Jurnal Ilmiah Sistem Informasi dan Teknik Informatika (JISTI) p-ISSN: 2620 – 5327
Volume 9, Nomor 1, April 2026 e-ISSN: 2715 – 5501
DOI : 10.57093/ jisti.v9i1.395

# Analisis Komparatif Performa Go, Bun, dan PHP dalam Menangani HTTP Request Data Besar dari Database MySQL

**Kamarudin[1] Nuzulul Afia Idris[2] Guntur[3] Yusri[4]**
Program Studi Teknik Informatika, Universitas Handayani Makassar[1,2]
Program Studi Sistem Komputer, Universitas Handayani Makassar[3]
Program Studi Manajemen, STIE Pelita Buana Makassar[4]
Jln. Adyaksa No. 01 Makassar, Sulawesi Selatan, 90231, Indonesia[1,2,3]
Jln. Laniang Blok F No. 4, Sulawesi Selatan, 90245, Indonesia[4]
kamarudin@handayani.ac.id[1], nakkekulo136@gmail.com[2], guntur@handayani.ac.id[3], yusri.acho@gmail.com[4]

**Kata Kunci :**
*Go*;
*Bun*;
PHP;
*Benchmark*;
*Grafana k6;*

**ABSTRAK**

Penelitian ini menyajikan analisis komparatif performa tiga *runtime backend Go*, *Bun*, dan PHP dalam menangani HTTP *request* dengan beban data besar dari *database* MySQL. Pengujian dilakukan menggunakan *Grafana* k6 dengan skenario *ramp up* hingga 250–300 *virtual users* (VU) selama 6–7 menit, di *deploy* pada platform *Railway.app* dengan spesifikasi 8 vCPU dan 8 GB RAM. Tiga skenario diuji: (1) *query* 10.000 baris × 40 *field* dengan 250 VU, (2) *query* 5.000 baris × 40 field dengan 250 VU, dan (3) *query* 1.000 baris × 50 *field* dengan 300 VU. Hasil menunjukkan bahwa Go secara konsisten unggul di seluruh skenario dengan *throughput* tertinggi mencapai 19,3 req/s pada skenario 10.000 baris dan konsumsi memori yang efisien (43–155 MB). Bun berada di posisi kedua dengan *throughput* 2,1–3,04 req/s namun mengalami konsumsi memori tertinggi (116–425 MB) akibat *double allocation* pada proses JSON.stringify(). PHP menunjukkan *throughput* terendah (3,5–4,8 req/s) dengan *error rate* yang sangat tinggi (81,4–99,9%) akibat keterbatasan arsitektur PHP-FPM yang *sinkronus* dalam menangani *concurrent requests*. *Root cause* analysis mengidentifikasi bahwa *streaming* JSON *encoder Go*, model *goroutine* yang ringan, dan *garbage collector* yang prediktabel menjadi faktor utama keunggulan performa Go. Temuan penelitian memberikan panduan empiris bagi pengembang dalam memilih *runtime* yang sesuai berdasarkan karakteristik beban data dan tingkat konkurensi yang dibutuhkan.

***Keywords***
*Go*;
*Bun*;
PHP;
*Benchmark*;
*Grafana k6;*

***ABSTRACT***

*This study presents a comparative performance analysis of three backend runtimes Go, Bun, and PHP in handling HTTP requests with large data loads from a MySQL database. Testing was conducted using Grafana k6 with a ramp-up scenario of up to 250–300 virtual users (VUs) for 6–7 minutes, deployed on the Railway.app platform with an 8 vCPU and 8 GB RAM specification. Three scenarios were tested: (1) a query of 10,000 rows × 40 fields with 250 VUs, (2) a query of 5,000 rows × 40 fields with 250 VUs, and (3) a query of 1,000 rows × 50 fields with 300 VUs. Results show that Go consistently outperforms in all scenarios with peak throughput of 19.3 req/s in the 10,000-row scenario and efficient memory consumption (43–155 MB). Bun ranks second with 2.1–3.04 req/s throughput but experiences the highest memory consumption (116–425 MB) due to double allocation in JSON.stringify() processing. PHP shows the lowest throughput (3.5–4.8 req/s) with a very high error rate (81.4–99.9%) due to architectural limitations of synchronous PHP-FPM in handling concurrent requests. Root cause analysis identifies Go's streaming JSON encoder, lightweight goroutine model, and*

*predictable garbage collector as the primary factors behind its performance superiority. Research findings provide empirical guidance for developers in selecting the appropriate runtime based on data load characteristics and required concurrency levels.*



## PENDAHULUAN

Pemilihan *runtime backend* merupakan keputusan arsitektur yang berdampak langsung terhadap kemampuan sistem dalam melayani pengguna secara simultan, efisiensi penggunaan sumber daya infrastruktur, serta biaya operasional jangka panjang. Dalam konteks pengembangan API yang mengembalikan data dalam jumlah besar dari database relasional, perbedaan antar *runtime* dapat memunculkan perbedaan performa yang sangat signifikan, bukan hanya dalam kondisi trafik rendah, melainkan terutama pada kondisi beban tinggi dengan ratusan pengguna konkuren.

