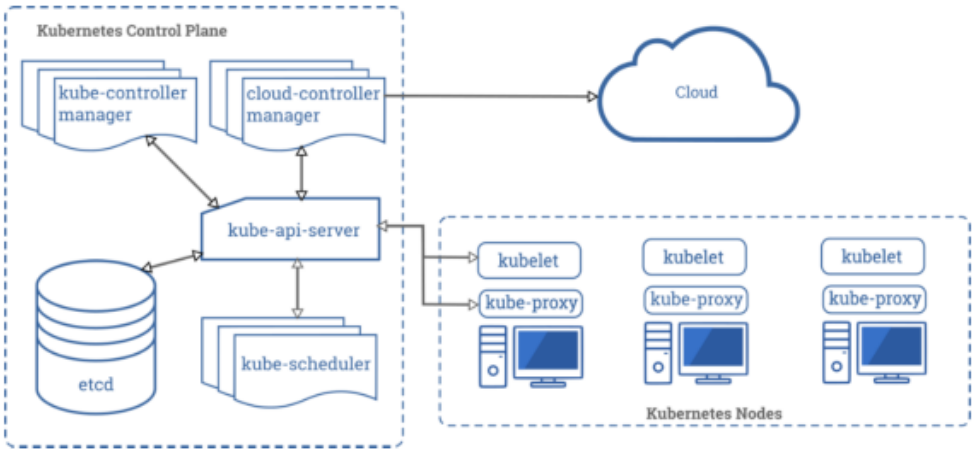
Penelitian [8] pada tahun 2021 yang berjudul "Deployment of Applications Using Nginx Ingress Controller" meneliti mengenai deployment (penyebaran) aplikasi menggunakan ingress controller Nginx. Penelitian ini menguji parameter skalabilitas kontainer yang diuji menggunakan Kubernetes dengan 1 master node

dan 2 worker node pada AWS Ubuntu bionic. Pengujian skalabilitas diambil menggunakan tools Prometheus Server dan Grafana untuk visualisasinya. Hasil yang didapatkan adalah Kubernetes adalah solusi untuk sebagian besar masalah terkait platform untuk mengotomatiskan penerapan, pengoperasian aplikasi kontainer, penskalaan, dan juga menyediakan infrastruktur sentris kontainer. Dengan Kontainer Orkertrasi, penskalaan atau penghapusan atau penambahan kontainer telah dilakukan menjadi lebih mudah.

2.2 DASAR TEORI

2.2.1 *Kubernetes*

Kubernetes adalah orkestrator *open source* untuk menerapkan aplikasi dalam *kontainer*. Awalnya dikembangkan oleh Google, terinspirasi oleh pengalaman selama satu dekade menerapkan sistem yang dapat diskalakan dan andal dalam wadah melalui API berorientasi aplikasi. Sejak diperkenalkan pada tahun 2014, Kubernetes telah berkembang menjadi salah satu proyek *open source* terbesar dan terpopuler di dunia. Ini telah menjadi API standar untuk membangun aplikasi *cloud-native*, hadir di hampir setiap *cloud* publik. Kubernetes adalah infrastruktur teruji untuk sistem terdistribusi yang cocok untuk *developer cloud-native* dari semua skala, mulai dari *cluster* komputer *Raspberry Pi* hingga gudang yang penuh dengan mesin terbaru. Kubernetes dapat menyediakan perangkat lunak yang diperlukan untuk berhasil membangun dan menerapkan sistem terdistribusi yang andal dan dapat diskalakan [9].



Gambar 2.1 Arsitektur Komponen Kubernetes [10]

Pada gambar 2.1 menunjukkan arsitektur komponen Kubernetes *cluster*. *Master node* dalam kluster Kubernetes ditetapkan sebagai *control plane*. Tugasnya adalah mengelola seluruh *cluster*, memastikan bahwa keadaan sebenarnya sesuai dengan keadaan yang diinginkan. Komponen kunci pertama di *control plane* adalah "*kube-apiserver*", yang berfungsi sebagai *gateway* ke API Kubernetes. Komponen "*kube-scheduler*" menangani pembuatan dan pendistribusian kontainer ke node yang tersedia. "*kube-controller-manager*" adalah komponen yang memastikan keadaan aktual sesuai dengan keadaan yang diinginkan. Komponen "*etcd*" adalah database tempat status *cluster* yang diinginkan disimpan. Komponen "*cloud-controller-manager*" adalah komponen opsional yang dapat digunakan untuk mengintegrasikan klaster dengan API vendor *cloud* [10].

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*.

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*) [11] [12].

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* [13].

Komponen Kubernetes *Node* merupakan *worker node* yang menjadi tempat beban kerja sebenarnya terjadi. Kontainer yang diterapkan ke kluster Kubernetes berjalan dalam sebuah *Pod*. Setiap *worker node* memiliki *kontainer runtime* yang terpasang, sehingga memungkinkan untuk menjalankan beban kerja kontainer. Kubernetes juga memiliki komponen yang disebut kubelet, yang digambarkan oleh Kubernetes sebagai agen yang berjalan di setiap *node* dalam kluster. Kubelet memastikan bahwa kontainer berjalan di dalam Pod. Komponen terakhir dalam *worker node* adalah kube-proxy yang bertanggung jawab untuk jaringan [10]. Kubernetes memberikan banyak manfaat dan *value* dalam penerapannya, antara lain :

1) *Workload Scheduling*

Kubernetes adalah orkestrator kontainer yang memiliki tujuan utama untuk memulai aplikasi berbasis kontainer, yang disebut *Pod* pada *Node* dalam sebuah *Cluster*. Tugas Kubernetes adalah untuk menemukan tempat yang paling tepat untuk menjalankan *Pod* di *Cluster*. Saat menjadwalkan *Pod* pada *Node*, perhatian utama adalah menentukan apakah sebuah *Node* memiliki sumber daya CPU dan memori yang cukup untuk menjalankan beban kerja yang ditetapkan.

