BAB II
TINJAUAN PUSTAKA

Bab ini mengulas landasan teoretis dan studi-studi relevan yang menjadi fondasi bagi penelitian ini. Bagian awal akan menyajikan **studi literatur** yang membahas penelitian-penelitian sebelumnya dalam bidang protokol komunikasi, pola desain perangkat lunak, dan integrasi kecerdasan buatan di Unity. Selanjutnya, akan dibahas secara mendalam **kerangka teori** yang mencakup konsep-konsep kunci seperti protokol WebSocket, *Reactive Pattern*, arsitektur berbasis kejadian, dan teknologi LLM yang digunakan. Bab ini ditutup dengan penjelasan mengenai **konteks penelitian**, **teknik pengumpulan data**, serta strategi **verifikasi dan validasi** yang akan diterapkan.

2.1 Studi Literatur

Penelitian ini dibangun di atas fondasi beberapa area studi yang relevan, mencakup protokol komunikasi jaringan, pola desain perangkat lunak, integrasi kecerdasan buatan dalam aplikasi interaktif, dan platform pengembangan Unity. Pemahaman mendalam mengenai protokol WebSocket, yang mekanismenya telah distandarisasi oleh W3C (2021)[20] dan dijelaskan melalui dokumentasi API modern seperti MDN (2024)[5], menjadi krusial. WebSocket menyediakan saluran komunikasi *full-duplex* melalui satu koneksi TCP, yang berkontribusi pada pengurangan latensi dan overhead dibandingkan HTTP untuk aplikasi real-time. Sejumlah penelitian telah menginvestigasi karakteristik dan kesesuaian WebSocket untuk aplikasi interaktif. Studi oleh Łasocha & Badurowicz (2021)[7] dan Alviando et al. (2023)[8] menyoroti keunggulan WebSocket dalam skenario yang membutuhkan responsivitas tinggi dan delay rendah. Dalam studi komparatif terbaru, Tsaqief dan Sutopo (2025) menyoroti karakteristik performa protokol komunikasi modern, di mana WebSocket menunjukkan keunggulan throughput yang lebih tinggi dibandingkan MQTT dalam kondisi muatan data (payload) yang

bervariasi, yang krusial untuk visualisasi dinamis[24]. Penelitian lain seperti yang dilakukan oleh Biradar et al. (2024)[11] dan Sharma & Sahoo (2022)[12] juga menegaskan peran transformatif WebSocket dalam komunikasi real-time modern. Aspek keamanan WebSocket juga menjadi perhatian penting, sebagaimana diulas oleh Wang et al. (2020)[6]. Validitas penggunaan WebSocket untuk sistem yang menuntut responsivitas tinggi juga didukung oleh penelitian Hlayel et al. (2025), yang menemukan bahwa protokol WebSocket mampu mempertahankan latensi rata-rata yang kompetitif dan efisiensi sumber daya yang lebih baik dibandingkan protokol industri standar seperti MQTT dan OPC UA dalam arsitektur berbasis *cloud*[25].

Pola desain arsitektural, khususnya yang berorientasi pada *event* dan konkurensi, menjadi penting dalam sistem terdistribusi. Konsep *event-driven architecture* (EDA), sebagaimana dibahas oleh Richards & Ford (2020)[9] dan Harrer (2021)[10], menawarkan fleksibilitas dan skalabilitas. Dalam konteks penanganan I/O jaringan secara efisien, prinsip-prinsip desain reaktif dan pola untuk sistem asinkron, yang memungkinkan penanganan *event* I/O secara konkuren dan non-blocking, menjadi sangat relevan. *Reactive Pattern*, sebagai manifestasi dari prinsip-prinsip ini, memungkinkan aplikasi untuk secara reaktif menangani berbagai sumber *input* tanpa *multi-threading* yang kompleks untuk setiap koneksi, dengan memanfaatkan mekanisme seperti *event demultiplexer* dan *dispatcher* untuk mengirimkan *event* ke *handler* yang sesuai. Diskusi mengenai komunikasi asinkron dan penggunaan antrian pesan dalam arsitektur *microservices* terdistribusi oleh Liu et al. (2023)[13] juga memberikan perspektif terkait pentingnya manajemen alur data asinkron.

