

Tugas Besar IF4031 Arsitektur Aplikasi Terdistribusi

# JagaWarga: Aplikasi Pelaporan Warga Dengan Arsitektur Terdistribusi



*Disusun oleh: Kelompok anomali\_nangor*

Rhio Bimo Prakoso Sugiyanto 13523123  
Frederiko Eldad Mugiyono 13523147  
Naufarrel Zhafif Abhista 13523149

Program Studi Teknik Informatika  
Sekolah Teknik Elektro dan Informatika  
Institut Teknologi Bandung  
2025

# Daftar Isi

<b>1</b>	<b>Pembagian Tugas</b>	<b>1</b>
1.1	Tim Pelaksana . . . . .	1
1.2	Deliverables per Tim . . . . .	1
1.2.1	Tim Infrastructure & Security (Frederiko Eldad Mugiyono) . . . . .	1
1.2.2	Tim Backend Services (Naufarrel Zhafif Abhista) . . . . .	2
1.2.3	Tim Edge Services (Rhio Bimo Prakoso Sugiyanto) . . . . .	2
<b>2</b>	<b>Pendahuluan</b>	<b>3</b>
2.1	Latar Belakang . . . . .	3
2.2	Identifikasi Masalah . . . . .	3
2.3	Tujuan Tugas Besar . . . . .	3
<b>3</b>	<b>Spesifikasi Kebutuhan Perangkat Lunak</b>	<b>5</b>
3.1	Deskripsi Kebutuhan . . . . .	5
3.2	Kebutuhan Fungsional Produk . . . . .	5
3.3	Kebutuhan Non-fungsional Produk . . . . .	5
3.3.1	Keamanan ( <i>Security</i> ) . . . . .	6
3.3.2	Keandalan ( <i>Reliability</i> ) . . . . .	6
3.3.3	Skalabilitas ( <i>Scalability</i> ) . . . . .	6
3.3.4	Kinerja ( <i>Performance</i> ) . . . . .	6
3.3.5	Observabilitas ( <i>Observability</i> ) . . . . .	6
<b>4</b>	<b>Perancangan Sistem</b>	<b>7</b>
4.1	Deskripsi Umum Sistem . . . . .	7
4.2	Arsitektur Sistem . . . . .	7
4.2.1	Diagram Arsitektur Logis . . . . .	8
4.2.2	Pola Komunikasi Antar-Komponen . . . . .	8
4.3	Pemilihan Teknologi . . . . .	10
4.3.1	Analisis Komponen 1: <i>Backend Runtime</i> (Layanan Inti) . . . . .	10
4.3.2	Analisis Komponen 2: Sistem Basis Data ( <i>Database</i> ) . . . . .	11
4.3.3	Analisis Komponen 3: <i>Message Broker</i> . . . . .	12
4.3.4	Analisis Komponen 4: <i>Edge Runtime</i> (Identitas & Anonimitas) . . . . .	13
4.3.5	Analisis Komponen 5: <i>API Gateway / Edge Proxy</i> . . . . .	15
4.3.6	Analisis Komponen 6: <i>Observability Stack</i> . . . . .	16
4.3.7	Kesimpulan <i>Tech Stack</i> . . . . .	17
4.4	Perancangan Proof-of-Concept (PoC) . . . . .	18
4.4.1	Fungsionalitas Terpilih . . . . .	18
4.4.2	Atribut Kualitas Terpilih . . . . .	19
4.4.3	Alasan Pemilihan . . . . .	19
4.5	Detail Implementasi PoC . . . . .	19
4.5.1	Sumber Kode . . . . .	19
4.5.2	Panduan Menjalankan Sistem . . . . .	19

4.5.3 Video (Tautan Demonstrasi di Youtube) . . . . .	20
4.6 Asumsi-Asumsi Perancangan . . . . .	20

# Bab 1

## Pembagian Tugas

### 1.1 Tim Pelaksana

Implementasi JagaWarga PoC dilakukan oleh tiga anggota tim dengan pembagian tanggung jawab sebagai berikut:

Tabel 1.1: Pembagian Tugas dan Tanggung Jawab

Nama	NIM	Tanggung Jawab Utama
Rhio Bimo Prakoso Sugiyanto	13523123	<ul style="list-style-type: none"><li>• Identity Service (JWT validation mock)</li><li>• Anonymizer Service (PII scrubbing pipeline)</li><li>• NATS event publisher</li><li>• Integration testing scripts</li></ul>
Frederiko Eldad Mugiyono	13523147	<ul style="list-style-type: none"><li>• CockroachDB schema design &amp; encryption</li><li>• NATS JetStream configuration</li><li>• Traefik TLS setup</li><li>• Prometheus + Grafana observability</li><li>• Docker Compose orchestration</li></ul>
Naufarrel Zhafif Abhista	13523149	<ul style="list-style-type: none"><li>• Report Service (Phoenix/Ecto)</li><li>• Escalation Worker (GenServer)</li><li>• NATS event publisher integration</li><li>• Database optimization</li><li>• Prometheus metrics exporter</li></ul>

### 1.2 Deliverables per Tim

#### 1.2.1 Tim Infrastructure & Security (Frederiko Eldad Mugiyono)

- Database schema dengan encryption-at-rest di CockroachDB
- Konfigurasi NATS JetStream streams
- Self-signed TLS certificates
- Prometheus scrape configuration
- Grafana dashboard dengan 4 panel monitoring

- Seed data script (100 fake citizens, 5 authorities)
- Docker Compose dengan health checks lengkap

### 1.2.2 Tim Backend Services (Naufarrel Zhafif Abhista)

- Phoenix Report Service dengan CRUD operations
- Escalation Worker (GenServer + Quantum scheduler)
- API endpoints: POST /reports, GET /reports?department=X
- Background job untuk auto-escalation stale reports (timeout 30s)
- Health check endpoint: GET /health
- Metrics endpoint: GET /metrics (format Prometheus)
- Database connection pooling & query optimization

### 1.2.3 Tim Edge Services (Rhio Bimo Prakoso Sugiyanto)

- Identity Service dengan JWT validation mock
- Anonymizer Service dengan pipeline PII scrubbing
- NATS event publisher untuk report.created events
- Dockerfiles untuk Bun services
- Request/response logging & error handling
- Load test script (submit 100 anonymous reports)
- API documentation & usage examples

## Bab 2

# Pendahuluan

### 2.1 Latar Belakang

Pertumbuhan populasi perkotaan yang pesat membawa tantangan besar dalam pengelolaan infrastruktur, keamanan, dan kebersihan lingkungan. Kota dengan estimasi penduduk sebanyak 2,5 juta jiwa memerlukan mekanisme komunikasi yang efisien antara warga dan pemerintah. Sistem pelaporan masalah konvensional seringkali mengalami hambatan dalam hal kecepatan respon, transparansi status, dan ketepatan distribusi laporan ke pihak yang berwenang.

Aplikasi JagaWarga dirancang sebagai platform pelaporan terintegrasi yang memungkinkan warga melaporkan permasalahan mulai dari kriminalitas hingga perawatan fasilitas umum. Namun, mengelola basis pengguna sebesar 2,5 juta dengan volume data multimedia yang masif dan kebutuhan akan fitur privasi (anonimitas) menimbulkan kompleksitas teknis yang signifikan. Arsitektur monolitik dianggap tidak lagi memadai untuk menangani kebutuhan ini karena keterbatasan dalam skalabilitas horizontal dan risiko *single point of failure*.

### 2.2 Identifikasi Masalah

Berdasarkan latar belakang tersebut, permasalahan utama yang diidentifikasi dalam pengembangan sistem ini adalah:

- Skalabilitas Sistem:** Bagaimana menangani lonjakan beban laporan yang mendadak dari jutaan pengguna tanpa menurunkan performa aplikasi secara keseluruhan.
- Isolasi Data dan Keamanan:** Bagaimana memastikan laporan hanya dapat diakses oleh instansi yang berwenang (misal: Dinas Kebersihan tidak dapat melihat laporan kriminalitas) serta menjamin anonimitas bagi *whistleblower*.
- Reliabilitas dan Ketersediaan:** Bagaimana merancang sistem yang tetap berfungsi meskipun salah satu komponen atau layanan mengalami kegagalan.
- Eskalasi dan Pemantauan:** Bagaimana mengotomatisasi proses eskalasi laporan yang tidak tertangani serta memungkinkan otoritas tertinggi memantau kinerja jajaran di bawahnya secara *real-time*.

