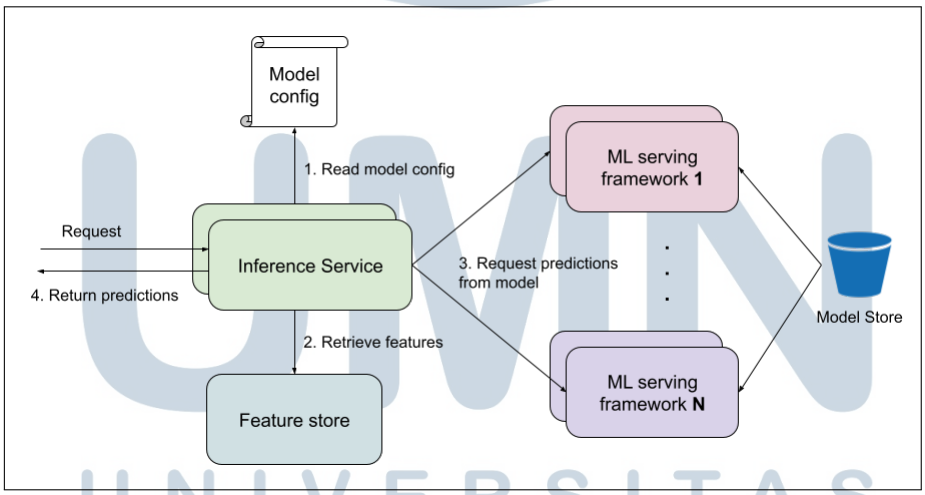
BAB 2
LANDASAN TEORI

2.1 Backend Inference pada Sistem Artificial Intelligence

Inference serving dalam sistem AI produksi menghadapi tantangan fundamental berupa *bottleneck* komputasi yang disebabkan oleh sifat sekuensial pemrosesan model dan overhead komunikasi data. Pada arsitektur tradisional dengan model *request-response* sinkron, setiap *request* harus menunggu penyelesaian seluruh *pipeline inference* sebelum *response* dikirim, membatasi *throughput* sistem dan menyebabkan peningkatan latensi eksponensial pada beban tinggi [8]. Sistem *high-load inference* memerlukan pendekatan arsitektural yang berbeda, meliputi *batching* dinamis, *caching* hasil prediksi, dan distribusi beban antar *replica* model [3]. Namun demikian, implementasi strategi optimasi ini dalam lingkungan produksi menuntut pertimbangan matang terhadap *trade-off* antara latensi, *throughput*, dan kompleksitas arsitektural, khususnya dalam konteks integrasi dengan ekosistem aplikasi yang lebih besar [2].



Gambar 2.1. Arsitektur *Inference Serving* untuk Sistem AI Produksi

Gambar 2.1 mengilustrasikan arsitektur tipikal *inference serving* yang terdiri dari beberapa komponen kunci yaitu *load balancer* untuk distribusi *request*, *model server* yang menjalankan proses inferensi, dan mekanisme *batching* untuk mengoptimalkan utilisasi GPU [21]. Arsitektur ini menunjukkan bagaimana *request* dari klien diproses melalui *pipeline* yang teroptimasi, dengan pemisahan jelas antara komponen penerima *request* dan komponen eksekusi model. Pemahaman

terhadap arsitektur ini menjadi fondasi untuk mengidentifikasi titik-titik *bottleneck* dan merancang strategi optimasi yang efektif, khususnya dalam konteks pemisahan alur sinkron dan asinkron yang menjadi fokus penelitian ini [3].

