PERANCANGAN ARSITEKTUR *MICROSERVICE* MENGGUNAKAN *CONTAINER ORCHESTRATION* UNTUK MENDUKUNG *HIGH SCALABLE APPLICATION*
(Studi Kasus : APLIKASI SAPAWARGA)

TESIS

Disusun sebagai salah satu syarat untuk
memperoleh gelar Magister Komputer
dari Sekolah Tinggi Manajemen Informatika dan Komputer LIKMI

Oleh :

FIRMANSYAH

NPM : 2020210082

PROGRAM STUDI PASCASARJANA
MAGISTER SISTEM INFORMASI
SEKOLAH TINGGI MANAJEMEN INFORMATIKA DAN KOMPUTER
LIKMI
BANDUNG

2022

PERANCANGAN ARSITEKTUR *MICROSERVICE* MENGGUNAKAN *CONTAINER ORCHESTRATION* UNTUK MENDUKUNG *HIGH SCALABLE APPLICATION*
(Studi Kasus : APLIKASI SAPAWARGA)

Oleh :

FIRMANSYAH

NPM : 2020210082

Bandung, 3 - Maret - 2022

Mengetahui

Dr. Hery Heryanto, S.Kom., M.Kom.

Pembimbing

PROGRAM STUDI PASCASARJANA
MAGISTER SISTEM INFORMASI
SEKOLAH TINGGI MANAJEMEN INFORMATIKA DAN KOMPUTER
LIKMI
BANDUNG
2022

ABSTRAK

Aplikasi yang baik salah satunya memperhatikan unsur skalabilitas untuk mendukung bisnis yang berkelanjutan dari sebuah organisasi. Meningkatnya jumlah pengguna aplikasi harus dapat dikelola dengan baik sehingga tidak menimbulkan dampak pada aplikasi yang susah diakses atau bahkan mati. Untuk itu diperlukan perencanaan arsitektur sistem yang matang.

Sapawarga adalah aplikasi satu pintu untuk memudahkan RW di Jawa Barat berkomunikasi dengan pemerintah daerah melalui berbagai fitur akses layanan publik, aspirasi, informasi, berpartisipasi, dan berinteraksi dengan RW lainnya. Pada rilis tahap awal sapawarga menggunakan arsitektur *monolith*. Pada tahap berikutnya sapawarga direncanakan akan dapat digunakan oleh seluruh warga jawa barat yang jumlah penggunanya sangat banyak. Dengan berbagai kekurangan arsitektur *monolith*, perlu dilakukan perencanaan perubahan arsitektur aplikasi ke *microservice*.

Arsitektur *microservice* merupakan arsitektur aplikasi yang memecah *service* besar ke dalam *service* yang lebih kecil yang otomatis dapat membagi beban secara fitur. Arsitektur *microservice* hadir untuk membantu salah satunya masalah skalabilitas yang sulit dilakukan pada arsitektur *monolith*. Untuk melakukan perubahan dari arsitektur *monolith* ke *microservice* perlu dilakukan pemisahan dari sebuah *service* besar ke dalam *service-service* kecil yang fokus menangani fitur yang spesifik. *Domain Driven Design* (DDD) membuat sebuah pendekatan sebagai panduan memisahkan kompleksitas sebuah aplikasi ke dalam domain terpisah. *Service* yang telah dipisah berdasarkan *domain* kemudian dibungkus dengan menggunakan *container* yang bertujuan memudahkan konfigurasi aplikasi. Untuk mengelola *container* yang dimungkinkan banyak bisa menggunakan *container orchestration* .

Salah satu fungsi *container orchestration* juga dapat memungkinkan untuk melakukan auto *scaling* dengan menambah *container* guna membantu mendistribusikan beban aplikasi, juga dapat mengurangi *container* jika aplikasi mengalami penurunan beban. Dari hasil pengujian dapat disimpulkan dengan menggunakan arsitektur *microservice* menggunakan *container orchestration* dengan Kubernetes dapat mengatur dan meningkatkan skalabilitas sebuah aplikasi. Proses HPA telah terbukti dapat meningkatkan skalabilitas aplikasi di mana dilakukan percobaan dengan jumlah *request* sebanyak 10.000 kali, proses *autoscaling* berjalan dengan baik.

Kata kunci: *Microservices*, Arsitektur, *Container*, *Orchestration*, Docker, Kubernetes, *Scalable*, Sapawarga.

ABSTRACT

A good application, one of which pays attention is a scalability to support a sustainable business of an organization. The increasing number of application users must be managed properly so that it does not have an impact on applications that are difficult to access or even down. For that we need a mature system architecture planning.

Sapawarga is a one-stop application to make it easier for RWs in West Java to communicate with local governments through various features of accessing public services, aspirations, information, participating, and interacting with other RWs. In the early stages of release, greetings use a monolith architecture. In the next stage, greetings are planned to be used by all residents of West Java, which has a very large number of users. With the various shortcomings of the monolith architecture, it is necessary to plan changes to the application architecture to microservices.

Microservice architecture is an application architecture that breaks down large services into smaller services that can automatically share the load feature. Microservice architecture is here to help, one of which is scalability problems that are difficult to do on a monolith architecture. To make a change from monolith architecture to microservices, it is necessary to separate from a large service into smaller services that focus on handling specific features. Domain Driven Design (DDD) makes an approach as a guide to separate the complexity of an application into separate domains. Services that have been separated by domain are then wrapped using containers that aim to facilitate application configuration. To manage many possible containers, you can use container orchestration.

One of the container orchestration functions can also allow for auto scaling by adding containers to help distribute the application load, it can also reduce containers if the application experiences a decrease in load. From the test results, it can be concluded that using a microservice architecture using container orchestration with Kubernetes can manage and improve the scalability of an application. The HPA process has been proven to improve application scalability where experiments were carried out with 10,000 requests, the autoscaling process running well.

Keyword: *Microservices, Aritektur, Container, Orchestration,* Docker, Kubernetes, *Scalable,* Sapawarga.

KATA PENGANTAR

Puji syukur kehadirat Allah Tuhan Yang Maha Esa atas segala nikmat yang diberikan terutama atas kelancaran penyelesaian tesis ini dengan baik dan tepat waktu. Shalawat serta salam terlimpahkan kepada Nabi Muhammad SAW yang selalu menjadi sosok teladan di kehidupan ini.

Penulis menyadari bahwa tesis dapat diselesaikan berkat dukungan dan bantuan dari berbagai pihak, oleh karena itu penulis berterima kasih kepada semua pihak yang secara langsung maupun tidak langsung memberikan kontribusi dalam menyelesaikan Tesis ini Maka dari itu penulis mengucapkan terima kasih kepada :

1. Bapak Dr. Hery Heryanto, S.Kom., M.Kom. atas segala bimbingan yang telah diberikan serta ketersediaan meluangkan waktu dalam menyelesaikan tesis ini.
2. Jabar Digital Service sebagai tempat bertumbuh dan menjadi bagian kecil untuk membangun solusi lewat teknologi.
3. Orang tua tercinta beserta keluarga besar bapak Muchtapen dan Ibu Srichati yang tidak tahu bahwa penulis sedang menempuh Pendidikan S2, semoga menjadi kabar kejutan bahagia pada saat kelulusan.
4. Rekan kerja sekaligus sahabat dan keluarga di PT Dunia Data Digital yang selalu mendukung baik moral, fasilitas dan doa terbaiknya selama ini.
5. Seluruh dosen program Pascasarjana Magister Sistem Informasi STMIK LIKMI yang telah bersedia memberikan arahan dan bimbingan bagi penulis dalam mendalami ilmu-ilmu sistem informasi.
6. Teman seperjuangan S2 STMIK LIKMI angkatan 2021, di mana penulis belajar menimba ilmu bersama, belajar bersama, dan bersama-sama saling memberikan semangat untuk segera menyelesaikan studinya.
7. Skal *Coffee* dengan kopinya yang nikmat untuk menjadi teman begadang saat pengerjaan tesis ini.

Penulis menyadari bahwa masih terdapat kekurangan dari penulisan tesis ini. Untuk itu saran serta kritikan yang membangun sangat diharapkan.

Penulis berharap tesis ini dapat berguna dalam rangka menambah wawasan serta pengetahuan khalayak banyak. Semoga tesis ini berguna sebagai bahan pembelajaran untuk membangun sistem yang lebih baik, handal dan bahan pembelajaran bagi penelitian selanjutnya.

Bandung, 3 - Maret - 2022

Penulis

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

DAFTAR SIMBOL

Simbol *Use Case Diagram*			
No.	Simbol	Nama	Keterangan
1		*Actor*	Mewakili perang seseorang atau juga sistem lain.
2		*Use case*	Fungsionalitas sebuah sistem
3		*Association*	Penghubung antara *use case* dengan *actor* tertentu.
4		*Include*	Menunjukkan bahwa *use case* satu merupakan bagian dari *use case* lainnya.
5		*Extend*	Menunjukkan arah panah secara putus-putus dari *use case* ke *base use case.*

Simbol Pada Topologi Arsitektur			
No.	Simbol	Nama	Keterangan
1		*Docker*	Sebuah *tools* untuk melakukan containerization
2		*Kubernetes*	Sebuah *tools* untuk melakukan *container orchestration*
3		*Server*	*Hardware* yang bersifat melayani permintaan dari para pengguna sistem.
4		*Database*	Tempat penyimpanan data
5		*Internet*	Media untuk melakukan sistem komunikasi secara global
6		*Versioning Control*	Merupakan sistem yang merekam perubahan dari sebuah *file*, berkas, dan aplikasi.

BAB I

PENDAHULUAN

1.1 Latar Belakang

Aplikasi Sapawarga dikembangkan oleh Jabar Digital Service atau Unit Pelayanan Teknis Pelayanan Digital, Data, dan Informasi Geospasial di bawah Dinas Komunikasi dan Informatika Pemerintah Provinsi Jawa Barat. Sapawarga adalah aplikasi publik yang bertujuan untuk akses layanan publik, aspirasi, informasi, berpartisipasi, dan saling berinteraksi dengan RW lainnya. Pada tahap rilis awal ini Sapawarga hanya bisa digunakan oleh Kepala RW. Pada tahun 2022 harapannya Sapawarga dapat digunakan oleh seluruh warga Jawa Barat, sehingga manfaat teknologi dapat terasa pada masyarakat luas. Melihat data penduduk menurut Badan Pusat Statistik (BPS) Jawa Barat menyebutkan, dari hasil dari kegiatan Sensus Penduduk 2020 (SP2020) diketahui penduduk Jawa Barat pada bulan September 2020 sebanyak 48,27 juta jiwa, dengan persentase penduduk usia produktif (15-64 tahun) 70,68 persen di tahun 2020. Untuk mempersiapkan Sapawarga dapat diakses oleh banyak pengguna dibutuhkan arsitektur infrastruktur dan aplikasi yang baik juga efisien.

Pada awal peluncuran, Sapawarga dikenalkan sebagai *SupperApp* sebagai lompatan transformasi digital. Istilah *superapp* sendiri dikenalkan oleh pendiri BlackBerry *Mike Lazaridis* yang berarti sebuah ekosistem yang tertutup yang terdiri banyak aplikasi, yang akan digunakan orang setiap hari karena menawarkan pengalaman yang mudah, terintegrasi, kontekstual dan efisien (*Lazaridis*, 2010). Secara konsep berarti, *superapp* Sapawarga dalam pengembangan ke depan akan dimungkinkan memiliki banyak sekali menu atau fitur yang saling terintegrasi, ini akan menambah kompleksitas aplikasi dalam sisi pengembangan dan juga pengelolaan *source code,* karena kode terus bertambah besar pada sebuah *repository*. Semakin banyak fitur pada aplikasi, maka akan semakin besar pula beban *server* yang didapat.

Mengembangkan aplikasi dengan jumlah konkurensi akses pengguna yang banyak adalah hal yang menantang. Sebagus apapun fitur pada aplikasi, kalau tidak bisa dibuka atau memiliki *load time* yang lama akan mengganggu pengguna, bahkan bisa

ditinggalkan oleh pengguna. Selain faktor fungsionalitas, aplikasi yang baik adalah aplikasi yang dapat memenuhi kriteria yang *scalable* yaitu aplikasi yang dapat menangani penambahan beban yang diberikan tanpa berdampak pada penurunan kinerja aplikasi.

Microservice adalah sebuah arsitektur yang memiliki pendekatan untuk mengembangkan aplikasi dengan memecah ke dalam *service-service* kecil, di mana setiap *service* dapat ditempatkan di *server* yang berbeda (*Independent*), namun tetap bisa saling berhubungan dengan *service* lainnya dengan menggunakan protokol REST (*Representational State Transfer*) API, RPC (*Remote Procedure Call*), atau juga Protobuf. Dengan terpisahnya servis maka, aplikasi dapat membagi beban *request* dari pengguna. Terdapat kekurangan dan kelebihan masing-masing dalam arsitektur *monolith* dan *microservice*, namun untuk aplikasi yang memiliki skala besar dan memiliki *concurrency* pengguna yang banyak, sehingga *server* tidak mampu lagi menangani *request* dan berakibat lambatnya akses, sangat disarankan untuk menerapkan arsitektur *microservice*.

Langkah berikutnya adalah bagaimana untuk mengelola *microservice* yang memungkinkan akan mempunyai banyak service, di mana setiap *service* memiliki biaya yang harus diperhitungkan. Penggunaan *orchestration container* berupa Kubernetes dapat membantu mengatur atau mengorkestrasikan banyak *container*, kapan *service* melakukan *scale up* untuk menunjang *request* yang banyak, kapan melakukan *scale down* untuk melakukan efisiensi. Proses *scaling* dapat dilakukan ketika *service* sudah melewati metrik utilisasi misal pemakaian CPU yang sudah melebihi dari batas yang sudah ditentukan.

Dengan mempertimbangkan fitur terus bertambah, tim *development* yang bertambah banyak, jumlah akses aplikasi yang terus meningkat, untuk meningkatkan skalabilitas aplikasi dan membuat tim yang *autonomous* yang fokus pada fitur yang spesifik, Penulis mengambil judul penelitian **"PERANCANGAN ARSITEKTUR *MICROSERVICE* MENGGUNAKAN *CONTAINER ORCHESTRATION* UNTUK MENDUKUNG *HIGH SCALABLE APPLICATION*"** Studi kasus pada aplikasi Sapawarga

1.2 Rumusan Masalah

Berdasarkan latar belakang yang sudah dijelaskan sebelumnya, maka dapat menyampaikan beberapa rumusan masalah, yaitu:

1. Bagaimana membangun arsitektur aplikasi Sapawarga dengan skala yang besar yang mampu melayani kebutuhan pengguna tanpa mengalami penurunan kinerja?
2. Bagaimana mengelola layanan arsitektur aplikasi Sapawarga dengan menggunakan arsitektur *microservice*?

1.3 Tujuan Penelitian

Tujuan dari penelitian ini adalah sebagai berikut :

1. Sebagai cetak biru untuk menggantikan arsitektur lama dari *monolith* menjadi *microservice*.
2. Menggunakan *orchestration container* Kubernetes untuk mengatur *scaling* pada *microservice* secara efisien, akan melakukan *scaling up* secara horizontal jika aplikasi mengalami *traffic* yang tinggi, dan melakukan *scaling down* jika aplikasi berada pada *traffic* yang rendah atau sedang.

1.4 Pembatasan Masalah

Untuk menghindari bahasan yang meluas dari topik utama, berikut beberapa batasan masalahnya:

1. Penggunaan *microservice* hanya pada lingkup aplikasi *backend* saja yang berbasi API, karena yang bertugas menangani *request* adalah dari sisi *backend.*
2. Penggunaan *database* menggunakan MySQL.
3. Fitur yang akan diubah menjadi *microservice* adalah beberapa fitur unggulan : *User*, Kegiatan RW, Usulan, Survei, *Polling*, Berita, dan Info Penting.

1.5 Jenis Penelitian dan Teknik Pengumpulan Data

Untuk jenis penelitian akan dilakukan observasi terhadap sistem lama, melakukan analisa terhadap teknologi arsitektur *microservice* kemudian melakukan perubahan dari

sistem *monolith* ke *microservice.* Setiap *service* yang telah terpisah akan dilakukan proses *containerize* dengan menggunakan Docker dan penambahan *container orchestration* menggunakan Kubernetes. Penggunaan Kubernetes untuk mendukung proses *auto scaling* secara *horizontal* sehingga aplikasi dan arsitektur dapat mendukung aspek skalabilitas.

1.6 Sistematika Penulisan

Adapun untuk sistematika penulisan dalam penyusunan tesis ini terdiri dari beberapa BAB, yaitu sebagai berikut :

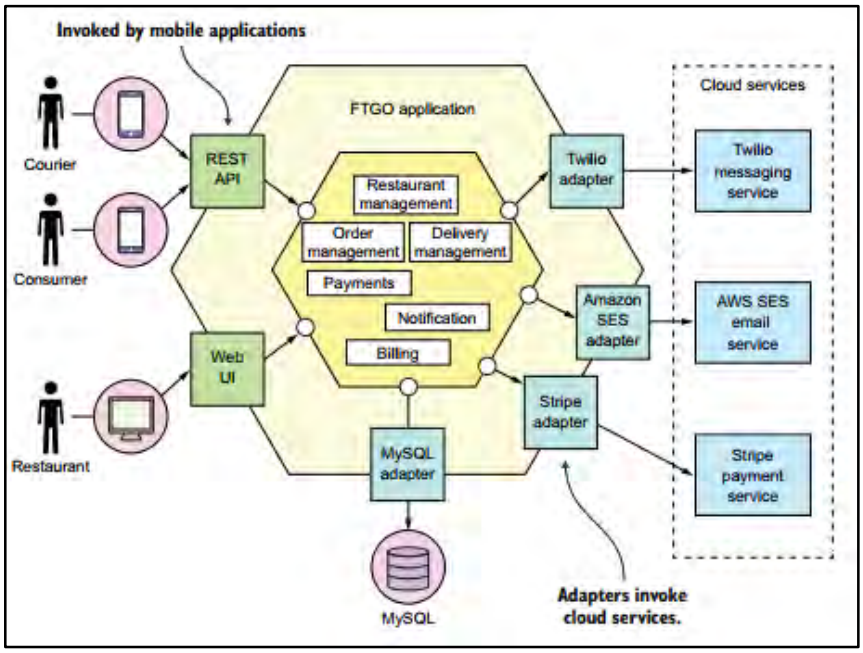
1. BAB I PENDAHULUAN : Pada bab ini akan menjelaskan latar belakang masalah berupa aplikasi Sapawarga yang ke depan harus siap sebagai *Superapp* dan tetap dapat mudah diakses oleh banyak pengguna untuk manfaat yang lebih luas dengan mempertimbangkan aspek skalabilitas.
2. BAB II LANDASAN TEORI : Bab ini berisi uraian tentang teori-teori keilmuan yang diambil dari referensi yang mendukung dan berhubungan dengan topik bahasan tesis. Pembahasan tentang *microservice, container, container orchestration, cloud native* dan teori lain yang mendukung.
3. BAB III OBJEK DAN METODE PENELITIAN : Pada bab ini membahas mengenai tempat dan juga objek penelitian beserta metodenya.
4. BAB IV HASIL DAN PEMBAHASAN : Bab ini menjelaskan hasil rancangan, temuan, nilai metrik yang dapat diukur sebagai tolak ukur skalabilitas dan efisiensi.
5. BAB V KESIMPULAN DAN SARAN : Bab ini memberikan kesimpulan mengenai penelitian yang dikerjakan, juga memberikan saran untuk penelitian selanjutnya.

BAB II

LANDASAN TEORI

2.1 Arsitektur *Monolithic*

Monolith adalah arsitektur di mana keseluruhan kode akan dikompilasi menjadi satu aplikasi. Aplikasi tersebut menjalankan seluruh proses yang dibutuhkan (Richardson, 2017). *Monolith* tidak membutuhkan *service* lain, karena semua *service* telah tergabung dalam sebuah cakupan atau keseluruhan kode pada sebuah aplikasi.



Gambar 2.1
Contoh Arsitektur M*onolith* Pada Sebuah Aplikasi
Sumber : *Microservice Pattern,* (Richardson, 2017)

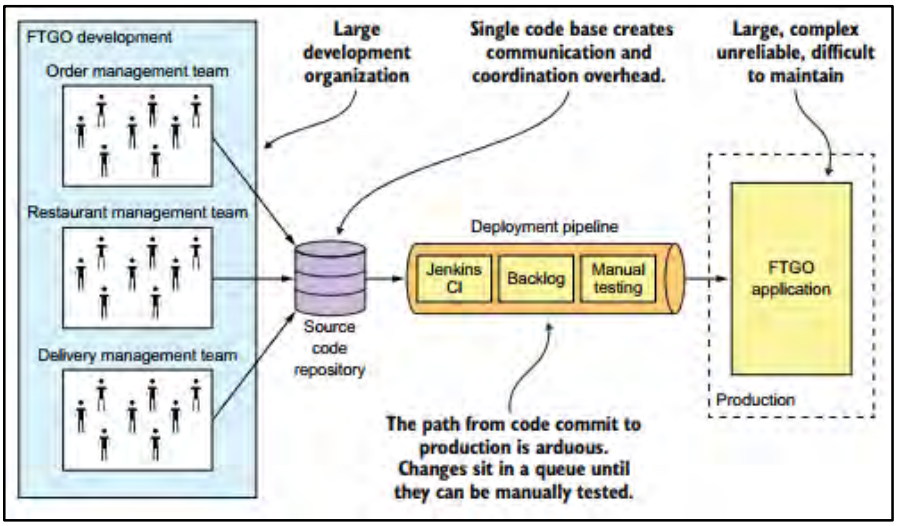
Kelebihan arsitektur *monolithic* :

1. Lebih mudah untuk diterapkan pada awal pengembangan aplikasi.
2. Mudah untuk di *deploy* karena tinggal menyalin semua *file* yang berada dalam satu paket aplikasi ke *server*.
3. Proses *deployment* yang lebih sederhana, karena hanya ada satu tempat pengembangan.