Go (Golang), yang dikembangkan oleh *Google* dan dirilis publik pada 2009, telah menjadi pilihan populer untuk layanan *backend* berkinerja tinggi berkat model konkurensi *goroutine*-nya yang ringan dan *garbage collector* yang efisien. Bun, *runtime JavaScript* berbasis *JavaScriptCore engine* yang mengklaim performa lebih tinggi dari *Node.js*, telah menarik perhatian komunitas *developer* sejak rilis stabil v1.0 pada September 2023. Sementara PHP, meskipun telah berusia lebih dari tiga dekade, tetap mendominasi ekosistem web server dan terus mendapatkan peningkatan performa melalui PHP 8.x dengan *OPcache* dan *JIT compiler*.

Namun, sebagian besar studi *benchmark* yang tersedia membandingkan *runtime* ini dalam skenario yang terlalu sederhana seperti *hello world response* atau JSON *serialization* tanpa beban database nyata. Kondisi ini menciptakan kesenjangan antara data *benchmark* yang tersedia dengan kebutuhan pengembang yang ingin memilih *runtime* untuk API data *heavy* yang sesungguhnya melayani *query* ke database relasional dengan *volume* baris yang besar.

Penelitian ini hadir untuk mengisi kesenjangan tersebut dengan menyajikan hasil *benchmark* yang dilakukan secara eksperimental pada infrastruktur *cloud* nyata (*Railway.app*) menggunakan Grafana k6, dengan skenario *query* MySQL yang merepresentasikan kondisi produksi: pengambilan ribuan hingga puluhan ribu baris data dengan banyak *field* secara bersamaan oleh ratusan *virtual users*. Go adalah bahasa pemrograman *statically typed* dan *compiled* yang dirancang dengan filosofi kesederhanaan dan performa tinggi. Keunggulan utama Go untuk layanan *backend* terletak pada model *goroutine*, yaitu: *lightweight thread* yang dikelola oleh Go *runtime scheduler* menggunakan model M:N *threading memapping N goroutine* ke M OS *thread*. Setiap *goroutine* hanya memerlukan 2–8 KB memori awal (berbeda dengan OS *thread* yang membutuhkan 1–2 MB), sehingga Go dapat menangani ratusan ribu koneksi *concurrent* secara efisien (Donovan & Kernighan, 2016).

Untuk serialisasi JSON ke HTTP *response*, Go menggunakan pendekatan *streaming* melalui *json.NewEncoder(w).Encode*() yang menulis langsung ke *http.ResponseWriter* tanpa perlu mengalokasikan *buffer string* perantara. Pendekatan *zero allocation* ini menjadi keunggulan signifikan ketika menangani dataset besar karena menghindari *double allocation* yang umum terjadi pada *runtime* lain (Go Documentation, 2024). Bun adalah *JavaScript runtime* modern yang menggunakan *JavaScriptCore* (JSC) *engine* dari *Apple WebKit*, berbeda dari *Node.js* yang menggunakan V8 dari *Google.* Bun dirancang sebagai *all-in-one toolkit* yang mencakup *runtime, package manager, bundler,* dan *test runner* dalam satu *binary.* Bun mengklaim performa *startup* yang lebih cepat dan penanganan I/O yang lebih efisien dibandingkan Node.js (Sumner, 2023).

Namun, untuk *workload* yang melibatkan serialisasi objek *JavaScript* besar ke JSON (menggunakan JSON.*stringify*()), Bun mengalami masalah *double allocation*: data asli tersimpan di memori, kemudian proses *stringify* menciptakan alokasi *string* baru yang berisi representasi JSON dari data tersebut (Ismail et al., 2023). Untuk dataset berukuran besar, ini berarti penggunaan memori yang mendekati 2× ukuran data asli (WWT, 2023). PHP menggunakan model eksekusi berbasis proses melalui PHP-FPM (*FastCGI Process Manager*), dimana setiap HTTP *request* ditangani oleh satu *worker process* secara sinkronus. Model ini memiliki batasan inheren dalam menangani *concurrent requests*: jumlah *request* yang dapat dilayani secara simultan dibatasi oleh jumlah *worker process* yang dikonfigurasi. Ketika semua *worker* sedang aktif, *request* baru harus mengantri hingga *worker* tersedia (Tatroe et al., 2020).

Pada kondisi beban tinggi dengan ratusan *virtual users*, antrian *request* PHP-FPM dapat melebihi batas *timeout,* mengakibatkan *error rate* yang tinggi. Meskipun PHP 8.x dengan *OPcache* dan *JIT compiler* memberikan peningkatan performa *CPU-bound* yang signifikan, keterbatasan model konkuren sinkronus ini tetap menjadi *bottleneck* utama untuk *workload* I/O-*bound* yang membutuhkan tingkat konkurensi tinggi (Codesoltech, 2024). Grafana k6 adalah *open source load testing tool* yang ditulis dalam Go dan menggunakan *JavaScript* sebagai bahasa *scripting* skenario. k6 dirancang untuk *developer* dan mendukung berbagai pola *load testing* termasuk *constant load, ramping VUs,* dan *spike testing. Output* k6 mencakup metrik komprehensif seperti: *throughput* (http_reqs), latensi *percentile* (p(50), p(95), p(99)), *error rate*, dan *transfer rate* (Grafana Labs, 2024).