Pemisahan *concern* antara penerimaan *request*, eksekusi *inference*, dan pengiriman *response* melalui arsitektur asinkron terbukti efektif dalam meningkatkan skalabilitas sistem, dengan studi menunjukkan peningkatan *throughput* hingga 370% dan pengurangan waktu respons API hingga 70–85% pada kasus pemrosesan citra [7]. Meskipun penelitian sebelumnya telah mendemonstrasikan efektivitas teknik-teknik individual seperti *asynchronous processing* atau *caching*, belum terdapat studi yang mengeksplorasi kombinasi kedua pendekatan dalam satu eksperimen terkontrol untuk *inference serving* AI, khususnya pada arsitektur *hybrid microservices* yang menggabungkan API *Gateway* berbasis Node.js dengan layanan *inference* Python/FastAPI.

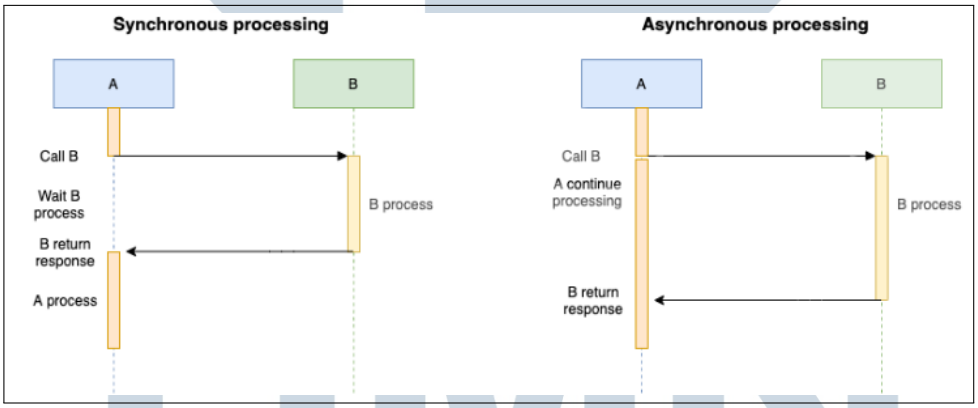
2.2 Arsitektur Microservices dalam Sistem AI

Arsitektur *microservices* memungkinkan isolasi komponen *inference* yang intensif komputasi dari komponen aplikasi lainnya, memberikan fleksibilitas dalam alokasi sumber daya dan pemilihan teknologi optimal untuk setiap komponen [4]. Pemisahan layanan *API Gateway* (Node.js) dan layanan *inference* (Python/FastAPI) memanfaatkan kekuatan masing-masing teknologi: Node.js unggul dalam menangani I/O asinkron dan konkurensi tinggi melalui *event loop*, sedangkan Python dengan ekosistem *machine learning*-nya optimal untuk eksekusi model [22]. Komunikasi antar layanan dapat dilakukan secara sinkron melalui HTTP/gRPC atau asinkron melalui *message broker*, dengan orkestrasi yang memerlukan mekanisme koordinasi robust untuk memastikan konsistensi sistem, termasuk manajemen siklus hidup model, distribusi beban, dan penanganan kegagalan melalui *retry* dan *circuit breaker* [22].

Meskipun arsitektur *microservices* telah banyak diterapkan pada sistem AI, mayoritas implementasi berfokus pada *environment* homogen atau platform *serverless* [23]. Belum terdapat studi yang secara spesifik mengkaji optimasi *backend inference* pada arsitektur *hybrid* yang menggabungkan Node.js API *Gateway* dengan layanan *inference* FastAPI, khususnya dalam konteks penerapan *asynchronous processing* dan *caching* terdistribusi untuk mengatasi *bottleneck* komputasi yang menjadi karakteristik utama *AI inference workload*.

