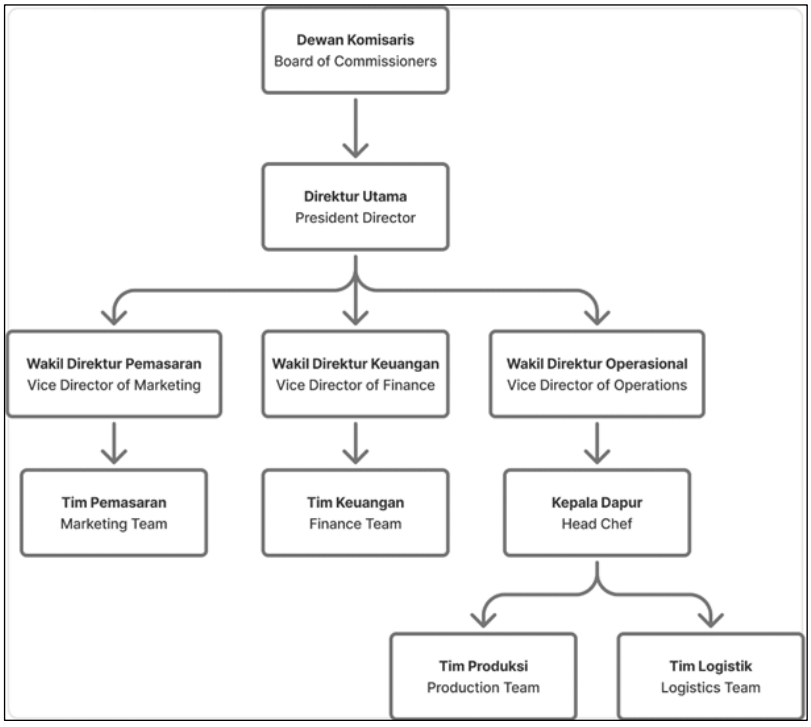
BAB II
TINJAUAN PUSTAKA

2.1. Dasar Teori

Teori dasar yang diuraikan pada bab ini merupakan aspek-aspek yang mendukung penelitian, bersumber dari jurnal, buku, dokumentasi resm dan referensi lainnya.

2.1.1. Nusantara Kuliner

"Nusantara Kuliner" merupakan sebuah Usaha Mikro, Kecil, dan Menengah (UMKM) yang bergerak di bidang penjualan makanan, jasa catering, dan penyediaan paket langganan makanan harian. UMKM ini berdiri dengan visi untuk memperkenalkan dan melestarikan cita rasa kuliner khas Nusantara melalui pendekatan modern yang berbasis digital. Dalam operasionalnya, "Nusantara Kuliner" menyediakan beragam produk kuliner yang mengusung konsep makanan tradisional dengan sentuhan modern. Menu yang disajikan mencakup hidangan harian, paket *catering* acara, serta menu langganan mingguan dan bulanan untuk pelanggan individu maupun korporasi. Selain itu, usaha ini juga menerima pesanan khusus untuk kebutuhan acara seperti seminar, rapat, pesta, dan perayaan keluarga.

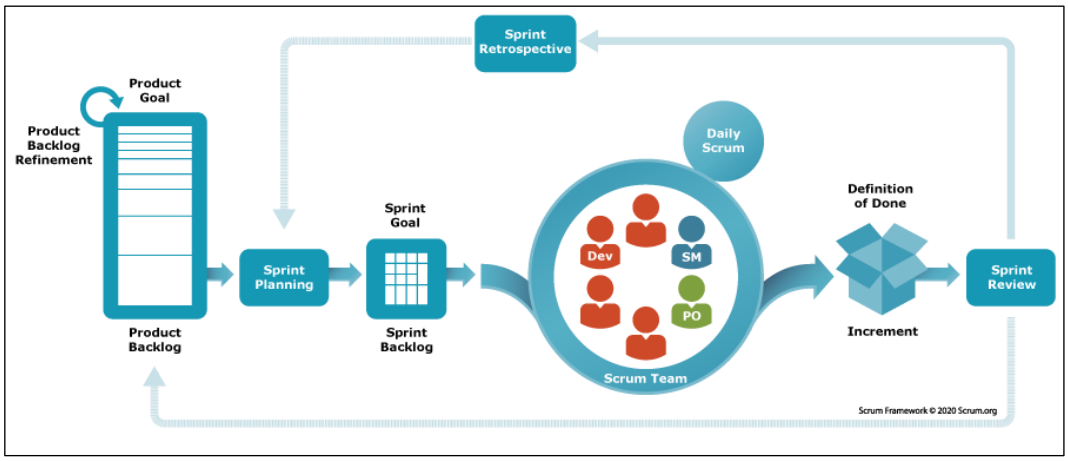


Gambar 2.1 Struktur Organisasi Nusantara Kuliner

Gambar 2.1 merupakan struktur organisasi dari UMKM “Nusantara Kuliner” yang memiliki beberapa peran penting dalam mendukung kelancaran proses bisnisnya, yaitu Dewan Komisaris, Direktur Utama, Wakil Direktur (yang terdiri dari Wakil Direktur Pemasaran, Wakil Direktur Keuangan, dan Wakil Direktur Operasional), serta divisi-divisi pelaksana seperti Tim Pemasaran, Tim Keuangan, Kepala Dapur, Tim Produksi, dan Tim Logistik. Dewan Komisaris merupakan unsur pengawas tertinggi yang memiliki wewenang untuk memberikan arahan strategis kepada Direktur Utama. Direktur Utama berperan sebagai pimpinan utama yang mengatur seluruh kegiatan operasional perusahaan secara menyeluruh. Di bawah Direktur Utama, terdapat tiga Wakil Direktur yang bertanggung jawab pada bidang fungsionalnya masing-masing. Wakil Direktur Pemasaran membawahi Tim Pemasaran yang menjalankan kegiatan promosi, branding, dan hubungan pelanggan. Wakil Direktur Keuangan mengoordinasikan Tim Keuangan yang bertugas dalam pencatatan transaksi, pengelolaan arus kas, dan penyusunan laporan keuangan. Sementara itu, Wakil Direktur Operasional mengawasi aktivitas produksi dan distribusi makanan melalui Kepala Dapur sebagai penanggung jawab teknis kegiatan dapur. Kepala Dapur dibantu oleh Tim Produksi yang fokus pada pengolahan dan penyajian makanan, serta Tim Logistik yang bertugas dalam pengemasan dan pengantaran makanan kepada pelanggan. Setiap peran dalam struktur ini saling terintegrasi dan memiliki keterkaitan untuk memastikan proses bisnis berjalan efisien dan efektif. Struktur organisasi yang jelas ini dirancang untuk memperkuat koordinasi antar bagian serta mendukung pengambilan keputusan yang cepat dan tepat dalam pengelolaan UMKM “Nusantara Kuliner”

2.1.2. Scrum

Scrum adalah kerangka kerja pengembangan perangkat lunak yang berjalan dalam siklus pendek (*sprint*) untuk menghasilkan *increment* yang siap dievaluasi secara berkala melalui mekanisme inspeksi dan adaptasi [19]. Kerangka ini mengatur peran, artefak, dan seremonial agar kolaborasi tim terstruktur serta tujuan produk tercapai secara bertahap [18]. Visualisasi alur Scrum ditunjukkan pada Gambar 2.2.