Adapun Kekurangan arsitektur *monolithic* adalah sebagai berikut:

1. Memiliki keterbatasan ukuran dan kompleksitas.

2. Aplikasi menjadi terlalu besar dan kompleks sehingga sulit untuk dipahami dan dilakukan perubahan, misal sebuah fungsi yang sudah dipakai di banyak tempat ketika ada perubahan, maka harus melakukan pengecekan pada semua tempat.
3. Ketika terjadi perubahan fitur, maka untuk proses *deployment* baik *staging* maupun *production* membutuhkan waktu yang lebih lama, karena mencakup keseluruhan aplikasi.
4. Memperbesar peluang konflik pada kode, karena semua bekerja pada *repository* yang sama, bahkan dapat bekerja pada *file* dan fungsional kode yang serupa.
5. Hanya dapat menggunakan sebuah teknologi, misal hanya bisa menggunakan satu bahasa pemrograman.

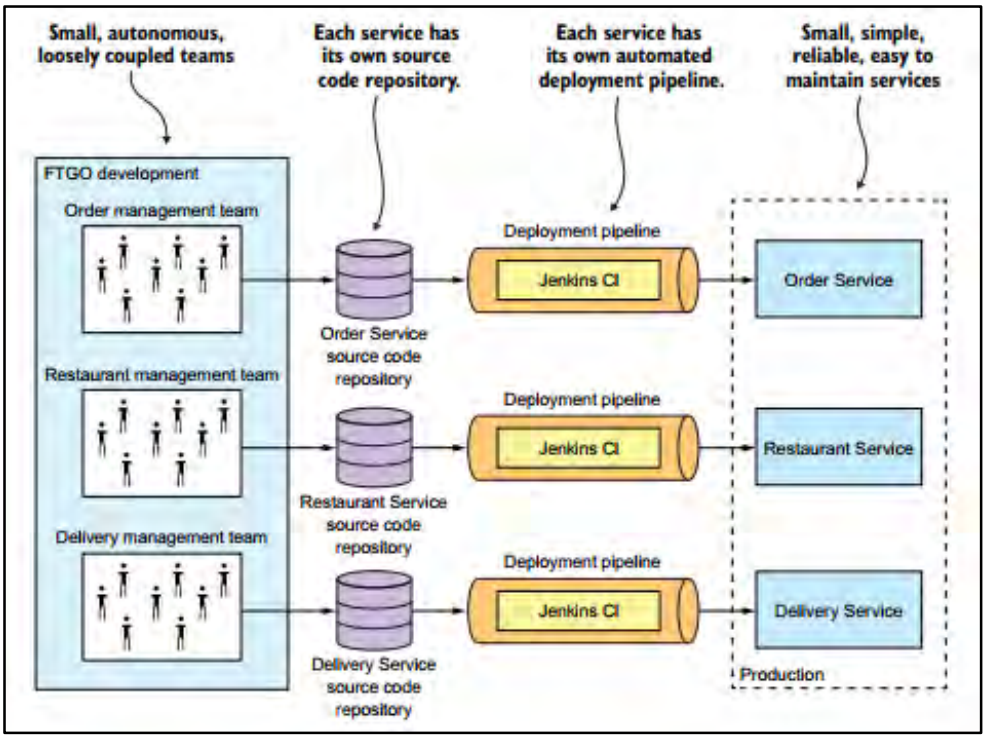


Gambar 2.2
Kendala Monolith Dalam Pengembangan Sistem Skala Besar
Sumber : *Microservice Pattern,* (Richardson, 2017)

2.2 Arsitektur *Microservice*

Microservice adalah pendekatan untuk mengembangkan aplikasi tunggal sebagai rangkaian *service* atau layanan kecil, masing-masing berjalan dalam prosesnya sendiri dan dapat saling berkomunikasi, sering kali merupakan API sumber daya HTTP (Fowler, 2017). Setiap layanan merupakan aplikasi kecil yang dibuat untuk menyelesaikan tanggung jawabnya masing-masing, sangat dimungkinkan setiap layanan memiliki bahasa pemrograman dan *database* yang berbeda tergantung dari fokus masalah apa yang ingin diselesaikan.

Kerumitan aplikasi yang di mana fiturnya terus bertambah banyak akan menyulitkan dalam pengembangannya. Kode yang bertambah banyak dan tim yang semakin banyak akan menambah kerumitan. *Microservice* merupakan salah satu solusi yang dapat digunakan untuk membantu meringankan kompleksitas pengembangan aplikasi.



Gambar 2.3
Contoh Penggunaan M*icroservice* Pada Aplikasi
Sumber : *Microservice Pattern,* (Richardson, 2017)

2.2.1 Karakteristik *Microservice*

Berikut beberapa karakteristik dari *microservice* :

1. *Independently deployable*, dapat melakukan *deployment* tanpa memiliki ketergantungan dengan *service* yang lain.
2. *Highly observable*, dapat melakukan *logging, monitoring, tracing* pada setiap layanannya.
3. *Loosely coupled*, *service* dapat menjalankan pekerjaannya dengan tidak bergantung pada *service* yang lain.

4. *Decentralized*, setiap fitur dapat berdiri sendiri, dan dapat berjalan pada berbagai *server*. Tim pengembang dapat memilih standar kode dan datanya spesifik setiap layanannya.
5. *Highly testable*, setiap layanan dapat dilakukan *automated testing* dengan lebih ringan karena fokus pada salah satu layanan saja.
6. *Highly maintainable*, sudah terpisahnya logika bisnis dalam layanan kecil yang terpisah, sehingga mudah untuk melakukan pengembangan dan perawatan pada masing-masing layanan.

2.2.2 Kelebihan *Microservice*

Berikut beberapa kelebihan dari *microservice* :

1. *Autonomy*, setiap layanan dapat dibangun dengan tim pengembang yang berbeda karena tidak akan mempengaruhi layanan lainnya.
2. *Agility* atau kelincahan, setiap layanan bisa memilih teknologi yang berbeda baik bahasa pemrograman dan *database*.
3. *Scalability*, dapat melakukan secara spesifik layanan mana yang akan melakukan *scaling,* sehingga tidak perlu harus melakukan pada seluruh sistem.
4. *High reliability*, kesalahan di salah satu layanan tidak menjadikan seluruh aplikasi mati.
5. Modul yang dapat digunakan kembali pada layanan lainnya.
6. *Asymmetric Service Scaling*, dapat melakukan *scaling* secara horizontal dan vertikal.

2.2.3 Kekurangan *Microservice*

Berikut kekurangan *microservice* :

1. Membutuhkan lebih banyak sumber daya dan waktu yang lebih lama daripada *monolith.*
2. Dalam tahap pengembangan aplikasi memerlukan *role devops* untuk menangani masalah seperti latensi , *api gateway* dan *load balancing*,

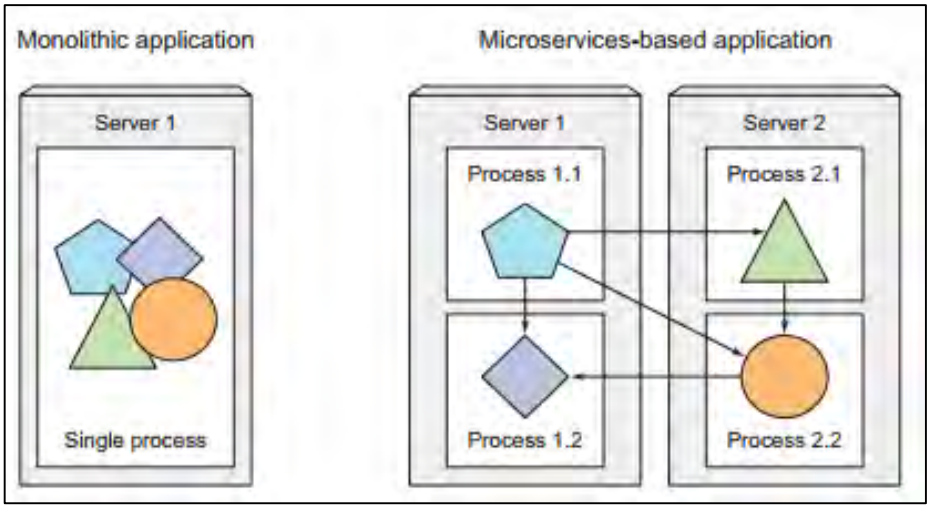
2.2.4 Perbandingan *Monolith* Dengan *Microservice*

Sebagaimana telah dibahas tentang kekurangan dan kelebihan masing-masing arsitektur *monolith* dan *microservice*, tidak ada keharusan bahwa saat ini semua harus

menggunakan *microservice* atau *monolith*, tidak ada yang salah atau benar pada salah satu arsitektur. Penerapan arsitektur yang benar adalah yang sesuai dengan masalah apa yang ingin diselesaikan berdasarkan karakteristik aplikasi.

Perbandingan yang sangat nyata adalah *microservice* dapat menangani *request* yang lebih banyak secara konkurensi dibanding dengan *monolith*, karena secara teori beban *request* telah terbagi menjadi banyak *service* yang sebelumnya hanya satu *service* saja. Beban *server* pada aplikasi maupun *database* telah terbagi mengikuti tiap *service,* namun masih dapat berkomunikasi antar *service* seperti terlihat pada Gambar 2.4.

Arsitektur *monolith* pada pengembangan aplikasi awal dengan spesifikasi yang tidak terlalu kompleks adalah pilihan yang baik, karena lebih sederhana dan cepat. Aplikasi yang memiliki karakteristik jumlah pertumbuhan pengguna yang banyak, harus *running* selalu dan memiliki banyak *request* sebaiknya harus menggunakan arsitektur *microservice*.



Gambar 2.4
Perbandingan Arsitektur *Monolith* Dengan *Microservice*
Sumber : Kubernetes *in Action*, (Luksa, 2017)

Sebagai sarana untuk memilih arsitektur dengan tepat pada sebuah aplikasi, perlu mempertimbangkan dan membandingkan beberapa aspek utama dalam pengembangan aplikasi. Pastikan masalah yang dihadapi cocok dengan spesifikasi masing-masing arsitektur aplikasi. Berikut tabel perbandingan dari beberapa aspek :

Tabel 2.1
Perbandingan M*onolith* dan M*icroservice*

Aspek	*Monolith*	*Microservice*
Deployment	Sekali *deploy* mencakup seluruh sistem	Bisa melakukan *deployment* pada salah satu layanan
Maintenance	Cukup memahami bahasa pemrograman	Membutuhkan kemampuan *devops,* Docker, Kubernetes*, cloud computing.*
Reliability	Kegagalan salah satu fitur dapat membuat seluruh sistem *down.*	Satu layanan mati, tidak berakibat pada layanan lainnya.
Scalability	Rendah, hanya vertikal.	Tinggi, vertikal dan horizontal
Agility	Sulit mengganti teknologi baru misal bahasa pemrograman, karena harus mengubah semua kode,	Memungkinkan untuk memakai bahasa program yang berbeda di tiap layanan.
Development	Tim terlibat dalam proses pengembangan secara bersamaan.	Tim yang berbeda dapat bekerja pada layanan yang lain.
Updates	*Update* memerlukan waktu yang lama, karena ketergantungan dengan *developer* lain yang bekerja pada *repository* yang sama.	*Update* bisa dengan cepat dilakukan karena bersifat otonom.
Testing	Dapat melakukan *end-to-end* pengujian	Setiap pelayanan harus dilakukan pengujian.
Security	Hanya memperhatikan keamanan pada tingkat aplikasi.	Harus memperhatikan komunikasi setiap layanan yang melalui API *Gateway*.

2.2.5 Kapan Memilih *Monolith* atau *Microservice*

Kapan sebaiknya menggunakan *monolith* :

1. Ketika hanya memiliki tim yang relatif kecil dan tim tidak ada keahlian dalam *microservice.*
2. Ketika akan melakukan pengembangan aplikasi yang sederhana,
3. Ketika akan melakukan pengembangan dalam waktu yang singkat.

Kapan sebaiknya menggunakan *microservice* :

1. Ketika akan mengembangkan aplikasi dengan skala yang besar dan *server* sudah dalam batas yang maksimal.

2. Mempunyai cukup waktu dan anggaran untuk membuat rencana dan mengembangkan *microservice*.
3. Ketika aplikasi membutuhkan beberapa metoda, *scaling vertical* dan *horizontal*.
4. Ketika tim pengembang mulai banyak, membutuhkan tim kecil untuk fokus akan masalah yang dihadapi

2.3 Service Oriented Architecture

Service Oriented Architecture (SOA) adalah gaya arsitektur untuk membangun solusi perusahaan berdasarkan layanan. Lebih khusus lagi, SOA berkaitan dengan konstruksi independen dari layanan yang selaras dengan bisnis yang dapat digabungkan ke dalam proses dan solusi bisnis tingkat tinggi yang bermakna dalam konteks perusahaan (Rosen, 2012). SOA menampilkan satu lapisan baru yang disebut lapisan *service*, yang berfungsi untuk memisahkan logika dari aplikasi dan logika bisnis. Hal ini terkait dengan salah satu karakteristik SOA, yakni meningkatkan aspek *loosely couple* di antara logika bisnis dan logika aplikasi. Ketika lapisan *service* dapat merepresentasikan logika dari bisnis dan aplikasi, maka ketergantungan langsung antara logika bisnis dan logika aplikasi rendah.

SOA muncul pada akhir 1990-an dan merupakan tahap penting dalam evolusi pengembangan aplikasi dan integrasi. Sebelum SOA menjadi pilihan, menghubungkan aplikasi monolitik ke data atau fungsionalitas di sistem lain memerlukan integrasi titik-ke-titik yang kompleks yang harus dibuat ulang oleh pengembang untuk setiap proyek pengembangan baru. Mengekspos fungsi-fungsi tersebut melalui SOA menghilangkan kebutuhan untuk membuat ulang integrasi mendalam setiap saat.

Sekilas terdapat persamaan antara kedua arsitektur yaitu sama-sama memecah ke dalam beberapa layanan, namun secara konsep keseluruhan terdapat banyak perbedaan yang mendasar. Data XML adalah bahan utama untuk solusi yang didasarkan pada arsitektur SOA. Aplikasi SOA berbasis XML dapat digunakan untuk membangun layanan web. Pada SOA pola komunikasi menggunakan protokol XML SOAP, namun pada *microservice* menggunakan REST ataupun RPC. Proses desain dan pembagian

fungsional yang mengedepankan pembagian berdasarkan *domain* fungsionalitas pada organisasi Berikut beberapa perbedaan antara SOA dan *microservice* dengan beberapa sudut pandang*:*

Tabel 2. 2
Perbandingan SOA dan M*icroservice*

Sudut Pandang	**SOA**	***Microservice***
Arsitektur	SOA memiliki konsep yang lebih luas, sehingga ruang lingkup masalah dengan gaya ini lebih besar. Tidak seperti layanan *microservice*, SOA terdiri dari layanan aplikasi yang digabungkan secara longgar yang berkomunikasi melalui mekanisme komunikasi umum	*Microservice* adalah artisektur yang mengembangkan sebuah aplikasi sebagai rangkaian *service* kecil dan independen yang dikembangkan dan digunakan secara independen.
Penyimpanan Data	Model SOA memiliki satu lapisan penyimpanan data yang digunakan bersama oleh semua layanan dalam aplikasi tersebut. Jadi tidak memiliki penyimpanan data yang independen	Memiliki penyimpanan data independen yang berarti setiap Layanan akan menjadi layanan independen dan tidak berbagi penyimpanan data yang umum di antara mereka.
Fleksibilitas	Memiliki fleksibilitas organisasi yang lebih besar dan implementasi yang spesifik untuk lingkungan, sehingga mereka dapat secara efektif merespons perubahan lingkungan bisnis. SOA mendistribusikan logika ke portal dan layanan individual.	Perubahan sistem atau sebuah fitur pada setiap *service* dapat dilakukan pengujian secara independen pada setiap *service*-nya.
Toleransi kesalahan	SOA memungkinkan integrasi komponen perangkat lunak yang ada dari berbagai sumber dengan lebih cepat sehingga toleransi kesalahan dapat dilakukan.	Lebih rentan karena lebih rumit, namun *service*-nya lebih independen dan otonom. Sehingga jika terjadi kesalahan tidak pasti berdampak pada *service* lain. Sebuah kesalahan pada *service* tidak pasti menjadikan seluruh sistem *down*

2.4 Domain Driven Design

Domain Driven Design (DDD) merupakan pendekatan pembangunan perangkat lunak dalam modeling suatu aplikasi, DDD membantu dalam proses pengembangan perangkat lunak menjadi lebih akurat sesuai dengan proses bisnis yang akan berjalan pada aplikasi (Evans, 2004). Perangkat lunak sederhana yang dapat diselesaikan dengan atensi

pada desain. Namun, ketika fitur yang terus bertambah dan kompleksitas mulai muncul, hal ini menjadi kesulitan bagi para pengembang.

Untuk merancang perangkat lunak yang baik, penting untuk mengetahui tentang domain bisnis perangkat lunak itu sendiri. DDD memiliki nilai untuk melakukan pemetaan kompleksitas sebuah konsep domain bisnis ke dalam perangkat lunak. Mengatur domain kode yang selaras dengan masalah bisnis dengan menggunakan bahasa umum yang sama berdasarkan konsensus antara *expert* dan *developer*.

Kompleksitas adalah istilah yang relatif, apa yang rumit bagi seseorang bisa jadi sederhana bagi orang lain. Namun, kompleksitas adalah masalah yang harus dipecahkan oleh DDD. Dalam konteks ini, kompleksitas berarti keterkaitan, banyak sumber data yang berbeda, tujuan bisnis yang berbeda. Pendekatan DDD hadir untuk memecahkan kompleksitas pengembangan perangkat lunak. Di sisi lain, dapat menggunakan desain yang muncul saat tantangannya sederhana.

2.4.1 Istilah Pada *Domain Driven Desain*

Pendekatan DDD lebih lanjut mendefinisikan beberapa istilah umum yang berguna saat menjelaskan dan merancangnya. Berikut adalah beberapa istilah umum yang digunakan pada pendekatan pemecahan *service* berdasarkan DDD :

1. *Context* : Pengaturan di mana kata atau pernyataan muncul yang menentukan maknanya. Pernyataan tentang suatu model hanya dapat dipahami dalam suatu konteks.
2. *Model* : Sebuah sistem abstrak yang menggambarkan aspek-aspek yang dipilih dari *domain* dan dapat digunakan untuk memecahkan masalah yang terkait dengan *domain* tersebut.
3. *Ubiquitous Language*: Model yang bertindak sebagai bahasa universal untuk membantu komunikasi antara *developers* dan *experts*. Tujuannya adalah *domain* dari *experts* dan *developers* bisa saling mengerti apa yang sedang didiskusikan karena mereka menggunakan istilah yang sama.
4. *Bounded Context* : Deskripsi pembatas, biasanya sub sistem, atau pekerjaan tim tertentu di mana model tertentu didefinisikan dan dapat diterapkan.

2.4.2 Tahapan implementasi *Domain Driven Design*

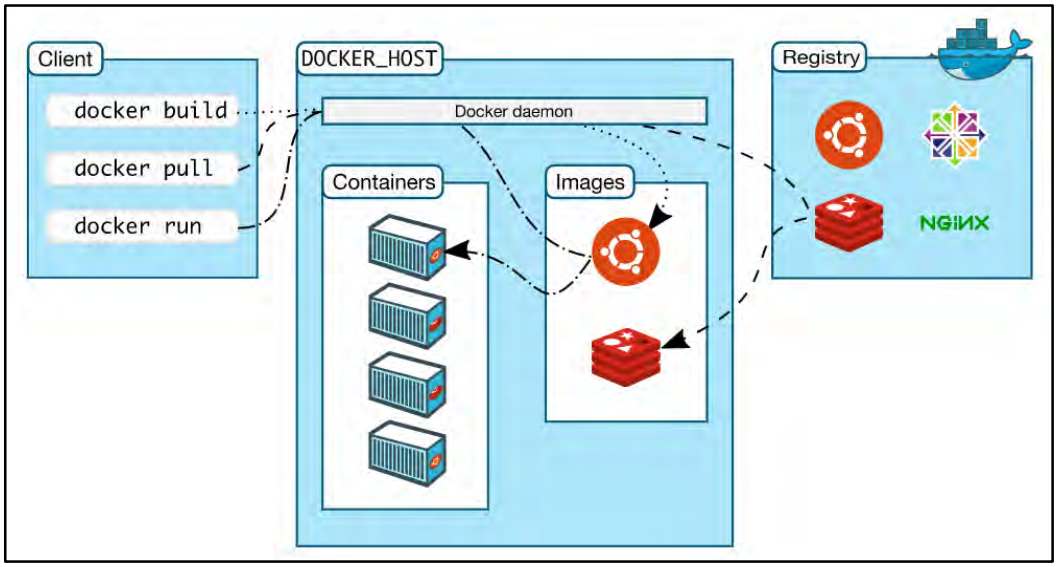
Tahapan pada pendekatan DDD adalah sebagai berikut :

1. *Analyze Domain* : Mulai dengan memetakan semua fungsi bisnis dan koneksinya. Membuat kolaborasi yang melibatkan *expert*, *developer*, dan pemangku kepentingan lainnya. Hasil berupa keseluruhan *domain* yang ada pada aplikasi *existing*.
2. *Define bounded contexts* : Selanjutnya, menentukan batasan konteks setiap konteks terikat berisi model *domain* yang mewakili *subdomain* tertentu dari aplikasi yang lebih besar.
3. *Define entities, aggregates, and services* : Mendefinisikan entitas kemudian melakukan pengecekan untuk fungsi agregat apakah ada antara *service* yang terlibat.
4. *Identify microservices* : Menggunakan hasil dari pemecahan *domain* yang dilakukan pada langkah sebelumnya untuk mengidentifikasi dan menerapkan *microservice*.