2.3 Asynchronous Processing pada Backend Service

Model *non-blocking* (asinkron) memungkinkan *thread* tunggal menangani banyak *request* secara konkuren dengan mendelegasikan tugas ke *background* dan melanjutkan pemrosesan *request* lain, berbeda dengan model *blocking* yang menyebabkan *thread idle* selama menunggu operasi lambat. Pendekatan *reactive programming* (*event-loop*) dengan *distributed caching* terbukti meningkatkan *throughput* sebesar 75% dan mengurangi penggunaan memori hingga 68% dibandingkan model *asynchronous thread-per-request*, menegaskan superioritas arsitektur *reactive* untuk skenario konkurensi tinggi pada operasi I/O intensif [9]. *Message queue* seperti RabbitMQ atau Kafka bertindak sebagai *buffer* antara *producer* dan *consumer*, memungkinkan dekopel temporal dan spasial antar komponen, dengan benchmarking menunjukkan RocketMQ lebih sesuai untuk skenario yang membutuhkan latensi rendah, sementara Kafka menawarkan *throughput* tertinggi untuk pemrosesan volume pesan yang masif [11].



Gambar 2.2. Perbandingan Model *Synchronous* dan *Asynchronous Processing*

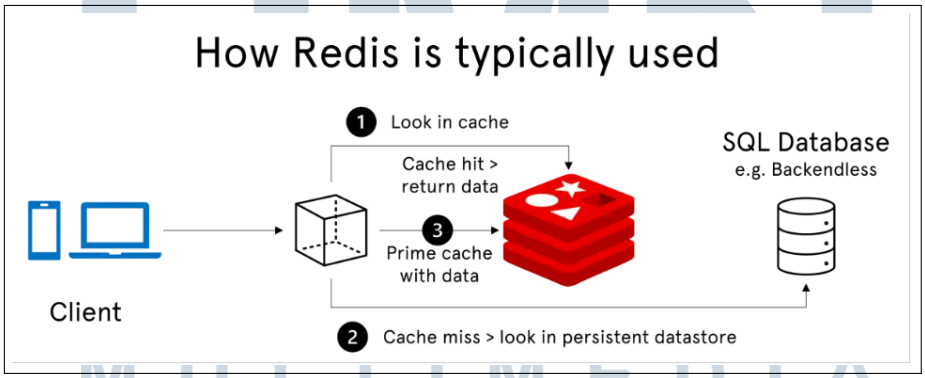
Gambar 2.2 menunjukkan perbedaan antara model pemrosesan *synchronous* dan *asynchronous*. Pada model *synchronous*, komponen A harus menunggu hingga proses pada komponen B selesai sebelum melanjutkan eksekusi, sehingga *thread* berada dalam kondisi *blocking*. Sebaliknya, pada model *asynchronous*, komponen A dapat melanjutkan proses lain setelah mengirim permintaan ke komponen B tanpa harus menunggu hasil secara langsung [24]. Pendekatan ini memungkinkan sistem menangani lebih banyak *request* secara konkuren dan mengurangi waktu tunggu pada operasi yang bersifat I/O atau komputasi intensif.

Implementasi *message queue* dan *dedicated worker process* memungkinkan *request* segera dimasukkan ke *queue* dengan *HTTP response* dikirim ke klien,

sementara *worker process* khusus melakukan *inference* secara asinkron, terbukti meningkatkan *throughput* hingga ratusan persen sambil menurunkan latensi persepsi pengguna [7]. Namun demikian, penelitian sebelumnya cenderung terbatas pada *environment* Java *Spring Boot* atau aplikasi web generik [9], sementara karakteristik unik *AI inference workload* seperti komputasi CPU/GPU intensif, *payload* besar, dan pola akses spesifik memerlukan pendekatan optimasi yang disesuaikan dengan *tech stack* Python/FastAPI yang umum digunakan untuk *inference serving*.

2.4 Caching pada Sistem Backend dan Redis

Caching hasil *inference* sangat efektif karena banyak aplikasi memiliki pola *request* yang berulang, dengan implementasi yang tepat dapat mengurangi beban komputasi drastis dan meningkatkan *response time* hingga ratusan kali lipat untuk *cache hit*. Redis dengan pola *cache-aside* pada aplikasi web *high-traffic* mencapai pengurangan *response time* sebesar 71.8% melalui *load testing* dengan 1.000 hingga 10.000 pengguna simultan, memvalidasi efektivitas *in-memory caching* dalam mengurangi latensi pada beban tinggi [12]. Analisis komparatif algoritma *distributed caching* (ARC, TLRU, LRU, LFU) menunjukkan implementasi ARC terdistribusi menghasilkan peningkatan *hit ratio* sebesar 12–18% pada *mixed workload* produksi, memberikan *insight* tentang pemilihan algoritma optimal berdasarkan karakteristik *workload* [25].