Penggunaan k6 dengan *mode ramp-up* memungkinkan simulasi kondisi trafik yang lebih realistis dibandingkan *load testing* dengan *concurrent users* yang *fixed* dari awal, karena mencerminkan pertumbuhan bertahap jumlah pengguna yang mengakses sistem dalam kondisi nyata Lemire (2023) membandingkan Go dan Bun untuk *web server hello-world* dan menemukan bahwa Bun (*Bun.serve*()) dapat lebih cepat dari Go net/http dalam skenario minimal tanpa I/O. Namun penelitian tersebut tidak menyertakan beban database. WWT (2023) membandingkan Bun, C#, Go, *Node.js*, dan *Python* dengan skenario yang melibatkan *in-memory data lookup*, menemukan Go dan Bun sangat kompetitif dalam *req/s* namun Go jauh lebih unggul dalam efisiensi memori. *TechEmpower Round* 23 (2024) menyajikan *benchmark* komprehensif dengan berbagai skenario termasuk database *query,* menunjukkan *framework* berbasis *compiled language* (Go, C#, Rust) secara konsisten unggul dalam *throughput* pada skenario database. Penelitian ini berbeda dari studi terdahulu karena: 1) menggunakan data nyata dari eksperimen di *cloud environment* (*Railway.app*); 2) menyertakan skenario *query* dengan *volume* data yang jauh lebih besar (hingga 10.000 baris × 40 *field*); dan 3) menganalisis *root cause* teknis secara mendalam berdasarkan karakteristik masing-masing *runtime*.

## METODE PENELITIAN

### A. Desain Penelitian

Penelitian ini menggunakan desain eksperimental kuantitatif dengan pendekatan *controlled benchmark* pada *cloud environment*. Variabel bebas adalah jenis *runtime* (Go, Bun, PHP). Variabel terikat meliputi *throughput* (*req/s*), latensi P95, *error rate*, dan konsumsi memori. Variabel kontrol meliputi spesifikasi infrastruktur (*Railway.app*: 8 vCPU, 8 GB RAM), jenis database (MySQL), *query* SQL yang identik, dan *tool* pengujian (Grafana k6).

p-**ISSN: 2620 – 5327**
**e-ISSN: 2715 – 5501**

### B. Spesifikasi Infrastruktur

Tabel 1: Spesifikasi Infrastruktur Pengujian

| Komponen | Spesifikasi |
|---|---|
| *Platform Deployment* | *Railway.app (Cloud PaaS)* |
| CPU | 8 Vcpu |
| *Memory* | 8 GB RAM |
| *Database* | MySQL (server terpisah di *Railway.app)* |
| *Network* | *Internal Railway.app network (latency 20–100 ms ke DB via internet)* |
| *Tool Load Testing* | *Grafana k6* (dijalankan dari mesin lokal penguji) |
| *Monitoring Metrics* | *Railway.app metrics dashboard + k6 output* |

### C. Konfigurasi *Runtime*

Tabel 2: Konfigurasi *Runtime* dan Pendekatan Teknis

| Runtime | Bahasa | Pendekatan Serialisasi JSON | Library DB |
|---|---|---|---|
| Go | Go 1.22+ | *json.NewEncoder(w).Encode()* *streaming* langsung ke *ResponseWriter* | *database/sql + go-sql-driver/mysql* |
| Bun | *JavaScript (TypeScript)* | JSON.*stringify*() *full allocation ke string buffer* | *mysql2 (Node.js compat layer)* |
| PHP | PHP 8.x + FPM | *json_encode*() *buffer* ke *string*, lalu *echo* | *PDO* dengan *ATTR_PERSISTENT* |

### D. Skenario *Benchmark*

Tiga skenario dirancang untuk menguji performa pada berbagai ukuran *payload* dan tingkat beban. Setiap skenario menggunakan *query* SQL SELECT yang identik dengan perbedaan pada jumlah baris yang dikembalikan.

Tabel 3: Konfigurasi Skenario *Benchmark*

| Skenario | Dataset | Virtual Users | Durasi | Threshold k6 |
|---|---|---|---|---|
| Skenario 1 | 10.000 baris × 40 *field* | 250 VU (*ramp-up*) | 7 menit | *error rate* < 0,5% \| P95 < 30 detik |
| Skenario 2 | 5.000 baris × 40 *field* | 250 VU (*ramp-up*) | 7 menit | *error rate* < 0,5% \| P95 < 30 detik |
| Skenario 3 | 1.000 baris × 50 *field* | 300 VU (*ramp-up*) | 6,5 menit | *error rate* < 0,5% \| P95 < 30 detik |

p-**ISSN: 2620 – 5327**
**e-ISSN: 2715 – 5501**

**Catatan Metodologi**

*Query database* dilakukan ke MySQL yang terhubung melalui internet dengan *latency* 20–100 ms per *query*. Kondisi ini merepresentasikan skenario *deployment* nyata di mana *database server* berada di lokasi terpisah dari *application server*, sehingga hasil *benchmark* mencerminkan performa *end-to-end* termasuk *overhead* jaringan bukan hanya performa *runtime* secara isolasi.