2) *Managing State*

Saat kode diterapkan ke Kubernetes, menentukan beban kerja yang perlu dijalankan, Kubernetes bertanggung jawab untuk memulai *Pod* dan sumber daya lain di Klaster dan menjaga Klaster dalam status yang diinginkan. Jika keadaan berjalan Cluster menyimpang dari keadaan yang diinginkan, Kubernetes akan mencoba mengubah keadaan berjalan Cluster untuk mendapatkan keadaan berjalan dari Cluster kembali ke keadaan yang diinginkan yang ditentukan.

3) *Consistent Deployment*

Menerapkan (*deploy*) aplikasi dengan kode memungkinkan proses berulang. Kode yang mendefinisikan penerapan adalah artefak konfigurasi dan dapat ditempatkan di kontrol sumber. Anda juga dapat menggunakan kode ini untuk menerapkan sistem yang identik di lingkungan tingkat bawah seperti lingkungan pengembangan atau bahkan antara sistem lokal dan *cloud*. Lebih lanjut tentang ini di bagian selanjutnya di Kubernetes API.

4) Kecepatan

Kubernetes memungkinkan penerapan yang cepat dan terkontrol, memulai Pod dalam klaster dengan cepat. Selain itu, di Kubernetes, aplikasi dapat diskalakan dengan cepat. Memperluas jumlah Pod yang mendukung aplikasi bisa sesederhana mengubah baris kode, dan ini bisa memakan waktu hanya beberapa detik.

5) Abstraksi infrastruktur

Kubernetes API menyediakan abstraksi atau pembungkus di sekitar sumber daya yang tersedia di sebuah Cluster. Saat menerapkan aplikasi, ada sedikit fokus pada infrastruktur dan lebih pada bagaimana aplikasi didefinisikan dan disebarkan serta menghabiskan sumber daya Cluster. Kode yang digunakan untuk penerapan akan menjelaskan bagaimana tampilan Deployment, dan Cluster akan mewujudkannya. Jika aplikasi membutuhkan sumber daya seperti alamat IP publik

atau penyimpanan, itu menjadi bagian dari Deployment, dan Cluster akan menyediakan sumber daya ini untuk penggunaan aplikasi.

6) *Persistent Service Endpoint*

Kubernetes menyediakan IP persisten dan penamaan DNS untuk aplikasi yang diterapkan di Cluster. Karena Pod dapat datang dan pergi karena operasi penskalaan atau bereaksi terhadap kejadian kegagalan, Kubernetes menyediakan abstraksi jaringan ini untuk mengakses aplikasi ini. Bergantung pada jenis Service yang digunakan, beban layanan menyeimbangkan lalu lintas aplikasi ke Pod yang mendukung aplikasi tersebut. Saat Pod dibuat dan dihancurkan, baik berdasarkan operasi penskalaan atau sebagai respons terhadap kegagalan dalam Cluster, Kubernetes secara otomatis memperbarui informasi di mana Pod menyediakan layanan aplikasi [14].

