BAB 2

LANDASAN TEORI

Bab ini membahas landasan teori yang menjadi dasar dalam perancangan dan pengembangan sistem *Human Resource* (HR) berbasis arsitektur microservices. Pembahasan teori bertujuan untuk memberikan pemahaman konseptual serta kerangka ilmiah yang mendukung analisis, perancangan, dan implementasi sistem yang dikembangkan pada penelitian ini.

Landasan teori yang dibahas meliputi konsep sistem informasi, arsitektur perangkat lunak, khususnya arsitektur *monolithic* dan *microservices*, karakteristik serta pola desain microservices, mekanisme konsistensi dan sinkronisasi data antar layanan, serta teori pengujian perangkat lunak yang mencakup unit testing, integration testing, dan API testing.

Dengan adanya landasan teori ini, diharapkan penelitian memiliki dasar ilmiah yang kuat dan relevan dalam menjawab permasalahan yang dihadapi oleh PT Annisaa Putri Kaligayam dalam pengembangan sistem HR yang lebih terstruktur, scalable, dan handal.

2.1 Arsitektur Perangkat Lunak

Arsitektur perangkat lunak merupakan struktur fundamental dari suatu sistem yang mencakup komponen, hubungan antar komponen, serta prinsip-prinsip desain yang digunakan dalam pengembangannya. Keputusan arsitektural memiliki dampak jangka panjang terhadap kualitas sistem, seperti skalabilitas, *maintainability*, dan *reliability* [1].

Dalam konteks sistem informasi modern, arsitektur tidak hanya berfungsi sebagai panduan teknis, tetapi juga sebagai dasar pengambilan keputusan organisasi dalam menghadapi perubahan kebutuhan bisnis dan teknologi.

2.2 Arsitektur Monolith dan Keterbatasannya

Arsitektur *monolith* merupakan pendekatan tradisional di mana seluruh fungsi sistem dibangun dan dijalankan sebagai satu kesatuan aplikasi. Pendekatan ini relatif mudah dikembangkan pada tahap awal, namun memiliki keterbatasan ketika sistem berkembang menjadi lebih kompleks.

Menurut Su et al. [2], arsitektur *monolith* cenderung sulit diskalakan, memiliki dependensi yang kuat antar modul, serta menyulitkan proses *deployment* dan *maintenance*. Kondisi ini sering mendorong organisasi untuk melakukan migrasi menuju arsitektur *microservices*.

2.3 Arsitektur Microservices

Microservices adalah pendekatan arsitektur di mana sistem dibangun sebagai kumpulan layanan kecil yang berdiri sendiri, memiliki tanggung jawab bisnis spesifik, dan berkomunikasi melalui mekanisme ringan seperti *REST* atau *message* broker [4].

Setiap microservice dapat dikembangkan, diuji, dan dideploy secara independen, sehingga meningkatkan fleksibilitas dan kecepatan pengembangan sistem [13].

2.3.1 Karakteristik Arsitektur Microservices

Arsitektur *microservices* memiliki sejumlah karakteristik utama yang membedakannya dari pendekatan arsitektur *monolith*. Karakteristik ini dirancang untuk mendukung pengembangan sistem yang berskala besar, kompleks, serta adaptif terhadap perubahan kebutuhan bisnis dan teknologi.

A Loose Coupling

Loose coupling merupakan karakteristik fundamental dari arsitektur *microservices*, di mana setiap layanan dirancang agar memiliki ketergantungan seminimal mungkin terhadap layanan lain. Setiap *microservice* berinteraksi

melalui antarmuka yang terdefinisi dengan jelas, umumnya berupa *Application Programming Interface* (API).

Dengan tingkat keterikatan yang rendah, perubahan atau pembaruan pada satu layanan tidak secara langsung mempengaruhi layanan lainnya. Hal ini meningkatkan fleksibilitas sistem dan mengurangi risiko kegagalan menyeluruh [4].

B Single Responsibility dan Business Capability

Setiap *microservice* dibangun berdasarkan prinsip *single responsibility*, yaitu hanya menangani satu fungsi atau kapabilitas bisnis tertentu. Pendekatan ini dikenal juga sebagai *decomposition by business capability*.