Gambar 2.3. Mekanisme *Cache-Aside* Menggunakan Redis

Gambar 2.3 menggambarkan mekanisme *cache-aside* yang umum digunakan pada sistem backend modern. Ketika klien mengirimkan permintaan data, sistem terlebih dahulu memeriksa apakah data tersedia di dalam cache. Jika terjadi *cache hit*, data langsung dikembalikan dari Redis sehingga waktu respons

menjadi sangat cepat. Sebaliknya, jika terjadi *cache miss*, sistem akan mengambil data dari basis data utama kemudian menyimpannya kembali ke dalam cache untuk permintaan berikutnya [26]. Mekanisme ini efektif dalam mengurangi beban komputasi dan meningkatkan performa sistem pada skenario dengan pola akses data yang berulang.

Redis sebagai *in-memory data structure store* dirancang untuk performa ekstrem dengan model *single-threaded event loop*, mampu menangani ratusan ribu operasi per detik, dengan dukungan struktur data fleksibel (*string*, *hash*, *list*, *set*) dan fitur *TTL* otomatis untuk menghindari *stale data* [12]. Meskipun efektivitas Redis untuk *caching* telah terbukti pada aplikasi web generik, fokus penelitian sebelumnya belum secara spesifik membahas *caching* hasil *inference* model AI yang memiliki karakteristik unik seperti ukuran *payload* besar dan pola akses berbeda [27]. Lebih lanjut, studi sebelumnya cenderung mengeksplorasi *caching* secara terpisah dari *asynchronous processing*, belum mengintegrasikan kedua teknik untuk mengeksplorasi sinergi potensial dalam meningkatkan performa *backend inference* AI.

2.4.1 Algoritma *Time-aware Least Recently Used* (TLRU)

TLRU (*Time-aware Least Recently Used*) merupakan algoritma eviksi *cache* yang diusulkan oleh Bilal dan Kang sebagai pengembangan dari LRU klasik dengan menambahkan komponen validitas temporal pada setiap entri *cache*. Bilal dan Kang mengidentifikasi kelemahan mendasar algoritma eviksi konvensional seperti LRU dan FIFO dalam konteks jaringan terdistribusi, yaitu ketidakmampuannya mempertahankan popularitas konten serta ketidaksadarannya terhadap batas waktu ketersediaan konten. Untuk mengatasi keterbatasan ini, TLRU didefinisikan sebagai *"content popularity and content life span aware eviction scheme"* sehingga eviksi tidak hanya mempertimbangkan *recency* akses tetapi juga batas waktu validitas konten yang tersimpan [28].

Pada TLRU, setiap entri *cache* dilengkapi dengan parameter TTU (*Time to Use*) yang menentukan batas waktu kebergunaan konten berdasarkan karakteristik lokal dan kebijakan administrator. Ketika kapasitas *cache* penuh, algoritma melakukan eviksi LRU pada subset entri yang masih valid, sehingga entri yang kurang populer dan mendekati kedaluwarsa diprioritaskan untuk dieviksi terlebih dahulu. Kondisi validitas entri dalam TLRU diformulasikan sebagai berikut.