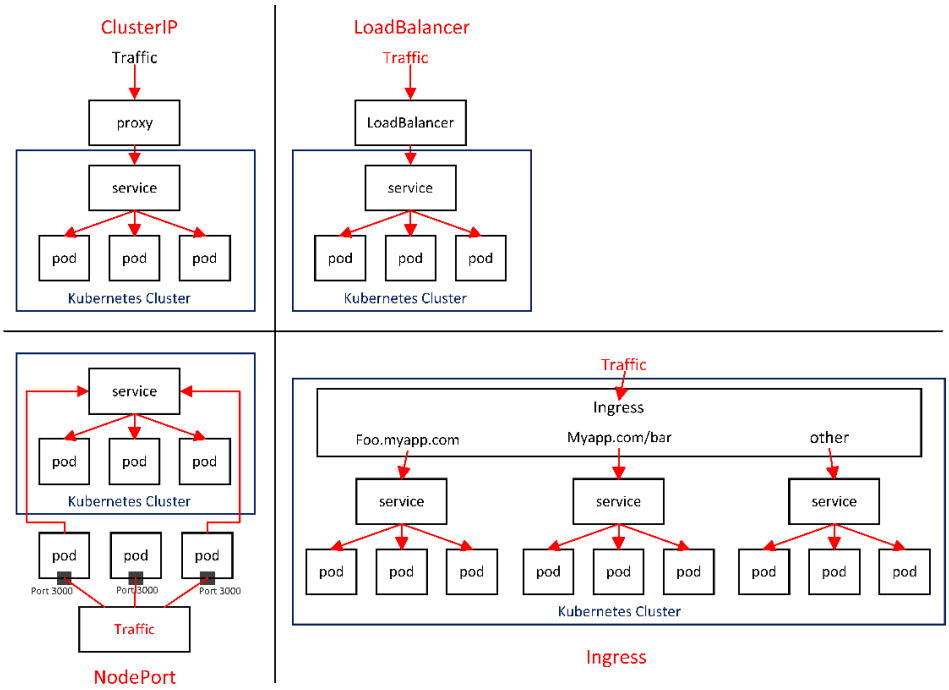
2.2.2 *Service Pada Kubernetes*

Service adalah metode untuk mengekspos aplikasi jaringan yang berjalan sebagai satu atau lebih *Pod* di klaster Kubernetes. Tujuan utama *service* di Kubernetes adalah pengguna tidak perlu memodifikasi aplikasi yang ada untuk menggunakan mekanisme penemuan *service* yang tidak dikenal. Dengan *service*, pengguna dapat menjalankan kode di *Pod*, baik itu kode yang didesain untuk *cloud-native*, atau aplikasi lama yang telah dikontainerisasi. *Service* dapat digunakan untuk membuat kumpulan Pod tersedia di jaringan sehingga klien dapat berinteraksi dengan *service* [15].

Service Kubernetes dapat dari berbagai jenis seperti yang diilustrasikan pada Gambar 2.2. Jenis service default disebut "IP *Cluster*". Service ini hanya dapat diakses dari dalam *cluster* Kubernetes, tidak bisa diakses dari luar cluster. IP Cluster menggunakan *proxy* untuk mengakses servicenya. Jenis *service* "*Node Port*" dibangun di atas *service Cluster* IP dan mengekspos *service* pada *port* yang sama dari setiap node *cluster*. *NodePort* tidak memerlukan konfigurasi, dan hanya merutekan lalu lintas pada *port* acak di *host* ke *port* acak di kontainer. Terakhir yaitu jenis *service* "*Load Balancer*" yang mengekspos *service* secara eksternal hanya ketika klaster berjalan di *cloud* publik. *LoadBalancer* akan menjaga koneksi

tetap terbuka untuk *pod* yang sedang aktif, dan menutup koneksi untuk yang sedang *down* [16] [17].

Kubernetes juga menyediakan cara lain yang disebut "*Ingress*" untuk mengakses *service* dari luar *cluster* Kubernetes. *Ingress* adalah kumpulan aturan untuk koneksi masuk untuk mencapai *service* tertentu di *cluster* yang didefinisikan sebagai *backend* untuk ingress. Agar *ingress* berfungsi, *ingress controller* harus diimplementasikan di *cluster* Kubernetes. Namun *ingress controller* bukan bagian dari Kubernetes. Terdapat berbagai macam *ingress controller* yang memiliki karakteristik dan kemampuan yang berbeda, seperti Kong, Istio, Nginx, Traefik, dan lainnya [16] [17].



Gambar 2.2 Arsitektur *Service Type* Kubernetes [18]

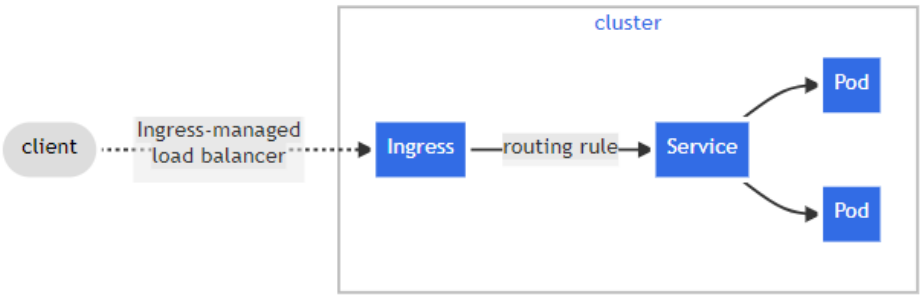
2.2.3 *Ingress*

Kubernetes pada dasarnya adalah orkestrasi/*scheduler* dan *server* API yang mendukung konfigurasi status yang diinginkan. Biasanya, klien mengirimkan permintaan sumber daya YAML deklaratif ke *server* API Kubernetes, dan Kubernetes menyediakan sumber daya yang diminta. Untuk setiap jenis sumber daya yang didukung oleh *server* API Kubernetes, seperti *deployment* atau *service* (layanan), *controller* Kubernetes *default* bertindak dan menyediakan permintaan sumber daya yang dikirimkan. Satu-satunya pengecualian adalah tipe sumber daya

ingress. Objek *Ingress* menentukan aturan perutean lalu lintas (misalnya, load balancing, terminasi SSL, perutean berbasis jalur) dalam satu sumber daya untuk mengekspos beberapa layanan di luar cluster [19]. Objek API yang mengelola akses eksternal ke layanan di *cluster*, biasanya HTTP. Ingress mengekspos rute HTTP dan HTTPS dari luar cluster ke layanan di dalam *cluster*. Perutean lalu lintas dikontrol oleh aturan yang ditentukan pada sumber daya *Ingress* [20].

Ingress adalah kumpulan aturan yang memungkinkan koneksi masuk mencapai titik akhir yang ditentukan oleh *backend*. *Ingress* dapat dikonfigurasi untuk memberikan layanan URL yang dapat dijangkau secara eksternal, trafik *load balance*, terminasi SSL, menawarkan *hosting* virtual berbasis nama, dll. *Ingress controller* bertanggung jawab untuk menyediakan permintaan *Ingress* yang dikirimkan. Kubernetes tidak dikirimkan dengan *Ingress controller default*. Vendor pihak ketiga (*third-party*) menyediakan implementasi yang banyak digunakan contohnya *nginx-ingress-controller* berdasarkan Nginx, tetapi *end user* bebas menggunakan *Ingress controller* yang mereka butuhkan [19].