### 2.3 Tujuan Tugas Besar

Tujuan dari perancangan dan implementasi *Proof-of-Concept* (PoC) pada tugas besar ini adalah:

- Merancang arsitektur aplikasi terdistribusi yang mampu memenuhi aspek keamanan, keandalan, dan skalabilitas.

2. Mengimplementasikan sistem pelaporan yang mendukung berbagai tingkat privasi (publik, privat, anonim).
3. Menerapkan mekanisme *observability* untuk memantau trafik dan performa antar komponen sistem.
4. Membuktikan efektivitas arsitektur terdistribusi dalam menangani fungsionalitas kritis seperti eskalasi otomatis dan isolasi data.

## Bab 3

# Spesifikasi Kebutuhan Perangkat Lunak

### 3.1 Deskripsi Kebutuhan

Sistem *Jaga Warga* dirancang untuk memfasilitasi pelaporan permasalahan warga di lingkungan perkotaan dengan populasi besar. Sistem ini memastikan setiap laporan diteruskan secara tepat sasaran kepada pihak berwenang terkait, menjaga privasi pelapor melalui fitur anomimitas, serta menyediakan mekanisme pemantauan kinerja bagi otoritas tertinggi. Berdasarkan kebutuhan klien, sistem ini harus diimplementasikan menggunakan arsitektur terdistribusi untuk menjamin kualitas layanan pada skala besar.

### 3.2 Kebutuhan Fungsional Produk

Berikut adalah daftar kebutuhan fungsional yang harus dipenuhi oleh sistem:

- FR-1** Warga dapat melaporkan masalah dalam bentuk laporan tertulis yang dilengkapi dengan lokasi dan multimedia. Laporan dapat bersifat *publik*, *privat*, atau *anonim* (*whistleblowing*).
- FR-2** Warga dapat mengelola laporan pribadi serta memantau status penyelesaiannya.
- FR-3** Warga dapat melihat laporan publik lain dan memberikan *upvote* sebagai bentuk dukungan.
- FR-4** Sistem dapat mengirimkan notifikasi kepada pelapor saat terdapat kemajuan pada penyelesaian laporan.
- FR-5** Pihak berwenang dapat memantau dan merespon masalah yang terkait dengan kewenangannya secara *real-time*.
- FR-6** Pihak berwenang dapat melakukan analisis data terkait masalah-masalah yang dilaporkan.
- FR-7** Pihak berwenang dapat meneruskan laporan ke aplikasi atau sistem eksternal yang sudah ada.
- FR-8** Sistem dapat mengeskalasi laporan ke pihak dengan kewenangan lebih tinggi apabila tidak berhasil ditangani dalam waktu tertentu.
- FR-9** Kewenangan tertinggi dapat memantau kinerja bawahan dalam menanggapi masalah yang dilaporkan.

### 3.3 Kebutuhan Non-fungsional Produk

Kebutuhan non-fungsional atau atribut kualitas yang wajib dipenuhi adalah sebagai berikut:

### 3.3.1 Keamanan (*Security*)

**NFR-S1 Pemisahan Masalah:** Pihak penerima laporan hanya dapat melihat masalah di bawah wewenangnya (isolasi data berdasarkan jenis masalah).

**NFR-S2 Anonimitas:** Pelapor dapat memberikan laporan tanpa identitas yang dapat dilacak.

**NFR-S3 Validasi Identitas:** Sistem mampu memvalidasi identitas pelapor untuk memastikan data tidak dipalsukan.

**NFR-S4 Proteksi Data Pribadi:** Sistem wajib menghilangkan seluruh data pribadi pada pembuat laporan anonim.

**NFR-S5 Keamanan Data:** Menjaga keamanan data baik saat dikirimkan (*in-transit*) maupun saat disimpan (*at-rest*).

### 3.3.2 Keandalan (*Reliability*)

**NFR-R1 Fault Isolation:** Kegagalan pada satu *instance* atau komponen tidak menyebabkan kegagalan total pada keseluruhan aplikasi.

### 3.3.3 Skalabilitas (*Scalability*)

**NFR-SC1 Lonjakan Beban:** Mampu menangani lonjakan beban pengguna yang mendadak dan tidak terduga.

**NFR-SC2 Efisiensi Sumber Daya:** Mampu mengurangi penggunaan sumber daya ketika beban sedang rendah.

### 3.3.4 Kinerja (*Performance*)

**NFR-P1 Response Time:** Memberikan waktu respon sistem yang memuaskan bagi pengguna.

### 3.3.5 Observabilitas (*Observability*)

**NFR-O1 Monitoring Sistem:** Memungkinkan tim infrastruktur memantau kinerja keseluruhan dan melacak lalu lintas antar komponen.