**E. Metrik yang Diukur**

1. *Throughput*: jumlah HTTP *request* berhasil per detik (*req/s*) yang diselesaikan *runtime*;
2. P95 *Response Time*: latensi dimana 95% *request* selesai pada waktu tersebut atau lebih cepat;
3. *Error Rate*: persentase *request* yang gagal atau *timeout* selama pengujian;
4. HTTP *Fail Rate*: persentase *request* dengan HTTP status *non*-2xx; dan
5. *Memory Peak*: konsumsi RAM tertinggi yang dicapai selama skenario berlangsung.

## HASIL PENELITIAN DAN PEMBAHASAN

### A. Hasil Penelitian

#### 1. Ringkasan Hasil Seluruh Skenario

Tabel 4 menyajikan hasil keseluruhan dari tiga *runtime* pada tiga skenario *benchmark*. Data ini merupakan hasil pengujian langsung menggunakan Grafana k6 pada *platform Railway.app* dengan spesifikasi yang telah ditetapkan.

Tabel 4: Hasil Lengkap *Benchmark Go* vs Bun vs PHP Semua Skenario

| Skenario | Runtime | Req/s | P95 Response | Error Rate | Memory Peak | HTTP Fail |
|---|---|---|---|---|---|---|
| 10k *row* 40 *field* 250 VU | Go | 19,3 | 18 detik | 70,38% | 77 MB | 70,38% |
| | Bun | 2,75 | 60 detik | 98,8% | 425 MB | 83,9% |
| | PHP | 3,5 | 120 detik | 99,9% | 74 MB | 92,3% |
| 5k *row* 40 *field* 250 VU | Go | 15,2 | 22,6 detik | 79,62% | 155 MB | 0% |
| | Bun | 3,04 | 104 detik | 96,2% | 392 MB | 1,35% |
| | PHP | 3,5 | 120 detik | 99,9% | 41 MB | 85,9% |
| 1k *row* 50 *field* 300 VU | Go | 2,15 | 1m 59s | 98,06% | 45 MB | 5,9% |
| | Bun | 2,1 | 2m 0s | 98,3% | 116 MB | 6,6% |
| | PHP | 4,8 | 1m 56s | 99,7% | 14 MB | 81,4% |

(**Sumber:** Grafana k6 + *Railway.app Metrics*)

**2. Analisis Skenario 1: Query 10.000 Baris × 40 Field (250 VU)**

Skenario 1 merupakan pengujian paling berat dengan *payload* terbesar setiap *request* mengembalikan 10.000 baris dengan 40 *field* per baris dari MySQL. Ini merepresentasikan kasus penggunaan nyata seperti ekspor laporan atau data *bulk* API.

Go mencatat *throughput* 19,3 *req/s* dengan P95 *response time* 18 detik. Meskipun latensi terlihat tinggi, ini disebabkan oleh waktu yang dibutuhkan untuk mentransfer *payload* JSON berukuran besar (estimasi 10–50 MB per *response*) melalui koneksi jaringan. Go berhasil menyelesaikan *request* tanpa *timeout* yang disebabkan oleh *concurrency failure error* 70,38% yang tercatat pada skenario ini merupakan HTTP *failure* yang terjadi akibat beban *payload* sangat besar yang melebihi *threshold timeout* k6 (30 detik), bukan akibat kegagalan *concurrency handling runtime* itu sendiri. *Memory peak Go* sebesar 77 MB menunjukkan efisiensi *streaming encoder* yang tidak menahan seluruh data di RAM sebelum mengirimkan *response*.

Bun hanya mencapai 2,75 *req/s* dengan P95 60 detik dan *error rate* 98,8%. Rendahnya *throughput* disebabkan oleh JSON.*stringify*() yang menciptakan alokasi *string* besar di memori sebelum *response* dapat dikirim. *Memory peak* 425 MB pada skenario ini mengkonfirmasi terjadinya *double allocation:* data MySQL di *load* ke memori *JavaScript* (sekitar 200 MB), kemudian JSON.*stringify*() menghasilkan *string* baru berukuran setara yang disimpan secara terpisah di memori. PHP mencatat *throughput* 3,5 req/s dengan P95 120 detik (*timeout* penuh) dan *error rate* 99,9%. Arsitektur sinkronus PHP-FPM tidak mampu menangani 250 VU bersamaan sebagian besar *request* mengantri melampaui *threshold timeout* 30 detik k6. *Memory* PHP relatif rendah (74 MB) karena model per *process* mengalokasikan memori secara terisolasi, namun ini justru menjadi pembatas karena jumlah *worker process* yang tersedia tidak mencukupi untuk melayani 250 *concurrent request*.