Integrasi Large Language Models (LLM) ke dalam aplikasi interaktif seperti game dan simulasi merupakan area riset yang berkembang pesat. Paduraru et al. (2024)[2] mengeksplorasi penggunaan LLM untuk otomatisasi pembuatan *unit test* dalam pengembangan game Unity, menyoroti adaptasi model seperti Code Llama untuk skenario spesifik game. Karya ini relevan karena menunjukkan potensi dan tantangan penggunaan LLM dalam ekosistem Unity. Sementara itu,

Tang et al. (2024)[14] dalam "Scaling On-Device GPU Inference for Large Generative Models" memperkenalkan ML Drift, sebuah *framework* yang dioptimalkan untuk menjalankan model generatif besar pada GPU perangkat, termasuk untuk aplikasi yang dibangun dengan Unity, yang menekankan pentingnya efisiensi inferensi untuk pengalaman pengguna yang baik. Kemajuan yang ditunjukkan oleh model seperti GPT-4[16] menjadi dasar pemahaman potensi LLM dalam berbagai tugas.

Dalam konteks spesifik Unity, pentingnya pola asinkron untuk menjaga responsivitas *main thread* telah lama diakui. Dokumentasi resmi Unity menyediakan panduan ekstensif mengenai pemrograman asinkron, termasuk penggunaan *Coroutines*, *async/await*, dan *Job System*. Penggunaan Ollama sebagai *framework* untuk menjalankan dan melayani LLM secara lokal menyederhanakan proses *deployment* dan akses ke model seperti sampri-custom:latest (berbasis Llama 3.2). Dokumentasi Ollama menjelaskan API yang tersedia, termasuk kemampuan untuk *streaming respons*. Chakraborty & Aithal (2023)[17] juga mendemonstrasikan penggunaan C# dan WebSocket dalam simulasi *smart home* di CoppeliaSim, yang menunjukkan relevansi kombinasi teknologi ini untuk interaksi *real-time*. Penelitian oleh Verma (2021)[18] mengenai integrasi aplikasi robotik menggunakan *Internet of Robotics Things* (IoRT) juga menyinggung pentingnya komunikasi efisien dan latensi rendah dalam sistem interaktif.

Meskipun terdapat banyak literatur mengenai masing-masing teknologi ini secara terpisah, studi yang secara spesifik menggabungkan integrasi LLM dengan Unity menggunakan protokol WebSocket dan secara eksplisit menerapkan serta menganalisis *Reactive Pattern* untuk manajemen komunikasi asinkron masih relatif terbatas. Oleh karena itu, penelitian ini diharapkan dapat memberikan kontribusi baru dengan menyajikan desain arsitektur, implementasi konkret, dan analisis efektivitas pola tersebut pada area persimpangan teknologi ini.

2.2 Kerangka Teori

Kerangka teori penelitian ini didasarkan pada pemahaman mendalam mengenai beberapa konsep fundamental yang saling terkait.

2.2.1 Protokol HTTP dan WebSocket

Protokol HTTP (*Hypertext Transfer Protocol*) merupakan protokol dasar yang telah lama digunakan untuk komunikasi data di *World Wide Web*. Versi HTTP awal memiliki keterbatasan dalam hal efisiensi untuk komunikasi *real-time* karena sifat request-response dan *overhead* koneksi. Perkembangan menuju HTTP/2 dan HTTP/3 telah membawa perbaikan signifikan, seperti *multiplexing*, kompresi header, dan penggunaan QUIC (pada HTTP/3) untuk mengurangi latensi dan meningkatkan efisiensi[3, 4]. Meskipun demikian, untuk komunikasi dua arah yang persisten dan berlatensi sangat rendah, model *request-response* fundamental HTTP masih memiliki keterbatasan dibandingkan dengan protokol yang dirancang khusus untuk itu[7, 12]. Kemampuan *streaming* HTTP/1.1, yang didukung Ollama, memungkinkan pengiriman respons LLM secara bertahap, yang penting untuk persepsi responsivitas, namun tetap beroperasi dalam paradigma *client-initiated request*.

Sebagai alternatif yang lebih sesuai untuk komunikasi *real-time* dan dua arah yang persisten, protokol WebSocket menyediakan saluran komunikasi *full-duplex* secara simultan antara klien dan *server* melalui satu koneksi TCP, setelah proses handshake awal melalui HTTP[5, 20]. Karakteristik utama WebSocket yang mendukung aplikasi interaktif meliputi:

a. **Koneksi Persisten:** Mengurangi latensi yang terkait dengan pembuatan koneksi berulang.

b. **Komunikasi Dua Arah Penuh (*Full-duplex*):** *Server* dapat mengirim data ke klien kapan saja tanpa menunggu permintaan baru

dari klien, sangat penting untuk notifikasi *real-time* dan *streaming* data berkelanjutan dari LLM.

c. ***Overhead* Pesan Rendah:** Setelah koneksi terjalin, *header* pesan WebSocket jauh lebih kecil dibandingkan HTTP/1.1, membuatnya efisien untuk pertukaran pesan kecil yang sering terjadi dalam percakapan *chatbot*[7, 11]. Efisiensi sumber daya WebSocket ini dikonfirmasi oleh studi Soewito et al. (2019), yang membuktikan bahwa metode WebSocket mengonsumsi bandwidth dan memori yang jauh lebih rendah dibandingkan teknik *polling* konvensional dalam skenario pemantauan *real-time*[21].

