BAB 1
PENDAHULUAN

Transformasi digital yang tengah berlangsung di hampir seluruh sektor industri telah menempatkan sistem informasi berbasis perangkat lunak sebagai elemen strategis yang tidak dapat dipisahkan dari keberlangsungan operasional organisasi. Di antara berbagai komponen sistem informasi tersebut, *Human Resource Information System* (HRIS) menempati posisi yang sangat krusial karena berkaitan langsung dengan pengelolaan aset manusia—sumber daya yang paling menentukan dalam pertumbuhan jangka panjang sebuah perusahaan [1]. Sistem ini mencakup berbagai fungsi yang sifatnya lintas departemen, mulai dari pencatatan kehadiran, pengajuan dan persetujuan cuti, rekap lembur, pengelolaan data medis, hingga alur persetujuan manajerial yang melibatkan banyak pihak.

Namun, tidak sedikit perusahaan yang saat ini masih menggantungkan operasional HR mereka pada sistem yang dibangun satu dekade atau lebih yang lalu—sistem yang dalam literatur rekayasa perangkat lunak dikenal sebagai *legacy system*. Keberadaan sistem *legacy* bukanlah sesuatu yang bisa diabaikan; sebaliknya, ia merepresentasikan akumulasi bertahun-tahun dari keputusan teknis, adaptasi bisnis, dan pengetahuan domain yang tertanam dalam jutaan baris kode. Justru di sinilah letak kompleksitasnya: sistem *legacy* tidak bisa begitu saja dibuang, tetapi juga sulit untuk terus dipertahankan tanpa konsekuensi yang semakin berat.

Kerja magang yang dilakukan oleh penulis di PT Kompas Gramedia pada periode 1 Januari hingga 31 Maret 2026 berkisar pada permasalahan inilah. Penulis ditugaskan sebagai *Backend Engineer* untuk berkontribusi pada proyek migrasi sistem *backend* HR dari platform PHP berbasis SAP ke layanan Go (*Golang*) modern dengan arsitektur yang terstruktur dan terintegrasi dengan Odoo ERP dan SQL Server 2014. Bab ini akan memaparkan konteks permasalahan yang melatarbelakangi keputusan modernisasi tersebut, tujuan yang ingin dicapai, serta gambaran umum mengenai waktu dan prosedur pelaksanaan magang.

1.1 Latar Belakang Masalah

1.1.1 Fenomena Legacy System dalam Industri Perangkat Lunak

Dalam literatur rekayasa perangkat lunak, tidak ada definisi tunggal yang diterima secara universal untuk istilah *legacy system*. Namun, sejumlah peneliti membangun konsensus di sekitar karakteristik tertentu: sistem *legacy* umumnya

dicirikan oleh usia yang sudah lanjut, ketergantungan pada teknologi yang sudah usang, dokumentasi yang minim atau tidak akurat, serta kesulitan yang signifikan dalam melakukan modifikasi dan pengujian [2]. Monaghan dan Bass menambahkan dimensi penting dengan memandang *legacy* bukan semata-mata sebagai masalah usia teknologi, melainkan sebagai manifestasi dari *technical debt* yang telah terakumulasi sedemikian rupa sehingga kemampuan sistem untuk beradaptasi dengan kebutuhan baru menjadi sangat terbatas [3].

Literatur rekayasa perangkat lunak secara konsisten menggambarkan pola yang berulang dalam siklus hidup *legacy system*. Seiring waktu, modifikasi demi modifikasi yang dilakukan tanpa disertai refactoring yang memadai mulai mengikis integritas arsitektur aslinya—sebuah fenomena yang dikenal sebagai *architectural erosion* [2]. Pengembang baru yang bergabung ke dalam tim menghadapi kode yang sulit dipahami karena developer aslinya sudah tidak ada, dokumentasi sudah kadaluarsa, dan logika bisnis tersebar tanpa struktur yang jelas. Akibatnya, upaya untuk menambahkan fitur baru atau memperbaiki *bug* menjadi semakin mahal dari segi waktu dan risiko: setiap perubahan berpotensi menimbulkan efek samping yang tidak terduga di bagian lain dari sistem.