Ingress controller bekerja pada lapisan 7 model OSI jaringan. Ingress controller menyediakan trafik routing yang dapat dikonfigurasi, terminasi *Transport Layer Security* (TLS), dan proksi terbalik. *Ingress Controller* memiliki aturan dan rute masuk ke service Kubernetes. Penggunaan umum dari *Ingress Controller* adalah kemampuan untuk merutekan dari satu alamat IP publik ke beberapa layanan dalam kluster Kubernetes. *Ingress controller* yang paling umum di Kubernetes adalah *Ingress controller Nginx* [21]. *Ingress controller* memastikan setiap service hanya memiliki satu IP yang dapat diakses dari internet, dan lalu lintas yang ditujukan untuk IP ini dialihkan ke service yang benar oleh ingress controller [22].

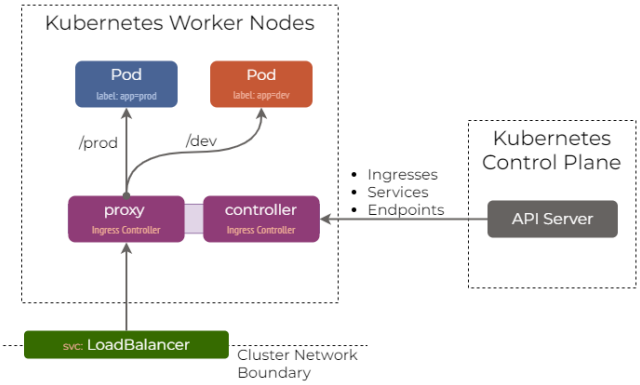


Gambar 2.3 Alur trafik Ingress [20]

Gambar 2.3 merupakan contoh sederhana di mana *Ingress* mengirimkan semua trafiknya ke satu *Service*. Permintaan berasal dari klien yang mencapai *ingress-managed load balancer*. Kemudian permintaan diproses oleh sumber daya *Ingress* berdasarkan awalan *service* yang ditentukan dalam *routing rule*. Kemudian permintaan dikirim ke *service* yang sebenarnya. Kemudian permintaan dikirim ke *pod backend* yang sebenarnya.

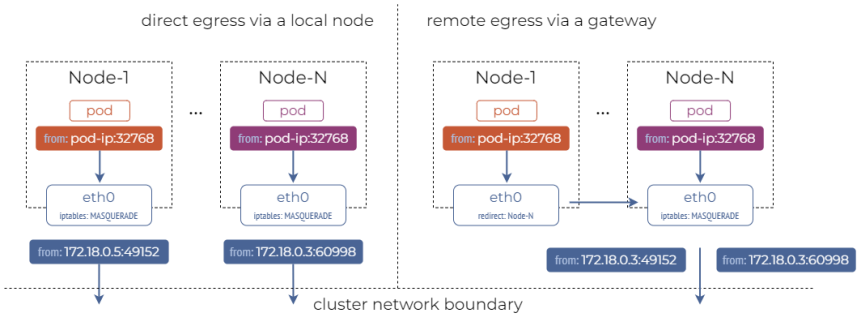
Sebuah *Ingress* dapat dikonfigurasi untuk memberikan *service* URL yang dapat dijangkau secara eksternal, trafik *load balance*, terminasi SSL/TLS, dan menawarkan *hosting virtual* berbasis nama. *Ingress controller* bertanggung jawab untuk memenuhi *Ingress*, biasanya dengan *load balancer*, meskipun mungkin juga mengonfigurasi *edge router* atau *frontend* tambahan untuk membantu menangani trafik. *Ingress* tidak mengekspos *port* atau protokol sembarangan. Mengekspos *service* selain HTTP dan HTTPS ke internet biasanya menggunakan *service* bertipe *NodePort* atau *LoadBalancer* [20].

Di Kubernetes, trafik *ingress* dan *egress* mengacu pada arah trafik jaringan dalam kaitannya dengan kluster Kubernetes. Trafik Ingress mengacu pada lalu lintas yang mengalir ke dalam klaster, dari titik akhir eksternal ke pod. Trafik Ingress biasanya digunakan untuk permintaan HTTP atau HTTPS yang masuk ke kluster Kubernetes, dan biasanya ditangani oleh Ingress controller. Ingress controller adalah komponen yang mendengarkan permintaan masuk dan mengarahkannya ke service yang sesuai berdasarkan URL atau informasi lainnya [23]. Pada Gambar 2.4 terlihat ingress memiliki beberapa komponen. Komponen *Proxy* merupakan komponen *control plane*, yang dikelola oleh *controller* (melalui API, *plugin*, atau file teks biasa), dapat diskalakan naik dan turun oleh Horizontal Pod *Autoscaler*. Komponen *Controller* terdapat proses yang berkomunikasi dengan *server* API dan mengumpulkan semua informasi yang diperlukan untuk berhasil menyediakan *proxy*-nya [24].