Gambar 2 5
Domain Analysis
Sumber : Build microservices on Azure (2019).

2.5 Docker *Container*

Docker adalah aplikasi yang berjalan pada Linux dan Windows yang dapat membuat, mengontrol sebuah *containerized* aplikasi (Poulton, 2019). Docker merupakan proyek *open source platform as a service (PaaS)*. Docker mengemas aplikasi ke dalam satuan standar yang disebut kontainer. Dalam kontainer terdapat semua yang dibutuhkan aplikasi agar dapat berfungsi dengan baik termasuk *library, environment*, kode, dan waktu proses.



Gambar 2.6
Docker A*rchitecture*
Sumber : https://docs.docker.com/get-started/overview/

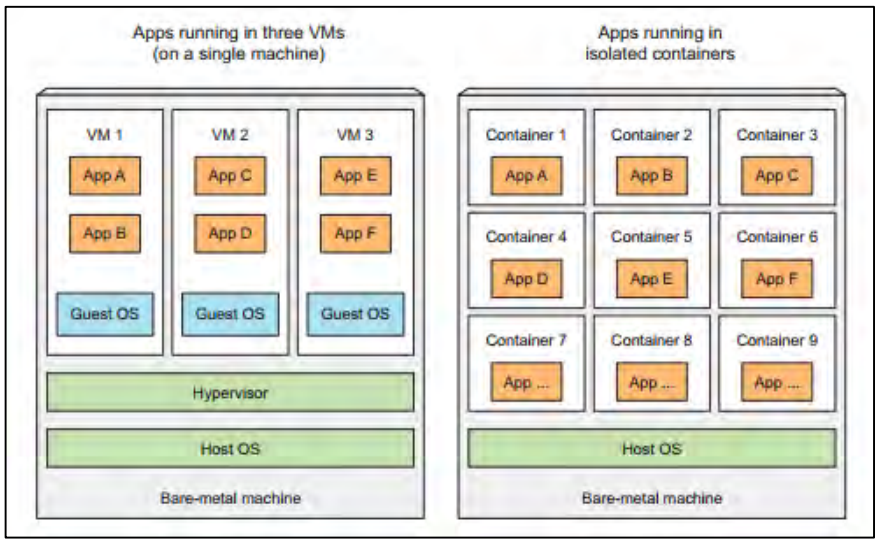
Docker menggunakan arsitektur berbasis *client-server*. *Docker client* adalah cara utama pengguna berinteraksi dengan Docker. Saat menjalankan perintah seperti docker run, klien mengirimkan perintah ini ke Docker, yang menjalankannya menggunakan API Docker. Klien Docker dapat berkomunikasi dengan lebih dari satu daemon. Terlihat pada gambar 2.5 , Docker *daemon* berjalan pada *host*, sehingga pengguna tidak dapat berinteraksi langsung. Untuk mengaksesnya, menggunakan Docker *client* yang merupakan tampilan utama untuk pengguna, sehingga pengguna dapat berkomunikasi dengan Docker *daemon*.

Berikut beberapa kelebihan menggunakan Docker *Container* :

1. Mempermudah pengembangan : Docker dapat mempermudah pekerjaan *developer* ketika mengembangkan aplikasi. Alasannya, Docker lebih hemat *resource* dan mampu menyediakan *environment* yang stabil untuk dijalankan di perangkat apapun, mulai dari *cloud server* hingga komputer pribadi.
2. Menyederhanakan konfigurasi : Docker tidak memiliki *overhead* sehingga *developer* bisa menjalankan aplikasi yang diuji tanpa konfigurasi tambahan.
3. Memudahkan pengembangan *pipeline* : *Developer* bisa memanfaatkan Docker *container* sebagai tempat pengujian kode *pipeline* beserta *tools* yang diperlukan dengan lebih mudah.

4. Bisa digunakan sebagai *debugging* : Jika terdapat sebuah masalah baik *error* pada aplikasi ataupun pada *library*, Docker menyediakan fitur dapat melakukan debugging. Pengguna dapat melihat *error log* pada sebuah *container* tersebut.
5. Mendukung *Multi Tenancy* : Adalah kemampuan untuk menjalankan beban kerja milik entitas yang berbeda. Docker cocok digunakan untuk membuat aplikasi berstruktur *multi tenance* seperti *Software as a Service* (SaaS).
6. Meningkatkan sumber daya dengan cepat : Karena telah terbungkus dengan standar yang sama, Docker dapat peningkatan sumber daya perangkat jika di setup pada mesin atau server yang berbeda.

Container adalah sebuah virtualisasi OS yang dapat membungkus suatu aplikasi beserta *dependency* dan *environment* setiap kontainer ini memiliki proses yang terisolir sehingga tidak mengganggu host OS ataupun kontainer yang lain. Prinsip kontainer ini mirip dengan kontainer yang ada di kapal kargo di mana kapal kargo tersebut diibaratkan sebagai sistem komputer. *Virtual machine* (VM) adalah sebuah *emulator* dari sebuah sistem komputer. Secara sederhana, VM membuat kita bisa membagi *resource hardware* dari satu hardware fisik menjadi beberapa sistem komputer.



Gambar 2.7
Perbedaan *Virtual Machine* (VM) dan *Container*
Sumber : Kubernetes *in Action*, Marko Luksa (2018)

Virtual Machine (VM) adalah sistem yang bertindak persis seperti komputer. VM adalah abstraksi dari perangkat keras fisik yang mengubah satu server menjadi banyak *server*. Hypervisor memungkinkan beberapa VM berjalan pada satu mesin. Setiap VM menyertakan salinan lengkap sistem operasi, aplikasi, binari, dan library yang diperlukan. Docker adalah aplikasi tidak memerlukan sistem operasi secara penuh setiap kali menjalankan sebuah kontainer baru. Docker bergantung pada *kernel* Linux OS *host* yang mana hampir semua distribusi Linux menggunakan model kernel standar yang mana kontainer dibangun dengan *kernel* tersebut, seperti Ubuntu, Debian, dan lain-lain.

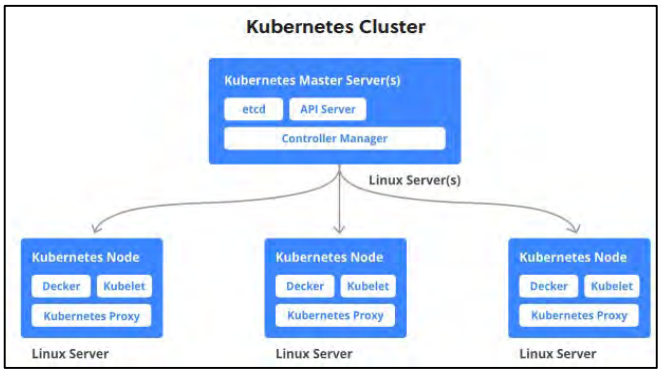
2.6 Kubernetes

Kubernetes adalah proyek open source dari Google yang dengan cepat muncul sebagai *orchestrator* dari *containerized* aplikasi (Poulton, 2017). Kubernetes dapat melakukan manajemen *workloads* pada kumpulan kontainer, serta menyediakan konfigurasi dan otomatisasi secara deklaratif. *Service*, *support*, dan perkakas Kubernetes tersedia secara luas. Platform ini pertama kali dikembangkan oleh *Google* pada tahun 2014 dan kini dikelola oleh *Cloud Native Computing Foundation* (CNCF) sebagai platform manajemen kontainer yang cukup populer.

Kubernetes dapat menjadwalkan dan menjalankan kontainer aplikasi pada kelompok mesin fisik atau virtual. Dengan menggunakan Kubernetes, para pengguna dapat memperoleh sejumlah keuntungan diantaranya skalabilitas, portabilitas, *deployment* yang konsisten, dan *separated automated operation* dan *development*. Kemampuan ini tentunya sangat berguna untuk aplikasi yang memiliki *traffic* sering tidak bisa diprediksi baik waktu maupun jumlah *traffic*. Aplikasi pada sebuah organisasi harus tetap berjalan melayani pengguna, misal pada layanan publik seperti pendaftaran masa sekolah, pembelian tiket kereta saat masa mudik yang dapat mengakibatkan lonjakan *traffic* meningkat drastis.

2.6.1 Klaster Kubernetes

Klaster adalah suatu kelompok berisi *server* fisik atau *virtual* (VPS) untuk menjalankan Kubernetes. Ada dua jenis *server* yang dibutuhkan, yaitu *master node* dan *worker node*.



Gambar 2.8
Klaster Pada Kubernetes
Sumber : *An Introduction to Kubernetes* (2019)

Master node adalah *server* utama yang mengatur semua operasi klaster menggunakan tiga komponen, yaitu :

1. Kube-apiserver : Validasi dan konfigurasi data untuk objek API, yaitu *pod, services, volume*, dan lainnya.
2. Kube-controller-manager : Melakukan *monitoring* klaster agar sesuai dengan konfigurasi data objek di dalam *node*.
3. Kube-scheduler : Menambah objek baru ke *node*, misalnya instalasi *pod* ke *node* tertentu.
4. Etcd : ruang penyimpanan *key value* konfigurasi data klaster.

Worker node adalah semua *server* yang bukan *master* yang berfungsi untuk menjalankan dua komponen, yaitu:

1. Kubelet : Komponen yang berjalan pada setiap node untuk memastikan kontainer beroperasi pada objek *pod*.
2. Kube-proxy : Memelihara aturan jaringan dan meneruskan koneksi ke suatu *host*.
3. Docker image : *File* dari aplikasi Docker yang berfungsi untuk membuat kontainer.

2.6.2 Objek Kubernetes

Beberapa istilah dalam Kubernetes :

1. *Pod*, adalah satu grup kontainer *instance,* yang dapat menjalankan beberapa kontainer dalam satu pod. Antar kontainer dalam satu pod bisa saling mengakses dengan menggunakan alamat lokal.

2. *Node*, adalah representasi dari satu mesin yang bisa berupa mesin virtual (VPS) atau fisik.
3. *Service*, merupakan mekanisme untuk mengekspos *pod* ke dunia luar. Aplikasi yang berjalan dalam *pod* tidak memiliki alamat IP yang tetap. Agar bisa diakses oleh aplikasi lain atau oleh pengguna, diperlukan alamat IP yang tetap.
4. *Volume*, adalah objek yang berfungsi untuk penyimpanan data suatu kontainer. Letaknya di luar kontainer
5. *Namespace*, adalah objek untuk memisahkan *resource* atau *environment* klaster. Dengan *namespace*, Anda dapat memisahkan tiap klaster *project* supaya tidak saling terganggu satu sama lain
6. *Label,* adalah seperangkat informasi metadata untuk mencari *pod* tertentu.
7. *Stage-production* : Label *stage* bisa gunakan untuk menentukan berbagai konfigurasi *environment deployment* aplikasi kita, misalnya pengembangan, pengujian, pengujian performa dan *production.*
8. *Selector* : Adalah *filtering* menggunakan label, misalnya ingin mencari semua *instance database* untuk aplikasi belajar yang berjalan di *production*.

2.6.3 Keunggulan Kubernetes

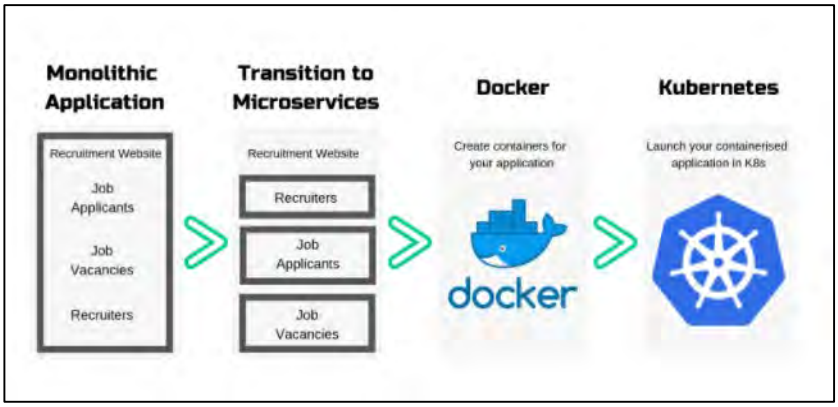
Berikut beberapa keunggulan Kubernetes :

1. *Service discovery* dan *load balancing*, dapat melacak kontainer dengan otomatis. *Load balancer* sebagai pengelolaan *traffic*.
2. *Storage orchestration*, dapat melakukan *mounting* pada *file system*.
3. *Automated rollouts* and *rollbacks*, dapat mengembalikan *deploy* sebelumnya ketika apabila terjadi kesalahan pada saat *deployment*.
4. *Automatic bin packing*, dapat mengatur utilisasi sumber daya CPU, RAM pada setiap kontainer.
5. *Self healing*, Kubernetes memiliki kemampuan untuk memeriksa kontainer yang ada, apakah dalam keadaan *running* atau mengalami *error*. Fitur ini penting bagi aplikasi yang terus berjalan selama 24 jam.

6. *Secret and configuration management*, Kubernetes memungkinkan untuk menyimpan data-data sensitif seperti *password, auth token* hingga SSH *keys* ke dalam Kubernetes *Secret*. Hal ini tentu jauh lebih aman dibanding menyimpannya di dalam kontainer *image*.

2.6.4 Hubungan Docker, *Microservice,* dan Kubernetes

Keterhubungan antara *microservice*, Docker dan Kubernetes yaitu *microservice* bertujuan secara konseptual untuk memecah aplikasi menjadi layanan-layanan kecil yang kemudian disimpan dengan membungkus aplikasi menggunakan Docker berbentuk kontainer *image*. Pada sebuah layanan, misal layanan produk dapat memiliki kontainer *image* ini dengan jumlah yang banyak, *Kubernetes*-lah dengan cara orkestrasi yang berfungsi untuk mengelola kontainer tersebut sesuai kebutuhan, menambah *service* atau mengurangi *service* sesuai aturan metrik yang dibuat.



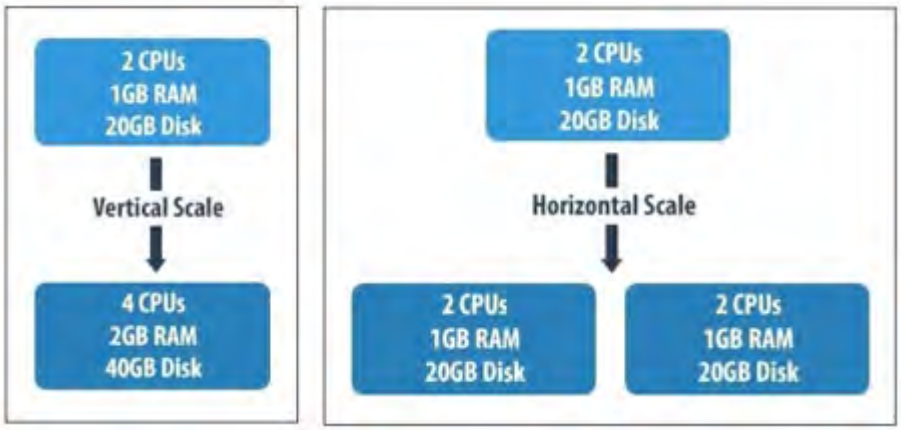
Gambar 2.9
Hubungan Antara *Microservice*, Docker dan Kubernetes
Sumber : *An Introduction to Kubernetes* (2019)

Docker bisa menjadi bagian dari Kubernetes namun tidak dapat sebaliknya. Dari kumpulan service yang telah terpecah dibungkus dengan berupa kontainer oleh Docker. Kubernetes berperan untuk mengelola kontainer yang dihasilkan oleh Docker, atau bisa disebut sebagai kontainer manajer. Kubernetes dapat mengelola banyak kontainer, membuat otomatisasi sesuai kebutuhan dan melakukan orkestrasi terhadap perubahan kontainer.

2.6.5 Horizontal Pod Autoscaler (HPA)

Aplikasi yang berjalan di pod dapat diskalakan secara manual dengan meningkatkan bidang replika di ReplicationController, ReplicaSet, Deployment, atau sumber daya lainnya. Pod juga dapat diskalakan secara vertikal dengan meningkatkan permintaan dan batas sebuah *resource container*, walaupun saat ini hanya dapat dilakukan pada waktu pembuatan pod, tidak saat pod sedang berjalan. Meskipun scaling manual tidak masalah untuk saat-saat ketika dapat mengantisipasi lonjakan beban sebelumnya atau ketika beban berubah secara bertahap dalam jangka waktu yang lebih lama, memerlukan intervensi manual untuk menangani peningkatan lalu lintas yang tiba-tiba dan tidak terduga tidaklah ideal.

Horizontal pod *autoscaling* adalah otomatisasi *scaling* dengan menambah atau mengurai jumlah pod yang dikelola oleh *controller*. Ini dilakukan oleh *controller* Horizontal, yang diaktifkan dan dikonfigurasi dengan membuat sumber daya HorizontalPodAutoscaler (HPA). *Controller* secara berkala memeriksa metrik pod, menghitung jumlah replika yang diperlukan untuk memenuhi nilai metrik target yang dikonfigurasi di sumber daya HorizontalPodAutoscaler, dan menyesuaikan bidang replika pada sumber daya target (Deployment, ReplicaSet, ReplicationController, atau StatefulSet)



Gambar 2.10
Vertical Scaling dan H*orizontal Ccaling*
Sumber : MongoDB High Availability (Mehrabani, 2014)

Vertical scaling adalah upaya untuk meningkatkan kapabilitas dari *single* server seperti menambah RAM dan lain sebagainya. Sedangkan Horizontal *scaling* adalah upaya

untuk meningkatkan kapabilitas dari banyak server di mana masing2 dari server tersebut tidak memiliki banyak perubahan dari segi spesifikasi. Vertikal *scaling* terbatas pada kapasitas satu mesin, scaling di luar kapasitas itu dapat melibatkan waktu henti dan memiliki batas keras atas, yaitu skala perangkat keras tempat server sedang berjalan. Secara teori, menambahkan lebih banyak mesin ke *poll* yang ada berarti tidak terbatas pada kapasitas satu unit server, sehingga memungkinkan untuk scaling dengan *downtime* yang lebih sedikit.

2.7 REST API

Representational state transfer (REST) adalah adalah seperangkat aturan yang menentukan bagaimana aplikasi atau perangkat dapat terhubung dan berkomunikasi satu sama lain (Fielding & Taylor, 2002). Protokol yang sering digunakan oleh REST adalah berupa HTTP.

API merupakan singkatan dari *application programming interfaces*, yang secara singkatnya API bisa digunakan oleh *developer* untuk mengizinkan dan mengintegrasikan aplikasi yang berada di platform yang berbeda ataupun perangkat yang berbeda untuk bisa saling terkoneksi antara satu dengan yang lainnya (Reddy, Martin, 2011). Tujuan penggunaan API lainnya yaitu untuk mempercepat proses pengembangan aplikasi dengan cara menyediakan sebuah *function* yang terpisah sehingga para developer tidak perlu lagi membuat fitur yang serupa

Adapun metode HTTP yang secara umum dipakai dalam REST API adalah GET, POST,

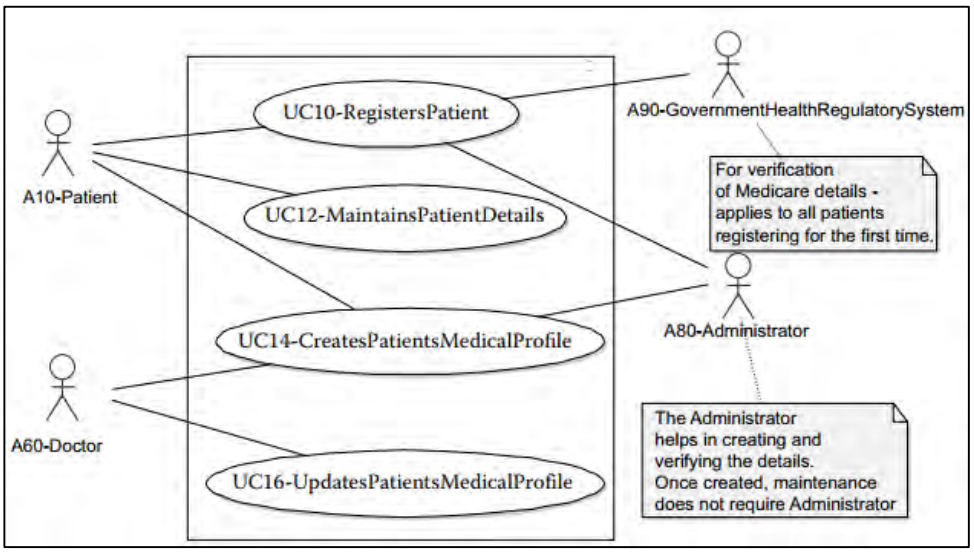
PUT, DELETE, dan OPTIONS. Untuk HTTP *Response Code* adalah kode standarisasi dalam menginformasikan hasil *request* kepada *client*. Secara umum terdapat 3 kelompok yang biasa kita jumpai pada RESTful API yaitu :

1. 2XX : adalah *response code* yang menampilkan bahwa *request* berhasil.
2. 4XX : adalah *response code* yang menampilkan bahwa *request* mengalami kesalahan pada sisi *client*.
3. 5XX : adalah *response code* yang menampilkan bahwa *request* mengalami kesalahan pada sisi server.