Gambar 2.2 Metode Scrum (Sumber: scrum.org)

a. *Product Backlog*

Product Backlog adalah daftar prioritas kebutuhan tingkat produk yang selalu hidup (*living document*) dan dikelola untuk mengarahkan pencapaian *product goal* [18]. Backlog disempurnakan secara berkelanjutan melalui *refinement* agar item jelas, dapat diperkirakan, dan siap dipilih pada *sprint planning* [20].

b. *Sprint Planning*

Sprint Planning adalah pertemuan awal sprint untuk menetapkan sprint goal, memilih item dari *product backlog*, dan menyusun rencana kerja tingkat tinggi tim selama satu periode *sprint* [20]. Hasil kegiatan ini berupa *sprint backlog* yang menggambarkan cakupan pekerjaan dan pendekatan implementasi untuk mencapai *sprint goal* [18].

c. *Daily Scrum*

Daily Scrum adalah sinkronisasi harian berdurasi singkat di antara anggota tim pengembang untuk meninjau kemajuan terhadap sprint goal dan menyesuaikan rencana harian [21]. Pertemuan ini menjaga visibilitas alur kerja sekaligus mempercepat penanganan hambatan yang muncul selama sprint [20].

d. *Sprint Review*

Sprint Review adalah forum di akhir *sprint* untuk menginspeksi *increment* bersama pemangku kepentingan dan mengumpulkan umpan balik nilai bisnis terkini [22]. Hasil umpan balik digunakan untuk memperbarui

product backlog sehingga arah produk tetap selaras dengan kebutuhan pengguna dan organisasi [19].

e. *Sprint Retrospective*

Sprint Retrospective adalah sesi refleksi tim untuk mengidentifikasi peluang perbaikan proses, kolaborasi, dan praktik teknis sebelum sprint berikutnya [23]. Rencana perbaikan yang disepakati diimplementasikan pada siklus selanjutnya guna memperkuat kemampuan tim beradaptasi dan meningkatkan kualitas hasil kerja [20].

2.1.3. Golang

Golang (atau Go) adalah bahasa pemrograman dari sumber terbuka yang dikembangkan oleh Google yang dirancang untuk memberikan performa tinggi, efisiensi, dan produktivitas dalam pengembangan perangkat lunak [17]. Bahasa ini menawarkan kompilasi cepat, eksekusi yang ringan, dan sintaksis yang sederhana namun ekspresif, menjadikannya pilihan yang sangat sesuai untuk pengembangan sistem *backend* dan *RESTful API* [24].

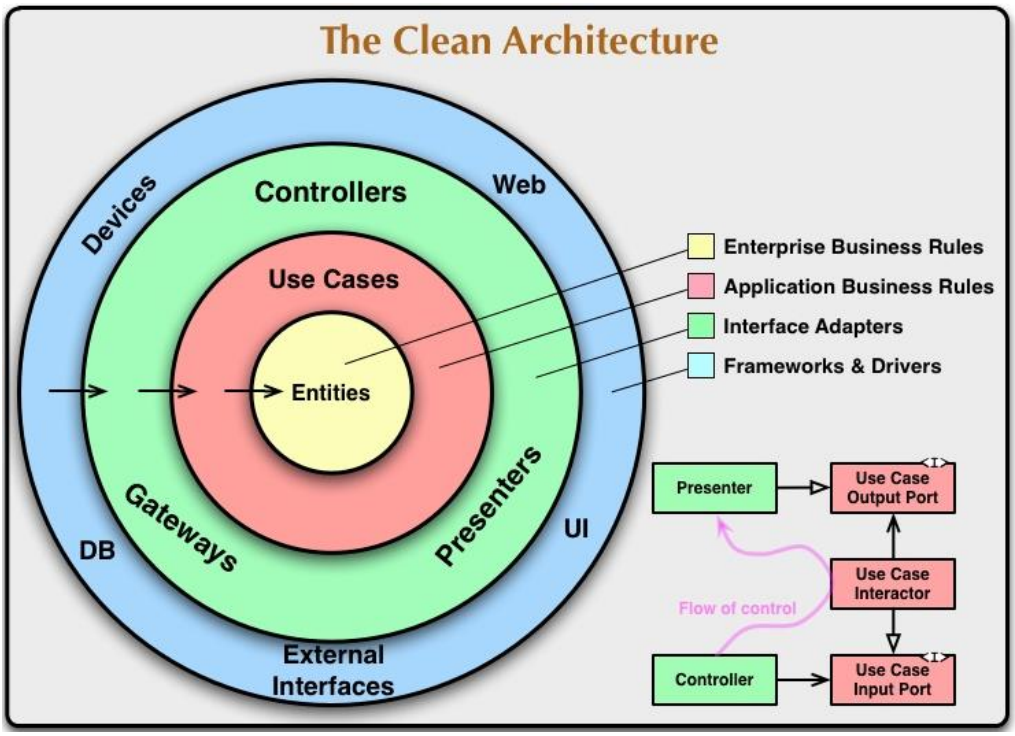
Salah satu fitur unggulan Go adalah dukungan *native* terhadap konkurensi melalui mekanisme *goroutines* dan *channels*, yang memungkinkan pengembang membangun aplikasi yang dapat menangani ribuan permintaan secara bersamaan dengan konsumsi sumber daya yang minimal [25]. Go juga mendukung penerapan *Clean Architecture*, suatu pendekatan arsitektural yang memisahkan kode ke dalam lapisan-lapisan yang independen dan memiliki tanggung jawab tunggal Implemen [26].