Gambar 2.4 Komponen pada *Ingress* [25]

Sedangkan Egress, mengacu pada trafik yang mengalir keluar dari sebuah *cluster*, dari *pod* ke titik akhir eksternal. Traffic *Egress* digunakan untuk mengakses layanan eksternal seperti *database*, API, dan layanan lain yang berjalan di luar *cluster* [23]. Pada Gambar 2.5 merupakan komponen *Egress*. Secara *default*, trafik yang meninggalkan *Pod* akan mengikuti *rute default Egress* dari *Node* dan akan disamarkan (SNAT'ed) ke alamat antarmuka *Egress*. Ini biasanya disediakan oleh opsi *plugin* CNI, misalnya opsi ipMasq dari *plugin bridge*, atau oleh agen terpisah seperti ip-masq-agent [24].



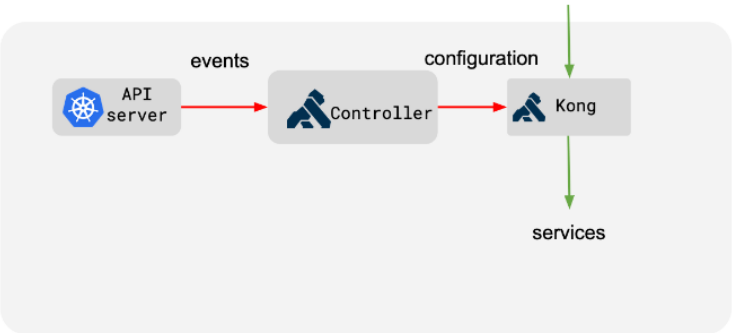
Gambar 2.5 Komponen pada *Egress* [24]

2.2.3.1 *Kong Ingress Controller*

Kubernetes *Ingress Controller* mengonfigurasi Kong *Gateway* menggunakan sumber daya *Ingress* atau API *Gateway* yang dibuat di dalam kluster Kubernetes. *Ingress Controller* Kubernetes terdiri dari dua komponen *high-level*, yaitu Kong (*proxy* inti yang menangani semua trafik) dan *Controller Manager* (serangkaian proses yang menyinkronkan konfigurasi dari Kubernetes ke Kong). Kong *Ingress Controller* Kubernetes melakukan lebih dari sekadar memproksi lalu lintas yang masuk ke kluster Kubernetes. Dimungkinkan untuk mengonfigurasi

plugin, penyeimbangan muatan, pemeriksaan kesehatan, dan memanfaatkan semua yang ditawarkan Kong dalam instalasi mandiri [26]. Kong *ingress controller* memiliki beberapa fitur, antara lain:

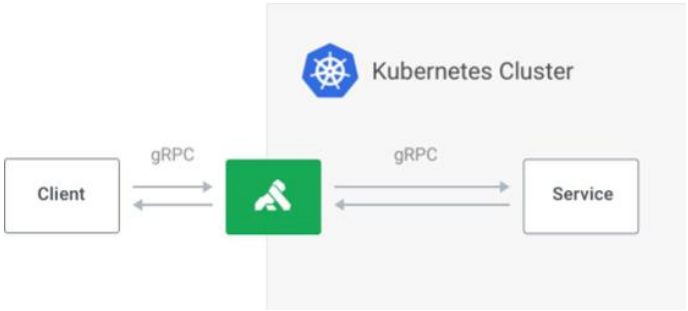
1) *Ingress routing* menggunakan sumber daya *Ingress* untuk mengonfigurasi Kong.
2) Pengingkatan manajemen API menggunakan *plugin* yang beragam untuk memantau, mengubah, dan melindungi lalu lintas pengguna.
3) *Native* gRPC mendukung trafik *Proxy* gRPC dan mendapatkan visibilitas ke dalamnya menggunakan *plugin* Kong.
4) Pemeriksaan kesehatan dan penyeimbangan *request load balancing* di seluruh *pod* pengguna dan mendukung pemeriksaan kesehatan aktif & pasif.
5) Transformasi permintaan/respons, menggunakan *plugin* untuk mengubah permintaan/respons dengan cepat.
6) Otentikasi melindungi *service*, menggunakan metode otentikasi pilihan pengguna.
7) Konfigurasi deklaratif untuk Kong *Configure* dari semua Kong menggunakan CRD di Kubernetes dan mengelola Kong secara deklaratif.
8) *Gateway Discovery*, memonitor Kong *Gateway* pengguna dan mendorong konfigurasi ke semua replika [27].



Gambar 2.6 Cara Kerja Kong *Ingress Controller* di Kubernetes [26]

Gambar 2.6 menujukkan cara kerja kong *ingress controller* di Kubernetes. *Controller Manager* mendengarkan perubahan yang terjadi di dalam *cluster* Kubernetes dan memperbarui Kong sebagai tanggapan atas perubahan tersebut untuk mem-*proxy* semua trafik dengan benar. Kong diperbarui secara dinamis untuk merespons perubahan seputar penskalaan, perubahan konfigurasi, kegagalan yang terjadi di dalam kluster Kubernetes [26]. API server yang terhubung ke

controller adalah proyek open-source yang dikelola oleh komunitas SIG-NETWORK. Gateway API adalah kumpulan sumber daya (resource) yang memodelkan jaringan layanan di Kubernetes. Sumber daya ini (GatewayClass, Gateway, HTTPRoute, TCPRoute, Service, dan lain-lain) bertujuan untuk mengembangkan jaringan layanan Kubernetes melalui interface ekspresif, dapat diperluas, dan interface role-oriented yang diimplementasikan oleh banyak vendor dan memiliki dukungan industri yang luas [28].