Narváez et al. [7] menyatakan bahwa pemisahan layanan berdasarkan kapabilitas bisnis membantu tim pengembang memahami domain sistem dengan lebih baik serta meningkatkan konsistensi desain arsitektur *microservices*.

C Independensi Deployment

Salah satu keunggulan utama *microservices* adalah kemampuan untuk melakukan *deployment* secara independen. Setiap layanan dapat diperbarui, diuji, dan *dideploy* tanpa harus melakukan *redeployment* terhadap seluruh sistem.

Karakteristik ini memungkinkan organisasi untuk menerapkan *continuous integration* dan *continuous deployment* (CI/CD), sehingga mempercepat siklus pengembangan perangkat lunak dan meningkatkan *responsivitas* terhadap kebutuhan pengguna [13].

D Desentralisasi Manajemen Data

Dalam arsitektur *microservices*, setiap layanan umumnya memiliki basis data tersendiri yang dikelola secara independen. Pendekatan ini dikenal dengan istilah *database per service*.

Menurut Tapia dan Gaona [8], desentralisasi data membantu mengurangi ketergantungan antar layanan serta memungkinkan setiap microservice memilih teknologi penyimpanan data yang paling sesuai dengan kebutuhannya.

E Komunikasi Antar Layanan

Microservices berkomunikasi satu sama lain melalui mekanisme komunikasi ringan, baik secara sinkron maupun asinkron. Komunikasi sinkron biasanya menggunakan protokol HTTP/REST atau gRPC, sedangkan komunikasi asinkron memanfaatkan *message broker* atau *event-driven architecture*.

Pemilihan mekanisme komunikasi yang tepat sangat penting untuk menjaga performa dan keandalan sistem, terutama pada sistem terdistribusi yang kompleks [6, 9].

F Skalabilitas Independen

Arsitektur *microservices* memungkinkan skalabilitas dilakukan secara selektif pada layanan tertentu yang membutuhkan sumber daya lebih besar. Hal ini berbeda dengan arsitektur *monolith* yang umumnya memerlukan *scaling* pada keseluruhan aplikasi.

Rodrigues et al. [11] menegaskan bahwa kemampuan scaling secara independen menjadi salah satu faktor utama adopsi *microservices* pada sistem dengan beban kerja yang tidak merata.

G Resiliensi dan Fault Isolation

Karena setiap *microservice* berjalan sebagai unit terpisah, kegagalan pada satu layanan tidak secara langsung menyebabkan kegagalan pada seluruh sistem. Konsep ini dikenal sebagai *fault isolation*.

Untuk meningkatkan *resiliensi*, *microservices* sering dilengkapi dengan mekanisme seperti *circuit breaker*, *retry*, dan *fallback*. Penerapan pola-pola ini membantu menjaga ketersediaan sistem meskipun terjadi gangguan pada sebagian layanan [12].

H Heterogenitas Teknologi

Arsitektur microservices memungkinkan penggunaan berbagai teknologi, bahasa pemrograman, dan *framework* yang berbeda untuk setiap layanan.

Fleksibilitas ini dikenal sebagai *polyglot programming*.

Pendekatan ini memberikan kebebasan bagi tim pengembang untuk memilih teknologi yang paling sesuai dengan kebutuhan layanan tertentu, namun juga menuntut standar integrasi dan pengelolaan yang baik [14].

I Kompleksitas Operasional

Meskipun memiliki banyak keunggulan, arsitektur *microservices* juga membawa kompleksitas operasional yang lebih tinggi dibandingkan *monolith*. Kompleksitas ini mencakup pengelolaan infrastruktur, *monitoring*, *logging* terdistribusi, serta keamanan antar layanan.

Su et al. [2] menekankan bahwa tanpa perencanaan dan tata kelola yang matang, kompleksitas ini dapat mengurangi manfaat yang diharapkan dari penerapan *microservices*.

2.3.2 Pola Desain Arsitektur Microservices

Pola desain arsitektur *microservices* merupakan solusi yang telah teruji untuk mengatasi permasalahan umum yang muncul dalam pengembangan sistem terdistribusi. Pola-pola ini membantu meningkatkan skalabilitas, keandalan, *maintainability*, serta kemudahan pengelolaan sistem *microservices* [5].