Dalam berbagai pengujian performa, Go telah menunjukkan keunggulan dibandingkan bahasa lainnya. Studi komparatif antara Go, PHP, dan JavaScript dalam konteks *backend* API mengonfirmasi bahwa Go unggul dalam hal *throughput* dan *latency* [27]. Pengujian pada ekosistem Go untuk koneksi konkuren menunjukkan capaian latensi dan *throughput* yang kompetitif [28]. Karakteristik-karakteristik ini membuat Go menjadi pilihan yang tepat untuk pengembangan sistem *backend* yang memerlukan skalabilitas dan maintainability yang baik.

2.1.4. Clean Architecture

Clean Architecture merupakan pendekatan arsitektur yang menekankan pemisahan tanggung jawab ke dalam beberapa lapisan sehingga aturan bisnis inti

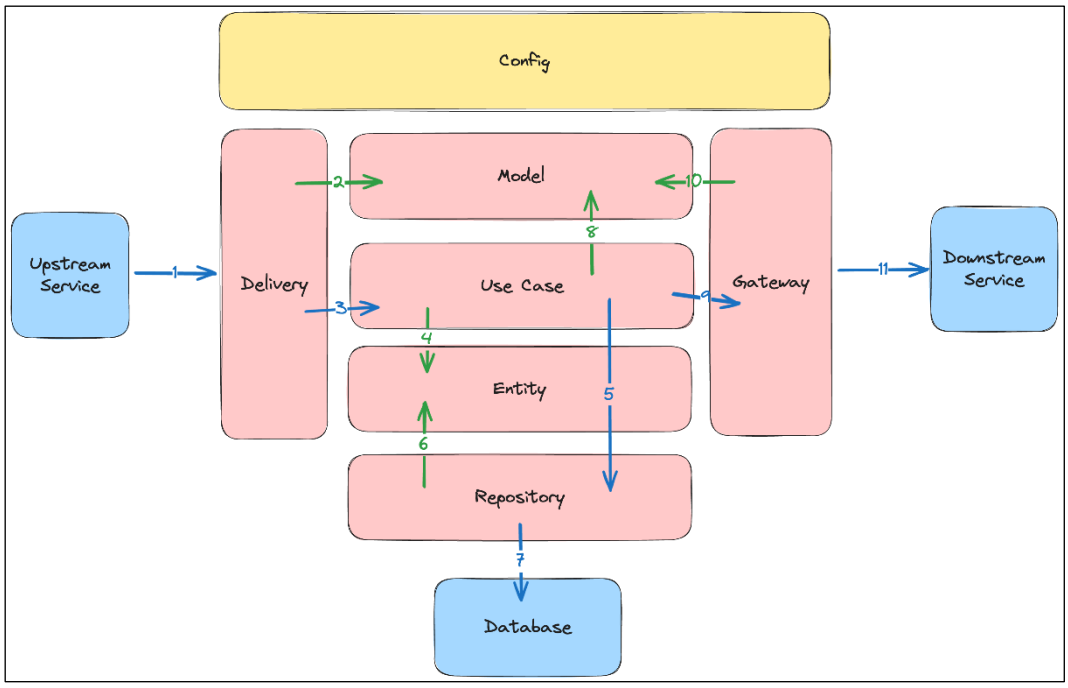
tetap terisolasi dari detail teknis. Pendekatan ini menekankan agar sistem tidak bergantung pada *framework* tertentu, mudah diuji tanpa antarmuka pengguna atau basis data, serta tetap stabil ketika terjadi perubahan pada antarmuka maupun teknologi penyimpanan [16].



Gambar 2.3 Konsep Clean Architecture (Sumber: R. C. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design, 2018)

Gambar 2.3 menampilkan konsep *Clean Architecture* dalam bentuk beberapa lingkaran konsentris yang terdiri dari Entities, Use Cases, Interface Adapters, serta Frameworks and Drivers. Entities memuat aturan bisnis paling umum, Use Cases mengatur aturan bisnis spesifik aplikasi dan alur data, Interface Adapters berperan mengonversi data antara format internal dan kebutuhan komponen eksternal, sedangkan Frameworks and Drivers memuat detail implementasi seperti web dan basis data. Aturan utama yang harus dijaga adalah dependency rule, yaitu ketergantungan kode harus selalu mengarah ke lapisan yang lebih dalam sehingga lapisan inti tidak bergantung pada detail lapisan luar dan perubahan pada lapisan luar tidak memaksa perubahan pada lapisan dalam [16].

Pada penelitian ini, konsep umum tersebut diadaptasi secara lebih spesifik ke dalam arsitektur *backend* Nusantara Kuliner sebagaimana ditunjukkan pada Gambar 2.4.



Gambar 2.4 Diagram Alur Clean Architecture (Sumber: https://github.com/khannedy/golang-clean-architecture)

Gambar 2.4 menggambarkan penerapan *Clean Architecture* pada *backend* Nusantara Kuliner. Komponen utama pada diagram ini meliputi Delivery sebagai antarmuka masuk seperti HTTP atau gRPC, *Use Case* sebagai pengatur alur aplikasi, *Entity* sebagai aturan bisnis inti, *Repository* sebagai port akses data, *Gateway* sebagai port ke layanan eksternal, *Database* sebagai driver penyimpanan, *Model* sebagai struktur transfer data, serta Config sebagai konfigurasi lintas komponen [17]. Arah panah menunjukkan arus data melintasi batas dengan ketergantungan yang konsisten menuju lapisan inti agar perubahan pada *framework* atau penyimpanan tidak memaksa perubahan pada domain [26]. Alur kerja sesuai diagram adalah sebagai berikut:

1. *External system* melakukan *request* (HTTP, gRPC, Messaging, dll).
2. *Delivery* membuat berbagai *Model* dari data *request*.
3. *Delivery* memanggil *Use Case*, dan mengeksekusinya menggunakan data *Model*.
4. *Use Case* membuat data *Entity* untuk *business logic*.
5. *Use Case* memanggil *Repository*, dan mengeksekusinya menggunakan data *Entity*.
6. *Repository* menggunakan data *Entity* untuk melakukan operasi *database*.

7. *Repository* melakukan operasi *database* ke *database*.
8. *Use Case* membuat berbagai *Model* untuk *Gateway* atau dari data *Entity*.
9. *Use Case* memanggil *Gateway*, dan mengeksekusinya menggunakan data *Model*.
10. *Gateway* menggunakan data *Model* untuk membangun *request* ke *external* system.
11. *Gateway* melakukan *request* ke *external system* (HTTP, gRPC, *Messaging*, dll).

Pemisahan *Use Case* dari *Delivery*, *Repository*, dan *Gateway* menjaga domain tetap murni serta memungkinkan pengujian unit tanpa ketergantungan pada *framework* atau basis data tertentu [17]. Praktik ini sangat relevan pada pengembangan *backend* transaksional karena batas antarlapisan yang jelas memungkinkan pergantian teknologi (misalnya basis data atau protokol komunikasi) tanpa perlu mengubah logika bisnis yang telah berjalan [26].