Gambar 2.7 Arsitektur Kong *Ingress Controller* [29]

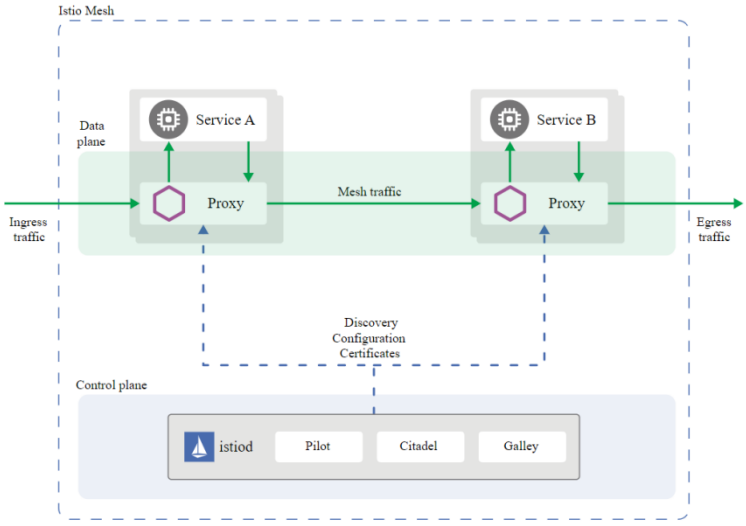
Sebagai aplikasi Kubernetes asli, Kong dipasang dan dikelola persis seperti sumber daya Kubernetes lainnya. Ini terintegrasi dengan baik dengan proyek-proyek CNCF lainnya dan secara otomatis memperbarui dirinya sendiri tanpa downtime sebagai respons terhadap peristiwa cluster seperti penyebaran pod. Ada juga ekosistem plugin yang bagus dan dukungan gRPC asli yang menghubungkan klien dengan Kong ingress controller di Kubernetes cluster [30]. Di gRPC, aplikasi klien dapat langsung memanggil metode pada aplikasi server di mesin lain seolah-olah itu adalah objek lokal, sehingga memudahkan user membuat aplikasi dan layanan terdistribusi. gRPC didasarkan pada ide untuk mendefinisikan layanan, menentukan metode yang dapat dipanggil dari jarak jauh dengan parameter dan jenis pengembaliannya (return). Di sisi server, server mengimplementasikan interface ini dan menjalankan server gRPC untuk menangani panggilan klien. Di sisi klien, klien memiliki rintisan (stub) yang menyediakan metode yang sama dengan server [31]. gRPC menyediakan plugin compiler buffer protokol yang menghasilkan kode sisi klien dan sisi server. pengguna gRPC biasanya memanggil API ini di sisi klien dan mengimplementasikan API yang sesuai di sisi server [32].

2.2.3.2 Istio *Ingress Controller*

Istio *Ingress* adalah solusi *service mesh* yang juga dapat berfungsi sebagai *ingress controller*, mengatur lalu lintas dari luar yang memasuki *kluster* Kubernetes. Istio memanfaatkan Envoy *proxy* yang ditempatkan dalam kontainer berdampingan bersama dengan setiap service yang terbuka di kluster. Envoy menawarkan trafik routing tingkat lanjut dan kemampuan pengamatan. Saat Istio digunakan sebagai solusi ingress, Istio berjalan sepenuhnya terpisah dari service, memotong semua trafik dan mengumpulkan metrik, melacak header, dan menerapkan otentikasi JSON Web Token (JWT) [22].

Istio mengimplementasikan Kubernetes Ingress Resources untuk mengekspos service dan membuatnya dapat diakses dari luar klaster. Di lingkungan Kubernetes, Kubernetes Ingress Resources memungkinkan pengguna menentukan layanan yang harus diekspos di luar klaster. Namun, spesifikasi Ingress Resources sangat minim, memungkinkan pengguna untuk menentukan hanya host, jalur, dan layanan pendukungnya. Terdapat beberapa batasan Istio ingress yang perlu diketahui:

1) Istio mendukung spesifikasi Kubernetes Ingress standar tanpa anotasi. Tidak ada dukungan untuk anotasi ingress.kubernetes.io dalam spesifikasi Ingress Resources. Anotasi apa pun selain kubernetes.io/ingress.class: istio akan diabaikan.
2) Ekspresi reguler di jalur tidak didukung.
3) Injeksi kesalahan pada Ingress tidak didukung [33].



Gambar 2.8 Arsitektur Istio *Ingress Controller* [34]

Pada Gambar 2.8 merupakan arsitektur dari Istio ingress controller. Istio service mesh secara logis dibagi menjadi data plane dan control plane. Data plane terdiri dari satu set *proxy* (*Envoy*) yang digunakan sebagai sidecars. Proxy ini memediasi dan mengontrol semua komunikasi jaringan antara layanan mikro (*microservice*). Envoy proxy juga mengumpulkan dan melaporkan telemetri pada semua lalu lintas mesh. Control plane dapat mengelola dan mengonfigurasi proxy untuk merutekan lalu lintas. Istio memiliki 2 (dua) komponen inti yaitu Envoy dan Istiod [34].