A API Gateway Pattern

API Gateway merupakan pola desain yang menyediakan satu titik masuk (*single entry point*) bagi klien untuk mengakses berbagai *microservice*. Pola ini berfungsi sebagai perantara antara klien dan layanan *backend*.

API Gateway bertanggung jawab terhadap fungsi-fungsi seperti *routing request*, *authentication*, *authorization*, *rate limiting*, dan *aggregation response*. Dengan adanya *API Gateway*, klien tidak perlu mengetahui detail lokasi dan jumlah *microservice* yang ada [4].

Pola ini sangat bermanfaat untuk sistem dengan banyak klien, seperti web dan aplikasi mobile, karena menyederhanakan komunikasi dan meningkatkan

keamanan sistem.

B Service Discovery Pattern

Service Discovery digunakan untuk mengatasi permasalahan penemuan lokasi layanan dalam lingkungan *microservices* yang dinamis. Karena microservice dapat mengalami perubahan alamat akibat *scaling* atau *redeployment*, mekanisme *hard-coded endpoint* menjadi tidak efektif.

Melalui *Service Discovery*, setiap layanan mendaftarkan dirinya ke *registry* dan layanan lain dapat menemukan layanan tersebut secara otomatis. Pola ini meningkatkan fleksibilitas dan mendukung otomatisasi dalam sistem *microservices* [14].

C Database per Service Pattern

Pola *Database per Service* mengharuskan setiap *microservice* memiliki basis data sendiri dan tidak mengakses basis data milik layanan lain secara langsung. Pendekatan ini mendukung prinsip *loose coupling* dan independensi layanan.

Tapia dan Gaona [8] menyatakan bahwa pemisahan basis data membantu menghindari konflik skema data dan memungkinkan setiap layanan memilih teknologi penyimpanan yang paling sesuai dengan kebutuhannya.

Namun, pola ini juga menimbulkan tantangan dalam menjaga konsistensi data antar layanan, sehingga sering dikombinasikan dengan pendekatan *eventual consistency*.

D Circuit Breaker Pattern

Circuit Breaker merupakan pola desain yang digunakan untuk mencegah kegagalan berantai (*cascading failure*) dalam sistem *microservices*. Pola ini bekerja dengan memantau kegagalan pada pemanggilan layanan dan menghentikan sementara *request* jika tingkat kegagalan melebihi ambang batas tertentu.

Dengan mekanisme ini, sistem dapat memberikan respons alternatif

(*fallback*) dan menjaga ketersediaan layanan meskipun terjadi gangguan pada salah satu *microservice* [12].

E Saga Pattern

Saga Pattern digunakan untuk mengelola transaksi terdistribusi yang melibatkan beberapa microservice. Alih-alih menggunakan transaksi ACID tradisional, saga memecah transaksi menjadi serangkaian langkah lokal yang saling terkoordinasi.

Jika salah satu langkah gagal, maka akan dijalankan langkah kompensasi untuk mengembalikan sistem ke kondisi yang konsisten. Pola ini umum digunakan pada sistem yang membutuhkan konsistensi data lintas layanan tanpa mengorbankan *skalabilitas* [6].

F Event-Driven Architecture Pattern

Event-Driven Architecture (EDA) merupakan pola desain di mana *microservice* berkomunikasi melalui peristiwa (*event*) yang dipublikasikan ke *message* broker. Layanan lain dapat berlangganan event tersebut tanpa adanya ketergantungan langsung antar layanan.

Menurut Hernandez [9], pola ini meningkatkan *loose coupling* dan memungkinkan sistem bereaksi secara asinkron terhadap perubahan state, sehingga cocok untuk sistem berskala besar dengan alur bisnis yang kompleks.

G *Strangler Pattern*

Strangler Pattern digunakan dalam proses migrasi dari arsitektur *monolith* ke *microservices*. Pola ini dilakukan dengan cara menggantikan bagian-bagian sistem *monolith* secara bertahap dengan *microservice* baru.