2.1.5. GoFiber

GoFiber adalah kerangka kerja HTTP ringan pada ekosistem Go yang memfasilitasi pembangunan layanan web dan REST API melalui mekanisme *routing*, *middleware*, dan penanganan permintaan yang efisien [26]. Penggunaan Go untuk merancang *web service* REST dipilih karena kesederhanaan kontrak HTTP serta kemudahan integrasi antar sistem pada aplikasi terdistribusi [26]. Studi perbandingan kinerja *backend* API menunjukkan implementasi berbasis Go mampu menghasilkan *throughput* tinggi dan waktu respons rendah pada skenario pengujian beban sehingga relevan untuk aplikasi transaksional [27].

Dalam penerapan arsitektur berlapis, komponen *framework* HTTP ditempatkan pada lapisan antarmuka agar logika bisnis dan akses data tetap terpisah sesuai prinsip *Clean Architecture* [17]. Riset terapan di Indonesia juga menunjukkan pemakaian GoFiber pada pengembangan API dalam proyek nyata sehingga dapat dijadikan acuan praktik penggunaan *framework* Go pada pengembangan layanan web [29].

2.1.6. PostgreSQL

PostgreSQL merupakan sistem manajemen basis data relasional (RDBMS) *open-source* yang powerful dan memiliki fitur lengkap untuk menangani berbagai

kebutuhan penyimpanan data. Sebagai *database* yang compliant dengan standar SQL, PostgreSQL menyediakan dukungan terhadap fitur-fitur *advanced* seperti *transaction*, *concurrency control*, *indexing*, dan *stored procedure* yang menjamin konsistensi dan keandalan data [30]. Keunggulan PostgreSQL dalam menangani data yang kompleks dan kemampuannya untuk skalabilitas vertikal membuatnya menjadi pilihan yang populer untuk aplikasi *backend* yang memerlukan performa tinggi [31]. Selain itu, PostgreSQL dikenal karena stabilitas dan kemampuannya dalam menangani beban kerja yang tinggi, sehingga banyak digunakan dalam lingkungan *production* berbagai aplikasi *enterprise*[32].

Dari segi arsitektur, PostgreSQL mengimplementasikan mekanisme yang robust dengan dukungan penuh terhadap *ACID properties* (*Atomicity*, *Consistency*, *Isolation*, *Durability*) yang menjamin keandalan transaksi data [33]. Sistem ini menggunakan mekanisme MVCC (*Multi-Version Concurrency Control*) yang memungkinkan akses data secara bersamaan oleh multiple users tanpa mengorbankan konsistensi data [34]. Fitur-fitur lanjutan PostgreSQL meliputi dukungan untuk tipe data JSONB, *full-text search*, spatial data melalui PostGIS, dan *extensibility* melalui fungsi kustom yang membuatnya cocok untuk berbagai macam aplikasi modern [25]. PostgreSQL juga menyediakan berbagai metode indexing seperti B-tree, Hash, GiST, dan GIN yang dapat dioptimalkan untuk mempercepat *query* pada data yang kompleks [35].

Dalam konteks pengembangan sistem informasi, implementasi PostgreSQL melibatkan perancangan basis data yang matang melalui pembuatan *entity* relationship diagram (ERD) dan physical data *model* untuk memetakan struktur data secara komprehensif [30]. Integrasi PostgreSQL dengan aplikasi *backend* Golang dapat dilakukan melalui driver *database* native atau ORM seperti GORM yang menyediakan abstraksi untuk operasi *database* [36]. Pada implementasi *Clean Architecture*, PostgreSQL berperan sebagai infrastruktur data yang diakses melalui *layer repository*, dimana logika bisnis pada *layer use case* tidak bergantung secara langsung pada implementasi *database* [26]. Konfigurasi *connection pooling* dan *query optimization* pada PostgreSQL mampu mendukung aplikasi dengan beban tinggi dan requirement performa yang ketat, menjadikannya solusi ideal untuk sistem yang memerlukan skalabilitas dan keandalan [25].

2.1.7. GORM

Object-Relational Mapping memetakan objek di memori ke tabel relasional sehingga operasi CRUD dapat dilakukan melalui abstraksi objek dan *query* builder [37]. Dalam praktik pengembangan layanan web, GORM adalah pustaka ORM di ekosistem Go yang menyediakan pemetaan struktur data ke tabel, migrasi skema, asosiasi, transaksi, serta dukungan strategi *eager* atau *lazy loading* sehingga akses data dapat terstandarisasi dan lebih mudah diuji [38]. Pemakaian ORM berelasi dengan pola *Data Mapper* dan *Active Record* yang memengaruhi jejak memori dan durasi eksekusi sehingga pemilihan dan penerapannya perlu mempertimbangkan kebutuhan domain serta beban transaksi aplikasi [37].

Dalam konteks arsitektur berlapis, GORM umumnya diletakkan di lapisan infrastruktur melalui antarmuka *repository* agar modul *use case* tidak bergantung pada detail basis data dan tipe *query* tertentu [17]. Pendekatan ini membantu pemeliharaan, isolasi pengujian, dan konsistensi kontrak data. Studi terapan di lingkungan Go menunjukkan kombinasi Go *framework* dengan GORM pada PostgreSQL menghasilkan kinerja penyimpanan yang kompetitif untuk skenario concurrent write sehingga relevan untuk sistem transaksi web yang memerlukan konsistensi dan skalabilitas menengah [38].

2.1.8. Redis

Redis adalah *in-memory data store* yang umum digunakan sebagai *cache key-value* untuk menurunkan latensi baca dan mengurangi beban basis data pada pola akses berulang di aplikasi web [39]. Evaluasi eksperimental menunjukkan bahwa Redis *caching* meningkatkan efisiensi akses dan *throughput* dibandingkan konfigurasi tanpa *cache* pada skenario beban tinggi [10]. Evaluasi pada aplikasi web menunjukkan peningkatan efisiensi akses ketika mekanisme *caching* diterapkan [40].

Penerapan Redis umumnya mengikuti pola *cache-aside*, yaitu aplikasi memeriksa *cache* terlebih dahulu, mengambil data dari basis data jika terjadi miss, lalu menyimpan kembali hasilnya ke *cache* dengan *time-to-live* yang diatur agar konsistensi tetap terjaga [39]. Kebijakan invalidasi perlu dirancang jelas agar perubahan pada sumber data segera tercermin pada *cache* sehingga perilaku sistem

tetap prediktif [39]. Dukungan pustaka klien memungkinkan operasi penyimpanan, pengambilan, dan pengaturan kedaluwarsa dijalankan secara konsisten [40].