**Temuan Utama Skenario 1**

Go unggul sangat signifikan dalam *throughput* (19,3 vs 2,75 vs 3,5 req/s). *Bottleneck* utama semua *runtime* pada skenario ini adalah kombinasi *payload* besar (10k baris) + *latency* jaringan ke MySQL (20–100 ms). Optimasi database (*indexing, connection pooling, read replica* lokal) berpotensi memberikan *improvement* 3–10× pada semua *runtime* sebelum optimasi *runtime* itu sendiri menjadi relevan.

**3. Analisis Skenario 2: Query 5.000 Baris × 40 Field (250 VU)**

Pengurangan beban data dari 10.000 ke 5.000 baris memberikan gambaran menarik tentang bagaimana masing-masing *runtime* berskala ketika *payload* berkurang setengahnya. Go menunjukkan perbaikan signifikan: *throughput* tetap tinggi di 15,2 req/s, namun yang paling menonjol adalah HTTP *fail rate* yang turun ke 0% berarti seluruh *request* yang diselesaikan Go berhasil tanpa *error* HTTP. *Memory peak Go* naik ke 155 MB pada skenario ini, yang mungkin disebabkan oleh peningkatan jumlah *goroutine* yang melayani request secara paralel.

Bun sedikit meningkat ke 3,04 req/s dengan HTTP *fail rate* 1,35% mendekati *threshold* maksimal yang ditetapkan (< 1%). *Memory peak* masih sangat tinggi di 392 MB, menunjukkan bahwa masalah *double allocation* tetap terjadi meskipun ukuran dataset berkurang. P95 *response time* 104 detik mengindikasikan bahwa mayoritas *request* memerlukan waktu sangat lama untuk diselesaikan.

PHP menunjukkan *throughput* identik dengan skenario 1 (3,5 req/s) meskipun beban data berkurang 50%. Ini mengkonfirmasi bahwa *bottleneck* PHP bukan pada pemrosesan data, melainkan pada model *concurrency* yang membatasi jumlah *request* yang dapat dilayani secara simultan. *Error*

p-**ISSN: 2620 – 5327**
**e-ISSN: 2715 – 5501**

*rate* 99,9% dan HTTP *fail rate* 85,9% pada skenario ini menunjukkan bahwa PHP-FPM *workers* kehabisan kapasitas menangani 250 VU bahkan dengan *payload* yang lebih kecil.

### 4. Analisis Skenario 3: Query 1.000 Baris × 50 Field (300 VU)

Skenario 3 memberikan hasil yang paling menarik dan kontras dengan dua skenario sebelumnya. Dengan beban data yang jauh lebih kecil (1.000 baris), PHP justru mencatat *throughput* tertinggi di antara tiga *runtime*: 4,8 req/s, mengalahkan Go (2,15 req/s) dan Bun (2,1 req/s). Namun, penting untuk memahami konteks angka ini.

Peningkatan VU dari 250 ke 300 pada skenario 3 memberikan tekanan tambahan pada semua *runtime*. Pada beban *query* 1.000 baris, kecepatan pemrosesan data tidak lagi menjadi *bottleneck* dominan *bottleneck* bergeser ke *latency* jaringan dan *overhead* koneksi database. PHP mendapatkan keuntungan relatif karena waktu *query* yang lebih singkat memungkinkan *worker processes*-nya berputar lebih cepat melayani *request* berikutnya.

Go dan Bun memiliki *throughput* yang hampir identik (2,15 vs 2,1 req/s) dan P95 yang serupa (1m59s vs 2m0s), mengindikasikan bahwa keduanya mencapai saturasi yang disebabkan oleh *overhead* koneksi MySQL yang menjadi *shared bottleneck. Error rate Go* (98,06%) dan Bun (98,3%) yang sangat tinggi pada skenario ini yang kontradiktif dengan performa baik di skenario sebelumnya mengindikasikan bahwa peningkatan VU ke 300 menciptakan tekanan *concurrency* yang melampaui kapasitas *connection pool database* yang dikonfigurasi.

**Catatan Penting: Konteks *Error Rate***

*Error rate* tinggi (> 95%) pada seluruh *runtime* di skenario 3 dengan 300 VU terutama disebabkan oleh *bottleneck* pada MySQL *connection pool* melalui koneksi internet dengan *latency* 20–100 ms. Ini bukan cerminan kegagalan *runtime* dalam mengelola *concurrency*, melainkan *reflection* dari keterbatasan kapasitas database yang diakses melalui jaringan internet. Dalam *deployment* produksi dengan database di jaringan lokal dan *connection pooling* yang optimal, angka ini akan jauh berbeda.

### 5. Analisis *Resource Usage*: Memori dan CPU

Profil *resource usage* mengungkapkan karakteristik arsitektur masing-masing *runtime* yang menjelaskan *trade-off* antara *throughput* dan efisiensi sumber daya.