$$\text{valid}(e) = \begin{cases} \text{true} & \text{jika } t_{now} - t_{created}(e) < TTL(e) \\ \text{false} & \text{jika sebaliknya} \end{cases} \tag{2.1}$$

di mana t_{now} adalah waktu saat ini, $t_{created}(e)$ adalah waktu pembuatan entri e, dan $TTL(e)$ adalah periode validitas yang ditetapkan untuk entri tersebut. Meskipun TLRU awalnya diusulkan dalam konteks *Information-Centric Networks* (ICN), prinsip *time-awareness* yang diperkenalkan yaitu penggabungan batas waktu validitas konten dengan kebijakan eviksi berbasis *recency* yang relevan untuk diterapkan pada sistem *backend inference* AI. Karakteristik ini sesuai karena model *machine learning* yang menghasilkan prediksi dapat diperbarui secara berkala, sehingga hasil prediksi yang melampaui periode validitasnya perlu dieviksi untuk menghindari pengembalian hasil dari versi model yang sudah tidak aktif.

2.5 Metrik Evaluasi Performa Sistem

Evaluasi performa sistem *inference* memerlukan pengukuran multi-dimensi yang mencakup latensi, *throughput*, *error rate*, dan efektivitas *caching*. Setiap metrik diformulasikan secara matematis untuk memastikan konsistensi pengukuran dan komparabilitas hasil antar skenario pengujian. Pemilihan metrik yang tepat sangat menentukan validitas kesimpulan yang ditarik dari hasil eksperimen, khususnya dalam konteks perbandingan dua arsitektur yang memiliki mekanisme pemrosesan yang berbeda secara fundamental [29].

2.5.1 Rata-rata *Response Time*

Rata-rata *response time* mengukur waktu rata-rata yang dibutuhkan sistem untuk memproses satu permintaan dari saat diterima hingga respons dikembalikan ke klien. Metrik ini memberikan gambaran umum performa sistem namun rentan terhadap distorsi akibat *outlier*, sehingga penggunaannya perlu dilengkapi dengan metrik distribusi seperti persentil [29]. Secara matematis, rata-rata *response time* mengikuti definisi rata-rata aritmetika sebagai berikut.

$$\bar{L} = \frac{1}{n} \sum_{i=1}^{n} L_i \tag{2.2}$$

di mana $\bar{L}$ adalah rata-rata *response time* dalam milidetik, n adalah jumlah total permintaan yang diproses, dan L_i adalah *response time* permintaan ke-i.

2.5.2 Metrik Persentil (*P50, P95, P99*)

Metrik persentil menggambarkan distribusi *response time* secara lebih representatif dibandingkan nilai rata-rata, khususnya untuk mengidentifikasi *tail latency* yang dialami sebagian kecil pengguna. P50 menunjukkan median *response time*, P95 menunjukkan nilai di mana 95% permintaan diselesaikan lebih cepat, dan P99 menggambarkan kondisi terburuk yang dialami 1% permintaan terakhir. Dalam aplikasi *real-time*, pengalaman pengguna lebih dipengaruhi oleh *tail latency* daripada nilai rata-rata, sehingga P95 dan P99 menjadi metrik utama dalam evaluasi sistem *inference* AI [13]. Diberikan dataset *response time* terurut menaik $X = \{x_1, x_2, \ldots, x_n\}$ di mana $x_1 \leq x_2 \leq \cdots \leq x_n$, nilai persentil ke-$p$ dihitung menggunakan metode *nearest rank* [30] sebagai berikut.

$$P_p = x_{\lceil \frac{p \cdot n}{100} \rceil} \tag{2.3}$$

di mana P_p adalah nilai persentil ke-p, n adalah jumlah total sampel, dan $\lceil \cdot \rceil$ adalah fungsi *ceiling* (pembulatan ke atas ke bilangan bulat terdekat).

2.5.3 *Throughput*

Throughput mengukur kapasitas sistem dalam memproses permintaan per satuan waktu dan dinyatakan dalam satuan *requests per second* (RPS). Nilai ini mencerminkan efisiensi keseluruhan sistem dan secara langsung dipengaruhi oleh mekanisme optimasi seperti *asynchronous processing* dan *caching* yang mengurangi waktu tunggu antar permintaan [9]. Peningkatan *throughput* yang lebih besar pada tingkat konkurensi yang lebih tinggi mengindikasikan bahwa sistem bersifat skalabel [29]. *Throughput* diformulasikan sebagai berikut.

$$\text{RPS} = \frac{N}{T} \tag{2.4}$$

di mana N adalah jumlah total permintaan yang berhasil diproses dan T adalah durasi pengujian dalam detik.