Pada tataran operasional, pemilihan kebijakan *eviction* dan penentuan prioritas objek *cache* dapat merujuk pada pendekatan penilaian berbasis karakteristik akses untuk menjaga hit ratio pada memori terbatas [40]. Pemantauan metrik seperti hit ratio, latency, dan pemakaian memori diperlukan untuk menilai efektivitas *cache* serta menentukan penyesuaian TTL atau kebijakan *eviction* berikutnya [10].

2.1.9. *RESTful API*

API (*Application Programming Interface*) merupakan antarmuka yang memungkinkan komunikasi dan pertukaran data antara dua atau lebih sistem perangkat lunak. Secara sederhana, API berfungsi sebagai perantara yang menghubungkan berbagai aplikasi, memungkinkan mereka untuk berinteraksi tanpa perlu mengetahui detail implementasi internal masing-masing [41]. Dalam konteks pengembangan web, API memfasilitasi integrasi antara *frontend* dan *backend*, serta memungkinkan sistem yang berbeda untuk saling berkomunikasi melalui protokol standar.

RESTful API (Representational State Transfer *Application Programming Interface*) adalah implementasi spesifik dari API yang mengikuti prinsip-prinsip arsitektur REST yang diperkenalkan oleh Roy Fielding [11]. Karakteristik utama *RESTful API* meliputi sifatnya yang stateless, dimana setiap *request* harus mengandung semua informasi yang diperlukan untuk diproses oleh *server*, serta penggunaan metode HTTP standar untuk operasi data. Metode HTTP yang umum digunakan meliputi GET untuk mengambil data, POST untuk membuat data baru, PUT untuk memperbarui seluruh data, PATCH untuk memperbarui sebagian data, dan DELETE untuk menghapus data [24].

Setiap metode HTTP dalam *RESTful API* memiliki semantik dan kegunaan yang spesifik. GET digunakan untuk operasi baca dan tidak boleh mengubah *state server*, POST biasanya untuk membuat sumber daya baru, PUT untuk mengganti seluruh sumber daya yang ada, PATCH untuk modifikasi parsial, sedangkan DELETE untuk menghapus sumber daya [8]. *RESTful API* juga menggunakan Uniform *Resource* Identifier (URI) untuk mengidentifikasi sumber daya dan

biasanya mengembalikan data dalam format JSON atau XML, yang memungkinkan pertukaran data yang efisien dan mudah dipahami oleh berbagai *platform* [11].

2.1.10. *Payment Gateway*

Payment gateway adalah komponen perantara yang menghubungkan aplikasi merchant dengan penyedia pembayaran melalui API HTTP atau REST untuk otorisasi, *capture* atau *settlement*, notifikasi, dan pelaporan transaksi secara real time [42]. Alur umum mencakup pembuatan charge dari sisi aplikasi, penyelesaian pembayaran oleh pengguna sesuai kanal yang dipilih, dan pengiriman *webhook* dari penyedia untuk memperbarui status transaksi pada sistem bisnis seperti *pending*, *settlement*, *cancel*, atau *expire* [42].

Pada penelitian ini, *payment gateway* digunakan untuk memproses pembayaran transaksi produk dan layanan Katering Langganan pada *Website* Nusantara Kuliner, dengan *backend* yang membuat transaksi dan menerima notifikasi untuk memperbarui status pesanan atau langganan secara otomatis [43]. *Backend* menyediakan *endpoint* inisiasi pembayaran dan *endpoint* notifikasi, lalu memetakan status yang diterima menjadi *state* domain agar alur *checkout* dan konfirmasi layanan berjalan konsisten [44]. Pemilihan penyedia *gateway* merujuk pada studi komparatif yang memetakan kecocokan metode bayar dan kategori bisnis untuk UMKM serta *e-commerce* di Indonesia [42].

2.1.11. *Continuous Integration*/*Continuous Deployment* (CI/CD)

Continuous Integration/*Continuous Deployment* (CI/CD) adalah praktik pengembangan perangkat lunak yang mengotomatiskan proses integrasi kode, pengujian, dan *deployment* ke lingkungan *production*. *Continuous Integration* (CI) merujuk pada praktik dimana pengembang secara teratur menggabungkan kode perubahan mereka ke dalam *repository* pusat, di mana automated *build* dan test dijalankan untuk mendeteksi masalah integrasi secara dini [11]. Faktor adopsi praktik DevOps di tingkat organisasi turut memengaruhi keberhasilan penerapan CI/CD [45]. Sementara *Continuous Deployment* (CD) memperluas konsep ini dengan secara otomatis menerapkan setiap perubahan kode yang lolos pengujian ke lingkungan *production*, memungkinkan rilis yang lebih cepat dan andal.

Implementasi CI/CD dalam pengembangan sistem *backend* modern mencakup berbagai tahapan automated *testing* termasuk pengujian unit, pengujian integrasi, dan pengujian fungsional untuk memastikan kualitas kode [46]. Pipeline CI/CD yang efektif juga mengintegrasikan pemeriksaan keamanan dan *performance testing* untuk mengidentifikasi kerentanan dan masalah performa sebelum *deployment* ke *production* [47]. Dalam pengembangan, CI/CD memungkinkan *deployment* independen untuk setiap layanan, memfasilitasi skalabilitas dan maintenance sistem yang lebih baik [48]. Penerapan CI/CD yang komprehensif tidak hanya mempercepat waktu rilis tetapi juga meningkatkan stabilitas sistem dengan meminimalkan human error dalam proses *deployment* [11].

2.1.12. *Black Box Testing*

Black-box testing merupakan metode pengujian perangkat lunak yang berfokus pada perilaku fungsional sistem tanpa mempertimbangkan struktur internal atau implementasi kode. Pendekatan ini menguji sistem dari perspektif pengguna akhir dengan memeriksa keluaran yang dihasilkan terhadap masukan yang diberikan, tanpa pengetahuan tentang cara sistem bekerja secara internal [47]. Metode ini sangat efektif untuk memvalidasi fungsionalitas sistem secara keseluruhan dan memastikan bahwa kebutuhan pengguna telah terpenuhi dengan baik.