Pendekatan ini meminimalkan risiko kegagalan migrasi dan memungkinkan sistem tetap berjalan selama proses transformasi arsitektur berlangsung [3].

H Sidecar Pattern

Sidecar Pattern melibatkan penggunaan komponen pendukung yang berjalan berdampingan dengan *microservice* utama. Komponen sidecar menangani aspek non-bisnis seperti *logging*, *monitoring*, konfigurasi, dan keamanan.

Dengan memisahkan *concern* teknis dari logika bisnis, pola ini meningkatkan konsistensi dan mengurangi kompleksitas kode pada *microservice* utama [14].

I Penerapan Pola Desain pada Sistem Informasi HRD

Dalam konteks Sistem Informasi HRD, pola desain *microservices* dapat diterapkan untuk memisahkan modul seperti rekrutmen, absensi, penggajian, dan penilaian kinerja ke dalam layanan terpisah.

Penggunaan *API Gateway* dan *Service Discovery* membantu mengelola komunikasi antar modul, sementara *Database per Service* dan *Saga Pattern* mendukung pengelolaan data HRD yang kompleks dan saling terkait [15].

2.4 Komunikasi dan Integrasi Microservices

Komunikasi antar *microservice* merupakan aspek krusial dalam sistem terdistribusi. Hernandez [9] menyatakan bahwa kompleksitas komunikasi meningkat seiring bertambahnya jumlah layanan.

Schwarz et al. [6] mengklasifikasikan teknik integrasi *microservices* menjadi komunikasi sinkron (REST, gRPC) dan asinkron (*message queue, event-driven architecture*), yang masing-masing memiliki kelebihan dan kekurangan.

2.5 Skalabilitas dan Performa Microservices

Skalabilitas merupakan salah satu alasan utama adopsi *microservices*. Rodrigues et al. [11] menunjukkan bahwa *microservices* memungkinkan scaling secara horizontal pada layanan tertentu tanpa mempengaruhi keseluruhan sistem.

Namun, studi lain menegaskan bahwa microservices juga menimbulkan

overhead jaringan dan kompleksitas operasional yang harus dikelola dengan baik [16].

2.5.1 Konsistensi Data dan Sinkronisasi Antar Microservices

Konsistensi data merupakan salah satu tantangan utama dalam penerapan arsitektur microservices. Berbeda dengan arsitektur *monolith* yang umumnya menggunakan satu basis data terpusat, microservices menganut prinsip desentralisasi data, di mana setiap layanan memiliki dan mengelola data secara mandiri [8].

Desentralisasi ini meningkatkan *loose coupling* dan independensi layanan, namun juga menimbulkan kompleksitas dalam menjaga konsistensi data antar layanan yang saling bergantung.

A Konsep Konsistensi Data

Konsistensi data mengacu pada kesesuaian dan keakuratan data yang disimpan pada berbagai layanan dalam sistem terdistribusi. Dalam konteks *microservices*, konsistensi tidak selalu bersifat kuat (*strong consistency*), melainkan sering bersifat *eventual consistency*, yaitu kondisi di mana sistem akan mencapai konsistensi dalam jangka waktu tertentu [6].

Pendekatan *eventual consistency* dipilih untuk menjaga *skalabilitas* dan ketersediaan sistem, terutama pada sistem berskala besar dan terdistribusi.

B CAP Theorem dalam Microservices

CAP Theorem menyatakan bahwa sistem terdistribusi tidak dapat secara bersamaan menjamin *Consistency*, *Availability*, dan *Partition Tolerance*. Dalam lingkungan *microservices* yang bersifat terdistribusi, *Partition Tolerance* menjadi kebutuhan utama, sehingga sistem harus memilih antara *consistency* atau *availability* [11].

Sebagian besar implementasi *microservices* memilih *availability* dengan mengorbankan *strong consistency*, dan mengandalkan mekanisme sinkronisasi data untuk mencapai konsistensi akhir.

C Sinkronisasi Data Antar Service

Sinkronisasi data antar *microservice* dilakukan melalui mekanisme komunikasi yang terkontrol, baik secara sinkron maupun asinkron. Pendekatan sinkron biasanya menggunakan *REST* atau *gRPC*, di mana satu layanan secara langsung meminta data ke layanan lain.