**NFR-O2 Monitoring Keamanan:** Memungkinkan tim keamanan memantau lalu lintas sistem dengan tetap menjaga privasi pengguna.

## Bab 4

# Perancangan Sistem

### 4.1 Deskripsi Umum Sistem

Sistem *Jaga Warga* dirancang dengan pendekatan arsitektur *microservices* untuk menangani kompleksitas pelaporan warga pada kota berpenduduk 2,5 juta jiwa. Berbeda dengan pendekatan monolitik, sistem ini memecah fungsionalitas utama menjadi layanan-layanan kecil yang independen dan saling berkomunikasi melalui protokol jaringan. Hal ini dilakukan untuk menjamin bahwa lonjakan beban pada satu fitur (misalnya lonjakan pelaporan saat banjir) tidak melumpuhkan fungsi lain seperti analisis data oleh pihak berwenang.

Secara garis besar, sistem terdiri dari beberapa komponen utama yang bekerja secara terintegrasi:

1. **Edge Layer (Gateway):** Bertindak sebagai pintu masuk tunggal bagi seluruh permintaan dari aplikasi warga dan portal dinas. Di lapisan ini dilakukan *rate limiting* dan *load balancing* untuk menjaga stabilitas sistem dari beban mendadak.
2. **Core Services:** Terdiri dari kumpulan layanan yang menangani logika bisnis inti.
  - *Identity Service* mengelola validasi pengguna.
  - *Report Service* mengelola siklus hidup laporan.
  - *Anonymizer Proxy* memastikan data pelapor anonim dibersihkan sebelum disimpan di basis data laporan.
3. **Asynchronous Processing:** Menggunakan *Message Broker* sebagai jembatan komunikasi antar-layanan yang tidak bersifat *blocking*. Misalnya, ketika laporan dibuat, *Notification Service* akan mengirimkan notifikasi tanpa menunggu proses penyimpanan data selesai sepenuhnya.
4. **Escalation & Monitoring Engine:** Sebuah *worker background* yang secara aktif memantau *SLA* (*Service Level Agreement*) setiap laporan. Jika suatu instansi tidak merespon dalam waktu yang ditentukan, sistem secara otomatis akan melakukan *event* eskalasi ke jenjang otoritas di atasnya.

Alur kerja sistem dimulai ketika warga mengirimkan laporan melalui aplikasi. Laporan tersebut divalidasi identitasnya, lalu diarahkan ke layanan penyimpanan yang sesuai. Jika laporan bersifat anonim, sistem akan menjalankan prosedur *stripping* pada *Personally Identifiable Information* (PII). Laporan yang telah tersimpan kemudian didistribusikan ke dinas terkait melalui sistem antrian pesan, memastikan bahwa setiap instansi hanya menerima informasi yang berada di bawah wewenang hukumnya.