2.5.4 *Error Rate*

Error rate mengukur persentase permintaan yang gagal diproses oleh sistem, mencakup respons dengan kode HTTP 4xx/5xx serta permintaan yang mengalami *timeout* akibat durasi pemrosesan yang melampaui batas waktu koneksi. Metrik ini merupakan indikator utama keandalan (*reliability*) sistem di bawah kondisi beban tinggi dan menjadi pembeda signifikan antara arsitektur *synchronous* dan *asynchronous* [7]. Nilai *error rate* yang tinggi pada arsitektur *synchronous* umumnya disebabkan oleh *connection timeout* ketika durasi inference melampaui batas waktu koneksi HTTP yang ditetapkan. *Error rate* diformulasikan sebagai berikut.

$$E = \frac{N_{error}}{N_{total}} \times 100\% \tag{2.5}$$

di mana N_{error} adalah jumlah permintaan yang gagal dan N_{total} adalah jumlah total permintaan yang dikirimkan selama pengujian.

2.5.5 Persentase Peningkatan Performa

Untuk mengkuantifikasi dampak optimasi yang diterapkan, perbedaan antara kondisi *baseline* dan kondisi setelah optimasi dinyatakan dalam persentase perubahan relatif terhadap nilai awal. Nilai positif pada formula ini menunjukkan penurunan metrik yang diinginkan seperti latensi dan *error rate*, sedangkan untuk metrik *throughput*, nilai positif menunjukkan peningkatan yang diinginkan dengan menukar posisi M_{before} dan M_{after} [29]. Pendekatan persentase relatif dipilih karena memungkinkan perbandingan yang adil antar metrik dengan satuan yang berbeda. Formula yang digunakan adalah sebagai berikut.

$$\Delta M = \frac{M_{before} - M_{after}}{M_{before}} \times 100\% \tag{2.6}$$

di mana ΔM adalah persentase perubahan metrik, M_{before} adalah nilai metrik pada arsitektur *synchronous*, dan M_{after} adalah nilai metrik pada arsitektur *asynchronous* dengan *warm cache*.

2.5.6 *Cache Hit Ratio*

Cache hit ratio mengukur efektivitas mekanisme *caching* dengan menghitung proporsi permintaan yang berhasil dilayani langsung dari *cache* tanpa perlu menjalankan proses inferensi ulang. Nilai CHR yang tinggi secara langsung berkorelasi dengan penurunan latensi dan beban komputasi karena setiap *cache hit* mengeliminasi satu siklus eksekusi model yang membutuhkan waktu puluhan detik [12]. Dalam konteks sistem inferensi AI, pola akses berulang dengan input yang identik menjadikan mekanisme *caching* sangat efektif dibandingkan aplikasi web generik karena biaya komputasi yang dihindari jauh lebih besar [27]. *Cache hit ratio* diformulasikan sebagai berikut.

$$\text{CHR} = \frac{N_{hit}}{N_{hit} + N_{miss}} \times 100\% \tag{2.7}$$

di mana N_{hit} adalah jumlah permintaan yang dilayani dari *cache* dan N_{miss} adalah jumlah permintaan yang tidak ditemukan di *cache* sehingga memerlukan eksekusi inferensi penuh.