2.8 *Unified Modeling Language* (UML)

UML adalah himpunan struktur dan teknik untuk pemodelan desain program berorientasi objek (OOP) serta aplikasinya (Kroenke, 1977) . Tujuan Penggunaan UML sendiri kurang lebih adalah sebagai berikut :

1. Memberikan bahasa pemodelan yang bebas dari berbagai bahas pemrograman dan proses rekayasa.
2. Menyatukan praktik-praktik terbaik yang terdapat dalam pemodelan.
3. Memberikan model yang siap pakai, bahasa pemodelan visual yang ekspresif untuk mengembangkan dan saling menukar model dengan mudah dan dimengerti secara umum.
4. UML bisa juga berfungsi sebagai sebuah (*blueprint*) cetak biru karena sangat lengkap dan detail. Dengan cetak biru ini maka akan bias diketahui informasi secara detail tentang kode program atau bahkan membaca program dan menginterpretasikan kembali ke dalam bentuk diagram (*reserve engineering*).



Gambar 2.11
Use Case Diagram Pada Sistem Informasi Rumah Sakit
Sumber : *Software Engineering with UML*, (Unhelkar, 2017)

Beberapa notasi penting dalam UML pada use case diagram adalah untuk mendefinisikan *actor*, *use case* ada pada gambar 2.11. System *boundary* diperlukan untuk menandai cakupan sebuah sistem, apakah sebuah *use case* atau aktor itu masuk ke dalam atau di luar sistem.

2.8.1 *Use Case* Diagram

Use Case Diagram adalah satu jenis dari diagram UML yang menggambarkan hubungan interaksi antara sistem dan aktor. *Use case* dapat mendeskripsikan tipe interaksi antara si pengguna sistem dengan sistemnya. *Use case* merupakan sesuatu yang mudah dipelajari. Langkah awal untuk melakukan pemodelan perlu adanya suatu diagram yang mampu menjabarkan aksi aktor dengan aksi dalam sistem itu sendiri, seperti yang terdapat pada u*se case.*

Use case diagram mempunyai 3 komponen ,yaitu :

1. Sistem : Menyatakan batasan sistem dalam relasi dengan aktor-aktor yang menggunakannya di luar sistem dan fitur-fitur yang harus disediakan dalam sistem.
2. Aktor : segala hal di luar sistem yang akan menggunakan sistem tersebut untuk melakukan sesuatu. Bisa merupakan manusia, sistem, atau perangkat yang memiliki peranan dalam keberhasilan operasi dari sistem.
3. *Use case* sendiri adalah gambaran fungsional dari sebuah sistem. Dengan demikian, antara konsumen dan juga pengguna pada sistem tersebut, akan mengerti atau paham mengenai fungsi sistem yang tengah dibangun.

2.9 Penelitian Terdahulu

Pada proses penelitian telah dilakukan terlebih dahulu observasi terhadap penelitian terdahulu yang berguna sebagai salah satu bahan acuan masukan. Melihat dari hasil penelitian dahulu berguna sebagai pembelajaran dan mendapatkan sudut pandang yang lain untuk mencapai solusi usulan yang diharapkan.

Pada Tabel 2.3 terdapat beberapa penelitian terdahulu yang dijadikan observasi baik metode penelitian ataupun objek penelitian. Terdapat kolom perbedaan setiap judul penelitian yang membahas tentang perbedaan yang spesifik antara penelitian terdahulu dengan penelitian ini.

Tabel 2.3
Hasil Penelitian Terdahulu

Judul Penelitian	**Obyek Penelitian**	**Fokus Penelitian**	**Perbedaan**
Analisa dan Implementasi *Microservice* pada *Container* Menggunakan *Docker* Stefanus Eko Prasetyo, Ardyansyah Wijaya (2021)	*Microservice*	Pengujian stabilitas Kemudahan *docker* pada *microservice*	Tidak ada pengujian skalabilitas dengan *scaling* dan Kubernetes
Pengamanan *Container Orchestration* Berbasis Kubernetes di Lembaga Penerbangan dan Antariksa Nasional. Arief Indriarto Haris, Rd. Angga Ferianda, Budhi Riyanto, Fajar Iman Nugraha, Januar Abadi (2021)	LAPAN	Fokus pada sisi *security* dari *Container Orchestration*	Beda subjek penelitian, dan berfokus pada masalah keamanan
Analisis Model Arsitektur *Microservice* Pada Sistem Informasi DPLK. Ghifari Munawar, Ade Hodijah (2018)	SI DPLK	Refactor	Tidak ada pengujian skalabilitas dengan *scaling* dan *orchestration*
A development process of enterprise applications with microservice F H Vera-Rivera (2018)	*Enterprise Application*	*Refactor* dengan *monitoring* dan *fault tolerant tools*	Tidak ada pengujian skalabilitas dengan *scaling*
Arsitektur *Microservice* untuk Resiliensi Sistem Informasi Hatma Suryotrisongko (2017)	Sistem Informasi	Refactor	Tidak ada pengujian skalabilitas dengan *scaling* dan orchestration

Perbedaan dari beberapa penelitian terdahulu adalah belum adanya pengujian bahwa dengan menggunakan arsitektur *microservice* menjadikan aplikasi yang mendukung skalabilitas. Pada penelitian terdahulu juga belum menggunakan *container orchestration.* Penggunaan Kubernetes telah dilakukan pada penelitian sebelumnya, namun belum menggunakan teknik Horizontal Pod Autoscaler (HPA yang dapat mengotomatisasi proses skalabilitas baik *scaling up* maupun *scaling down.*

BAB III

OBJEK DAN METODE PENELITIAN

3.1 Aplikasi Sapawarga

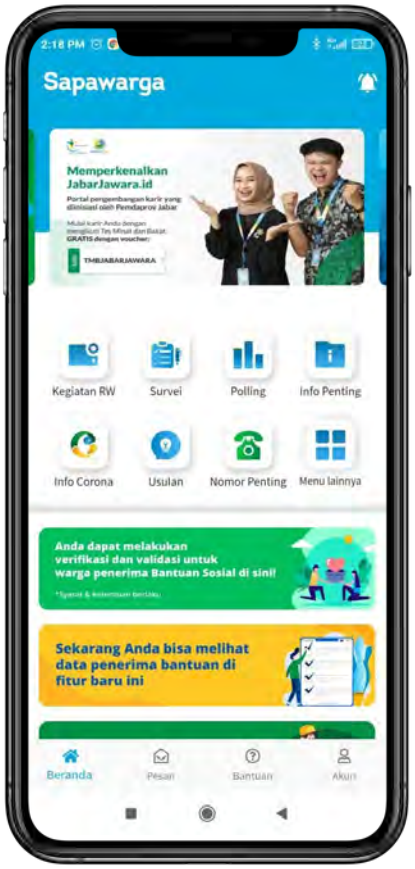
Aplikasi Sapawarga dikembangkan oleh Jabar Digital Service (JDS) atau Unit Pelayanan Teknis Pelayanan Digital, Data, dan Informasi Geospasial di bawah Dinas Komunikasi dan Informatika Pemerintah Provinsi Jawa Barat. Sapawarga merupakan aplikasi satu pintu untuk memudahkan warga Jabar berkomunikasi dengan pemerintah daerah melalui berbagai fitur yang dapat digunakan oleh pengurus RW. Tujuan Sapawarga ini adalah untuk mempermudah masyarakat (RW) untuk akses layanan publik, aspirasi, informasi, berpartisipasi, dan berinteraksi dengan RW lainnya.

Saat ini, pengguna Sapawarga adalah pengurus RW yang ada di wilayah Provinsi Jawa Barat. Aplikasi ini adalah inisiatif dan salah satu upaya untuk menjadikan Jawa Barat sebagai Provinsi Digital, bermula dari lingkup RW. Berdasarkan data statistik keaktifan pengguna Sapawarga RW dan Desa, jumlah RW yang pernah *login* ke Sapawarga per 16 November 2020 sebanyak 40.381 atau 69,3 persen (persentase terhadap jumlah RW). Secara bulanan pengguna aktif Sapawarga mencapai 17.426 per Oktober 2020.

Gambar 3. 1
Fitur Aplikasi Sapawarga

Pada aplikasi Sapawarga memiliki tiga tujuan yang ingin dicapai secara umum, yaitu sebagai :

1. Media Informasi dan Komunikasi : Sebagai wadah komunikasi langsung lintas pemerintahan (Pemerintah ke masyarakat / *top-down*). Sumber informasi umum yang valid, *update*, dan diverifikasi dari pemerintah dan sesuai target *audiens*
2. Wadah Aspirasi : Menampung aspirasi dan laporan dari berbagai jenis lapisan masyarakat dan sebagai media untuk memverifikasi berita *hoaks*
3. Pelayanan Publik Digital : Sebagai jalan pintas ke layanan publik daring lainnya seperti aplikasi SAMBARA untuk pembayaran pajak daring. Efisiensi dan transparansi pelayanan administrasi publik (e-KTP, akta kelahiran, surat keterangan, pajak kendaraan bermotor)



Gambar 3. 2
Tampilan Antarmuka Sapawarga

Berikut Fitur-fitur Sapawarga:

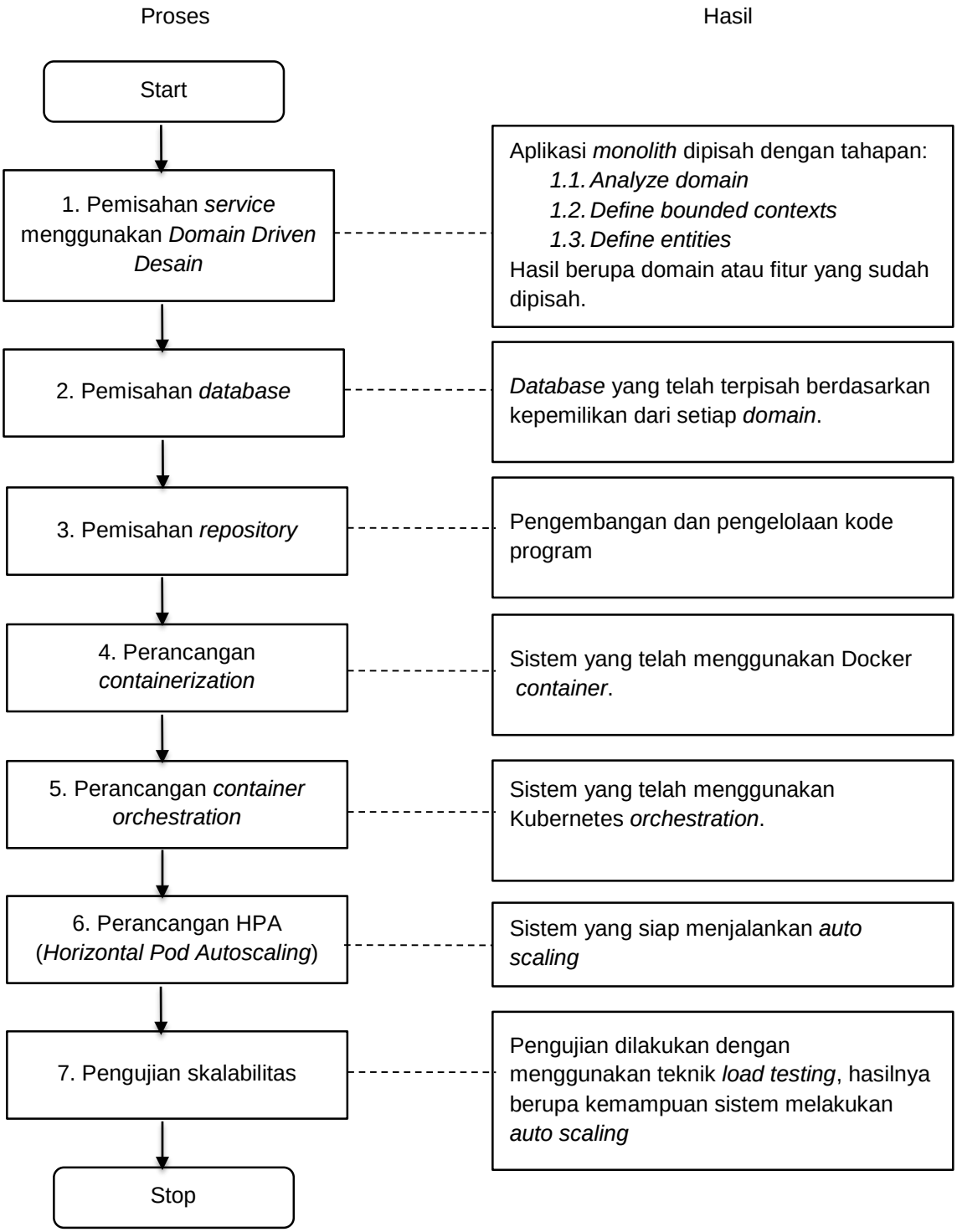
1. Kegiatan RW : RW dapat saling membagikan kegiatan lewat foto, dan juga dapat berinteraksi lewat kolom komentar.
2. Usulan : RW dapat menyampaikan aspirasi usulan.
3. Survei : RW dapat berpartisipasi dalam survei yang dibuat pemerintah.
4. *Polling* : RW dapat mengikuti *polling* yang diadakan oleh pemerintah.
5. Berita : RW mendapatkan berita-berita aktual dari portal dan sumber terpercaya.
6. Info Penting : Dapatkan info lowongan kerja, beasiswa, jadwal Samsat keliling.

Pada tahap pengembangan berikutnya, Sapawarga diproyeksikan agar dapat digunakan oleh seluruh warga Jawa Barat, dengan begitu manfaat teknologi akan lebih terasa mempermudah kehidupan masyarakat. Jawa barat merupakan provinsi dengan jumlah penduduk terbanyak di Indonesia sebanyak 5,42 juta (data BPS tahun 2021).

Pada saat ini, fitur-fitur pada aplikasi pun akan terus bertambah mengingat banyaknya masukan dari masyarakat juga dari para *stakeholders* untuk menciptakan solusi yang lebih inovatif. Dengan banyaknya fitur yang akan muncul di Sapawarga, maka dari itu Sapawarga diproyeksikan juga menjadi *superapp* di Jawa Barat.

3.2 Metode Penelitian

Penelitian ini dilakukan dengan melakukan 7 tahapan dari mulai pemisahan *monolith* menjadi *microservice* sampai pada akhirnya melakukan pengujian skalabilitas pada aplikasi untuk membuktikan bahwa dengan menggunakan *microservice* dan juga tools yang tepat dapat menjadikan solusi bagi aplikasi yang membutuhkan aspek skalabilitas tinggi. Berikut tahapan-tahapan proses perancangan perubahan dari *monolith* ke *microservice* ditunjukkan dalam Gambar 3.3.



Gambar 3. 3
Tahapan Metode Penelitian

3.2.1 Pemisahan *Service* Menggunakan *Domain Driven Design* (DDD)

Untuk memisahkan setiap *service* salah satunya dengan menggunakan pendekatan DDD. Untuk mendefinisikan DDD, pertama-tama harus menetapkan apa yang kita maksud dengan *domain* dalam konteks aplikasi. Definisi kamus umum tentang *domain* adalah lingkungan pengetahuan atau aktivitas. Menelusuri sedikit dari itu, *domain* di ranah rekayasa perangkat lunak biasanya mengacu pada area subjek di mana aplikasi tersebut dimaksudkan untuk diterapkan. Artinya dengan kata lain, selama pengembangan aplikasi, *domain* adalah lingkup pengetahuan dan aktivitas di mana logika aplikasi berputar.

Ambiguitas pada sebuah domain sangat mungkin terjadi karena berbedanya sudut pandang antara pengembang, bisnis, dan dari pengguna. Dibutuhkan sebuah kesepakatan nama domain yang bisa menjadi acuan untuk memisahkan setiap domain sehingga fokus pada masalah yang akan diselesaikan.

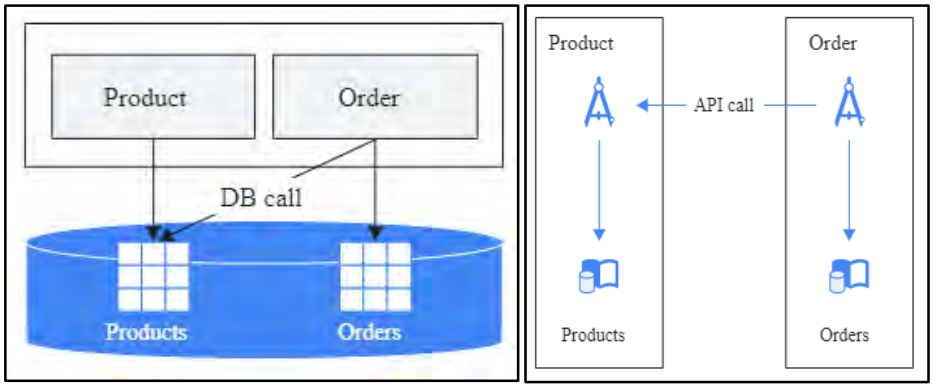
Istilah umum lainnya yang digunakan selama pengembangan perangkat lunak adalah lapisan *domain* atau logika *domain*, yang mungkin lebih dikenal oleh banyak pengembang sebagai logika bisnis. Logika bisnis aplikasi mengacu pada aturan tingkat yang lebih tinggi tentang bagaimana objek bisnis berinteraksi satu sama lain untuk membuat dan memodifikasi data yang dimodelkan.

3.2.2 Pemisahan *Database* Setiap *Service*

Beberapa *pattern microservice* ada yang menggunakan *database* secara berbagi dan ada juga yang terpisah berdasarkan layanannya. Untuk meningkatkan fleksibilitas baiknya setiap *database* khusus untuk menyimpan data dari sebuah service. *Database* terpisah memiliki keuntungan setiap *database* dapat berdiri sendiri, mengurangi ketergantungan pada *database* lain, jika pun ada komunikasi data dilakukan antar *service* melalui protokol REST atau RPC.

Teknik untuk melakukan pemisahan *database* adalah dengan menghilangkan *foreign key*, *constraint* yang biasanya sebagai penghubung antara tabel. Pemisahan dilakukan dengan memisahkan berdasarkan kepemilikan data, misal tabel produk adalah kepunyaan hanya *service* produk. Fitur lain yang membutuhkan data pengguna harus

berkomunikasi via *service* tidak boleh langsung *service* lainnya mengambil data pengguna langsung ke *database*.



Gambar 3. 4
Komunikasi Data Sebelum dan Setelah *Database* Dipisah
Sumber : https://cloud.google.com/architecture/microservices-architecture-refactoring-monoliths

3.2.3 Pemisahan *Repository Source Code*

Microservice juga dapat menyelesaikan masalah pada organisasi yang besar. Sebuah tim yang mempunyai jumlah yang besar akan kesulitan mengembangkan aplikasi dengan hanya sebuah *repository* untuk menyimpan kode. Setiap *service* yang telah dipecah sebaiknya disimpan dengan *repository* yang berbeda antara satu dengan yang lainnya. Tujuan pemisahaan repository untuk memudahkan pengembangan aplikasi dengan tidak tercampurnya kode dan tidak saling tergantung secara keseluruhan. *Repository* yang umum dipakai adalah Github, Gitlab, Bitbucket. *Repository* dapat berupa disimpan sebagai *project private* ataupun publik.

Setiap layanan *repository* juga dapat melakukan proses CI / CD (*Continuous Integration, Continuous Delivery*) *pipeline*. CI / CD merupakan metode untuk mengirimkan aplikasi ke pelanggan secara rutin dengan menggunakan proses otomatisasi ke dalam tahapan pengembangan aplikasi.