Tabel 5: Profil *Resource Usage* Masing-masing *Runtime*

| *Runtime* | *Memory Peak* (Skenario 1) | *Memory Peak* (Skenario 2) | *Memory Peak* (Skenario 3) | CPU *Usage* | Karakteristik |
|---|---|---|---|---|---|
| Go | 77 MB | 155 MB | 45 MB | 0,5–0,9 vCPU | *Streaming encoder, goroutine ringan, GC predictable* |
| Bun | 425 MB | 392 MB | 116 MB | 0,4–0,8 vCPU | *Double allocation JSON.stringify, GC pressure* tinggi |
| PHP | 74 MB | 41 MB | 14 MB | 0,1–0,2 vCPU | Per *process isolation,* efisien per *worker* tapi tidak *scalable* |

Go mencapai utilisasi CPU yang tinggi (0,5–0,9 vCPU) dengan *throughput* yang proporsional ini adalah tanda efisiensi: CPU digunakan secara produktif untuk memproses *request*, bukan untuk *garbage collection overhead* yang berlebihan. Bun menggunakan CPU dalam *range* yang mirip (0,4–0,8 vCPU) namun dengan *throughput* yang jauh lebih rendah, mengindikasikan bahwa sebagian besar CPU *time* Bun terpakai untuk GC akibat tekanan alokasi memori yang tinggi dari JSON.*stringify*().

PHP menunjukkan utilisasi CPU yang sangat rendah (0,1–0,2 vCPU) bukan karena efisien, melainkan karena sebagian besar waktu dihabiskan dalam keadaan menunggu respons database (I/O *wait*) CPU tidak dapat dimanfaatkan oleh *request* lain selama satu *worker process* menunggu MySQL *response* karena model eksekusi sinkronusnya.

**6. *Root Cause* Analysis: Mengapa Go Unggul?**

Berdasarkan data *benchmark* dan analisis karakteristik teknis, terdapat tiga faktor utama yang menjelaskan keunggulan Go pada skenario data-*heavy*:

a. *Streaming* JSON *Encoder*: *json.NewEncoder(w).Encode()* menulis data langsung ke HTTP *ResponseWriter* secara *streaming,* tanpa perlu mengalokasikan *buffer string* perantara di memori. Ini menghilangkan *double allocation* yang terjadi pada JSON.*stringify*() Bun atau *json encode*() PHP yang harus membangun *string* JSON lengkap di memori sebelum mengirimkannya;

b. Model *Goroutine* yang Ringan: Setiap *request* HTTP ditangani oleh *goroutine* dengan *overhead* memori hanya 2–8 KB. *Goroutine* yang sedang menunggu *respons* database di-*suspend* oleh Go *scheduler*, membebaskan CPU *thread* untuk mengeksekusi *goroutine* lain yang siap tanpa *blocking*. Ini memungkinkan ribuan *request concurrent* dikelola dengan *overhead* minimal;

c. *Garbage Collector* yang Prediktabel: GC Go menggunakan algoritma *tricolor concurrent mark-and-sweep* dengan GC *pause time* yang sangat rendah (biasanya < 1 ms). Dengan tekanan alokasi memori yang lebih rendah (karena *streaming encoder*), GC Go jarang harus bekerja keras, menjaga latensi *request* tetap stabil bahkan di bawah beban tinggi.

**7. *Root Cause* Analysis: Keterbatasan Bun dan PHP**

Bun mengalami performa di bawah Go terutama karena masalah memori akibat JSON.*stringify*(): *array* besar *JavaScript* yang dimuat dari MySQL *driver* mengonsumsi memori untuk representasi objek *JavaScript*-nya, kemudian JSON.*stringify*() mengalokasikan *string* baru yang berukuran setara atau lebih besar untuk representasi JSON-nya. Pada dataset 10.000 baris × 40 *field*, ini menghasilkan *peak memory* 425 MB lebih dari 5× *memory peak* Go. GC *JavaScript* (JSC) yang harus berjalan lebih sering untuk mengelola memori yang besar ini juga menyebabkan CPU lebih banyak terpakai untuk GC daripada untuk memproses *request* baru. Perlu dicatat pula bahwa mysql2 yang digunakan Bun adalah *Node.js compatibility* layer, bukan *native Bun driver*, sehingga ada *overhead* konversi tambahan.

PHP gagal di beban *concurrent* tinggi bukan karena lambat dalam memproses satu *request*, melainkan karena arsitektur PHP-FPM yang sinkronus membatasi jumlah *request* yang dapat dilayani secara bersamaan oleh jumlah *worker process* yang tersedia. Dengan 250 VU secara simultan dan setiap *request* membutuhkan beberapa detik untuk menyelesaikan *query* MySQL, *worker process* PHP habis dalam hitungan detik dan *request* baru harus mengantri hingga melebihi *timeout*. PDO

p-**ISSN: 2620 – 5327**
**e-ISSN: 2715 – 5501**

ATTR_PsERSISTENT tidak mampu mengatasi masalah ini karena *bottleneck* ada pada model *concurrency*, bukan pada *overhead* pembukaan koneksi database.