2.5.7 *Load Testing* dan *Stress Testing*

Load testing mensimulasikan kondisi beban normal untuk mengukur stabilitas dan performa sistem dalam kondisi operasional, sedangkan *stress testing* mendorong sistem hingga melampaui kapasitasnya untuk mengidentifikasi titik kegagalan dan batas kemampuan sistem. Kombinasi kedua metode ini memungkinkan pengukuran seluruh metrik yang telah didefinisikan pada subbagian sebelumnya secara komprehensif pada berbagai tingkat beban [29]. Evaluasi ini memungkinkan identifikasi *trade-off* antara peningkatan performa dan kompleksitas sistem, khususnya dalam penerapan *asynchronous processing* dan Redis *caching* pada arsitektur *hybrid* Node.js dan FastAPI [7, 12].

2.6 Tinjauan Literatur

Tinjauan literatur dilakukan untuk mengidentifikasi pendekatan-pendekatan yang telah digunakan dalam optimasi *backend* pada sistem berbasis AI dan arsitektur *microservices*. Analisis ini bertujuan untuk memahami tren penelitian terkini, metode yang digunakan, serta capaian performa yang telah dilaporkan pada

studi sebelumnya. Dengan membandingkan metode, arsitektur, fokus penelitian, dan hasil utama, dapat diidentifikasi posisi penelitian ini terhadap penelitian terdahulu serta celah penelitian (*research gap*) yang masih terbuka. Ringkasan komparatif penelitian terdahulu disajikan pada Tabel 2.1.

Tabel 2.1. Perbandingan penelitian terdahulu terkait optimasi *backend inference*

No	Peneliti	Metode	Fokus	Hasil Utama	Keterbaruan Penelitian Ini
1	Balivada (2025)	*Async Worker Processes, Microservices*	*Image Processing*	+370% *throughput*, -70–85% *latency*	Menambahkan Redis *caching* untuk eksplorasi sinergi *async* + *cache* [7].
2	Zbarcea et al. (2026)	*Reactive Programming, REST API (Java)*	*High Concurrency*	+75% *throughput*, -68% memori	Implementasi pada *tech stack* Python/FastAPI untuk AI *inference* [9].
3	Kaptosv (2025)	*Redis Cache-Aside, Cloud Infrastructure*	*High-Traffic Web Apps*	-71.8% *response time*	Fokus pada *caching* hasil *inference* AI dengan *payload* besar [12].
4	Maharjan et al. (2023)	*Benchmark Message Queue, Microservices (Kafka, Redis, RabbitMQ)*	*Message Queue Performance*	Redis: *lowest latency*, Kafka: *highest throughput*	Integrasi *message queue* dengan Redis *caching* dalam satu sistem [31].
5	Chaplia & Klym (2025)	*Improved Routing, Node.js Microservices*	REST API *Internal Routing*	Mengurangi kompleksitas sistem	Arsitektur *hybrid* Node.js API *Gateway* + FastAPI *inference* [32].
6	Mayer & Richards (2025)	*Distributed Caching Algorithms (ARC, TLRU, LRU, LFU)*	*Microservices*	+12–18% *cache hit ratio*	Evaluasi algoritma *caching* pada konteks *inference* AI [25].
7	AllMulti Journal (2025)	*Redis vs Memcached, Kubernetes Microservices*	*Caching Strategy Comparison*	Redis: struktur data kompleks; Memcached: *lightweight*	Implementasi Redis untuk *inference* dengan TTL dinamis [33].

Tabel 2.1. Perbandingan penelitian terdahulu (Lanjutan)

No	Peneliti	Metode	Fokus	Hasil Utama	Keterbaruan Penelitian Ini
8	Manivelmurugan (2025)	*Distributed Caching, Cloud-Native*	Skalabilitas dan resiliensi	Mengurangi *latency* dan beban *backend*	Kombinasi *distributed caching* dengan *async processing* [27].
9	Fu et al. (2021)	*Message Queue Comparison, Distributed Systems*	*Inference Workload*	RocketMQ ¡ 10 ms *latency*	Pemilihan *message queue* optimal untuk *inference workload* [11].
10	Naayini (2024)	*Adaptive Caching & Provisioning, Serverless (AWS Lambda)*	AI Model *Inference Serving*	5–10x *throughput*, pengurangan biaya	Implementasi pada arsitektur *non-serverless* (Node.js + FastAPI) [13].