3.2.4 Perancangan Docker *Container*

Salah satu *best practice* pada arsitektur *microservice* adalah penggunaan kontainer. Penggunaan kontainer sangat mempermudah ketika sebuah aplikasi dalam

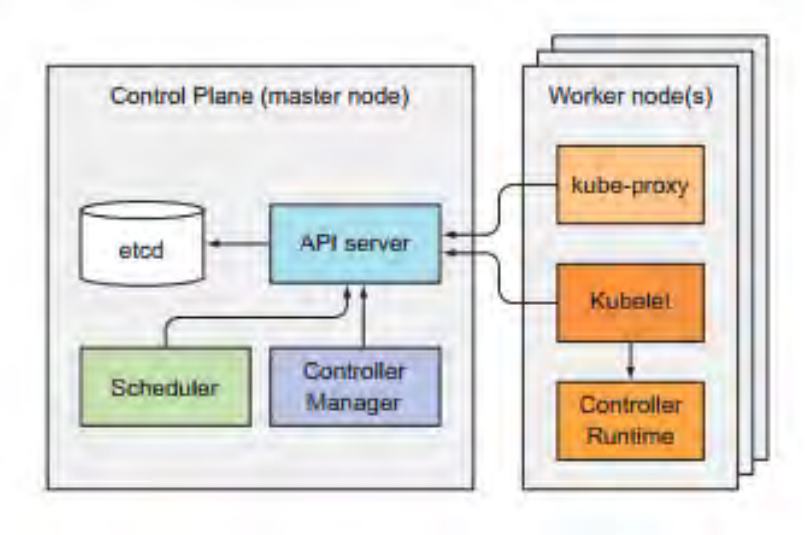
pengembangan dengan tidak ada isu lagi tentang di komputer lokal jalan, di server *production* aplikasi tidak jalan karena perbedaan *environment*.

Perancangan Docker ini berfokus pada penggunaan standarisasi *environment* yang digunakan baik pada saat pengembangan dan juga pada saat di *production*. Langkah-langkah untuk melakukan perancangan kontainer dengan Docker adalah sebagai berikut :

1. *Install* Docker pada sebuah *server* beserta Docker Compose.
2. Pembuatan Dockerfile pada setiap *service*, langkah ini adalah menentukan *environment* apa saja yang dibutuhkan. Kebutuhan *library* pada sebuah bahasa pemrograman ataupun library yang dibutuhkan aplikasi pada sebuah OS, menentukan web server bisa dilakukan di sini.
3. Pembuatan Docker Compose untuk melakukan *build image*. Pada tahapan ini dilakukan penentuan pemakaian nama kontainer, nomor *port* yang digunakan, *network*, lokasi dan penyimpanan *storage* pada *volume*.

3.2.5 Perancangan *Container Orchestration* Kubernetes

Melakukan persiapan *server master* yang berfungsi sebagai *node master*, *master* di sini berfungsi untuk melakukan *deploy* kontainer ke dalam *node worker* Kubernetes. Persiapan selanjutnya adalah melakukan setup pada *server node* yang berfungsi sebagai *node worker.* Sebuah *node* dapat berupa VM atau server fisik tergantung dari *kluster*-nya. Setiap *node* berisi beberapa *object* seperti *pod*, *service*, *volume*, *namespace* dan lainnya yang diatur oleh komponen-komponen yang dimiliki oleh *master node*.



Gambar 3. 5
Topologi *Master Node* dan *Worker Node*
Sumber : Kubernetes *in Action*, (Luksa, 2017)

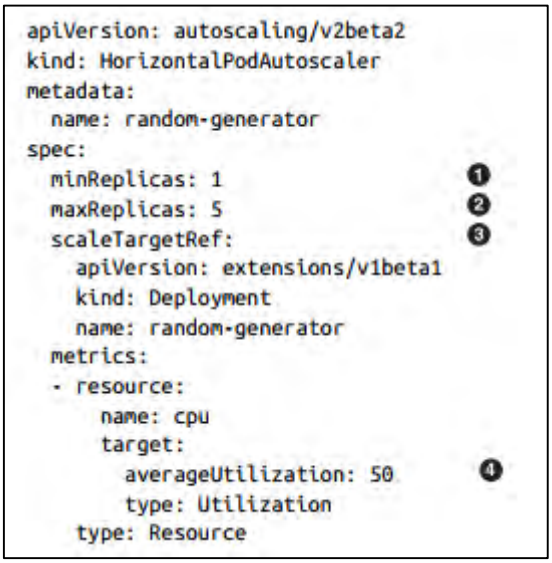
3.2.6 Perancangan *Horizontal Pod Autoscaler* (HPA) Kubernetes

Salah satu manfaat utama yang didapat dari Kubernetes adalah orkestrasi tingkat tinggi pada banyak kontainer. Secara khusus, kemampuan untuk secara otomatis *scaling* dan mengatur beban kerja yang dikerahkan pada sebuah *server* berjalan dapat menghilangkan kesulitan mengatur banyak kontainer pada sebuah aplikasi. Secara teori, kita dapat menentukan beberapa parameter yang mendorong aktivitas *scaling* dengan mendefinisikan *matrik*, lalu dengan otomatis Kubernetes melakukan pekerjaannya sebagai container *orchestration manager* yang dapat menambah atau mengurai kontainer berdasarkan metrik yang telah dicapai. Jika tidak memiliki otomatisasi ini, maka seorang *devops* atau bagian infrastruktur harus menjalankan replika secara manual dan ini bisa sebanyak puluhan bahkan ratusan kali tergantung banyaknya beban *server* terpenuhi.

Perancangan HPA sendiri adalah dengan mulai membuat sebuah *server* metrik API dan *server* metrik. *Server* ini bertugas untuk menyimpan dan melakukan agregat dengan mengumpulkan CPU metrik terkait memori dari Kubelet yang berjalan pada setiap *node* klaster pada interval reguler (secara *default*, setiap menit). *Server* metrik berjalan di klaster Kubernetes sama seperti beban kerja lainnya dan metrik yang dikumpulkan selanjutnya diekspos melalui metrik API untuk konsumsi pengontrol HPA.

Ketika server metrik sudah siap, selanjutnya adalah menyiapkan konfigurasi berupa *file* dengan extensi .yml (YAML). Gambar 3.X adalah contoh konfigurasi HPA yang akan secara otomatis melakukan *scaling-up* maksimal sebanyak 20 kali ketika sumber daya berupa CPU menyentuh pemakaian rata-rata seluruh *server* sebanyak 60%.

Pada Gambar 3.5 telah dilakukan *setup* orkerstrasi *container* dengan menggunakan Kubernetes. Penjelasan dari *script* tersebut adalah memasang minimal 1 replika dan maksimal 5 replika *pod.* Server *pod* akan bertambah otomatis sebanyak maksimal 5 buah Ketika matrik pemakaian CPU sudah melebihi 50 %.



Gambar 3. 6
Contoh Konfigurasi HPA Kubernetes
Sumber : Kubernetes *Patterns Reusable Elements for Designing Cloud-Native Applications* (Ibryam & Huß, 2019)

3.2.7 Pengujian Skalabilitas Dengan *Load Testing*

Load testing adalah proses menempatkan *request* pada sebuah sistem sistem dan kemudian mengukur waktu respon dari sebuah *request*. *Load testing* dilakukan untuk mengetahui perilaku sistem di bawah kedua kondisi beban puncak normal dan diantisipasi. Ini membantu untuk mengidentifikasi kapasitas operasi maksimum aplikasi serta adanya kemacetan dan menentukan unsur yang menyebabkan degradasi. *Load testing* biasanya jenis pengujian non fungsional meskipun dapat digunakan sebagai uji fungsional untuk memvalidasi kesesuaian sistem untuk digunakan.

Pengujian sistem dilakukan dengan menentukan beban yang sanggup ditangani oleh sebuah aplikasi. Beratnya beban yang sanggup ditangani oleh sebuah aplikasi direpresentasikan dengan banyaknya pengguna mengakses sebuah aplikasi dalam waktu tertentu atau bisa juga merepresentasikan beratnya sebuah permintaan respon data.

Pengujian skalabilitas adalah melakukan testing pada sebuah aplikasi dengan sejumlah *request* sampai pada metrik yang ditentukan. Pengujian bisa dengan menjalankan *request* sebanyak 100 kali yang dapat membuat utilisasi CPU naik menjadi 70%, apa yang terjadi ketika melakukan *request* sebanyak 200 kali, apakah sistem akan melakukan *scaling* atau tidak. Aspek skalabilitas yang baik sistem akan membuat replikasi *server* sampai rata-rata utilisasi *server* kembali di bawah 70%.

BAB IV

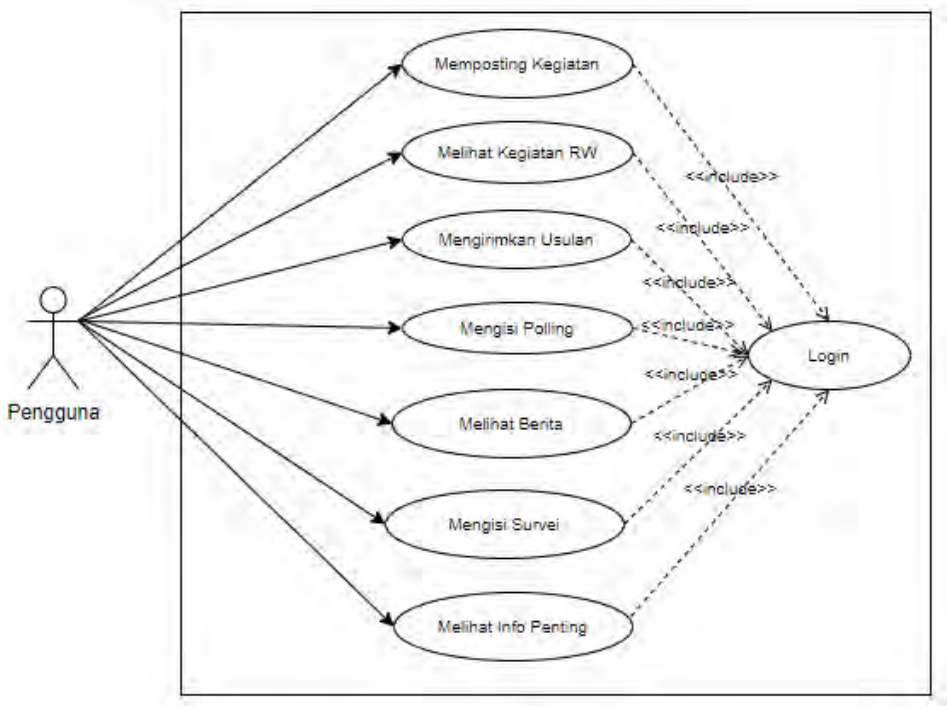
PERANCANGAN HASIL DAN PEMBAHASAN

4.1. Pemisahan *Service* Menggunakan *Domain Driven Design (DDD)*

Pemisahan Service menggunakan DDD dilakukan untuk memecah *service* sesuai dengan domain yang sama, Service dengan domain yang sama akan mempermudah proses bisnis sebuah fitur, karena sebuah domain hanya akan fokus mengembangkan fitur pada domain tersebut dan sebisa mungkin tidak berhubungan dengan domain lain.

4.1.1. *Analyze Domain*

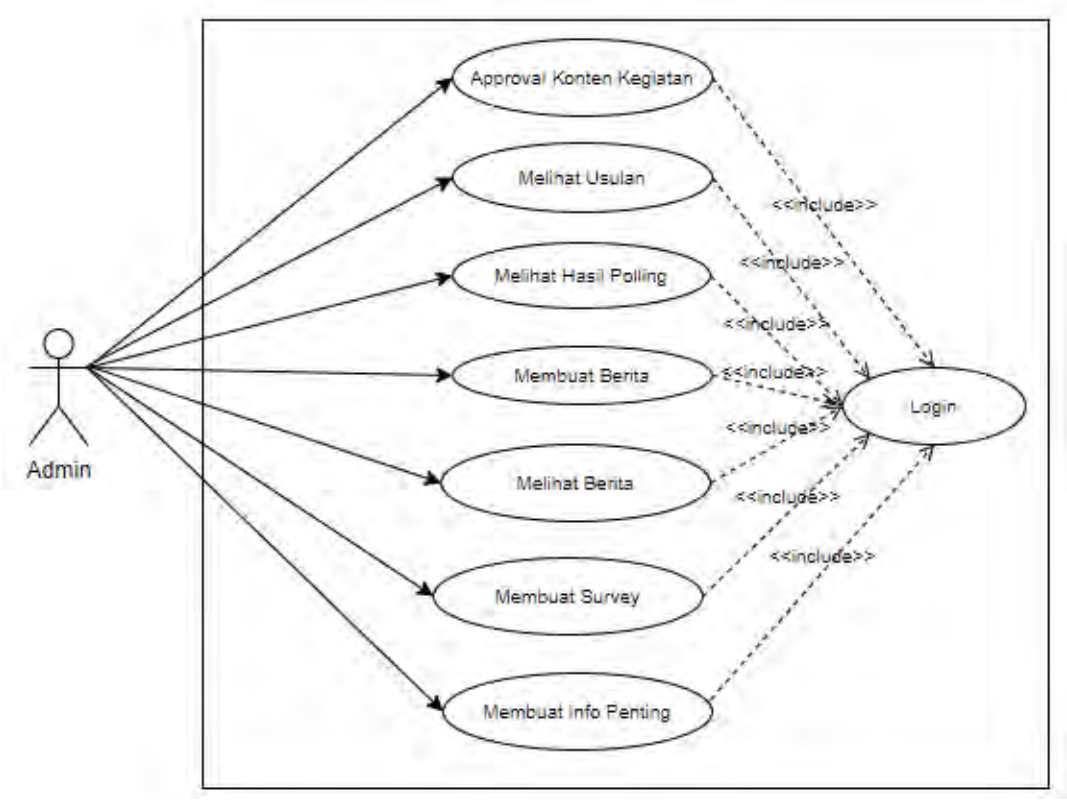
Pada tahap ini akan dilakukan *Analyze Domain* dengan mulai memetakan semua fungsi bisnis dan koneksinya. Melihat fungsi bisnis bisa dilakukan menggunakan *use case diagram,* di mana sudut pandang dilakukan berdasarkan sudut pandang pengguna, admin dan eksekutif. Membuat kolaborasi yang melibatkan *expert*, *developer*, dan pemangku kepentingan lainnya. Hasil berupa keseluruhan *domain* yang ada pada aplikasi *existing*.



Gambar 4. 1
Use Case Diagram Dari Sudut Pandang Pengguna

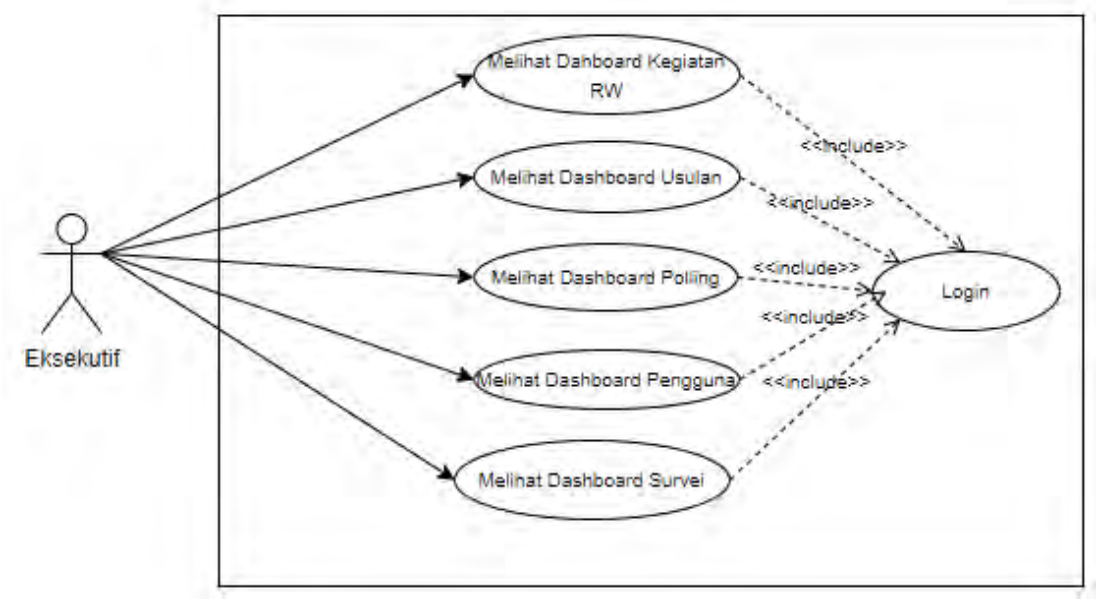
Sudut pandang pengguna seperti terlihat pada Gambar 4.1. Pengguna yang dimaksud adalah dengan *role* sebagai masyarakat umum yang dapat mengakses fitur-fitur

utama dengan diharuskan *login* terlebih dahulu. Adapun aksi yang bisa dilakukan Ketika berhasil *login* adalah dapat melihat semua daftar menu dan juga melakukan *posting* konten pada menu kegiatan, usulan dan *polling*.



Gambar 4. 2
Use Case Diagram dari Sudut Pandang *Admin* Sapawarga

Admin adalah orang yang bertugas sebagai pengelola konten secara *back office* dari Sapawarga. *Role admin* sendiri adalah orang internal dari organisasi. Adapun tugas-tugasnya terlihat seperti pada Gambar 4.2 sebagai *approval* dan *mediator* dari konten yang diterbitkan oleh pengguna (RW) jika ada konten yang tidak sesuai petunjuk, maka konten akan dinonaktifkan. Admin juga bertugas menerbitkan konten berupa berita, info penting, dan juga melihat hasil *polling*.



Gambar 4. 3
Use Case Diagram Dari Sudut Pandang Eksekutif

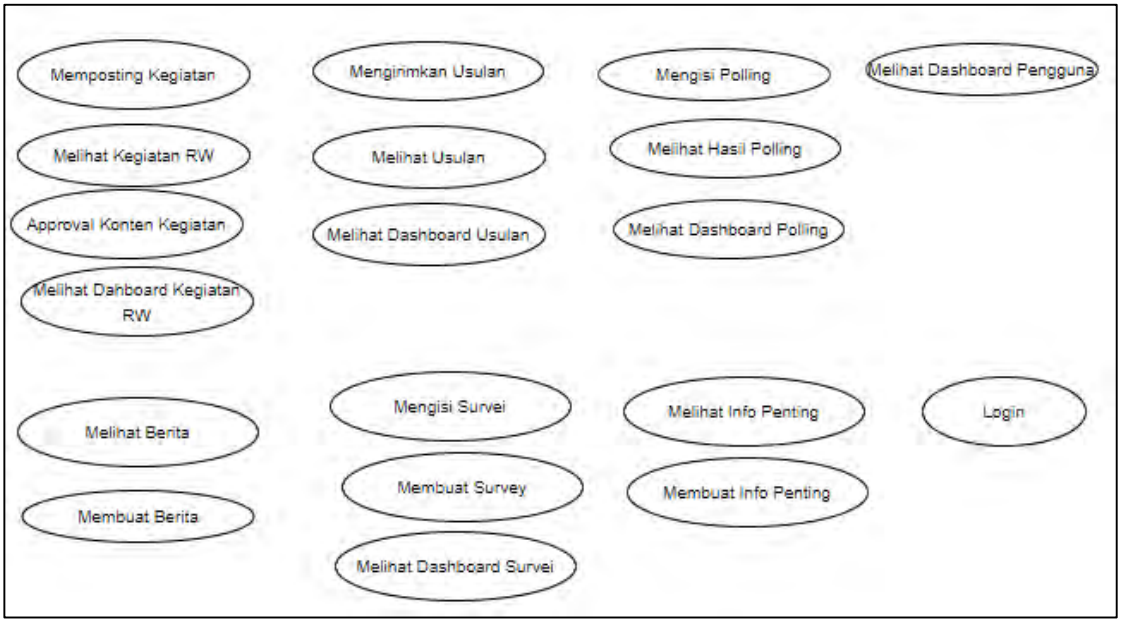
Role yang terakhir adalah *role* eksekutif yang ditujukan untuk para *stakeholder* melihat penggunaan Sapawarga. Eksekutif dapat melihat *dashboard* pengguna secara umum seperti jumlah pengguna baik secara demografi maupun secara gender. Para *stakeholder* dapat melihat usulan yang diberikan masyarakat dalam berbagai kategori misal : kesehatan, infrastruktur, pendidikan, dan usulan lainnya.

Usulan tersebut dapat dikalkulasi oleh aplikasi Sapawarga sehingga dapat menjadikan data yang berguna. yang dapat dijadikan landasan untuk membuat suatu kebijakan baru. Pada akhirnya *stakeholder* dapat membuat kebijakan yang benar-benar dibutuhkan masyarakat, begitu pula masyarakat mempunyai wadah untuk menyalurkan masukan kepada para *stakeholder*.