Kondisi ini bukan hanya masalah estetika kode, melainkan masalah bisnis yang nyata. Studi yang dilakukan oleh Da Silva et al. [4] dalam konteks re-engineering sistem perpajakan di Brasil melaporkan bahwa sebelum modernisasi dilakukan, tim membutuhkan 267 hari dan biaya hingga R$107.000 untuk menangani sejumlah bug. Setelah modernisasi, angka tersebut turun menjadi 48 hari dan R$19.000—penurunan drastis yang menggambarkan betapa besarnya biaya yang ditanggung organisasi akibat mempertahankan sistem *legacy* tanpa strategi modernisasi yang jelas. Asatiani et al. mempertegas temuan ini dengan argumentasi bahwa *technical debt* yang tidak dikelola secara aktif pada akhirnya dapat menghambat seluruh agenda transformasi digital organisasi [5].

1.1.2 Profil Sistem Legasi PHP di Lingkungan HR

Sistem *backend* HR yang dihadapi dalam konteks kerja magang ini merupakan representasi nyata dari permasalahan *legacy* yang diuraikan di atas. Sistem tersebut dibangun di atas PHP—sebuah bahasa pemrograman yang telah berjasa besar dalam ekosistem web sejak era 1990-an, tetapi memiliki keterbatasan inheren yang mulai terasa beban saat digunakan dalam konteks *backend* modern yang menuntut keandalan tinggi dan kemampuan penanganan beban konkuren yang

besar.

Kode *legacy* yang ada—lebih dari 240 file PHP berisi modul ESS (*Employee Self-Service*), MSS (*Manager Self-Service*), proses payroll, pengelolaan data personal, dan berbagai workflow HR lainnya—mencerminkan arsitektur yang tumbuh secara organik tanpa rancangan struktural yang konsisten. Logika bisnis, query SQL, dan respons HTTP bercampur dalam file yang sama, tanpa pemisahan yang jelas antara lapisan presentasi, aplikasi, dan data. Pola semacam ini dalam kajian arsitektur perangkat lunak dikenal sebagai *spaghetti code* atau *God object*—kondisi di mana satu komponen menanggung terlalu banyak tanggung jawab sehingga setiap modifikasi berisiko merusak bagian lain yang tidak terduga.

Di samping masalah struktural, sistem tersebut juga bergantung pada koneksi SAP RFC (*Remote Function Call*) untuk sejumlah operasi penting. Ketergantungan pada SAP RFC ini menambahkan lapisan kompleksitas tersendiri: selain membutuhkan lisensi yang mahal, RFC merupakan protokol tertutup (*proprietary*) yang sulit diuji secara otomatis dan memerlukan infrastruktur tambahan yang tidak fleksibel. Ini adalah bentuk *coupling* yang tidak diinginkan antara logika bisnis HR dan infrastruktur komunikasi sistem lama.

1.1.3 Problematika Performa dan Skalabilitas PHP untuk Layanan Backend Modern

Keputusan untuk me-*rewrite* sistem bukan semata-mata didorong oleh masalah arsitektur, tetapi juga oleh kebutuhan performa yang tidak dapat dipenuhi oleh PHP dengan model eksekusinya. PHP secara tradisional beroperasi dalam model *single-threaded per request*: setiap permintaan yang masuk membutuhkan proses PHP tersendiri, yang berarti kemampuan penanganan konkuren dibatasi oleh jumlah proses yang tersedia. Model ini berbeda secara fundamental dari bahasa yang dikompilasi seperti Go, yang mengimplementasikan model konkuren berbasis *goroutine*—unit eksekusi ringan yang dapat beroperasi secara paralel dengan *overhead* memori yang sangat rendah.

Penelitian empiris yang dilakukan oleh Yang dan Aklani [6] memberikan bukti kuantitatif yang signifikan tentang perbedaan ini. Dalam perbandingan langsung antara Go/Gin dan PHP/Laravel, Gin mampu memproses lebih dari 86 kali lebih banyak permintaan per detik dibandingkan Laravel dalam kondisi beban yang sama. Meski angka tersebut mencerminkan perbandingan framework bukan bahasa secara murni—Laravel dikenal memiliki overhead yang lebih besar dibandingkan

framework PHP yang lebih ringan—besaran gap yang dilaporkan tetap secara jelas memperlihatkan keunggulan arsitektural Go untuk layanan *backend* yang padat lalu lintas.