Pada komponen Envoy, Istio menggunakan versi perpanjangan dari proxy Envoy. Envoy adalah proxy berkinerja tinggi yang dikembangkan di C++ untuk memediasi semua lalu lintas masuk dan keluar untuk semua layanan di service mesh. Proksi Envoy adalah satu-satunya komponen Istio yang berinteraksi dengan lalu lintas data plane. Proxy Envoy dikerahkan sebagai sidecars ke layanan, secara logis menambah layanan dengan banyak fitur bawaan Envoy, misalnya:

a) *Dynamic service discovery*
b) *Load balancing*
c) TLS *termination*
d) HTTP/2 dan gRPC *proxies*
e) *Circuit breakers*
f) *Health checks*
g) *Staged rollouts* dengan %-*based traffic split*
h) *Fault injection*
i) *Rich metrics* [34].

Penyebaran sidecars ini memungkinkan Istio untuk melaksanakan keputusan *policy* dan mengekstraksi telemetri yang dapat dikirim ke sistem monitoring untuk memberikan informasi tentang perilaku seluruh jaringan. Model *proxy sidecars* juga memungkinkan *user* untuk menambahkan kemampuan Istio ke penerapan yang sudah ada tanpa mengharuskan *user* merancang ulang atau menulis ulang kode. Beberapa fitur dan tugas Istio yang diaktifkan oleh *proxy* Envoy meliputi:

a) Fitur *traffic control*: terapkan kontrol lalu lintas terperinci dengan aturan perutean untuk lalu lintas HTTP, gRPC, WebSocket, dan TCP.

b) Fitur *network resiliency* (ketahanan jaringan): percobaan ulang penyiapan, failover, pemutus sirkuit, dan injeksi kesalahan.

c) Fitur *security* dan *authentication*: menerapkan kebijakan keamanan, kontrol akses, dan pembatasan kecepatan yang ditentukan melalui API konfigurasi.

d) Model ekstensi *pluggable* berdasarkan WebAssembly yang memungkinkan penerapan kebijakan khusus dan pembuatan telemetri untuk lalu lintas mesh [34].

Komponen Istiod, Istiod menyediakan *service discovery*, konfigurasi, dan manajemen sertifikat. Istiod mengonversi aturan perutean tingkat tinggi yang mengontrol perilaku lalu lintas menjadi konfigurasi khusus Envoy, dan menyebarkannya ke sidecars saat *runtime*. Pilot mengabstraksi mekanisme *service discovery* khusus platform dan mensintesisnya ke dalam format standar yang dapat digunakan oleh *sidecars* mana pun yang sesuai dengan API Envoy. Istio dapat mendukung *discovery* untuk beberapa lingkungan seperti Kubernetes atau VM.

User dapat menggunakan *Traffic Management* API Istio untuk menginstruksikan Istiod menyempurnakan konfigurasi Envoy untuk melakukan kontrol yang lebih terperinci atas lalu lintas di *service mesh*. Keamanan Istiod memungkinkan autentikasi *service-to-service* dan *end-user* yang kuat dengan identitas bawaan dan manajemen kredensial. *User* dapat menggunakan Istio untuk memutakhirkan lalu lintas yang tidak terenkripsi di *service mesh*.

Dengan menggunakan Istio, operator dapat menerapkan *policy* berdasarkan identitas *service* daripada identitas jaringan lapisan 3 atau lapisan 4 yang relatif tidak stabil. Selain itu, *user* dapat menggunakan fitur otorisasi Istio untuk mengontrol siapa yang dapat mengakses layanan *user*. Istiod bertindak sebagai *Certificate Authority* (CA) dan menghasilkan sertifikat untuk memungkinkan komunikasi mTLS yang aman di *data plane* [34].

Pada tabel 2.1 dapat terlihat perbedaan dari *Ingress controller* Kong dan Istio dengan beberapa matrik yang menjadi perbandingan.

Tabel 2.1 Perbedaan *Ingress controller* Kong dan Istio [35]

Metrik	Kong	Istio
Protocols	http/https, http2, grpc, tcp	http/https, http2, grpc, tcp/udp, tcp+tls, mongo, mysql, redis
Based on	nginx	envoy
Traffic routing	host, path, method, header*	host, path, method, header (all with regex)
Namespace limitations	Specified namespace	All cluster or specified namespaces
Traffic distribution	canary, acl, blue-green, proxy caching*	canary, a/b, shadowing, http headers, acl, whitelist
Upstream probes	active, circuit breaker	retry, timeouts, active checks, circuit breakers
Load balancing	weighted-round-robin, sticky sessions	round-robin, sticky sessions, weighted-least-request, ring hash, maglev, random, limit conn, limit req
Authentication	Basic, HMAC, Key, LDAP, OAuth 2.0, PASETO, OpenID Connect**	Basic, mutual tls, OpenID, custom auth
Paid subscription	Ada	Tidak ada
GUI	Ada * **	Tidak ada
JWT (JSON Web Token) *validation*	Ada **	Ada
Basic DDoS *protection*	advanced rate limit*, rate limit, *request size limit*, *request termination*, *response rate limit*	acl, whitelist, rate limit
Requests tracing	Ada	Ada

Metrik	Kong	Istio
Config customization	Ada	Ada
WAF	Wallarm	ModSecurity

Keterangan :

* Hanya dalam versi berbayar.