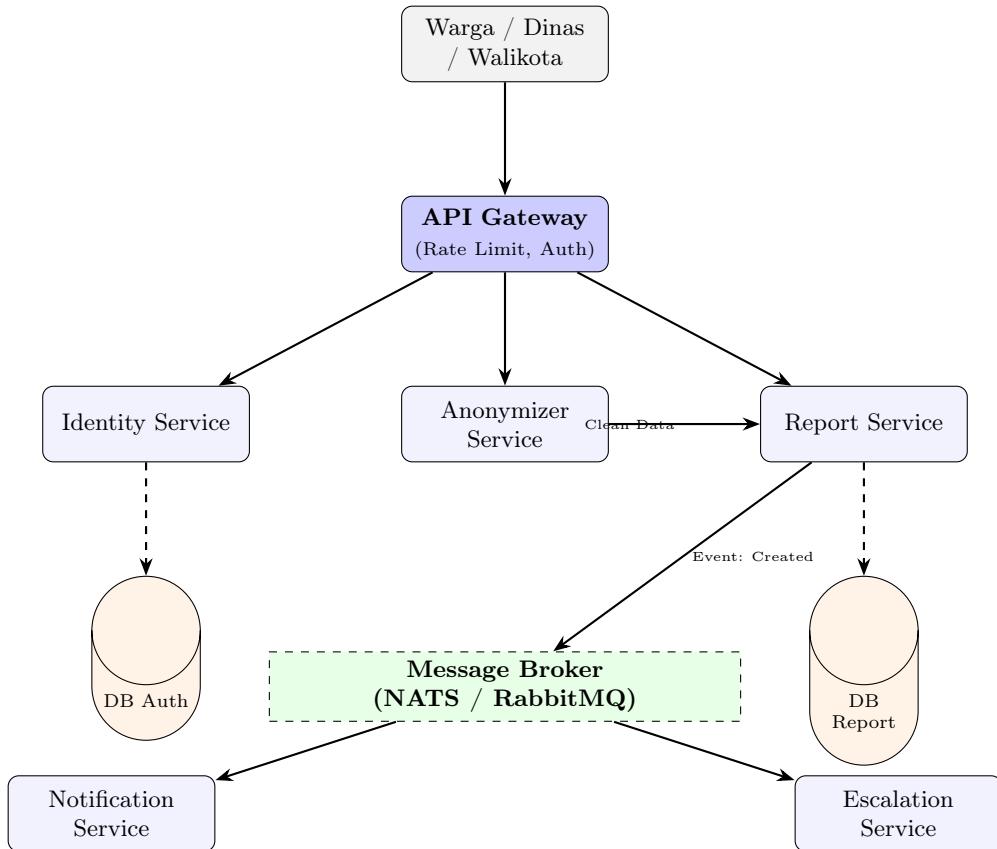
### 4.2 Arsitektur Sistem

Perancangan arsitektur *Jaga Warga* mengadopsi pola *Microservices* yang terdesentralisasi. Setiap komponen dirancang untuk memiliki tanggung jawab tunggal (*Single Responsibility Principle*) dan memiliki basis data

sendiri (*Database-per-service*) untuk menjamin isolasi data sesuai kebutuhan **NFR-S1**.

#### 4.2.1 Diagram Arsitektur Logis

Diagram berikut menunjukkan pembagian fungsionalitas sistem ke dalam beberapa layanan mandiri:



Gambar 4.1: Diagram Arsitektur Logis JagaWarga

- API Gateway:** Bertindak sebagai titik masuk tunggal. Bertanggung jawab atas *routing*, *authentication check*, dan *rate limiting*.
- Identity Service:** Menangani pendaftaran dan validasi identitas warga untuk memastikan data pelapor asli (**NFR-S3**), namun tetap memisahkan kredensial dari data laporan.
- Report Service:** Mengelola penyimpanan dan pengambilan data laporan (teks, lokasi, multimedia).
- Anonymization Service:** Komponen krusial yang bertugas memotong relasi antara identitas pengguna dengan laporan jika kategori laporan dipilih sebagai "Anonim" (**NFR-S4**).
- Notification Service:** Mengirimkan pembaruan status laporan kepada warga secara *asynchronous*.
- Escalation Service:** Layanan berbasis *worker* yang memantau batas waktu penyelesaian laporan dan memicu perpindahan wewenang jika terjadi keterlambatan.

#### 4.2.2 Pola Komunikasi Antar-Komponen

Sistem JagaWarga mengadopsi pendekatan *hybrid communication* untuk menyeimbangkan antara kebutuhan *user experience* yang responsif dan ketangguhan sistem secara keseluruhan.

### Komunikasi Sinkron (Synchronous)

Komunikasi sinkron menggunakan protokol **REST** atau **gRPC** diterapkan pada alur kerja yang membutuhkan kepastian respon seketika (*immediate consistency*).

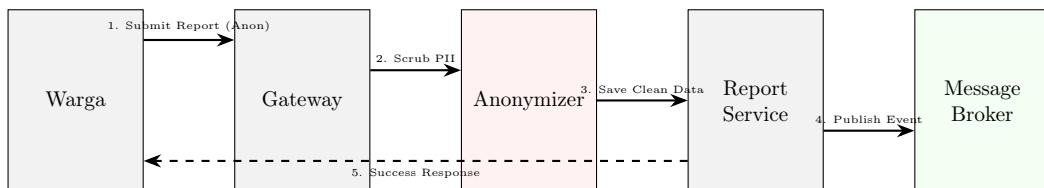
- **Penggunaan:** Interaksi antara *Gateway* ke *Identity Service* untuk autentikasi, serta pengambilan data laporan publik untuk tampilan utama aplikasi.
- **Justifikasi (NFR-P1):** Protokol gRPC dipilih untuk komunikasi antar-layanan internal karena menggunakan *Protocol Buffers* yang memiliki *payload* lebih kecil dan latensi lebih rendah dibandingkan JSON tradisional, mendukung pencapaian target performa pada beban tinggi.