Perbedaan ini bersumber dari sifat fundamental kedua bahasa. Go dikompilasi langsung ke *machine code*, sementara PHP diinterpretasikan saat *runtime*. Go memiliki *garbage collector* yang dioptimalkan untuk latensi rendah. Go mendukung konkurensi secara native melalui *goroutine* dan *channel*, memungkinkan server untuk melayani ribuan koneksi simultan dengan konsumsi memori yang jauh lebih efisien. Dalam analisis komparatif bahasa untuk *microservice*, Hussein et al. [7] menempatkan Go sebagai pilihan optimal untuk layanan dengan kebutuhan latensi rendah dan konkuren tinggi, satu posisi yang bahkan tidak menjadi pertimbangan bagi PHP dalam konteks arsitektur yang sama.

Untuk lingkungan HR di mana sejumlah besar karyawan mungkin mengakses sistem secara bersamaan—misalnya di awal bulan untuk melihat slip gaji, atau di periode puncak pengajuan cuti—kemampuan menangani beban konkuren yang tinggi bukan sekadar keunggulan teknis, melainkan kebutuhan fungsional yang nyata.

1.1.4 Tantangan Keamanan dan Kepatuhan pada Sistem Legasi

Di samping persoalan performa dan biaya pemeliharaan, sistem *legacy* seringkali menjadi mata rantai terlemah dalam postur keamanan informasi organisasi. Penggunaan bahasa pemrograman dan *framework* versi lama—seperti PHP era lama dalam konteks ini—membawa risiko kerentanan yang tidak lagi menerima pemutakhiran keamanan (*security patches*) dari komunitas pengembang aslinya. Dalam lanskap ancaman siber kontemporer, ketergantungan pada teknologi *deprecated* merupakan pelanggaran terhadap prinsip *security-by-design* yang diamanatkan oleh berbagai standar sertifikasi internasional seperti ISO/IEC 27001 [2].

Sistem HR, yang mengelola data pribadi karyawan (*Personally Identifiable Information* - PII), memiliki kewajiban hukum dan etis yang tinggi untuk mematuhi regulasi perlindungan data. Arsitektur sistem *legacy* yang cenderung monolitik dan kurang memiliki lapisan abstraksi keamanan yang granular seringkali sulit untuk mengimplementasikan kontrol akses modern seperti *Role-Based Access Control* (RBAC) yang ketat atau audit log yang komprehensif. Modernisasi ke Go memungkinkan penerapan standar keamanan terbaru, termasuk TLS 1.2+, enkripsi

data saat istirahat (*at rest*), serta validasi input yang lebih ketat melalui sistem tipe bahasa yang statis.

1.1.5 Urgensi Modernisasi: Dari Technical Debt ke Keputusan Strategis

Keputusan untuk memodernisasi sistem *legacy* tidak pernah sederhana. Schneidewind dan Ebert [8] menyatakan dengan tepat bahwa sistem *legacy* ”tidak dihentikan atau dirancang ulang tanpa alasan yang kuat” karena sistem tersebut menyimpan logika bisnis yang kritis dan pengetahuan domain yang mungkin tidak pernah terdokumentasi di tempat lain. Setiap strategi modernisasi membawa risikonya sendiri: rewrite besar-besaran (*big bang*) rentan gagal karena kompleksitas yang tidak terprediksi, sementara pendekatan bertahap membutuhkan koordinasi yang lebih cermat namun menawarkan kontrol risiko yang lebih baik.

Dalam kasus ini, strategi yang dipilih adalah migrasi *strangler fig* yang bertahap: sistem PHP lama tetap beroperasi sebagai referensi selama sistem Go baru dikembangkan secara paralel, dengan setiap modul yang sudah selesai dimigrasikan mulai mengambil alih tanggung jawab secara bertahap. Pendekatan ini selaras dengan rekomendasi dari Fonseca et al. [9], yang berdasarkan pengalaman industri nyata menyimpulkan bahwa migrasi inkremental dengan validasi berkelanjutan memberikan hasil yang lebih dapat diprediksi dibandingkan pendekatan *big bang*.

