BAB 1

PENDAHULUAN

1.1 Latar Belakang Masalah

Continuous Integration dan *Continuous Deployment (CI/CD)* merupakan serangkaian praktik pengembangan perangkat lunak yang akan membantu pengembang untuk merilis perangkat lunak dengan handal, aman, dan efisien. Biasanya kedua praktik ini dijalankan secara berurutan, dimulai dari *CI* kemudian jika berhasil maka akan dilanjutkan dengan proses *CD. CI* biasanya terdiri atas praktik *testing* dan *build* perangkat lunak secara otomatis setelah repositori perangkat lunak terhubung dengan *server CI.* Sedangkan *CD* merupakan serangkaian proses perilisan perangkat lunak ke lingkungan produksi secara otomatis (Farid and Anugrah, 2021).

Untuk membantu pengembang dalam menerapkan praktik *CI/CD,* dibutuhkan *tools CI/CD* seperti *Gitlab CI, Jenkins, Travis CI, AWS CodePipeline, Go CD,* dan *Circle CI* (Toba et al., 2022). Jenkins merupakan salah satu jenis *tool CI/CD* yang cukup populer untuk saat ini dikarenakan dukungan komunitas yang cukup besar dan jumlah *plugin* yang cukup banyak. Sebagaimana *tool CI/CD* pada umumnya, *Jenkins* juga mendukung arsitektur multi-node, yaitu arsitektur yang memungkinkan *Jenkins* memiliki lebih dari satu mesin pengeksekusi tugas-tugas atau *jobs CI/CD,* sehingga tugas-tugas tersebut nantinya tidak hanya dieksekusi bertumpuk di satu node jenkins saja melainkan bisa terdistribusi secara merata ke *node-node jenkins* yang lain.

Dalam arsitektur *multi-node Jenkins*, terdapat *node* yang bertindak sebagai *master* dan *node* yang bertindak sebagai *slave*. *Node Jenkins* yang bertindak sebagai *master* bertugas untuk mendistribusikan *job-job CI/CD* ke *node-node* yang bertindak sebagai *slave*, dan *node slave* akan bertugas untuk mengeksekusi *job-job CI/CD* tersebut.

Secara umum, terdapat 2 jenis perangkat lunak jika ditinjau dari sisi pengemasan aplikasi pada saat proses produksi. Jenis pertama adalah perangkat lunak yang berbasis *container*, sedangkan jenis kedua adalah perangkat lunak yang tidak berbasis *container*. Banyak sekali manfaat dari perangkat lunak yang dikemas dalam bentuk *container*, salah satunya adalah adanya jaminan konsistensi kondisi lingkungan tempat perangkat lunak dijalankan, baik perangkat lunak tersebut dijalankan pada saat pengembangan, testing, hingga produksi. Salah satu jenis platform yang dapat digunakan untuk membangun hingga mengelola container adalah *Docker*. *Jenkins* mendukung implementasi *CI/CD* untuk perangkat lunak berbasis *container*, dikarenakan *Jenkins* dapat terintegrasi dengan *Docker* dengan sangat baik (Alpery and Ridha, 2021).

Pada saat praktik *CI* perangkat lunak berbasis *container*, biasanya terdapat proses *build image* yang fungsinya adalah untuk mengemas perangkat lunak tersebut menjadi sebuah *docker image*. Durasi waktu yang dibutuhkan untuk melakukan proses *build image* suatu perangkat lunak biasanya tergantung dari seberapa banyak *dependency* yang dibutuhkan oleh perangkat lunak tersebut. Artinya, semakin banyak *dependency* yang dibutuhkan maka semakin lama juga

proses *build image* yang akan dilakukan. *Docker* sendiri memiliki fitur *image layer caching* yang dapat dimanfaatkan untuk mengoptimasi durasi waktu proses *build image* suatu perangkat lunak. Namun secara *default* fitur ini hanya akan berjalan dengan baik pada suatu perangkat lunak jika sebelumnya perangkat lunak ini sudah pernah dilakukan proses *build image* di *node* yang sama. Sehingga apabila suatu perangkat lunak belum pernah sama sekali dilakukan proses *build image* di suatu *node jenkins*, maka fitur *image layer caching* untuk perangkat lunak ini tidak akan dapat berjalan di *node jenkins* tersebut walaupun sebenarnya perangkat lunak ini sudah pernah dilakukan proses *build image* di *node-node* yang lain. Sehingga dapat disimpulkan bahwa fitur *docker* image *layer caching default* tidak akan dapat berjalan secara efektif pada *jenkins* yang sudah menerapkan arsitektur *multi-node* dikarenakan bisa jadi *job build image* suatu perangkat lunak dieksekusi oleh *node* yang berbeda dengan *node* yang sebelumnya pernah melakukan *build image* perangkat lunak tersebut.

Oleh karena itu, akan dilakukan penelitian tentang proses optimasi durasi waktu *build image* pada *CI* terhadap perangkat lunak berbasis *container* menggunakan Jenkins dengan arsitektur *multi-node*. Penelitian ini memanfaatkan fitur *docker image layer caching* dengan melakukan penambahan kostumisasi dari sisi perintah dan infrastruktur.

1.2 Rumusan Masalah

Berdasarkan dari latar belakang masalah yang telah disebutkan, rumusan masalah penelitian ini adalah bagaimana mengoptimasi durasi waktu *build image*

suatu aplikasi berbasis *container* pada *multi-node Jenkins* dengan menggunakan metode *cache* ?

1.3 Ruang Lingkup

Penelitian ini hanya terbatas pada beberapa hal berikut :

1. Fokus pada proses *build image* yang merupakan bagian dari proses *CI* pada *Jenkins.*
2. Pengujian menggunakan 3 aplikasi yang sudah berbasis *container* dengan bahasa pemrograman *Javascript (Node JS), Python, dan Ruby.*
3. Pengujian dengan skenario penggunaan *Jenkins* sebelum dan sesudah optimasi.
4. Aplikasi yang menjadi bahan uji merupakan aplikasi dengan source code yang sederhana namun memiliki jumlah dependensi minimal 20 buah untuk menambah waktu *build time* aplikasi. Namun penelitian ini tidak fokus untuk membahas bagaimana *business logic* dari aplikasi.

1.4 Tujuan Penelitian

Adapun tujuan dari penelitian ini adalah agar durasi proses *build image* perangkat lunak berbasis *container* pada *multi-node Jenkins* dapat berjalan lebih cepat dengan memanfaatkan metode *cache*.

1.5 Manfaat Penelitian

Manfaat dari penelitian ini adalah sebagai berikut :

1. Dapat menjadi rujukan bagi *DevOps engineer* atau profesi terkait dalam penerapan metode cache untuk optimasi durasi waktu proses *build image* pada praktik *CI*.
2. Dapat dijadikan sebagai referensi untuk penelitian lain yang berkaitan dengan *Jenkins* ataupun *CI/CD* secara umum.