Sebaliknya, pendekatan asinkron memanfaatkan *event-driven architecture*, di mana perubahan data pada satu layanan dipublikasikan sebagai *event* dan dikonsumsi oleh layanan lain tanpa ketergantungan langsung [9].

D Eventual Consistency dan Event-Driven Architecture

Event-driven architecture merupakan pendekatan yang umum digunakan untuk mencapai *eventual consistency*. Setiap perubahan *state* pada suatu *microservice* akan menghasilkan *event* yang dipublikasikan ke *message* broker.

Layanan lain yang berlangganan *event* tersebut akan memperbarui data *lokalnya* sesuai kebutuhan. Pendekatan ini meningkatkan *loose coupling* dan mendukung *skalabilitas* sistem, namun memerlukan perancangan yang cermat untuk menghindari duplikasi atau kehilangan *event* [6].

E Saga Pattern untuk Konsistensi Transaksional

Saga Pattern merupakan solusi umum untuk menangani transaksi terdistribusi dalam arsitektur *microservices*. Setiap transaksi besar dipecah menjadi serangkaian transaksi lokal pada masing-masing layanan.

Jika salah satu transaksi gagal, maka akan dijalankan transaksi kompensasi untuk membatalkan perubahan sebelumnya, sehingga sistem tetap berada dalam kondisi yang konsisten [12].

Saga dapat diimplementasikan menggunakan pendekatan orkestrasi atau koreografi, tergantung pada kompleksitas dan kebutuhan sistem.

F Masalah Umum dalam Sinkronisasi Data

Beberapa permasalahan yang sering muncul dalam sinkronisasi data antar microservice antara lain:

- *Data duplication* antar layanan
- Latensi dalam propagasi perubahan data
- *Inconsistency* sementara (*temporary inconsistency*)
- Kompleksitas penanganan kegagalan komunikasi

Menurut Su et al. [2], permasalahan ini sering menjadi alasan kegagalan adopsi *microservices* jika tidak diantisipasi sejak tahap perancangan arsitektur.

G Relevansi Konsistensi Data pada Sistem Informasi HRD

Sistem Informasi HRD memiliki kebutuhan konsistensi data yang tinggi, khususnya pada data karyawan, penggajian, dan absensi. Ketidaksinkronan data antar layanan dapat berdampak langsung pada akurasi perhitungan gaji dan pelaporan manajemen.

Oleh karena itu, penerapan mekanisme sinkronisasi data yang tepat, seperti *event-driven architecture* dan *Saga Pattern*, menjadi faktor krusial dalam keberhasilan implementasi *microservices* pada sistem informasi HRD [15].

2.6 Resiliensi dan Keandalan Sistem Microservices

Resiliensi sistem *microservices* menjadi tantangan tersendiri karena sifatnya yang terdistribusi. Muzeeb [12] mengidentifikasi berbagai pola pemulihan seperti retry mechanism, fallback, dan circuit breaker sebagai solusi umum untuk meningkatkan keandalan sistem.

Penerapan pola-pola ini penting untuk menjaga kontinuitas layanan pada sistem informasi kritikal seperti HRD.

2.7 Pengujian Perangkat Lunak

Pengujian perangkat lunak merupakan proses sistematis yang bertujuan untuk memastikan bahwa perangkat lunak berfungsi sesuai dengan kebutuhan yang telah ditentukan serta bebas dari kesalahan yang dapat mempengaruhi kualitas sistem. Dalam arsitektur microservices, pengujian memiliki peran yang sangat penting karena sistem terdiri dari banyak layanan yang saling berinteraksi [4].

Pendekatan pengujian yang tepat membantu meningkatkan keandalan, keamanan, dan *maintainability* sistem, khususnya pada sistem terdistribusi.

2.7.1 Unit Testing

Unit Testing adalah metode pengujian yang dilakukan pada unit terkecil dari perangkat lunak, seperti fungsi, metode, atau kelas, secara terisolasi. Tujuan utama unit testing adalah untuk memastikan bahwa setiap unit logika program bekerja sesuai dengan spesifikasi yang diharapkan.