Assunção et al. [10] dalam survei sistematis mereka terhadap strategi modernisasi perangkat lunak kontemporer menemukan bahwa tekanan dari ketidakmampuan sistem lama untuk beradaptasi dengan perubahan bisnis merupakan pendorong utama keputusan modernisasi, lebih dominan dibandingkan faktor teknis semata. Ini sesuai dengan konteks yang ada: sistem HR harus dapat mengakomodasi perubahan kebijakan ketenagakerjaan, integrasi dengan platform baru seperti Odoo ERP, serta penambahan modul-modul baru yang tidak diantisipasi saat sistem PHP dibangun. Fleksibilitas arsitektur menjadi kebutuhan nyata, bukan sekadar aspirasi teknis.

1.1.6 Clean Architecture sebagai Fondasi Modernisasi

Dalam menjawab kebutuhan pemeliharaan jangka panjang, sistem Go yang baru dirancang menggunakan prinsip *Clean Architecture*—sebuah pendekatan arsitektur perangkat lunak yang menekankan pemisahan tanggung jawab (*separation of concerns*) yang tegas melalui pelapisan (*layering*) yang eksplisit dan

aturan ketergantungan yang jelas: lapisan luar boleh bergantung pada lapisan dalam, tetapi tidak sebaliknya.

Bukti empiris mendukung nilai pendekatan ini. Spray et al. [11] mendemonstrasikan dalam penelitian mereka tentang *Abstraction Layered Architecture*—sebuah varian konseptual yang sangat dekat dengan *Clean Architecture*—bahwa pemisahan lapisan yang jelas secara konsisten meningkatkan modularitas, *reusability*, *analysability*, dan *testability* dalam jangka menengah hingga panjang, meski mungkin menambah kompleksitas awal. Khalilipour et al. [12] memperkuat hal ini melalui studi refactoring sistem *legacy* untuk pemisahan lapisan: proses tersebut terbukti meningkatkan *loose coupling* dan *cohesion*, dua indikator utama dalam pengukuran *maintainability* menurut standar ISO 25010.

Arango dan Loaiza [13] menunjukkan hasil yang konkret dalam konteks proyek nyata: penerapan *Clean Architecture* dalam kerangka Scrum berhasil mengurangi duplikasi kode dari 4,8% menjadi 3,5% dan *technical debt* dari 6,3% menjadi 1,3% dalam beberapa iterasi pengembangan. Angka-angka ini, meski berasal dari konteks yang berbeda, memberikan indikasi yang persuasif bahwa investasi awal dalam arsitektur yang terstruktur memberikan imbal balik yang terukur dalam pemeliharaan jangka panjang.

Dalam implementasi di proyek ini, *Clean Architecture* diterjemahkan ke dalam empat lapisan utama yang tersegregasi secara tegas:

1. **Lapisan *Handler*** — bertanggung jawab atas validasi input HTTP, parsing parameter permintaan, dan pemetaan hasil ke dalam format respons yang sesuai. Lapisan ini tidak mengandung logika bisnis apapun.

2. **Lapisan *Service*** — mengorkestrasikan logika bisnis (*use-case*) secara murni, tanpa ketergantungan pada *framework* HTTP atau detail implementasi infrastruktur apapun.

3. **Lapisan *Integration/Repository*** — mengisolasi seluruh komunikasi dengan sistem eksternal, baik Odoo ERP melalui HTTP, maupun SQL Server 2014 melalui koneksi database. Detail protokol komunikasi tidak bocor ke lapisan atas.

4. **Lapisan *Domain*** — mendefinisikan model data, antarmuka (*interface*), dan *domain error* yang menjadi bahasa bersama seluruh lapisan, bebas dari ketergantungan pada infrastruktur apapun.

Setiap lapisan berinteraksi melalui antarmuka yang terdefinisi dengan kontrak yang eksplisit, sehingga penggantian implementasi di satu lapisan—misalnya mengganti penyedia data dari SQL Server ke sistem baru—dapat dilakukan tanpa mengubah lapisan lainnya. Ini adalah keunggulan arsitektural yang sangat berharga dalam konteks modernisasi bertahap di mana perubahan akan terus terjadi sepanjang masa transisi.