d. **Efisiensi untuk *Streaming*:** Sangat cocok untuk mengirimkan respons LLM yang dihasilkan secara bertahap. Karakteristik ini, ditambah dengan dukungan keamanan melalui WebSocket Secure (WSS)[6], menjadikannya pilihan yang kuat secara arsitektural untuk aplikasi interaktif seperti chatbot yang menuntut respons cepat dan berkelanjutan, yang akan dikelola oleh *Reactive Pattern* dalam penelitian ini.

2.2.2 *Reactive Pattern* dan Arsitektur Berbasis Kejadian

Istilah "*Reactive Pattern*" dalam judul penelitian ini merujuk pada penerapan prinsip-prinsip desain perangkat lunak yang memungkinkan sistem untuk menjadi responsif, tangguh, elastis, dan berorientasi pada pesan (*message-driven*), khususnya dalam menghadapi event asinkron. Dalam pengembangan aplikasi jaringan modern, terutama yang melibatkan interaksi pengguna secara *real-time* dan komunikasi dengan layanan eksternal, arsitektur berbasis kejadian (EDA) menjadi sangat penting untuk mencapai skalabilitas dan responsivitas[9, 10]. Prinsip dasar EDA adalah sistem bereaksi terhadap kejadian (event) yang terjadi. Pola *seperti publish-subscribe, event streaming*, dan penggunaan antrian pesan (*message*

queues) adalah komponen umum dalam EDA[13].

Dalam konteks penanganan I/O jaringan secara non-blocking di Unity, penerapan *Reactive Pattern* berarti merancang sistem klien agar dapat secara efisien menangani event yang datang dari koneksi WebSocket (misalnya, pesan baru dari *server* LLM) tanpa memblokir *main thread*. Ini melibatkan mekanisme untuk mendelegasikan operasi I/O ke *thread* latar belakang dan kemudian memproses hasilnya di *main thread* melalui antrian *event* dan *callbacks*. Mekanisme DispatchMessageQueue() dari pustaka NativeWebSocket, yang dipanggil dalam Update() loop Unity, merupakan implementasi praktis dari prinsip ini. Ia bertindak sebagai event loop sederhana yang mengambil event dari antrian dan mengeksekusi handler yang sesuai di *main thread*. Pendekatan ini memastikan bahwa aplikasi tetap responsif terhadap input pengguna dan pembaruan game lainnya sambil tetap menangani komunikasi jaringan secara efisien. Keberhasilan pola ini dalam menjaga responsivitas dan mengelola kompleksitas asinkron menjadi fokus analisis dalam penelitian ini.

2.2.3 Integrasi Unity C# dan Komunikasi Asinkron

Integrasi komunikasi jaringan ke dalam aplikasi yang dikembangkan menggunakan Unity Engine memerlukan perhatian khusus terhadap sifat asinkron dari operasi jaringan tersebut. Hal ini disebabkan oleh arsitektur fundamental Unity yang sebagian besar operasinya dijalankan pada satu *main thread* tunggal. Setiap operasi yang berjalan lama atau bersifat *blocking* jika dieksekusi secara langsung pada *main thread* ini akan menyebabkan aplikasi berhenti merespons. Oleh karena itu, menjadi sebuah keharusan untuk menangani semua komunikasi jaringan secara asinkron. Unity dan bahasa pemrograman C# menyediakan beberapa mekanisme dan fitur untuk memfasilitasi pemrograman asinkron, seperti

Coroutines dan konstruksi *async/await* yang dibangun di atas .NET Task Parallel Library (TPL). Pustaka pihak ketiga NativeWebSocket digunakan dalam proyek ini untuk menyediakan fungsionalitas WebSocket, yang biasanya menangani operasi jaringan di *thread* terpisah dan menyediakan mekanisme *callback* atau antrian pesan untuk berinteraksi dengan *main thread* Unity, sejalan dengan implementasi *Reactive Pattern* yang diusulkan.