Dalam implementasinya, *black-box testing* telah banyak diaplikasikan pada berbagai jenis sistem. Pengujian terhadap *website* EPOS PT XYZ menggunakan *black-box testing* berhasil memverifikasi fungsionalitas sistem secara komprehensif [47]. Demikian pula pada sistem informasi parkir berbasis web, *black-box testing* digunakan untuk memvalidasi seluruh fungsionalitas antarmuka pengguna [46]. Aplikasi lain tercapai pada sistem informasi point of sales dimana *black-box testing* berperan penting dalam memastikan kelayakan sistem sebelum diimplementasikan [49].

Karakteristik utama *black-box testing* meliputi pengujian berdasarkan spesifikasi kebutuhan, independensi dari kode program, serta fokus pada antarmuka eksternal system [46]. Teknik-teknik yang umum digunakan dalam *black-box testing* antara lain *equivalence partitioning, boundary value analysis, decision table testing,* dan *state transition testing* [49]. Dalam konteks pengujian *website* atau

RESTful API, *black-box testing* dapat diterapkan untuk memverifikasi respons sistem terhadap berbagai skenario *input*, termasuk kasus uji normal, *boundary*, dan *invalid input*, sehingga memastikan kehandalan sistem dalam kondisi penggunaan yang beragam.

2.1.13. User Acceptance *Testing*

User Acceptance Testing (UAT) adalah pengujian penerimaan yang dilakukan untuk memastikan sistem sudah sesuai dengan kebutuhan pengguna sebelum dinyatakan layak digunakan. UAT menekankan validasi dari perspektif pengguna atau pihak yang mewakili pengguna, dengan berfokus pada kecocokan fungsi sistem terhadap kebutuhan operasional, sehingga hasil akhirnya menjadi dasar bahwa sistem dapat diterima dan digunakan pada lingkungan sebenarnya. UAT sering diposisikan setelah pengujian teknis seperti unit *testing* dan *black box testing* sebagai tahap validasi penerimaan [50].

Pada sistem *backend*, UAT tetap relevan karena pengguna atau pemangku kepentingan dapat mengevaluasi apakah layanan API sudah mendukung alur kerja yang dibutuhkan, misalnya proses autentikasi, pengelolaan data, dan keluaran informasi sesuai aturan bisnis. Dengan demikian, UAT pada *backend* dapat difokuskan pada skenario penggunaan dan kriteria penerimaan berbasis fungsi, yaitu apakah *endpoint* dan responsnya sudah memenuhi kebutuhan pengguna dan mendukung proses operasional yang ditargetkan [51].

2.1.14. k6

k6 merupakan alat *load testing* yang digunakan untuk mensimulasikan trafik pengguna secara lebih detail agar kinerja layanan dapat diukur secara objektif dan mendalam. Dalam pengujian performa web, indikator yang umum diamati mencakup *response time* dan *throughput*, serta dapat diperluas ke pemantauan CPU usage dan memory consumption untuk melihat kemampuan sistem saat menerima beban akses yang merepresentasikan kondisi nyata [12].

Secara konseptual, skenario pengujian dengan k6 dapat mencakup *load test* (kenaikan beban bertahap hingga stabil untuk mengamati respons sistem pada beban normal), *spike test* (lonjakan pengguna mendadak untuk menguji ketahanan sistem terhadap *traffic surge*), *soak test* (beban stabil jangka panjang untuk mendeteksi degradasi kinerja seperti memory leak), serta *performance test*

(peningkatan beban progresif untuk menilai skalabilitas dan kapasitas sistem). Hasil pengujian biasanya dianalisis menggunakan metrik seperti *response time*, *throughput*, *error rate*, dan *requests per second* untuk menggambarkan kecepatan respons, kapasitas pemrosesan, serta reliabilitas layanan pada berbagai kondisi beban [12].

2.1.15. Unified Modeling Language (UML)

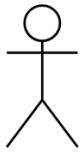
Unified Modeling Language (UML) adalah bahasa pemodelan visual standar yang digunakan untuk merepresentasikan, mendokumentasikan, dan merancang sistem perangkat lunak. UML menyediakan notasi grafis yang terstandarisasi untuk memvisualisasikan desain sistem, memungkinkan pengembang dan pemangku kepentingan untuk berkomunikasi secara efektif tentang struktur dan perilaku sistem [52]. Sebagai alat bantu dalam rekayasa perangkat lunak, UML memfasilitasi pemahaman yang lebih baik tentang kompleksitas sistem melalui berbagai diagram yang dapat merepresentasikan aspek-aspek berbeda dari pengembangan perangkat lunak [53].

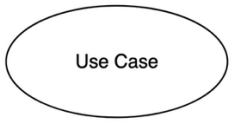
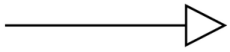
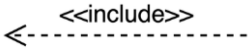
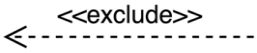
Alat bantu yang digunakan dalam perancangan berorientasi objek berbasis UML pada penelitian ini adalah sebagai berikut:

a. Use Case Diagram

Use case diagram digunakan untuk memodelkan interaksi antara aktor (pengguna sistem) dan sistem, serta menunjukkan *use case* (fungsionalitas) yang disediakan oleh sistem [52]. Diagram ini membantu dalam mengidentifikasi kebutuhan fungsional sistem dari perspektif pengguna. Notasi pada *use case diagram* dapat dilihat pada Tabel 2.1.

Tabel 2.1 Notasi Use Case Diagram

Simbol	Nama	Pengertian
	*Actor*	Entitas di luar sistem (manusia atau sistem eksternal) yang berinteraksi dengan sistem; memicu *use case* atau menerima hasilnya.

Simbol	Nama	Pengertian
	*Use Case*	Layanan/fungsi bernilai bisnis yang disediakan sistem; menggambarkan urutan interaksi untuk menghasilkan keluaran yang jelas.
	Association	Relasi partisipasi antara actor dan *use case* yang menandakan adanya komunikasi/keterlibatan.
	*Generalization*	Pewarisan/specialization pada actor atau *use case*; turunan mewarisi relasi dan perilaku induk.
	*Includes*	Hubungan ketika suatu *use case* selalu menyertakan perilaku *use case* lain sebagai langkah wajib (pemfaktoran ulang langkah umum).
	*Extends*	Hubungan penambahan perilaku opsional ke *use case* dasar pada kondisi tertentu melalui *extension point*.

b. Robustness Diagram

Robustness diagram digunakan untuk memodelkan interaksi antara objek-objek dalam sistem pada tingkat yang lebih detail, menghubungkan *use case* dengan desain objek. Diagram ini membantu dalam mengidentifikasi objek-objek yang terlibat dalam sebuah *use case* dan bagaimana mereka berinteraksi [52]. Notasi pada *robustness diagram* dapat dilihat pada Tabel 2.2.