1.1.7 Integrasi Multi-Sumber Data: Odoo ERP dan SQL Server 2014

Salah satu tantangan arsitektural paling signifikan dalam proyek ini adalah keharusan mengintegrasikan dua sumber data yang secara fundamental berbeda: Odoo ERP sebagai sistem HR utama yang baru, dan SQL Server 2014 sebagai basis data *legacy* yang tidak dapat sepenuhnya digantikan dalam periode magang.

Odoo ERP merupakan sistem ERP *open-source* yang memiliki modul HR komprehensif, akses data melalui REST API yang terautentikasi dengan mekanisme *token-based authentication*. SQL Server 2014, di sisi lain, adalah basis data relasional yang sudah berusia lebih dari satu dekade dan menyimpan data historis yang belum dapat dimigrasikan sepenuhnya dalam waktu singkat. Situasi ini memaksa sistem baru untuk berperan sebagai *gateway* yang cerdas: mengagregasikan, menggabungkan, dan menormalisasi data dari kedua sumber tersebut sebelum menyajikannya ke konsumen API dalam format yang konsisten.

Integrasi dengan Odoo diimplementasikan melalui mekanisme autentikasi *service-to-service*: *backend* Go mengambil token Odoo secara internal menggunakan kredensial layanan, mengelola siklus hidupnya (termasuk *cache* dan *refresh* saat token kedaluwarsa), serta melakukan *retry* secara otomatis ketika menerima respons 401/403 dari *upstream*. Ini adalah desain yang lebih aman dibandingkan pendekatan *legacy* yang meneruskan kredensial pengguna secara langsung.

Untuk meningkatkan ketahanan (*resilience*) sistem terhadap gangguan *upstream* Odoo, *circuit breaker* diimplementasikan menggunakan library `sony/gobreaker`. Mekanisme ini memungkinkan sistem untuk mendeteksi kegagalan *upstream* yang berulang dan secara otomatis menghentikan aliran permintaan ke sistem yang sedang tidak sehat, memberikan waktu bagi sistem tersebut untuk pulih sebelum percobaan koneksi ulang dimulai. Kegagalan dengan kode HTTP 4xx (termasuk 401 dan 403) dengan sengaja *tidak* dihitung sebagai kegagalan *circuit breaker* karena bersifat klien, bukan indikasi gangguan sistem.

1.1.8 RESTful API sebagai Kontrak Komunikasi

Seluruh antarmuka layanan yang dibangun menggunakan gaya arsitektur REST (*Representational State Transfer*), yang pertama kali diformulasikan oleh Roy Fielding dalam disertasi doktoralnya dan kemudian menjadi fondasi dominan dalam desain layanan web modern [14]. REST menekankan antarmuka yang seragam (*uniform interface*), *statelessness*, dan kemampuan *caching*, tiga sifat yang secara langsung mendukung skalabilitas dan interoperabilitas sistem enterprise.

Sahara et al. [15] mendemonstrasikan dalam konteks integrasi sistem akademik bahwa penggunaan REST Web Services secara signifikan meningkatkan efisiensi sinkronisasi data dibandingkan proses manual atau metode integrasi konvensional. Dalam konteks HR di PT Kompas Gramedia, desain RESTful API yang konsisten memungkinkan aplikasi mobile dan web *frontend* untuk mengonsumsi data dari berbagai modul—kehadiran, cuti, lembur, substitusi—melalui satu *endpoint* yang terdokumentasi dengan baik dalam format OpenAPI 3.0.3.

Vinoski [14] menekankan pentingnya konsistensi dalam desain RESTful API: penggunaan metode HTTP yang semantis, kode status yang akurat, dan struktur respons yang dapat diprediksi. Prinsip-prinsip ini diterapkan pada puluhan *endpoint* dalam proyek `hr-backend`, termasuk penanganan *error* yang terstandarisasi dengan format JSON yang seragam di berbagai modul.