2.6.1 Optimasi Asynchronous Processing untuk Image Processing

Penelitian mendemonstrasikan efektivitas *asynchronous worker processes* dengan *message queue* pada sistem pemrosesan citra, menghasilkan peningkatan *throughput* sebesar 370% dan pengurangan waktu respons API hingga 70–85% [7]. Pendekatan ini membuktikan bahwa pemisahan alur HTTP *response* dari eksekusi tugas komputasi intensif secara signifikan meningkatkan kapasitas sistem dalam menangani beban tinggi. Implementasi dilakukan dengan menempatkan *request* ke dalam antrian, mengirim respons segera ke klien, sementara *dedicated worker process* melakukan pemrosesan citra secara asinkron di *background*. Namun, penelitian ini tidak mengeksplorasi kombinasi *asynchronous processing* dengan mekanisme *caching* untuk mengurangi redundansi komputasi pada *request* yang berulang.

2.6.2 Reactive Programming untuk High Concurrency pada Java

Analisis performa antara model *asynchronous* (*thread-per-request*) dan *reactive programming* (*event-loop*) dalam konteks aplikasi Java dengan Redis *distributed caching* menunjukkan hasil yang signifikan [9]. Pendekatan *reactive* meningkatkan *throughput* sebesar 75% dan mengurangi penggunaan memori

hingga 68% dibandingkan model *asynchronous thread-per-request*. Studi ini menegaskan bahwa arsitektur *reactive* superior untuk skenario konkurensi tinggi, khususnya pada operasi I/O intensif seperti *caching*. Eksperimen dilakukan pada *environment* Java *Spring Boot* dengan berbagai skenario beban, termasuk operasi CPU-*bound* dan I/O-*bound*. Akan tetapi, penelitian ini terbatas pada *tech stack* Java dan belum mengkaji implementasi pada Python/FastAPI yang umum digunakan untuk *inference serving* AI.

2.6.3 Redis Cache-Aside untuk High-Traffic Web Applications

Implementasi Redis dengan pola *cache-aside* pada aplikasi web *high-traffic* mencapai pengurangan *response time* sebesar 71.8% melalui *load testing* dengan 1.000 hingga 10.000 pengguna simultan [12]. Penelitian ini memvalidasi efektivitas Redis dalam mengurangi latensi pada beban tinggi dan memberikan bukti empiris tentang pentingnya strategi *caching* yang tepat. Pola *cache-aside* memungkinkan aplikasi memeriksa *cache* terlebih dahulu sebelum mengakses sumber data utama, dengan mekanisme otomatis untuk memperbarui *cache* ketika data berubah. Meskipun demikian, fokus penelitian ini adalah aplikasi web generik dan tidak secara spesifik membahas *caching* hasil *inference* model AI yang memiliki karakteristik unik seperti ukuran *payload* besar dan pola akses yang berbeda.

2.6.4 Comparative Analysis of Distributed Caching Algorithms

Analisis komparatif terhadap berbagai algoritma *distributed caching* (ARC, TLRU, LRU, LFU) dalam arsitektur mikroservis menunjukkan hasil yang menarik [25]. Implementasi ARC (*Adaptive Replacement Cache*) terdistribusi menghasilkan peningkatan *hit ratio* sebesar 12–18% pada *mixed workload* produksi dibandingkan dengan algoritma tradisional seperti LRU. Penelitian ini memberikan *insight* mendalam tentang pemilihan algoritma *caching* yang optimal berdasarkan karakteristik *workload*, dengan ARC menunjukkan adaptabilitas superior dalam menangani pola akses yang bervariasi. Namun, studi ini tidak mengintegrasikan analisis *caching* dengan optimasi *asynchronous processing*, sehingga belum mengeksplorasi sinergi antara kedua teknik tersebut dalam konteks sistem AI.