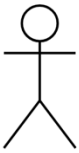
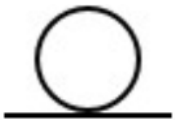
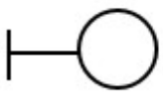
Tabel 2.2 Notasi Robustness Diagram

Simbol	Nama	Pengertian
	*Boundary*	Elemen antarmuka antara aktor dan sistem (UI, API handler, adapter) yang menerima *input* dan menyajikan *output*.
	*Entity*	Elemen yang merepresentasikan data/aturan bisnis yang persisten; biasanya berkorespondensi dengan tabel/rekaman pada basis data.
	*Controller*	Pengendali alur *use case*; mengoordinasikan interaksi boundary–*entity*, mengatur urutan proses, tanpa menyimpan state jangka panjang.

c. Sequence Diagram

Sequence diagram digunakan untuk memodelkan interaksi antar objek dalam sistem berdasarkan urutan waktu [53]. Diagram ini menunjukkan bagaimana objek-objek berkomunikasi melalui pertukaran pesan dalam skenario tertentu. Notasi pada *sequence diagram* dapat dilihat pada Tabel 2.3.

Tabel 2.3 Notasi Sequence Diagram

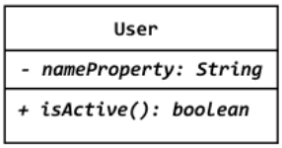
Simbol	Nama	Pengertian
	*Actor*	Pelaku eksternal yang memulai interaksi dan/atau menerima respons dari sistem.
	*Entity Class*	Kelas domain yang menyimpan/mengelola data persisten; menerima operasi baca/tulis.
	*BoundaryClass*	Kelas antarmuka (view/API) yang menjembatani aktor dengan sistem dan meneruskan permintaan.

Simbol	Nama	Pengertian
	*Control Class*	Kelas pengendali logika proses/*use case*; mengatur transaksi dan memanggil boundary/*entity*/layanan lain.
	*Life Line*	Garis vertikal yang menunjukkan eksistensi objek sepanjang waktu eksekusi; *activation bar* menandai periode eksekusi operasi.
	Message	Panah komunikasi antarlifeline yang merepresentasikan pemanggilan operasi/kiriman data (sinkron/asinkron) dan/atau *return*.

d. Class Diagram

Class diagram digunakan untuk memodelkan struktur statis sistem dengan menunjukkan kelas-kelas, atribut, operasi, dan hubungan antar kelas [53]. Diagram ini memberikan gambaran tentang arsitektur sistem dalam hal kelas-kelas yang membentuk sistem. Notasi pada *class diagram* dapat dilihat pada Tabel 2.4.

Tabel 2.4 Notasi Class Diagram

Simbol	Nama	Pengertian
	*Class*	Cetak biru struktur dan perilaku objek; berisi atribut dan operasi dengan visibilitas (+ public, − private, # protected).

Simbol	Nama	Pengertian
	*Association dan One Way Association*	Hubungan struktural antar kelas yang menunjukkan keterkaitan/navigasi; *one-way* memiliki arah akses dari sumber ke target; kardinalitas menggambarkan multiplisitas.
	*Composition*	Hubungan *whole–part* kuat (♦ hitam); siklus hidup bagian tergantung pada keseluruhan—jika *whole* dihapus, *part* ikut hilang.
	*Aggregation*	Hubungan *whole–part* lemah (♦ putih); bagian dapat berdiri sendiri dan dapat dibagikan ke *whole* lain.
	*Dependency*	Ketergantungan penggunaan sementara; perubahan pada pemasok dapat memengaruhi klien; ditunjukkan garis putus-putus berpanah.

2.1.16. Conceptual Data *Model* (CDM)

Conceptual Data *Model* (CDM) merupakan representasi tingkat tinggi dari struktur data yang berfokus pada konsep bisnis dan aturan tanpa mempertimbangkan implementasi teknis basis data. *Model* ini berfungsi sebagai fondasi untuk merancang basis data yang efektif dengan menangkap kebutuhan data dari perspektif bisnis dan menyajikannya dalam bentuk yang mudah dipahami oleh pemangku kepentingan non-teknis [54]. CDM menggambarkan entitas-entitas penting dalam domain bisnis, hubungan antar entitas, serta batasan-batasan bisnis yang berlaku, sehingga menjadi panduan untuk pengembangan logika dan physical data *model* selanjutnya [55].

CDM terdiri dari tiga komponen utama yaitu entitas, atribut, dan relationship. Entitas merupakan objek atau konsep dalam dunia nyata yang dapat dibedakan dan memiliki keberadaan independen seperti Pelanggan atau Produk dalam suatu sistem [55]. Atribut adalah karakteristik atau properti yang mendeskripsikan entitas misalnya atribut nama dan alamat untuk entitas Pelanggan. Relationship menggambarkan hubungan logis antara dua atau lebih entitas seperti hubungan antara entitas Pelanggan dan Produk yang dihubungkan melalui relationship Membeli [54].

Pemodelan konseptual ini sangat penting dalam fase awal pengembangan sistem karena membantu dalam memahami kebutuhan data secara menyeluruh sebelum masuk ke tahap implementasi teknis [55]. Dengan CDM, pengembang dapat memastikan bahwa struktur basis data yang akan dibangun benar-benar sesuai dengan kebutuhan bisnis dan dapat mengakomodasi semua persyaratan fungsional sistem [54]. Selain itu, CDM juga berperan dalam mempromosikan pendekatan data-driven dengan menyediakan kerangka kerja yang standar untuk mengelola dan berbagi data antar berbagai pemangku kepentingan [54].

2.1.17. Physical Data *Model* (PDM)

Physical Data *Model* (PDM) merupakan representasi implementasi fisik dari basis data yang mencakup detail teknis seperti tipe data, ukuran kolom, kunci, indeks, dan constraint [55]. *Model* ini berfokus pada aspek teknis implementasi basis data pada *Database* Management System (DBMS) tertentu seperti PostgreSQL atau MySQL, berbeda dengan CDM yang berfokus pada konsep bisnis

[54]. PDM menerjemahkan konsep dari CDM ke dalam struktur tabel, kolom, dan hubungan yang siap diimplementasikan dalam basis data fisik.