1.1.9 Konteks HRIS: Lebih dari Sekadar Masalah Teknis

Penting untuk memahami bahwa modernisasi sistem HRIS bukan sekadar persoalan teknis. Shukur et al. [1] dalam studi implementasi HRMS universitas mereka menunjukkan bahwa dimensi kemanfaatan (*usability*) dan kepercayaan pengguna terhadap sistem sama pentingnya dengan parameter teknis seperti performa dan keandalan. Sistem yang dibangun harus tidak hanya cepat dan andal, tetapi juga mampu menyajikan data yang akurat dan konsisten kepada karyawan yang mengandalkannya untuk keputusan-keputusan penting seperti pengajuan cuti, konfirmasi substitusi, atau verifikasi status persetujuan.

Migrasi yang dilakukan dengan hati-hati dan terstruktur—mempertahankan kompatibilitas semantik dengan perilaku sistem lama yang sudah dikenal pengguna, sambil secara bertahap memperkenalkan kemampuan baru—adalah bentuk penghormatan terhadap kebutuhan pengguna yang sesungguhnya, bukan

semata-mata kepuasan teknis pengembangnya.

1.2 Maksud dan Tujuan Kerja Magang

Maksud dari kerja magang ini adalah sebagai berikut.

1. Memenuhi salah satu persyaratan akademik untuk kelulusan mata kuliah Kerja Magang di Program Studi Informatika Universitas Multimedia Nusantara.
2. Memberikan kontribusi nyata pada proyek migrasi sistem *backend* HR dari platform PHP *legacy* ke layanan Go (*Golang*) modern yang sedang dikerjakan oleh tim *backend engineering* perusahaan.
3. Mengaplikasikan pengetahuan rekayasa perangkat lunak yang diperoleh selama perkuliahan—khususnya dalam aspek desain arsitektur, pemrograman berbasis bahasa bertipe statis, pengujian unit, dan integrasi sistem—dalam lingkungan proyek nyata dengan kompleksitas dan tekanan jadwal yang sesungguhnya.
4. Memperdalam pemahaman tentang tantangan dan strategi modernisasi sistem *legacy* dalam konteks industri, termasuk pengambilan keputusan teknis yang mempertimbangkan *tradeoff* antara risiko, kecepatan, dan kualitas.

Tujuan dari kerja magang ini adalah melakukan *rewrite* sistem *backend* HR dari PHP ke Go dengan pendekatan *Clean Architecture* dan integrasi multi-sumber data.

1.3 Waktu dan Prosedur Kerja Magang

1.3.1 Waktu Pelaksanaan

Kerja magang dilaksanakan selama tiga bulan penuh, terhitung sejak 1 Januari 2026 hingga 31 Maret 2026 (sekitar 13 minggu), dengan status penuh waktu (*full-time*) sebagai *Backend Engineer* di PT Kompas Gramedia di bawah bimbingan lapangan FX Endri Harmanto, sesuai dengan ketentuan program PRO-STEP Universitas Multimedia Nusantara. Selama periode tersebut, penulis bekerja dengan jam kerja reguler yang berlaku di perusahaan.

1.3.2 Prosedur Pelaksanaan

Pelaksanaan kerja magang di PT Kompas Gramedia mengikuti serangkaian prosedur yang ditetapkan baik oleh Universitas Multimedia Nusantara maupun oleh perusahaan.

Proses Rekrutmen dan Penerimaan

Proses magang diawali dengan tahap seleksi yang dilakukan oleh PT Kompas Gramedia. Profil penulis sebagai mahasiswa Informatika dengan minat dan pengalaman di bidang pemrograman *backend* dinilai relevan dengan kebutuhan tim yang saat itu sedang mengerjakan proyek modernisasi sistem HR. Setelah melalui proses seleksi, penulis resmi bergabung pada tanggal 1 Januari 2026 sebagai *software engineer intern* dengan peran sebagai *Backend Engineer*, di bawah bimbingan langsung FX Endri Harmanto.

Orientasi dan Pengenalan Proyek

Pada minggu pertama magang, penulis menjalani sesi orientasi yang mencakup:

1. Pengenalan struktur organisasi tim *engineering* dan jalur komunikasi yang berlaku.
2. Pemahaman mendalam tentang arsitektur proyek `hr-backend`: struktur direktori, konvensi kode, standar *static analysis* dengan `golangci-lint` v2, serta panduan pengembangan yang telah ditetapkan oleh tim.
3. Pengantar sistem *legacy* PHP yang menjadi referensi migrasi: memahami pemetaan fungsionalitas antara modul PHP lama dan *endpoint* Go baru berdasarkan dokumentasi teknis internal yang tersedia.
4. *Setup* lingkungan pengembangan lokal, termasuk koneksi to SQL Server 2014 dan konfigurasi variabel lingkungan untuk integrasi Odoo ERP.