** Modul tersedia

2.2.4 *Parameter Quality of Service (QoS)*

Quality of Service (QoS) adalah metode pengukuran kolektif dari kinerja layanan yang menentukan tingkat kepuasan pengguna layanan, yang dapat didefinisikan dalam berbagai parameter seperti *throughput, delay, jitter, packet loss,* dan lain-lain. Kata *Quality* didefinisikan sebagai totalitas karakteristik suatu entitas yang mendukung kemampuannya untuk memuaskan kebutuhan yang dinyatakan dan tersirat. *Internet Engineering Task Force* (IETF) menganggap QoS sebagai kemampuan untuk mengelompokkan lalu lintas atau membedakan antara jenis lalu lintas agar jaringan memperlakukan aliran lalu lintas tertentu secara berbeda dari yang lain. QoS mencakup kategorisasi layanan dan kinerja keseluruhan jaringan untuk setiap kategori [36].

2.2.4.1 *Delay*

Delay yaitu waktu yang dibutuhkan data/paket untuk menempuh jarak dari pengirim ke tujuan. Delay dapat dipengaruhi oleh jarak, media fisik, kongesti atau juga waktu proses yang lama [37]. Dalam pengukuran delay pada penelitian ini dilakukan menggunakan *tools* Wireshark, di mana satuannya yaitu *second*. Nilai *delay* yang baik dalam kinerja kontainer adalah semakin kecil *delay* maka akan semakin bagus kinerja dari kontainer tersebut. Rumus menghitung *delay* ditunjukkan pada persamaan 2.1 [38].

$$\text{Delay rata-rata} = \frac{total\ delay}{total\ paket\ yang\ diterima} \quad (2.1)$$

Tabel 2.2 Standar TIPHON Untuk *Delay* [39]

Kategori	Besar Delay (ms)	Indeks
Sangat bagus	< 150 ms	4
Bagus	150 s/d 300 ms	3
Sedang	300 s/d 450 ms	2
Jelek	> 450 ms	1

2.2.4.2 *Throughput*

Throughput merupakan nilai rata-rata pengiriman yang sukses dalam interval waktu tertentu yang diukur dalam satuan *bit per second* (bps atau bit/s) [38]. *Throughput* yaitu jumlah rata-rata *byte* yang ditransfer setiap detik dari *server* ke semua pengguna yang disimulasikan [40]. Dalam pengukuran *throughput* pada penelitian ini dilakukan menggunakan *tools Wireshark*, di mana satuannya yaitu *bytes*. Nilai *throughput* yang baik dalam kinerja *kontainer* adalah semakin besar *throughput* maka akan semakin bagus kinerja dari *kontainer* tersebut. Rumus menghitung *throughput* ditunjukkan pada persamaan 2.2 [38].

$$\text{Throughput} = \frac{jumlah\ data\ yang\ diterima\ (bit)}{waktu\ pengiriman\ data\ (second)} \quad (2.2)$$

Tabel 2.3 Standar TIPHON Untuk *Throughput* [39]

Kategori	Besar *Throughput* (bps)	Indeks
Sangat bagus	>2,1 Mbps	4
Bagus	1200 Kbps – 2,1 Mbps	3
Sedang	700 – 1200 Kbps	2
Buruk	338 – 700 Kbps	1
Sangat buruk	0 – 338 Kbps	0

2.2.4.3 *Memory*

Batasan dan permintaan memori diukur dalam *byte*. *User* Kubernetes dapat mengekspresikan memori sebagai bilangan bulat biasa atau sebagai angka titik tetap menggunakan salah satu sufiks kuantitas berikut: E, P, T, G, M, k. *User* juga dapat menggunakan persamaan pangkat dua: Ei, Pi, Ti, Gi, Mi, Ki. Misalnya, jika *user* meminta 400m memori, ini adalah permintaan 0,4 byte. *User* yang mengetik itu

mungkin bermaksud meminta 400 *mebibyte* (400Mi) atau 400 *megabyte* (400M). [41]. Dalam pengukuran *memory* pada penelitian ini dilakukan menggunakan *tools* Htop, di mana satuannya yaitu MB (*megabyte*). Rumus menghitung *memory* ditunjukkan pada persamaan 2.3 [42].

$$\text{Memory} = total\ memori - jumlah\ memori\ yang\ digunakan \quad (2.3)$$

2.2.4.4 *CPU*

Batasan dan permintaan sumber daya CPU diukur dalam CPU unit. Di Kubernetes, 1 unit CPU setara dengan 1 inti CPU fisik, atau 1 inti virtual, tergantung pada apakah node tersebut adalah host fisik atau mesin virtual yang berjalan di dalam mesin fisik. CPU *usage* dilakukan untuk mendapatkan informasi tentang beban proses yang diterima setiap *web server* [38]. Sumber daya CPU selalu ditentukan sebagai jumlah absolut sumber daya, tidak pernah sebagai jumlah relatif. Misalnya, 500m CPU mewakili jumlah daya komputasi yang kira-kira sama, terlepas dari apakah kontainer itu berjalan pada mesin *single-core*, *dual-core*, atau 48-*core* [41]. Dalam pengukuran CPU pada penelitian ini dilakukan menggunakan *tools* Htop, di mana satuannya yaitu persen (%). Jika persentase CPU mendekati 0 maka itu berarti CPU tidak memiliki banyak beban dan jika mendekati 100 untuk waktu yang lama, itu berarti CPU sedang memuat [42]. Rumus menghitung *memory* ditunjukkan pada persamaan 2.4 [42].

$$\text{CPU} = \frac{jumlah\ proses\ yang\ digunakan}{total\ proses}\ x\ 100\% \quad (2.4)$$