4.1.2. *Define Bounded Contexts*

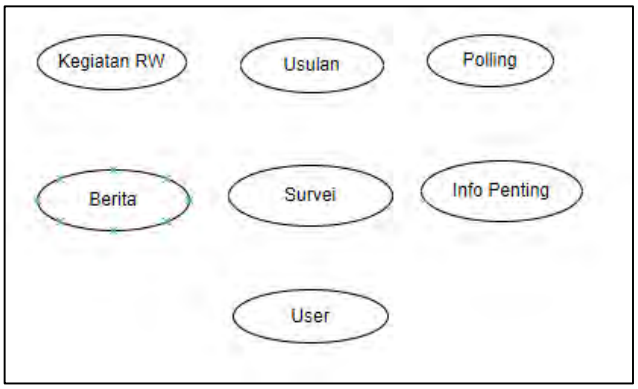
Tahap ini adalah tahapan untuk menentukan batasan konteks. Setiap konteks terikat berisi model *domain* yang mewakili *subdomain* tertentu dari aplikasi yang lebih besar. Mendefinisikan konteks bisa melihat dari semua use case atau proses bisnis yang terjadi. Semua proses bisnis dikumpulkan dari semua aktor, kemudian *use case* tersebut dikelompokkan berdasarkan kedekatan domain. Walaupun aksi sebuah use case

berbeda di setiap aktor, Ketika memiliki domain yang sama akan dikelompokkan menjadi satu domain.



Gambar 4. 4
Semua *Use Case* Dari Semua *Role*

Gambar 4.4 terlihat semua *use case* ketika digabungkan dari semua sudut pandang aktor dan dikelompokkan berdasarkan *domain* masing-masing. Ada dua buah domain yang penamaan fiturnya berbeda namun masih memiliki kedekatan data yaitu *dashboard* pengguna dengan *login*, yang bisa dijadikan sebuah *service* yang sama dengan sebutan domain *user*.

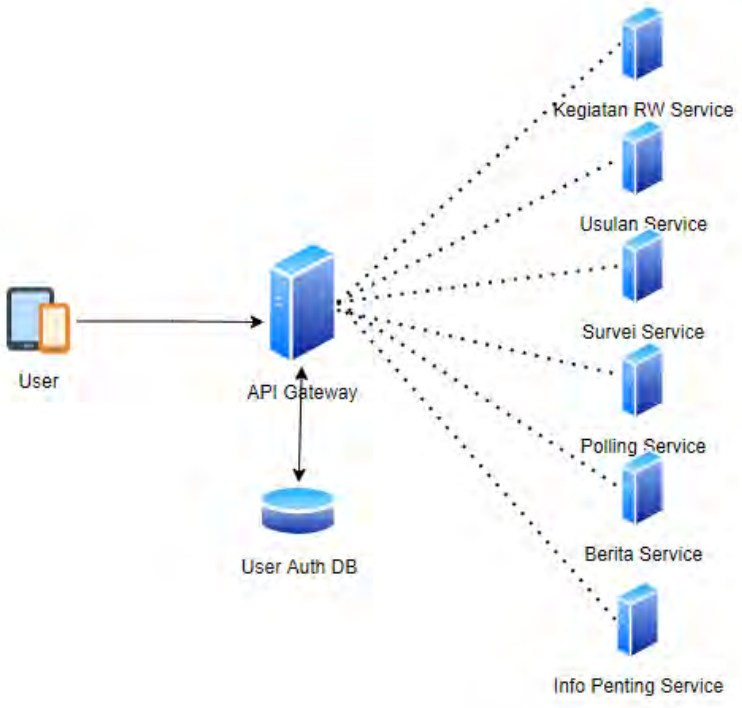


Gambar 4. 5
Pengelompokan *Domain* Berdasarkan Proses Bisnis

4.1.3. *Define Entities, Aggregates*, dan *Services*

Mendefinisikan entitas kemudian melakukan pengecekan untuk fungsi agregat apakah ada antara *service* yang terlibat. *Aggregate* dalam DDD merupakan sekelompok objek domain yang dapat diperlakukan satu unit. Dalam *aggregate*, dibuat satu kelas *root* yang memiliki identitas global untuk semua kelas dalam *agregate* tersebut, sedangkan kelas lainnya memiliki identitas lokal. Jika kita akan melakukan transaksi di kelas eksternal, kita hanya diizinkan untuk akses *reference* ke kelas *root* saja, tidak langsung ke *reference* kelas *child*.

Mendefinisikan *entities* dilakukan berdasarkan domain yang sudah dipisah berdasarkan pada Gambar 4.5. Kemudian adalah mendefinisikan *service* yang juga berdasarkan pengelompokan domain. Terlihat service yang akan dirancang adalah berjumlah 7 *service* dengan ditambahkan API *Gateway* yang bertugas mengarahkan *request* data berdasarkan masing-masing *service*.



Gambar 4. 6
Pemisahan *Service* Berdasarkan *Domain*

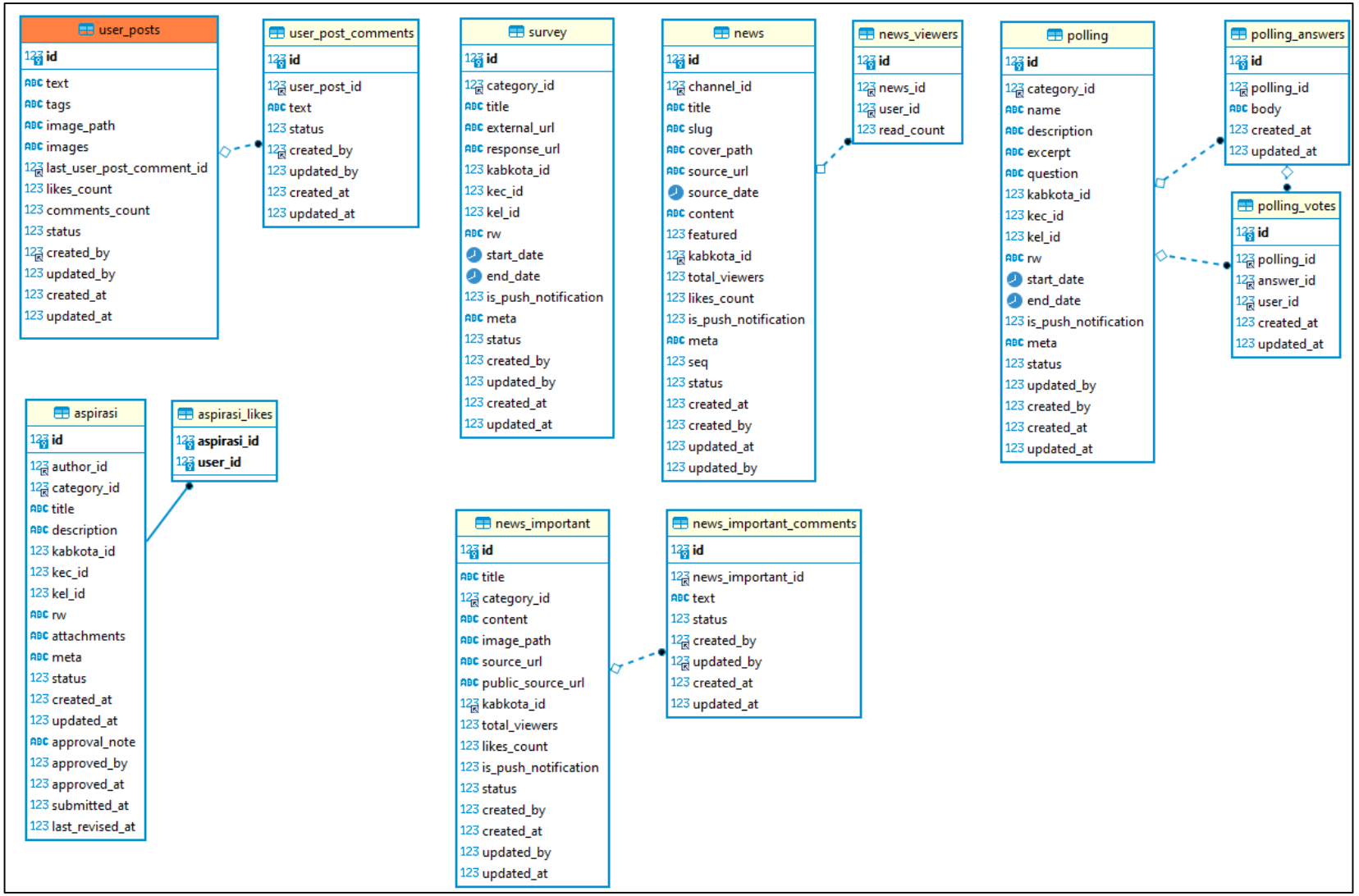
4.2. Pemisahan *Database* Setiap *Service*

Kondisi *database* saat ini adalah hanya menggunakan satu database yang digunakan untuk menyimpan seluruh data dari aplikasi. Pemisahan *database* dilakukan untuk memisahkan ketergantungan data dan memecah alur data sesuai tugas masing-masing *service*.

Pada Tabel 4.1 telah dilakukan pengelompokan antara domain, aksi, aktor dan nama tabel yang selanjutnya tabel-tabel *database* tersebut akan dibuatkan *database* baru berdasarkan domain masing-masing.

Tabel 4.1
Pengelompokan domain dengan nama table

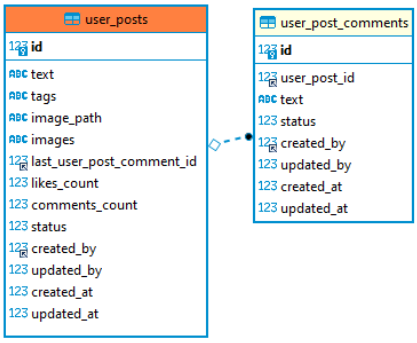
Domain	Aksi	Aktor	Nama Table
Kegiatan RW	Memposting Kegiatan	Pengguna	user_post user_post_comment
	Melihat Kegiatan RW	Pengguna	
	Approval Konten Kegiatan	Admin	
	Melihat Dashboard Kegiatan RW	Eksekutif	
Usulan	Mengirimkan Usulan	Pengguna	aspirasi
	Melihat Usulan	Admin	
	Melihat Dashboard Usulan	Eksekutif	
Polling	Mengisi Polling	Pengguna	polling polling_answers polling_votes
	Melihat Hasil Polling	Admin	
	Melihat Dashboard Polling	Eksekutif	
Berita	Melihat Berita	Pengguna	news news_viewer
	Membuat Berita	Admin	
	Melihat Berita	Admin	
Survei	Mengisi Survei	Pengguna	news
	Membuat Survey	Admin	
	Melihat Dashboard Survei	Eksekutif	
Info Penting	Melihat Info Penting	Pengguna	news_important news_important_comments
	Membuat Info Penting	Admin	



Gambar 4. 7
Database Fitur Sapawarga

Gambar 4.7 terlihat semua tabel yang menjadi bagian dari fitur-fitur Sapawarga yang akan dipecah *database* berdasarkan domain. Tidak adanya keterkaitan data yang pada setiap service memudahkan proses pemecahan *database*. Berikut usulan pemecahan *database* berdasarkan kepemilikan data :

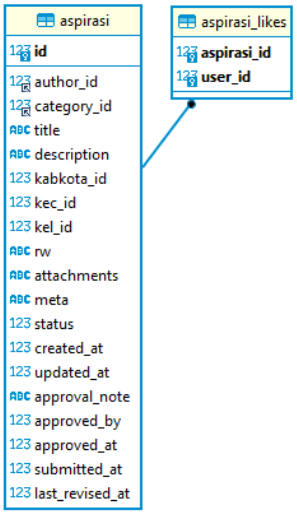
1. *Database* Kegiatan RW



Gambar 4. 8
Database Kegiatan RW

Gambar 4.8 merupakan *database* untuk *service* kegiatan rw yang di dalamnya mempunyai dua buah tabel yaitu : *user_post* untuk penyimpanan data kegiatan, dan *user_post_comment* untuk pengguna dapat memberikan komentar. Hubungan dari kedua buah tabel tersebut adalah *one to many*.

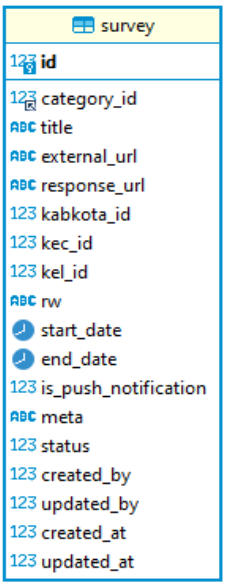
2. *Database* Usulan



Gambar 4. 9
Database Usulan

Gambar 4.9 merupakan *database* untuk *service* usulan. Pada *database* tersebut terdapat dua buat tabel yaitu : aspirasi, untuk menyimpan data semua usulan dari pengguna. Tabel *aspirasi_likes* untuk memberikan *likes* oleh pengguna kepada sebuah usulan. Hubungan antara kedua buah tabel tersebut adalah *one to many*, di mana sebuah usulan bisa mempunyai banyak *likes*.

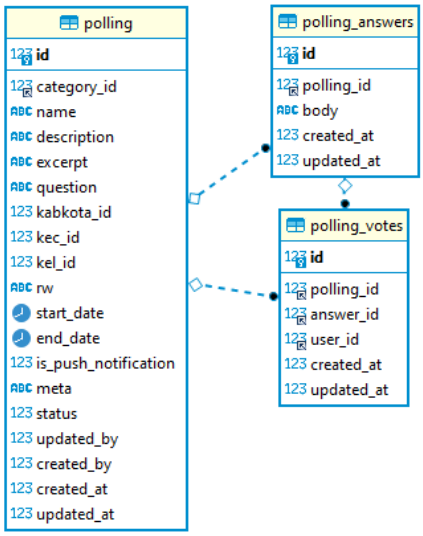
3. *Database* Survey



Gambar 4. 10
Database Survey

Gambar 4.10 merupakan *database* untuk untuk *service survey*. Pada *database survey* terdapat hanya sebuah tabel yaitu tabel *survey* yang digunakan untuk menyimpan data *survey*. Pada penggunaannya fitur *survey* ini hanya menampilkan *text* dan *link* yang kemudian *link* tersebut.diarahkan ke aplikasi *external* seperti Google *form*.Penempatan *link external* tersebut disimpan pada field *external_url.*

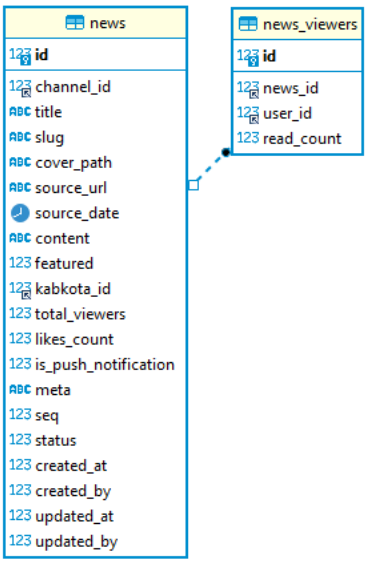
4. *Database* Polling



Gambar 4. 11
Database Polling

Gambar 4.11 adalah *database* untuk *service polling. Database polling* memiliki tiga buah tabel yaitu : *polling* untuk menyimpan data *polling*, *polling_answer* untuk menyimpan pilihan jawaban, terakhir *poling_votes* untuk menyimpan data pilihan *polling* dari pengguna. Relasi polling dengan answer dan *votes* adalah *one to many*.

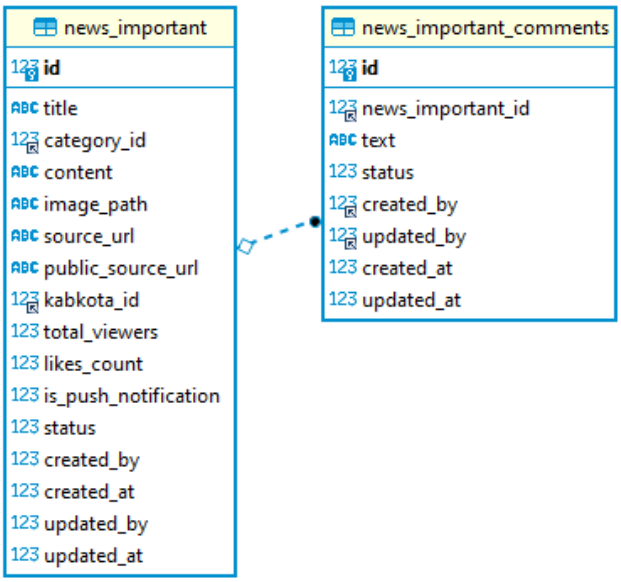
5. *Database* Berita



Gambar 4.12
Database Berita

Gambar 4.12 merupakan *database* untuk *service* berita. *Database* berita terdiri dari dua buah tabel yaitu : *news*, untuk menyimpan data berita dan *news_viewer* untuk mencatat jumlah pembaca pada berita tersebut. Hubungan antara *news* dan *news_viewer* adalah *one to many*, karena sebuah berita dapat dilihat banyak pengguna.

6. *Database* Info Penting

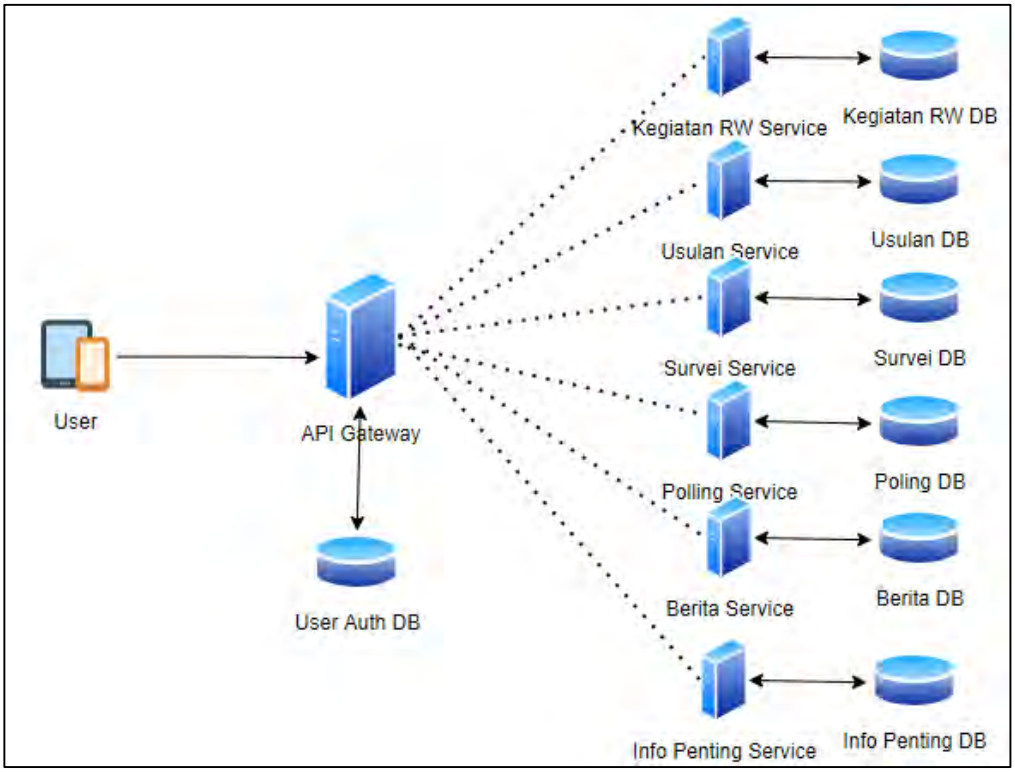


Gambar 4. 9
Database Info Penting

Gambar 4.9 adalah *database* untuk *service* info penting.*Database* terdiri dari dua buah tabel yaitu : *news*_important untuk menyimpan data info penting, juga *news_important_comment* untuk melakukan komentar atau pertanyaan pada info penting tersebut. Info penting yang ditampilkan biasa berkaitan dengan Kesehatan, lowongan kerja, dan bidang Pendidikan.

Setelah *database* terpisah berdasarkan kepemilikan data dari sebuah *domain* atau *service* maka dari sisi *service* akan dilakukan proses pencatatan konfigurasi *database*. Pencatatan konfigurasi *database* meliputi : nama *database*, *host database*, *user database* dan *password* untuk mengakses *database*. Jika perlu konfigurasi khusus biasanya dapat juga berupa penambahan konfigurasi *port database*.

Gambar 4.9 adalah *update* arsitektur sampai pada tahap pemisahan *database*. Setiap *service* sudah mempunyai *database* masing-masing. Tidak ada lagi ketergantungan data dalam sebuah *database*. Jika suatu *database* mengalami gangguan maka hanya *service* tersebut yang mengalami gangguan dan tidak *service* lain masih dapat berjalan dengan seharusnya.



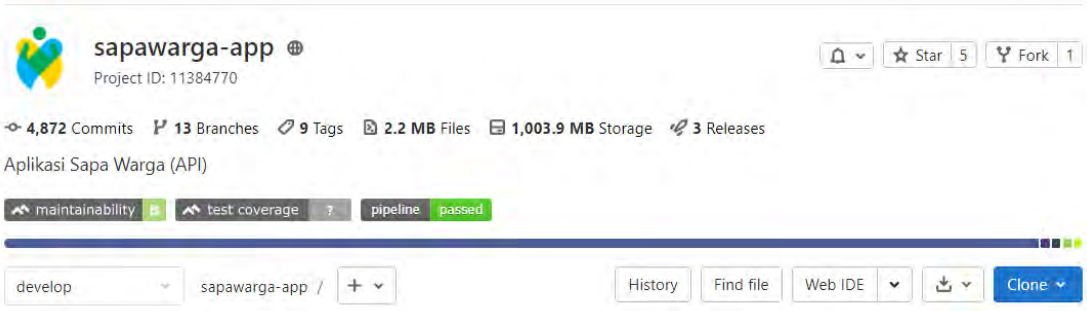
Gambar 4.9
Pemisahan D*atabase* Berdasarkan Kepemilikan Data

Gambar 4.9 merupakan *update* arsitektur setelah pemecahan *database* sesuai masing-masing *service*. Sudah tidak ada lagi ketergantungan data antara *database* satu dan yang lainnya. Bila sebuah *service* dibutuhkan mengambil *database* dari *service* lain, tidak diperkenankan untuk mengakses *database* langsung. Cara komunikasi untuk mendapatkan *database service* lain harus menggunakan protokol REST API yang melakukan *hit* ke *service* lain.