Metodologi Kerja

Pengerjaan tugas mengikuti pendekatan iteratif yang terstruktur. Setiap modul yang dikerjakan melewati tahapan berikut:

1. **Analisis Kode Legacy**: Membaca dan memahami file PHP yang bersesuaian untuk mengidentifikasi logika bisnis, query SQL, parameter yang diperlukan, dan perilaku yang harus direplikasi di sistem baru.

2. **Desain Arsitektur**: Menentukan struktur lapisan yang sesuai—apakah *endpoint* cukup menggunakan pola *generic proxy* ke Odoo, atau membutuhkan *handler* dan *service* spesifik dengan query ke SQL Server.

3. **Implementasi**: Menulis kode di setiap lapisan secara berurutan (domain → integration/repository → service → handler → router), memastikan setiap lapisan memperturut aturan ketergantungan yang ditetapkan arsitektur.

4. **Pengujian Unit**: Membuat unit test di direktori `test/unit/...` dengan struktur *mirror* mengikuti struktur `internal/...`, menggunakan pendekatan *black-box testing* dengan paket `xxx_test`.

5. **Validasi Kualitas Kode**: Menjalankan `goimports -w .` untuk format dan konsistensi impor, diikuti `golangci-lint run` untuk memastikan tidak ada pelanggaran terhadap aturan lint yang ditetapkan, dan `go test ./...` untuk memverifikasi seluruh pengujian lulus.

6. **Pembaruan Dokumentasi**: Memperbarui `openapi.yaml` dengan spesifikasi *endpoint* baru, termasuk *request/response schema* yang akurat, serta menambahkan contoh HTTP ke `example.http`.

7. **Tinjauan dan Umpan Balik**: Menyerahkan pekerjaan untuk ditinjau oleh pembimbing lapangan dan menerima umpan balik untuk penyempurnaan.

Standar Kualitas yang Diterapkan

Seluruh kode yang dikerjakan wajib memenuhi standar kualitas yang ditetapkan tim, yang mencakup:

1. Kepatuhan terhadap konfigurasi `golangci-lint` v2 yang mencakup lebih dari 30 kumpulan aturan, mulai dari deteksi *error* yang tidak ditangani (`errcheck`), potensi celah keamanan (`gosec`), manajemen sumber daya (`bodyclose`, `sqlclosecheck`), hingga konvensi dokumentasi kode (`revive`).

2. Setiap fungsi publik dan fungsi utama non-trivial wajib memiliki komentar dokumentasi dalam bahasa Indonesia yang menjelaskan tujuan dan niat logika, bukan sekadar mengulang sintaks kode.
3. Komentar berlabel langkah (`// 1. Ambil parameter...`, `// 2. Validasi input...`) wajib digunakan pada blok logika yang memiliki lebih dari satu tahap penting, untuk memfasilitasi navigasi kode bagi anggota tim yang akan memeliharanya di masa depan.

Alat Pengembangan

Selama pelaksanaan magang, beberapa alat pengembangan utama yang digunakan adalah:

1. **Go 1.24** — bahasa utama pengembangan *backend*.
2. **Gin Framework** — framework HTTP yang ringan dan berkinerja tinggi untuk Go.
3. **`goimports`** — untuk format kode dan konsistensi pengelolaan impor paket.
4. **`golangci-lint` v2** — untuk *static analysis* dan penjagaan kualitas kode secara otomatis.
5. **`air`** — untuk *live-reload* selama pengembangan lokal.
6. **Microsoft SQL Server 2014** — basis data *legacy* yang tetap diintegrasikan.
7. **Odoo ERP** — sistem HR utama yang menjadi sumber data primer melalui REST API.
8. **`go-sqlmock`** — untuk simulasi *database* dalam pengujian unit.
9. **`go.uber.org/mock`** — untuk pembuatan *mock* antarmuka dalam pengujian unit.