2.2.4 Llama 3.2 dan Ollama

Model sampri-custom:latest, yang merupakan hasil *fine-tuning* berbasis Llama 3.2[1] dan digunakan dalam penelitian ini, merupakan salah satu contoh kemajuan terkini dalam bidang kecerdasan buatan. Alasan utama pemilihan model berbasis Llama 3.2 dan *framework* Ollama untuk penelitian ini adalah kebutuhan agar virtual assistant dapat dijalankan secara lokal dan offline, serta tetap ringan untuk menjaga performa aplikasi Unity. Ollama adalah sebuah tool atau platform yang dirancang untuk menyederhanakan proses pengunduhan, konfigurasi, dan eksekusi berbagai LLM populer secara lokal. Ollama mengekspos fungsionalitas LLM yang dijalankannya melalui sebuah API HTTP/1.1, termasuk kemampuan untuk melakukan generasi teks secara *streaming*. Fitur *streaming* ini sangat krusial untuk mengurangi persepsi latensi oleh pengguna akhir. Dalam arsitektur sistem yang diusulkan dalam penelitian ini, Ollama bertindak sebagai *server backend* yang bertanggung jawab untuk menjalankan model sampri-custom:latest dan melayani permintaan inferensi. Kemampuan LLM secara umum, seperti yang ditunjukkan oleh model seperti GPT-4[16], terus berkembang dan membuka potensi aplikasi yang luas.

2.3 Konteks Penelitian

Penelitian ini akan diimplementasikan dan diuji dengan fokus pada analisis implementasi *Reactive Pattern* dan validasi fungsional sistem. Infrastruktur komputasi yang digunakan akan mendukung jalannya model sampri-custom:latest (berbasis Llama 3.2) secara lokal melalui Ollama. Versi perangkat lunak yang digunakan meliputi Unity 6000.0.45f1 dan Python 3.10 untuk backend proxy. Kondisi jaringan untuk pengujian akan menggunakan modul Wifi yang sama.

2.4 Teknik Pengumpulan Data

Pengumpulan data dalam penelitian ini berfokus pada pengukuran kuantitatif terhadap kinerja arsitektur *Reactive Pattern*, evaluasi metrik jaringan, serta validasi fungsional sistem secara keseluruhan.

a. **Pengukuran Metrik Kinerja Grafis (*Frametime* dan FPS):** Pengumpulan data kinerja dilakukan menggunakan skrip *profiler* khusus (FrametimeLogger) yang diintegrasikan ke dalam klien Unity. Skrip ini bertugas merekam waktu komputasi render per *frame* (*frametime* dalam milidetik) dan laju bingkai (FPS) secara *real-time* saat sistem menerima aliran data. Data ini akan membandingkan skenario sistem reaktif dengan skenario *baseline* non-reaktif (simulasi *blocking I/O*).

b. **Pengukuran Latensi dan *Overhead* Komunikasi:** Data *overhead* jaringan dikumpulkan melalui metode pencatatan waktu (*timestamping*) presisi tinggi atau metode *Apple-to-Apple*. Waktu pengiriman data terakhir dari sisi *server* akan dicatat dan dikomparasikan dengan waktu penyelesaian pemrosesan di sisi klien, guna membandingkan efisiensi latensi antara protokol WebSocket dan HTTP/1.1.

c. **Analisis Struktur Kode:** Struktur kode klien Unity (seperti modul OllamaUnifiedClient.cs) dianalisis untuk mengevaluasi implementasi arsitektur reaktif, khususnya pada mekanisme DispatchMessageQueue dalam memindahkan beban eksekusi dari *background thread* ke *main thread* secara aman.

d. **Validasi Fungsional:** Mengumpulkan log sistem dan dokumentasi interaksi untuk memastikan sistem terintegrasi mampu mengirim *prompt*, menerima respons *streaming* secara utuh dari LLM, dan mengeksekusi animasi karakter serta modul *Text-to-Speech* secara sinkron.

2.5 Verifikasi dan Validasi

Proses verifikasi dan validasi dilakukan secara terukur dan bertahap untuk menjamin validitas hasil penelitian.

a. **Verifikasi Komponen:** Pengujian unit pada modul jaringan klien Unity untuk memastikan koneksi, transmisi pesan asinkron, dan fungsi *dispatcher* berjalan tanpa *error* logika. Verifikasi juga mencakup pengujian stabilitas *server proxy* Python dan proses *parsing* data JSON.

b. **Validasi Kinerja *Reactive Pattern*:** Implementasi arsitektur reaktif divalidasi secara empiris menggunakan ambang batas kritis kinerja *game engine* (Gregory, 2018). Sistem dinyatakan valid dan responsif apabila garis metrik *frametime* mampu dipertahankan secara konsisten di bawah 16,6 milidetik selama proses I/O jaringan berlangsung, mencegah terjadinya penahanan utas utama (*main thread*).

c. **Validasi Fungsional Keseluruhan:** Memastikan fungsionalitas sistem *end-to-end* berjalan sesuai tujuan, di mana *virtual assistant* mampu memproses input pengguna dan menghasilkan respons *audio-visual* yang koheren tanpa mengorbankan stabilitas performa aplikasi.