Dalam konteks *microservices*, unit testing *difokuskan* pada pengujian logika bisnis internal dari *masing-masing* layanan tanpa melibatkan dependensi eksternal seperti basis data atau layanan lain. Dependensi tersebut biasanya disimulasikan menggunakan teknik *mocking* atau *stubbing*.

Menurut Waseem et al. [4], unit testing membantu mendeteksi kesalahan sejak tahap awal pengembangan, sehingga mengurangi biaya perbaikan dan meningkatkan kualitas kode secara keseluruhan.

2.7.2 Integration Testing

Integration Testing bertujuan untuk menguji interaksi antar komponen atau layanan dalam suatu sistem. Pada arsitektur *microservices*, *integration testing* dilakukan untuk memastikan bahwa komunikasi antar *microservice* berjalan dengan benar dan sesuai dengan kontrak yang telah ditentukan.

Pengujian ini mencakup verifikasi alur data, format pesan, serta penanganan kesalahan dalam komunikasi antar layanan. *Integration testing* menjadi krusial karena kegagalan sering terjadi bukan pada logika internal layanan, melainkan pada integrasi antar layanan [1].

Pendekatan yang umum digunakan dalam *integration* testing *microservices* meliputi *contract testing* dan penggunaan lingkungan *staging* yang menyerupai kondisi produksi.

2.7.3 API Testing

API Testing merupakan jenis pengujian yang berfokus pada antarmuka layanan atau *Application Programming Interface (API)*. Pengujian ini bertujuan untuk memastikan bahwa API dapat menerima *request*, memproses data, dan mengembalikan *response* sesuai dengan spesifikasi yang telah ditentukan.

Dalam arsitektur microservices, API menjadi sarana utama komunikasi antar layanan, sehingga kualitas API sangat mempengaruhi stabilitas sistem secara keseluruhan. API testing mencakup pengujian validasi input, status kode HTTP, struktur response, performa, dan keamanan API [5].

API testing biasanya dilakukan secara otomatis dan diintegrasikan dalam *pipeline CI/CD* untuk memastikan bahwa setiap perubahan kode tidak merusak fungsionalitas layanan yang telah ada.

2.7.4 Peran Pengujian dalam Arsitektur Microservices

Pengujian perangkat lunak pada arsitektur *microservices* tidak dapat bergantung pada satu jenis pengujian saja. Kombinasi unit testing, *integration* testing, dan API testing diperlukan untuk memastikan kualitas sistem secara menyeluruh.

Pendekatan pengujian berlapis ini membantu mendeteksi kesalahan pada berbagai tingkat sistem, mulai dari logika internal layanan hingga interaksi antar layanan, sehingga mendukung penerapan *microservices* yang andal dan berkelanjutan [4].

2.8 Microservices pada Sistem Informasi HRD

Sistem Informasi HRD memiliki karakteristik kompleksitas data, proses bisnis yang saling terintegrasi, serta kebutuhan akan keandalan tinggi. Gumilar et al. [15] menunjukkan bahwa penerapan *microservices* pada sistem HRD berbasis

web dapat meningkatkan fleksibilitas pengembangan dan kemudahan pemeliharaan sistem.

Pendekatan *microservices* memungkinkan pemisahan modul HRD seperti rekrutmen, absensi, penggajian, dan penilaian kinerja ke dalam layanan terpisah, sehingga sistem lebih mudah dikembangkan seiring pertumbuhan organisasi.

2.9 Ringkasan Tinjauan Pustaka

Berdasarkan studi pustaka yang telah dibahas, dapat disimpulkan bahwa arsitektur *microservices* menawarkan keunggulan dalam hal skalabilitas, fleksibilitas, dan *maintainability* dibandingkan arsitektur *monolith*. Namun, implementasinya memerlukan perencanaan matang, terutama terkait komunikasi, integrasi, dan resiliensi sistem.

Tinjauan pustaka ini menjadi dasar teoritis dalam perancangan dan implementasi arsitektur *microservices* pada Sistem Informasi HRD yang dibahas pada bab selanjutnya.