4.3. Pemisahan *Repository Source Code*

Pada tim pengembang aplikasi yang mempunyai jumlah anggota yang banyak akan kesulitan untuk melakukan koordinasi pada implementasi menulis kode program.

Setiap kode dalam sebuah *repository* merupakan suatu kesatuan, jika terdapat *error* pada satu tempat maka akan berakibat *error* pada seluruh aplikasi.



Gambar 4.10
Repository Sapawarga Saat Ini

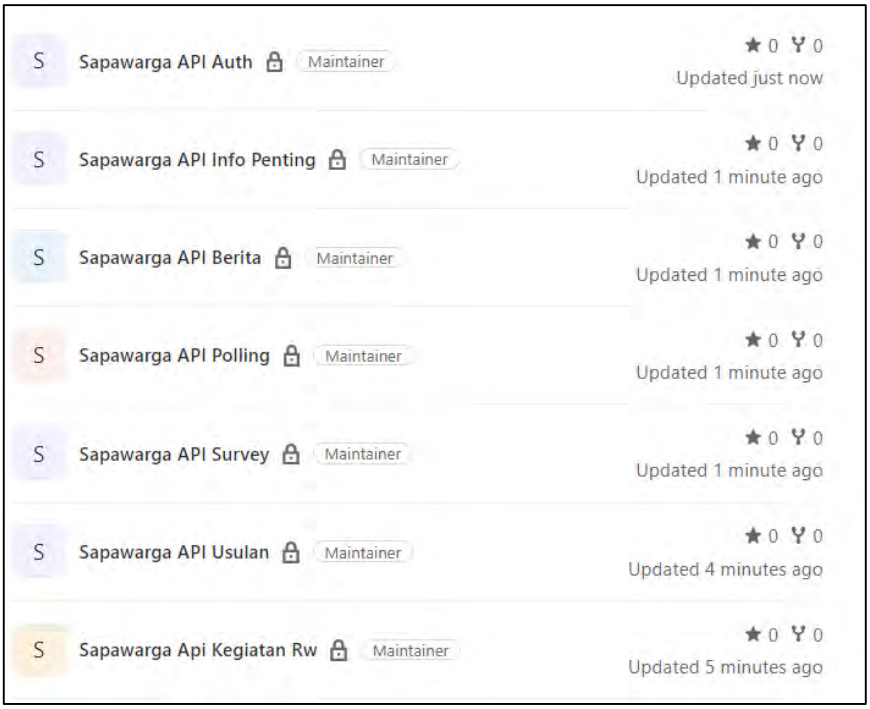
Pada tahapan ini akan dilakukan pemisahan *repository* yang sebelumnya *mono repository* seperti pada Gambar 4.10 menjadi *multi repository*. Adapun pemisahan didasarkan kepada *domain* yang sebelumnya telah dipisah. *Mono*-repo adalah pola kontrol sumber di mana semua kode disimpan dalam satu *repository*. Mudah untuk memberikan keseluruhan aplikasi kepada para pengembang dengan hanya sekali melakukan *clone* pada *version control system*, namun ini cocok untuk

Gambar 4.11
Sapawarga Menggunakan CI / CD Pada Proses *Development*

Penggunaan *repository* sapawarga telah diimplementasikan *best practice* dalam pengembangan aplikasi yaitu menggunakan CI / CD, fungsinya untuk melakukan otomatisasi *deployment* yang biasanya dilakukan manual oleh *developer*. Berikut beberapa tahapan yang dilakukan setelah *code* di *push* sampai *code* ter-*deploy* ke server *production* seperti pada Gambar 4.11:

1. Tahapan *versioning* : Membuat versi aplikasi

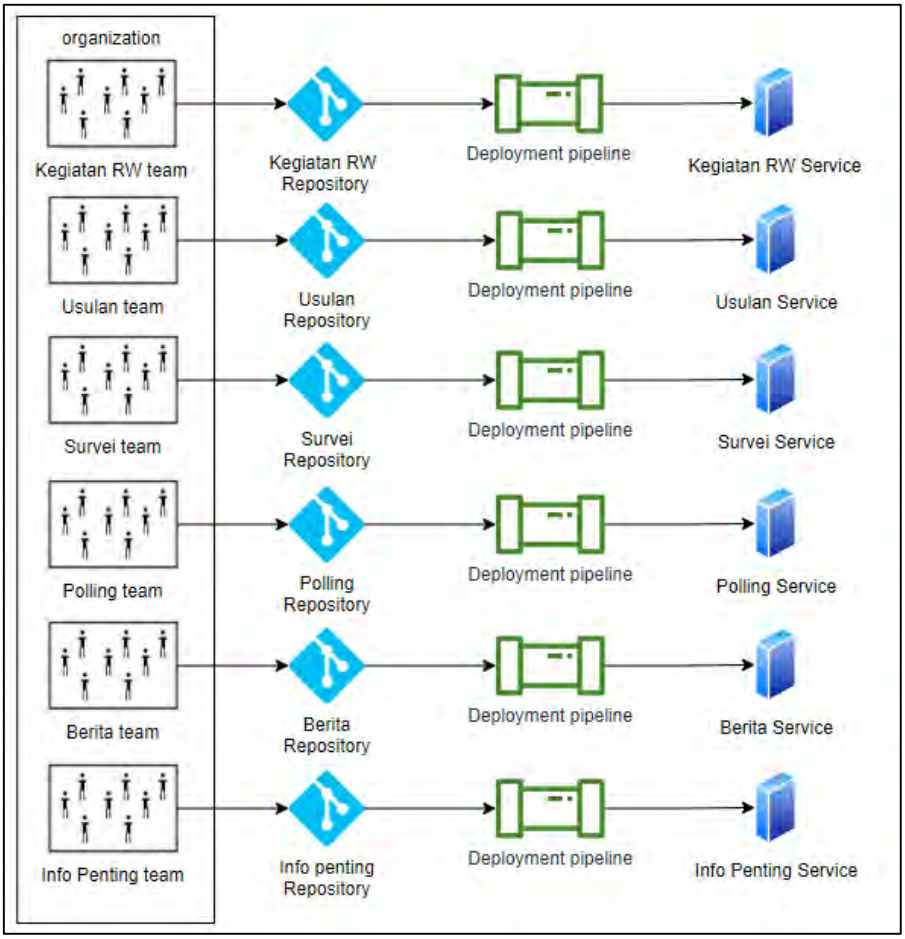
2. Tahapan *code_style* : Melakukan pengecekan terhadap gaya dan aturan membuat kode. Terdapat penilaian pada akhir pengecekan, apakah sesuai dengan konsensus atau tidak.
3. Tahapan *code_quality* : Pengecekan kualitas kode
4. Tahapan *build* : Membuat docker image baru, terkait dengan perubahan kode yang baru dilakukan.
5. Tahapan *test* : Melakukan pengecekan fungsional aplikasi menggunakan unit *test* yang telah dibuat sebelumnya oleh *software engineer*.
6. Tahapan *deploy_staging* : Proses *update* kode pada *server staging*, untuk dilakukan pengetesan terlebih dahulu oleh QA untuk memastikan fitur tidak ada *bug*.
7. Tahapan *deploy_prod* : Jika sudah lolos QA maka otomatis akan segera naik ke *server production* yang dipakai oleh publik.



Gambar 4.12
Pemisahan R*epository* Berdasarkan *Domain*

Gambar 4.11 menjelaskan *repository* usulan yang dipisah berdasarkan domain. *Repository* dibuat dengan bersifat *private* karena untuk menjaga keamanan dari pihak yang lain yang tidak bertanggung jawab. Antara *repository* satu dan yang lainnya juga dibatasi

anggotanya, hanya yang bertugas pada fiturnya yang dapat akses repository, missal tim development API Berita hanya bisa melihat repository berita saja.



Gambar 4. 13
Pemisahan *Repository* Untuk Setiap *Development Team*

Melakukan pemisahan *repository* pada implementasi *microservice* akan menyelesaikan permasalan juga dari sisi organisasi seperti pada Gambar 4.12. Pada organisasi yang memiliki tim *development* yang besar akan kesulitan untuk mengembangkan aplikasi dengan mono *repository*, karena aplikasi akan menjadi sebuah aplikasi kesatuan. Banyak orang yang melakukan *push*, *commit*, *deploy* akan saling ketergantungan pada fitur lain yang sedang dikembangkan. Kemungkinan untuk terjadi *conflict* kode antara *developer* satu dengan yang lainnya akan lebih besar.

Faktor kecepatan *delivery* produk untuk rilis pun akan meningkat, karena setiap kode yang di *push* ke *staging* maupun *production* akan melalui *pipeline* yang mempunyai tujuh tahapan pengecekan seperti pada Gambar 4.14 Pemisahan *repository* menjadikan

tim lebih *autonomous,* lebih *independent* terhadap fiturnya masing-masing, karena mengurai ketergantungan pada *service* lain. Tim akan fokus pada fitur masing-masing, jika ada kesalahan kode akan hanya berdampak pada *service* tersebut dan tidak berdampak pada *service* yang lain.

4.4. Perancangan *Docker Container*

Pada tahap perancangan Docker harus diinisialisasi pada semua *service* yang dibuat. Penggunaan Docker dapat mempermudah dan mempercepat setup aplikasi pada sebuah server. Dahulu untuk membuat sebuah aplikasi berbasis web dibutuhkan beberapa konfigurasi yang dilakukan manual misal meng-*install* dahulu *web server* bisa Apache untuk Linux atau Internet Information Services (IIS) untuk Windows. Setelah *web server* siap baru selanjutnya membuat konfigurasi dari sisi aplikasi dan *database* yang harus dilakukan manual. Perbedaan *environment* seperti versi *database*, versi bahasa pemrograman, versi *library* yang dipakai sering membuat aplikasi *error*. Bayangkan jika terdapat kebutuhan untuk melakukan *scaling* dengan menambah server, sebanyak itu pula harus melakukan konfigurasi manual.

Penggunaan Docker akan meminimalkan setup yang dahulu dijalankan manual. Docker juga pasti menjalankan *file* konfigurasi yang sama Ketika aplikasi akan dilakukan *scaling*. Sebanyak apapun kebutuhan penambahan server, ketika sudah membungkus aplikasi atau *service* menggunakan Docker akan lebih mudah dilakukan, karena tinggal menjalankan sebuah perintah Docker yang kemudian Docker akan mengeksekusi apa saja yang sudah ada di *file* konfigurasi docker-compose.

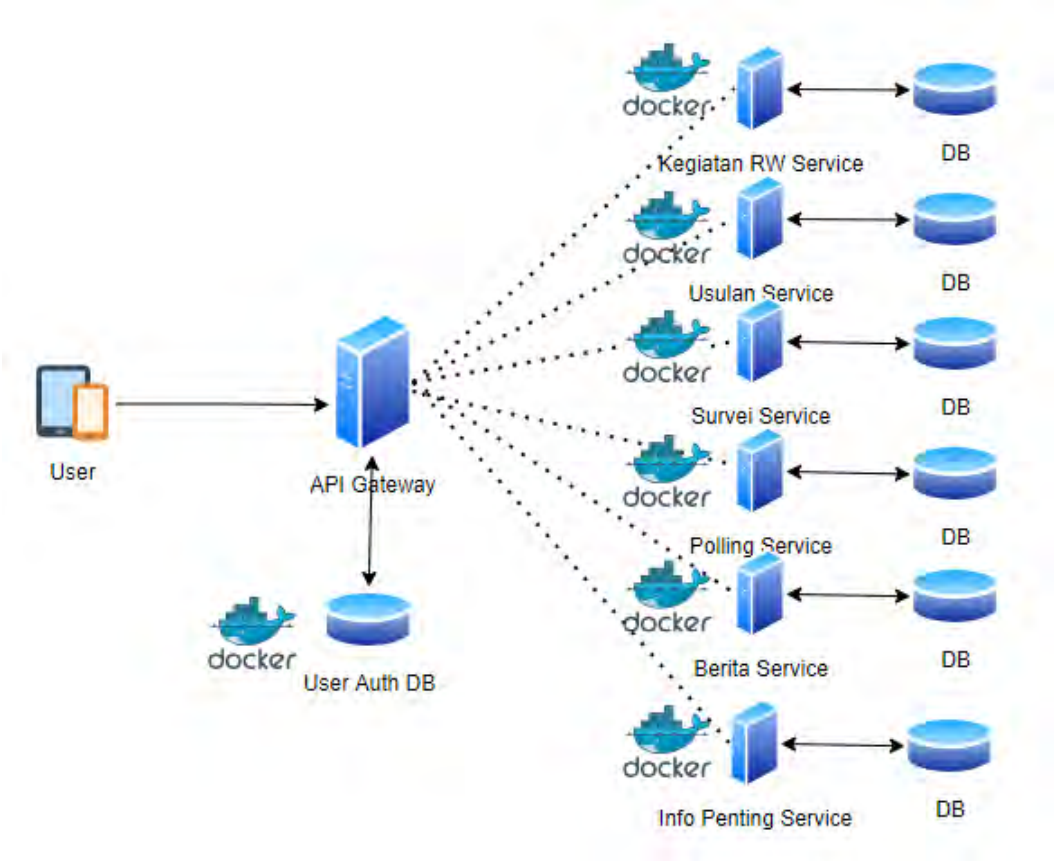
Ada beberapa cara menggunakan Docker untuk menjalankan sebuah *container*. Bisa dengan menjalankan docker-run, bisa juga menjalankan lewat docker-compose. Penggunaan docker-compose dapat membuat konfigurasi *container* secara *custom* dan bisa disesuaikan dengan kebutuhan.

Gambar 4. 14
Docker-compose Pada Setiap *Service*

Berikut penjelasan dari konfigurasi docker-compose berdasarkan baris fungsi :

1. *Version* : merupakan versi dari docker-compose, saat ini versi terbaru docker-compose sudah pada versi 3.9, namun untuk kebutuhan *stable* dapat menggunakan versi 3.
2. *Service* : Cakupan *service* ini adalah mendefinisikan *container* apa saja yang ingin dibuat. Pada Gambar 4.15 akan membuat *container* dengan nama api yang di dalamnya hanya terdapat aplikasi berupa API dari sebuah fitur.
3. API : Merupakan nama dari sebuah *service*. Docker akan membuat sebuah *container* dengan nama tersebut.
4. *Build* : Merupakan bagian untuk mendefinisikan lokasi konfigurasi sebuah docker-compose berada.
5. *Network* : Untuk mendefinisikan nama *network* dari sebuah *container.*

6. *Env_file* : Setiap aplikasi yang menggunakan *framework* misal Laravel pada *framework* PHP mempunyai konfigurasi *environment*, yang umumnya Bernama .env di dalamnya berisi kumpulan konfigurasi *file*.
7. *Volumes* : Untuk mendefinisikan lokasi *persistence volume*
8. *Port* : Untuk mendefinisikan penggunaan *port* pada aplikasi. Penggunaan port 81:80 artinya *network* yang masuk dari *external* melalui port 81 kemudian oleh Docker *network*-nya diteruskan ke dalam *container* dengan port 80.
9. *Restart* : Merupakan konfigurasi yang membuat *container* akan me-*restart* dengan sendirinya ketika ditemukan kondisi *failure*.
10. *Container_name* : Merupakan tempat untuk memberi nama *container*, pemberian nama dibutuhkan jika dalam sebuah server ada beberapa *container*.



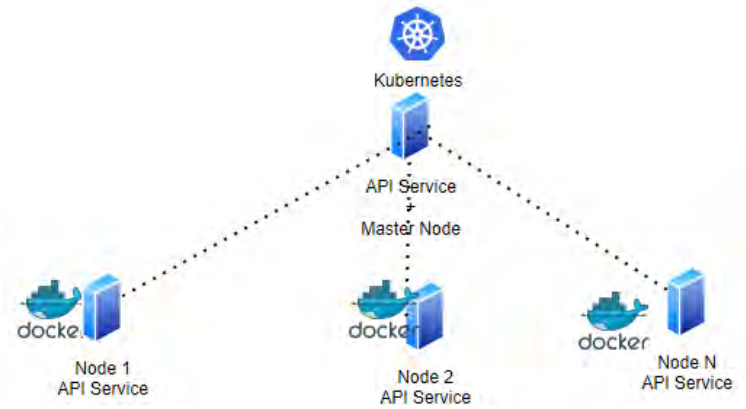
Gambar 4.15
Penggunaan *Docker Container* Pada Semua *Service*

Gambar 4.16 adalah usulan tentang penggunaan Docker *container* pada semua *service*. Ini sangat memudahkan proses pengembangan dan *maintenance.* Jika ada server sebuah *service* yang mati maka tinggal membuat server baru, meng-*install* Docker kemudian tinggal melakukan build lewat docker-compose, maka *service* tersebut akan jalan kembali dengan *effort* konfigurasi yang minimal. Untuk menjalankan Docker pada setiap *service* dilakukan dengan men-*setup file docker-compose.yml* seperti pada Gambar 4.15.

Jika tanpa Docker *container* diharuskan meng-*install web server*, melakukan konfigurasi web server seperti setup *max_execution_time*. Ketika *web server* sudah *running*, maka harus lanjut meng-*install dependency library* yang sangat membutuhkan waktu dan tenaga lebih.

4.5. Perancangan *Container Orchestration* Menggunakan Kubernetes

Untuk merancang arsitektur menggunakan *container orchestration* dibutuhkan sebuah *master node* dan minimal satu *worker node*.Untuk spesifikasi hardware yang digunakan sebaiknya memory minimal 2gb dan CPU memiliki 2 CPU.



Gambar 4. 16
Topologi Perancangan *Master Node* dan *Worker Node*

Gambar 4.16 adalah topologi simple untuk membangun sebuah Kubernetes klaster, dimana terdapat sebuah master node dan terdapat tiga buah *worker*. Jumlah *worker* dimungkinkan banyak tergantung kebutuhan sebuah sistem.

Master node adalah *server* utama yang melakukan kontrol terhadap tiap *node* yang terhubung kepada *master node*. *Master node* mempunyai beberapa komponen penting yang akan memberikan perintah pada setiap *worker node*, untuk melakukan

pekerjaan yang diinstruksikan yaitu API, Etcd dan *controller manager*. Langkah-langkah melakukan setup pada sebuah Kubernetes klaster di mana terdapat *master node* dan *worker node* adalah sebagai berikut :

1. Mempersiapkan 4 buah server atau VM
2. Login ke dalam masing-masing server bisa menggunakan SSH
3. Menamai masing-masing *hostname* dengan menggunakan perintah : hostnamectl set-hostname [nama_host].
4. Lakukan instalasi Docker
5. *Install* Kubectl pada masing-masing *node*
6. Inisialiassi Kubernetest Klaster menggunakan Kubeadm dari *master node* dengan perintah : kubeadm init
7. Melakukan *deploy* add-on pod network Calico pada *node master.*

4.6. Perancangan Horizontal Pod Autoscaler (HPA) Kubernetes

Jumlah pengguna akses aplikasi memang tidak bisa diprediksi secara tepat. Terkadang pemakaian aplikasi bisa sibuk dengan banyaknya pengguna, namun juga jumlah pengguna aplikasi terkadang sepi. Jika jumlah pengguna aplikasi sepi, aplikasi akan berjalan dengan lancar, namun Ketika jumlah pengguna meningkat drastis akan mengakibatkan konsumsi penggunaan *memory* atau *cpu* tinggi, maka akan ada kemungkinan performa aplikasi akan turun dan sulit diakses. Pada saat itulah *scaling* pada sisi infrastruktur diperlukan untuk menambah kapasitas *server*.