### 8. Perbandingan Lintas Skenario: Tren Performa

Tabel 6: Ringkasan Komparatif Lintas Skenario (Data *Riil Benchmark*)

| Metrik | Go | Bun | PHP | Catatan |
|---|---|---|---|---|
| ***Req*/s S1 (10k *rows*)** | **19,3** | **2,75** | **3,5** | *Go 7× > Bun, 5,5× > PHP* |
| ***Req*/s S2 (5k *rows*)** | **15,2** | **3,04** | **3,5** | *Go 5× > Bun, 4,3× > PHP* |
| ***Req*/s S3 (1k *rows*)** | **2,15** | **2,1** | **4,8** | *PHP tertinggi — VU naik, DB bottleneck dominan* |
| ***Memory* S1** | **77 MB** | **425 MB** | **74 MB** | *Bun 5,5× > Go; PHP efisien per-process* |
| ***Memory* S2** | **155 MB** | **392 MB** | **41 MB** | *Pola serupa, Bun tetap paling boros* |
| ***Memory* S3** | **45 MB** | **116 MB** | **14 MB** | *Semua turun proporsional dengan dataset* |
| **CPU *Usage*** | **0,5–0,9 Vcpu** | **0,4–0,8 vCPU** | **0,1–0,2 vCPU** | *PHP rendah = banyak waktu I/O wait* |
| **HTTP *Fail* S2** | **0%** | **1,35%** | **85,9%** | *Go zero error — sistem paling stabil* |

## B. PEMBAHASAN

### 1. Implikasi untuk Pengembang

Data *benchmark* ini memiliki implikasi praktis yang jelas untuk pengembang yang membangun API data-*heavy*. Go terbukti secara empiris sebagai pilihan optimal untuk *endpoint* yang mengembalikan data dalam jumlah besar dari database relasional, terutama ketika tingkat konkurensi tinggi. Keunggulan Go bukan hanya pada angka *throughput*, tetapi juga pada stabilitas dan prediktabilitas performa HTTP fail rate 0% pada Skenario 2 menunjukkan bahwa Go dapat diandalkan bahkan di bawah beban yang signifikan.

Bun menunjukkan potensi untuk API dengan *payload* lebih kecil (di bawah 5.000 baris), namun membutuhkan perhatian serius terhadap optimasi memori sebelum digunakan di produksi skala besar. Strategi seperti *streaming response* (menggunakan *Readable Stream* alih-alih JSON.*stringify* seluruh dataset sekaligus) dan implementasi *pagination* yang ketat dapat secara signifikan mengurangi tekanan memori Bun.

PHP tetap relevan dan bahkan kompetitif untuk *endpoint* dengan dataset kecil dan tingkat konkurensi moderat. Namun, untuk *endpoint* yang melayani data besar kepada banyak pengguna simultan benar, PHP-FPM murni bukan pilihan yang tepat. PHP + *Laravel Octane* dengan *Swoole*

(*persistent worker model*) atau migrasi *endpoint* berat ke Go secara bertahap merupakan strategi yang lebih pragmatis.

**2. Catatan Penting: *Bottleneck Database***

Hasil benchmark ini mengungkapkan fakta krusial yang sering diabaikan dalam diskusi pemilihan *runtime*: dalam kondisi pengujian ini, *bottleneck* terbesar bukan pada *runtime* itu sendiri, melainkan pada *query* database yang berjalan melalui koneksi internet dengan *latency* 20–100 ms. Seluruh *runtime* mengalami *timeout* dan *error rate* tinggi pada skenario 3 bukan karena kelemahan *runtime*, melainkan karena 300 VU membutuhkan 300 koneksi MySQL simultan melalui jaringan dengan *latency* tinggi.

Ini berarti: optimasi infrastruktur database (menempatkan database dalam jaringan privat yang sama dengan *application server*, implementasi *connection pooling* seperti: *PgBouncer/ProxySQL*, penggunaan *read replica*, penambahan *indexing* yang tepat) berpotensi memberikan *improvement* performa 3–10× pada semua *runtime* jauh lebih besar dari perbedaan antar *runtime* itu sendiri. Pilihan *runtime* adalah faktor penting, namun arsitektur database dan infrastruktur jaringan sama pentingnya dalam konteks data *heavy* API.

**3. Rekomendasi Pemilihan *Runtime***

Tabel 7: Rekomendasi Pemilihan *Runtime* Berdasarkan Data Empiris

| Kasus Penggunaan | Rekomendasi | Alasan Berdasarkan Data |
|---|---|---|
| API *endpoint* data *heavy* (> 5k *rows per request*) | Go | *Throughput* 7× lebih tinggi dari Bun, 5,5× dari PHP; *memory streaming* efisien; zero HTTP *fail rate* pada S2 |
| API data ringan (< 1k *rows*) dengan tim *JavaScript* | Bun | *Throughput* kompetitif dengan Go pada dataset kecil; *developer experience TypeScript* yang familiar |
| *Endpoint* sederhana, aplikasi monolitik | PHP (dengan *Octane+Swoole*) | Ekosistem Laravel yang matang; *Octane* meningkatkan *throughput* signifikan dibanding FPM murni |
| Migrasi bertahap dari PHP | Go (*endpoint* berat), PHP (*endpoint* ringan) | Strategi *hybrid*: Go untuk *endpoint* yang menjadi *bottleneck*, PHP untuk sisanya |
| *Prototyping* cepat dengan performa cukup | Bun (*Elysia*) | *Developer velocity* tinggi; performa jauh di atas *Node.js* untuk dataset sedang |
| Produksi dengan *budget* infrastruktur terbatas | Go | Konsumsi memori terendah; lebih banyak *request* per MB RAM dibanding *runtime* lain |