PDM mencakup komponen-komponen teknis seperti tabel, kolom, tipe data, primary key, foreign key, indeks, dan constraint [30]. Setiap entitas dalam CDM diimplementasikan sebagai tabel dalam PDM, sedangkan atribut menjadi kolom-kolom dalam tabel tersebut. Relationship dalam CDM diimplementasikan melalui *foreign key* yang menghubungkan tabel-tabel terkait [54]. PDM juga mempertimbangkan aspek optimasi performa melalui penentuan indeks yang tepat dan normalisasi tabel untuk menghindari redundansi data serta memastikan integritas data [55].

Pemodelan fisik ini sangat penting dalam fase implementasi sistem karena memberikan panduan yang jelas untuk pembuatan basis data fisik [54]. Dengan PDM, pengembang dapat memastikan bahwa basis data yang diimplementasikan memiliki struktur yang optimal, performa yang baik, dan memenuhi semua persyaratan teknis dari sistem. Implementasi PDM yang baik akan memfasilitasi proses transaksi data yang efisien dan andal, seperti yang terbukti dalam sistem transaksi perbankan dimana PDM berperan penting dalam mengelola alur transaksi yang kompleks [30].

2.2. Penelitian Terdahulu

Penelitian Terdahulu 1	
Judul	*Performance Evaluation of Backend Frameworks for REST API: A Comparative Study of Spring Boot, Flask, Express.js, Laravel FrankenPHP, and Gin* [13].
Tahun	2025
Tujuan Penelitian	Mengevaluasi kinerja berbagai *framework backend* untuk REST API dengan pengujian kinerja terstandar.
Latar Belakang	Pemilihan bahasa dan *framework* memengaruhi performa dan efisiensi layanan *web* dalam pengembangan sistem.
Metode Penelitian	Benchmark *throughput* dan waktu respons API menggunakan k6 pada beberapa *framework* populer termasuk Gin (Go).
Hasil Penelitian	Studi melaporkan perbandingan kinerja antarkerangka yang dapat dijadikan dasar pemilihan stack untuk layanan REST.
Relevansi	Memberi dasar empiris bahwa Go/Gin kompetitif untuk *RESTful API*, sehingga sejalan dengan pemilihan Golang + GoFiber.

Penelitian Terdahulu 2	
Judul	Implementing *Continuous Integration* and *Deployment* Strategy: Cloversy.id *RESTful API* [11]
Tahun	2024
Tujuan Penelitian	Menerapkan strategi CI/CD pada *RESTful API* untuk mengotomatisasi proses *build*, *testing*, dan *deployment* agar lebih konsisten dan terukur.
Latar Belakang	Proses rilis manual rentan terhadap human error dan membutuhkan waktu lama; diperlukan *pipeline* otomatis untuk meningkatkan efisiensi dan reliabilitas.
Metode Penelitian	Studi kasus dengan perancangan *pipeline* otomatis yang mencakup quality check, *build*, *testing*, packaging, hingga *deployment*.
Hasil Penelitian	Proses rilis menjadi lebih cepat, terstandarisasi, dan dapat direproduksi; beban kerja manual berkurang signifikan.
Relevansi	Menjadi acuan desain *pipeline* CI/CD *backend* Nusantara Kuliner yang diselaraskan dengan ritme sprint Scrum untuk memastikan setiap increment dapat di-deploy dengan aman.
Penelitian Terdahulu 3	
Judul	Three-Tier Application Development Using Scrum Method on Employee Attendance Application at Glints Academy [56]
Tahun	2022
Tujuan Penelitian	Menerapkan *framework* Scrum pada pengembangan aplikasi presensi karyawan berbasis three-tier architecture untuk menjamin ketercapaian *product backlog* dan visibilitas progres tim.
Latar Belakang	Diperlukan proses pengembangan yang terstruktur dan transparan untuk memastikan koordinasi tim yang efektif.
Metode Penelitian	Scrum
Hasil Penelitian	Fungsi inti aplikasi tercapai sesuai *product backlog*; ritme kerja tim dan transparansi progres terjaga dengan baik
Relevansi	Menjadi rujukan operasionalisasi Scrum untuk perencanaan sprint, *sprint review*, dan retrospective pada pengembangan *backend* Nusantara Kuliner.
Penelitian Terdahulu 4	
Judul	Implementasi Golang *Clean Architecture* pada Perancangan *Backend* Point of Sales *Website* [26].
Tahun	2024
Tujuan Penelitian	Penelitian ini bertujuan mengimplementasikan konsep *Clean Architecture* menggunakan bahasa pemrograman Golang untuk merancang *backend* pada *website* Point of Sales (POS).

Latar Belakang	Latar belakangnya adalah kebutuhan perusahaan penjualan akan pengelolaan penjualan yang lebih efisien, pemantauan inventaris yang akurat, serta analisis data yang lebih mendalam untuk mendukung pengambilan keputusan bisnis. *Website* POS dipaparkan dapat mengumpulkan data penjualan untuk analisis bisnis yang lebih baik.
Metode Penelitian	Agile Scrum
Hasil Penelitian	Hasil penelitian menyatakan bahwa penerapan *Clean Architecture* pada *backend* POS menghasilkan struktur yang lebih jelas dan memudahkan pengelolaan kode serta pengembangan fitur lanjutan.
Relevansi	Paper ini relevan dengan penelitian yang akan dikerjakan karena sama-sama berfokus pada perancangan dan implementasi *backend website* menggunakan Golang dengan pendekatan *Clean Architecture*. Struktur *layer* seperti *repository*, usecase, dan handler menunjukkan pemisahan tanggung jawab yang jelas sehingga memudahkan pemeliharaan, pengujian, dan pengembangan fitur secara bertahap. Selain itu, penggunaan Agile Scrum selaras dengan pengembangan berbasis sprint yang dapat digunakan untuk mengelola backlog, iterasi implementasi, serta evaluasi increment sistem *backend* pada setiap sprint.
Penelitian Terdahulu 5	
Judul	Integration of Design Thinking and Scrum in Development of Retail *Marketplace Website* [57]
Tahun	2022
Tujuan Penelitian	Mengintegrasikan pendekatan Design Thinking dan *framework* Scrum untuk membangun *website marketplace* ritel yang berorientasi pada kebutuhan pengguna
Latar Belakang	Diperlukan mekanisme untuk menghubungkan umpan balik pengguna secara sistematis dengan iterasi pengembangan yang terstruktur
Metode Penelitian	Design Thinking dan Scrum
Hasil Penelitian	Fitur inti *marketplace* berhasil direalisasikan dengan mekanisme umpan balik pengguna terintegrasi di setiap sprint
Relevansi	Menjadi rujukan untuk pendekatan user-driven backlog dalam konteks UMKM kuliner; menunjukkan keselarasan antara kebutuhan pengguna dan implementasi Scrum pada penelitian ini