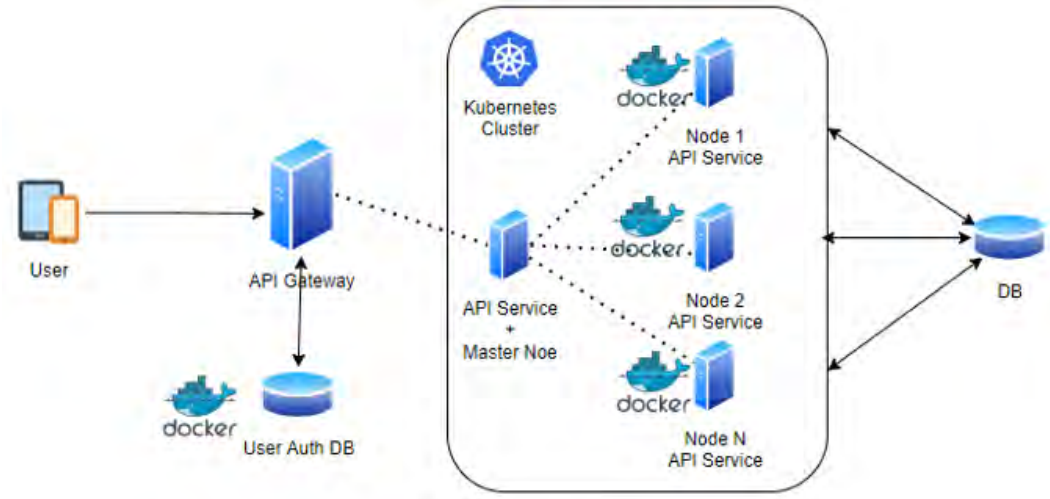
Scaling secara *vertical* mempunyai batasan untuk menambahkan kemampuan *hardware*. Misal sebuah *server* memiliki 4 *slot* RAM, saat ini diisi dengan 1 *slot* 8gb dan ketika dibutuhkan untuk proses *scaling* masing-masing diisi 8gb, maka *scaling* maksimal pada server tersebut adalah 4 x 8gb = 32gb.

Pada penggunaan *scaling* secara horizontal, proses *scaling* tidak memiliki batasan pada *hardware* sampai *server* yang dimiliki digunakan semua. Beban *server* dapat terbagi dengan penambahan server secara horizontal.

```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: api-service
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: api-service
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 20
```

Gambar 4.17
Konfigurasi HPA Pada Sebuah *Service*

Gambar 4.16 merupakan konfigurasi untuk setup HPA pada sebuah server. Penjelasan beberapa *syntax* adalah bahwa dalam sebuah server akan menggunakan *autoscaling* versi 1. Penamaan *pod* menggunakan nama api-service. Konfigurasi tersebut menjelaskan bahwa minimal penggunaan *replica* adalah 1 *pod* dan maksimal jika melakukan *autoscaling* adalah sebanyak 10 *pod*.



Gambar 4. 18
Perancangan HPA Pada Kegiatan RW *Service*

Gambar 4.18 menjelaskan di mana sebuah konfigurasi HPA disimpan pada *master node* yang berguna membaca metrik utilisasi berupa CPU dan *memory*. Jika *request* dari *user* masih dalam metrik yang diperbolehkan sebanyak 20%, maka *pod* dalam sebuah *node* tidak akan bertambah. Jika utilisasi CPU sudah melampaui 20 % maka HPA akan

menambahkan pod sampai rata-rata penggunaan CPU Kembali di bawah 20%. Penambahan *pod* dan *node* maksimal tergantung dari konfigurasi awal dan ketersediaan *node*.

4.7. Pengujian skalabilitas dengan HPA

Untuk uji skalabilitas akan digunakan *Request Generator* yaitu Hey, di mana *Web Server* akan di-*request* oleh Hey dengan *user* yang sudah ditentukan. Adapun parameter yang akan digunakan ialah load Testing, dengan cara mengukur *resource* utilisasi CPU dari masing-masing c*ontainer orchestration* . Dengan skenario memasukkan *request* yang telah di tentukan adalah 10.000 *request* .

Pengujian dilakukan menggunakan metoda *load testing*. Tujuan *Load Testing* adalah untuk melihat skalabilitas dari masing-masing c*ontainer orchestration*, dengan cara melihat beban pada utilisasi CPU dengan *user* yang sudah ditentukan sebelumnya. Utilisasi CPU menunjukkan berapa persentase yang dibutuhkan dalam proses yang dilakukan pada percobaan.

4.7.1. Melakukan Konfigurasi HPA

Untuk melakukan pengujian HPA pada sebuah server diperlukan konfigurasi terlebih dahulu. Versi HPA yang digunakan adalah versi *autoscaling*/v1. API versi 1 juga merupakan versi *stable* sehingga lebih stabil jika digunakan untuk *staging* dan *production*.

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: api-server-v1
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: api-server
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 20
```

Gambar 4. 19
Mendefinisikan HPA Pada Sebuah *Server*

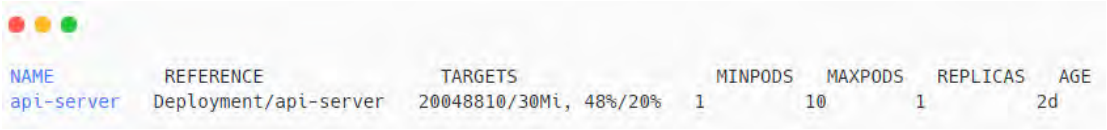
Gambar 4.19 berarti bahwa dalam sebuah server menggunakan *autoscaling* versi 1. Penamaan pod menggunakan nama api-server. Konfigurasi tersebut menjelaskan

bahwa minimal penggunaan *replica* adalah 1 *pod* dan maksimal adalah 10 *pod*. HPA akan melakukan autoscaling Ketika beban utilisasi server sudah memasuki lebih dari 20%.

4.7.2. Hasil Pengujian

Melakukan pengujian dengan mengirimkan *request* sebanyak 10.000 dengan menggunakan lima *worker*. Pengujian dilakukan dengan menggunakan *tools* Hey dengan menjalankan "$ hey -n 10000 -c 5 http://localhost:8000/"' pada sebuah shell atau terminal. Lakukan pengecekan efek dari *load testing* dengan menggunakan perintah :

$ kubectl get hpa web-servers.

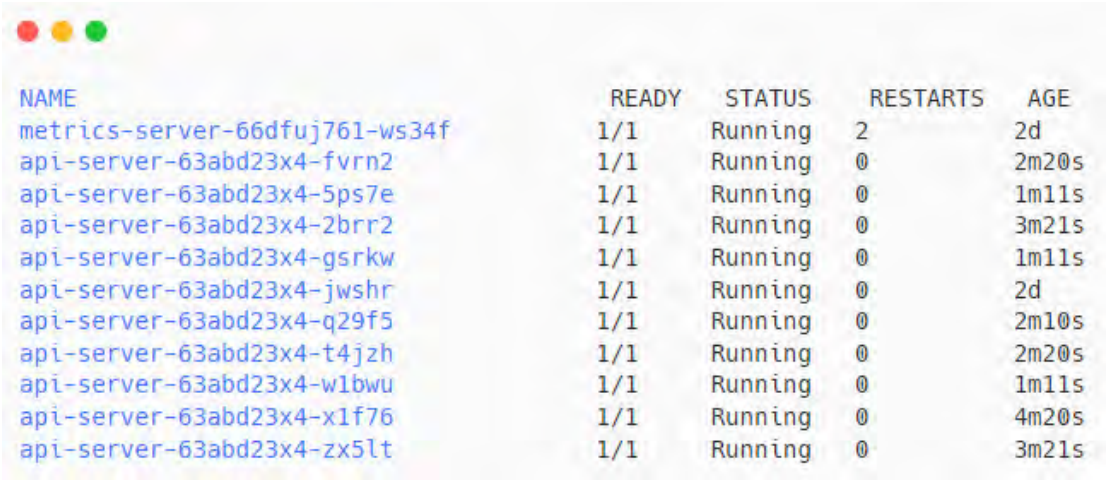


Gambar 4. 20
Utilisasi CPU dan *Memory* Meningkat Setelah Sesaat Dilakukan *Load Testing*

Gambar 4.20 menunjukan peningkatan setelah sesaat dilakukan *load testing*. Server dengan beban diawal masih bisa bekerja dengan sebuah *replica*, namun *load testing* masih terus berjalan *sampai* kondisi berhasil mengirimkan 10.000 *request*. Replika yang berjalan saai ini hanya membuat 1 replika, terlihat informasi berupa status replicas=1. Informasi AGE adalah informasi tentang usia sebuah *pod* telah *running* berapa lama.

Gambar 4. 21
HPA Melakukan *Autoscaling* Dengan Maksimal Pod

Setelah dilakukan pemantauan pada *load testing* yang sudah dijalankan lama, terjadi peningkatan jumlah *replica* yang asalnya hanya 1 bertambah menjadi 10 seperti terlihat pada Gambar 4.21. Kondisi saat ini adalah kondisi di mana HPA berhasil melakukan *autoscaling* terhadap beban yang masuk ke *server*.



```
NAME                                  READY   STATUS    RESTARTS   AGE
metrics-server-66dfuj761-ws34f        1/1     Running   2          2d
api-server-63abd23x4-fvrn2            1/1     Running   0          2m20s
api-server-63abd23x4-5ps7e            1/1     Running   0          1m11s
api-server-63abd23x4-2brr2            1/1     Running   0          3m21s
api-server-63abd23x4-gsrkw            1/1     Running   0          1m11s
api-server-63abd23x4-jwshr            1/1     Running   0          2d
api-server-63abd23x4-q29f5            1/1     Running   0          2m10s
api-server-63abd23x4-t4jzh            1/1     Running   0          2m20s
api-server-63abd23x4-w1bwu            1/1     Running   0          1m11s
api-server-63abd23x4-x1f76            1/1     Running   0          4m20s
api-server-63abd23x4-zx5lt            1/1     Running   0          3m21s
```

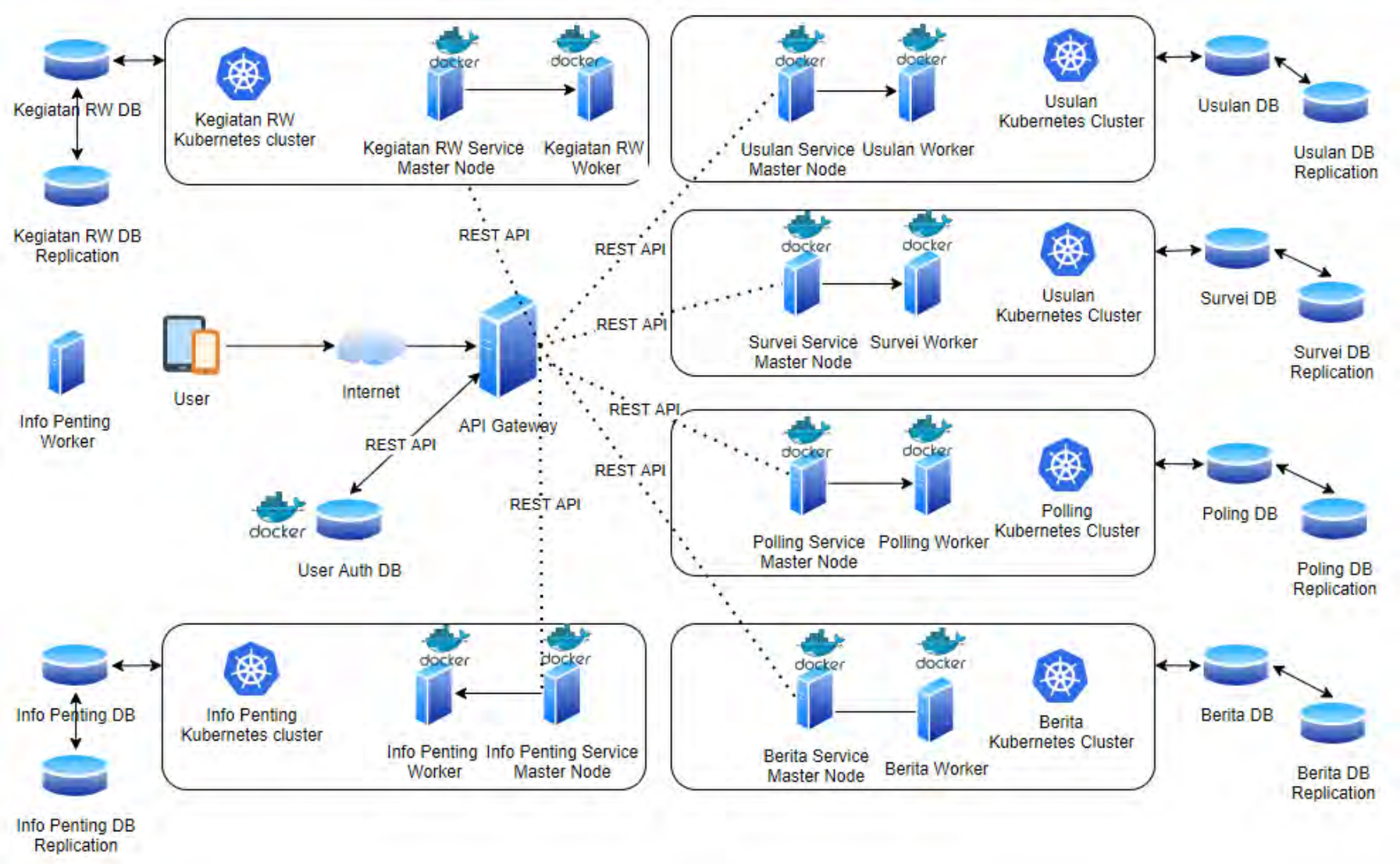
Gambar 4. 22
Hasil Pengujian Skalabilitas

Gambar 4.22 terlihat bahwa dengan melakukan *load testing* berupa pengiriman *request* data dengan jumlah tertentu server dapat melakukan *autoscaling* secara *horizontal*. Berhasilnya server melakukan *autoscaling* dapat membuat aplikasi terus berjalan dengan baik dan menjalankan *request*. Jumlah pod akan berkurang mengikuti aturan metrik utilisasi jika utilisasi CPU mengalami penurunan.

Informasi AGE adalah usia sebuah *pod*, terlihat perbedaan *age* antara *pod* satu dengan lainnya. Usia *pod* yang berbada diakibatkan olah perbedaan jumlah *request* dan waktu yang *datang* tidak bersamaan. Misal pada menit pertama terdapat 200 *request*, maka hanya dibutuhkan dua *pod* saja, jika pada menit ke 3 *request* ditambah maka pada saat itu akan dibuatkan *pod* baru dengan *age* yang muncul pada saat *pod* dibuat. Mau seberapa banyak request yang diterima bisa disesuaikan dengan konfigurasi oleh Kubernetes secara deklaratif.

4.8. Rekomendasi Usulan Arsitektur

Rekomendasi susulan berupa perubahan dari arsitektur *monolith* ke *microservice* harus dilakukan untuk mengantisipasi tantangan kebutuhan yang semakin meningkat baik dari jumlah pengguna maupun fitur aplikasi. Berdasarkan pengujian *autoscaling* dengan menggunakan *container orchestration* telah terbukti dapat menambah kemampuan aplikasi dalam menangani jumlah pengguna yang banyak. Kemudahan *scaling* yang dilakukan secara otomatis menjadi nilai lebih dalam rekomendasi usulan arsitektur ini.



Gambar 4. 23
Usulan Akhir Arsitektur *Microservice* Sapawarga Menggunakan *Container Orchestration*

Berikut penjelasan dari usulan akhir arsitektur *microservice* menggunakan *Container Orchestration* Kubernetes berdasarkan nomor pada Gambar 4.19 :

1. API g*ateway* bertugas untuk memecah aliran *request* yang berasal dari *frontend*,. Misal ada sebuah *request* dengan *wildcard* '*/polling*' maka *api gateway* akan mengarahkan *request* tersebut kepada *service polling*.
2. API *gateway* akan melakukan pengecekan terhadap akses *user*. Jika berhasil *login* maka akan diteruskan ke *request* selanjutnya.
3. API *gateway* meneruskan *request* menggunakan protokol HTTP REST API dengan fungsi-fungsi yang dipakai dapat berupa GET digunakan untuk mendapatkan data, POST untuk memasukan data, PUT untuk melakukan mengubah data, DELETE untuk melakukan penghapusan data.
4. Rekomendasi pada setiap domain harus memiliki sebuah klaster Kubernetes tersendiri, karena beban setiap domain pastilah berbeda. Misal pada waktu tertentu ada pengumuman untuk melakukan *polling*, maka otomatis penggunaan *service polling* meningkat dari pada *service* lainnya. Ketika beban *polling* meningkat, maka hanya *service polling* saja yang mengalami *autoscaling* dengam menambahkan *pod* secara otomatis melalui fungsi HPA.
5. Klaster Kubernetes berisikan *master node* dan *worker node*, file konfigurasi HPA tersimpan pada *master node*.
6. Penempatan konfigurasi Docker *container* terdapat di setiap *worker*.
7. *Database* yang digunakan hanya memiliki satu server. *Database* yang digunakan meliputi proses *read* & *write*.
8. Jika ke depan ada kebutuhan untuk memisahkan proses *read* & *write* dapat digunakan topologi *database slave master replication*, yaitu memisahkan antara proses *write* yang disimpan pada *database master* dan *read* yang disimpan pada *database slave*. DB Replication juga bisa berfungsi sebagai *Database Backup* dari *database master* atau utama. Jumlah *database replication* juga dapat digunakan lebih dari satu *database*.

BAB V

KESIMPULAN DAN SARAN

5.1. Kesimpulan

Berdasarkan hasil analisis, perancangan dan melakukan pengujian pada sebuah arsitektur *microservice* menggunakan *orchestration container* berupa Kubernetes dengan menambahkan fitur HPA maka dapat diambil kesimpulan berupa :

1. Dengan menerapkan usulan arsitektur aplikasi Sapawarga menjadi *microservices* dapat menambah skalabilitas aplikasi. Kubernetes dengan HPA yang bertugas untuk melakukan *autoscaling* ketika beban server meningkat telah memberi solusi otomasi untuk meningkatkan skalabilitas aplikasi.
2. Pada tahun 2022, Sapawarga direncanakan akan dirilis secara publik dan menjadi superapp di Jawa Barat. Fitur Sapawarga diperkirakan akan mengalami banyak penambahan fitur berkaitan dengan masukan dari publik dan *stakeholder*. Proses pengembangannya dimungkinkan untuk menambah tim baru baik dari *internal* organisasi maupun *external*. Dengan menggunakan usulan arsitektur *microservice* akan mempermudah proses pengembangan karena *service* dan *repository* yang telah dipisah sehingga tim dalam sebuah fitur akan fokus pada fitur masing-masing.

5.2. Saran

Adapun saran dari hasil kesimpulan dalam penelitian ini adalah :

1. Pada penelitian selanjutnya dapat melakukan uji coba perbandingan menggunakan HPA dengan tanpa menggunakan HPA dalam perhitungan biaya pengeluaran server pada *cloud provide* (AWS / GCP) yang berslogan "*Pay as you go*" atau hanya membayar apa yang sesuai dipakai.
2. Dapat melanjutkan penelitian usulan arsitektur *microservice* dengan menggunakan *cloud native roadmap*.

DAFTAR PUSTAKA

Evans, E., & Evans, E. J. (2004). *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional.

Fielding, R. T., & Taylor, R. N. (2002). Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, *2*(2), 115–150.

Fowler, M. (2017). Microservice Trade-offs, July 1, 2015. *URL Https://Martinfowler. Com/Articles/Microservice-Trade-Offs. Html. Accessed*, 4–19.

Ibryam, B., & Huß, R. (2019). *Kubernetes patterns: reusable elements for designing cloud-native applications*. O'Reilly Media.

Kroenke, D., & Kroenke, D. (1977). *Database processing*. Science Research Associates Reading, MA.

Luksa, M. (2017). *Kubernetes in action*. Simon and Schuster.

Mehrabani, A. (2014). *MongoDB High Availability*. Packt Publishing Ltd.

Poulton, N. (2017). *The Kubernetes Book*. http://leanpub.com/thekubernetesbook

Poulton, N. (2019). *Docker deep dive*. JJNP Consulting Limited.

Richardson, C. (2017). Microservices Patterns. In *Online* (Vol. 104, Issue 25). http://www.ncbi.nlm.nih.gov/pubmed/20608803

Rosen, M., Lublinsky, B., Smith, K. T., & Balcer, M. J. (2012). *Applied SOA: service-oriented architecture and design strategies*. John Wiley & Sons.

Unhelkar, B. (2017). *Software engineering with uml*. CRC Press.

Richardson, C. (2017). Microservices Patterns. In *Online* (Vol. 104, Issue 25). http://www.ncbi.nlm.nih.gov/pubmed/20608803

Rosen, M., Lublinsky, B., Smith, K. T., & Balcer, M. J. (2012). *Applied SOA: service-oriented architecture and design strategies*. John Wiley & Sons.

Wasson, M., & Schonning, N, 2019. Build microservices on Azure

Chen, Lianping , 2018. Microservice: Architecting for Continuous Delivery and DevOps. The IEEE International Conference on Software Architecture (ICSA 2018). IEEE.

Giallorenzo, Saverio , Nicola Dragoni, Alberto Lluch Lafuente, Manuel Mazzara Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina, 2016, microservice: yesterday, today, and tomorrow

Newman, Sam (2015-02-20). Building microservice. O'Reilly Media. ISBN 978-1491950357.

Richardson, Chris, 2018 1.4.1 Scale cube and microservice". Microservice Patterns. Manning Publications.

Wolff, Eberhard, 2016, microservice: Flexible Software Architectures.

Z. Ding, S. P. Architect, S. Sanjabi, and A. S. Engineer, 2016, “Using Docker in High Performance Computing in OpenPOWER Environment,” 2016 IEEE Sixth Int. Conf. Commun. Electron., pp. 1–17, 2016.

Mike Rosen; Boris Lublinsky; Kevin T. Smith; Marc J. Balcer, 2008, “Applied SOA Service-Oriented Architecture and Design Strategies”

Reddy, Martin (2011). API Design for C++. Elsevier Science. p. 1. ISBN 9780123850041.