## SIMPULAN DAN SARAN

Penelitian ini telah menyajikan data *benchmark* empiris yang dihasilkan dari pengujian langsung Go, Bun, dan PHP menggunakan Grafana k6 pada *platform Railway.app* dengan skenario *query* MySQL yang merepresentasikan beban data nyata. Berdasarkan hasil pengujian dan analisis yang dilakukan, kesimpulan penelitian ini adalah sebagai berikut:

1. Go secara konsisten mengungguli Bun dan PHP dalam skenario data-*heavy*, mencatat *throughput* 19,3 req/s pada *query* 10.000 baris dengan 250 VU 7× lebih tinggi dari Bun (2,75 req/s) dan 5,5× dari PHP (3,5 req/s). *Streaming* JSON *encoder*, model *goroutine* ringan, dan GC yang prediktabel merupakan faktor teknis utama keunggulan Go;
2. Bun berada di posisi kedua dengan *throughput* 2,1–3,04 req/s namun mengalami konsumsi memori tertinggi (116–425 MB) akibat *double allocation* pada proses JSON.*stringify*(). Bun lebih sesuai untuk API dengan *payload* sedang hingga kecil dan membutuhkan optimasi memori sebelum digunakan pada skala besar;
3. PHP menunjukkan *throughput* terendah (3,5 req/s) dengan *error rate* sangat tinggi (81–99%) pada skenario beban *concurrent* tinggi, disebabkan oleh keterbatasan arsitektur PHP-FPM sinkronus. PHP tetap kompetitif pada dataset kecil (1.000 baris) dengan *throughput* 4,8 req/s mengalahkan Go dan Bun, namun dengan *error rate* yang masih sangat tinggi (99,7%);
4. *Bottleneck database* (MySQL melalui koneksi internet dengan *latency* 20–100 ms) terbukti menjadi faktor pembatas yang signifikan pada seluruh *runtime*, terutama pada skenario 300 VU. Optimasi infrastruktur database berpotensi memberikan peningkatan performa 3–10× yang melampaui perbedaan antar *runtime*; dan
5. Penelitian lanjutan disarankan untuk menguji runtime yang sama dengan konfigurasi database dalam jaringan *privat* (tanpa *latency* internet), implementasi *streaming response* pada Bun, PHP dengan *Octane*+*Swoole*, serta variasi *connection pool size* yang berbeda untuk mengukur dampak optimasi infrastruktur terhadap performa masing-masing *runtime*.

## DAFTAR PUSTAKA

Codereliant. (2023). Battle of the frameworks: Benchmarking high-performance HTTP libraries. Codereliant Substack. https://www.codereliant.io/p/battle-of-the-frameworks-benchmarking-high-performance-http-libraries

Codesoltech. (2024). PHP 8.x performance benchmarks: JIT, OPcache, and real-world speed gains explained. https://www.codesoltech.com/blog/php-8-x-performance-benchmarks/

Donovan, A. A. A., & Kernighan, B. W. (2016). The Go programming language. Addison-Wesley Professional.

Go Documentation. (2024). encoding/json — Package json. The Go Programming Language. https://pkg.go.dev/encoding/json

Grafana Labs. (2024). Grafana k6 documentation: Load testing for engineering teams. https://k6.io/docs/

Ismail, I., Nusri, A. Z., & Rahman, S. (2023). Sistem smart trash pemilah sampah organik dan anorganik berbasis Internet of Things. Jurnal Saintekom, 13(2), 193-201.

Kamarudin, Prasetyo, R. D., & Fitriani, A. R. (2024). Laporan benchmark Go vs Bun vs PHP — Perbandingan performa request data [Laporan teknis internal]. Universitas Handayani Makassar.

Lemire, D. (2023). Web server 'hello world' benchmark: Go vs Node.js vs Nim vs Bun. Daniel Lemire's Blog. https://lemire.me/blog/2023/10/07/web-server-hello-world-benchmark-go-vs-node-js-vs-nim-vs-bun/

Railway.app. (2024). Railway infrastructure documentation: Deployment, metrics, and scaling. https://docs.railway.app/

Sumner, J. (2023). Bun 1.0. Bun Blog. https://bun.sh/blog/bun-v1.0

Tatroe, K., MacIntyre, P., & Lerdorf, R. (2020). Programming PHP: Creating dynamic web pages (4th ed.). O'Reilly Media.

TechEmpower. (2024). TechEmpower web framework benchmarks — Round 23. TechEmpower Inc. https://www.techempower.com/benchmarks/#section=data-r23

W3Techs. (2024). Usage statistics of server-side programming languages for websites. https://w3techs.com/technologies/overview/programming_language

WWT. (2023). Performance benchmarking: Bun vs. C# vs. Go vs. Node.js vs. Python. World Wide Technology. https://www.wwt.com/blog/performance-benchmarking-bun-vs-c-vs-go-vs-nodejs-vs-python