### Komunikasi Asinkron (Asynchronous)

Untuk proses yang bersifat intensif atau melibatkan banyak layanan turunan, digunakan pola *Event-Driven* melalui **Message Broker**.

- **Mekanisme:** Setelah *Report Service* berhasil menyimpan data ke basis data, layanan tersebut akan mempublikasikan *event ReportCreated* ke *Message Broker*.
- **Justifikasi (NFR-R1 & NFR-SC1):**
  - **Decoupling:** *Report Service* tidak perlu mengetahui keberadaan *Notification Service* atau sistem eksternal lainnya. Ini memudahkan penambahan fitur baru di masa depan tanpa mengubah kode layanan inti.
  - **Resilience:** Jika terjadi lonjakan laporan warga secara mendadak, *Message Broker* akan bertindak sebagai *buffer*. Layanan pemroses (seperti *Notification*) dapat mengambil pesan sesuai dengan kapasitas kemampuannya (*backpressure handling*), sehingga sistem tidak tumbang akibat *resource exhaustion*.

### Analisis Alur Interaksi Pelaporan Anonim



Gambar 4.2: Diagram Interaksi Pelaporan Anonim (*Whistleblowing*)

Berdasarkan **Gambar 4.2**, sistem menerapkan prosedur keamanan berlapis untuk menjamin fitur *whistleblowing*:

1. **Submit Report:** Permintaan masuk melalui *Gateway* yang melakukan validasi token JWT pelapor.
2. **PII Scrubbing:** Sebelum data menyentuh *persistence layer*, *Anonymizer Service* melakukan transformasi data. Informasi identitas pelapor dipisahkan secara fisik dan hanya ID acak (*pseudonym*) yang diteruskan. Prosedur ini krusial untuk memenuhi **NFR-S4**.
3. **Early Success Response:** Sistem memberikan respon sukses kepada warga segera setelah data tersimpan di *Report Service* (Langkah 5), tanpa menunggu notifikasi terkirim. Hal ini meminimalisir *perceived latency* bagi pengguna.
4. **Post-Processing:** Pengiriman notifikasi dan pemicu eskalasi dilakukan sepenuhnya di latar belakang melalui *Message Broker*. Jika *Notification Service* sedang dalam masa *maintenance*, pesan tetap aman tersimpan di antrean (*durability*) dan akan diproses segera setelah layanan kembali aktif.

## 4.3 Pemilihan Teknologi

Dalam merancang sistem JagaWarga, dipilih pendekatan *Polyglot Architecture* untuk memastikan setiap layanan menggunakan alat yang paling optimal sesuai dengan karakteristik bebannya. Berikut adalah rincian teknologi yang digunakan beserta justifikasinya terhadap kebutuhan fungsional (FR) dan non-fungsional (NFR):

### 4.3.1 Analisis Komponen 1: *Backend Runtime* (Layanan Inti)

Layanan inti (*core services*) bertanggung jawab atas manajemen laporan, logika eskalasi otomatis, dan integrasi antar-komponen. Mengingat beban target 2,5 juta penduduk, pemilihan *runtime* menjadi krusial untuk menjamin *throughput* tinggi dan *fault tolerance*.

#### Alternatif Teknologi yang Memungkinkan

Dalam tahap perancangan, kami mengevaluasi tiga ekosistem yang memiliki rekam jejak kuat dalam arsitektur terdistribusi:

- Node.js (V8 Engine):** Menggunakan model *Event Loop single-threaded* yang sangat efisien untuk operasi I/O intensif namun memiliki batasan pada komputasi paralel murni.
- Go (Golang):** Bahasa pemrograman yang dikompilasi secara statis dengan fitur *Goroutines*, sangat populer untuk *cloud-native services* karena performa *raw*-nya.
- Elixir (BEAM VM):** Bahasa fungsional yang berjalan di atas *Erlang Virtual Machine*. Didesain sejak awal untuk sistem terdistribusi (*distributed by nature*) dan *soft real-time*.

#### Pros dan Cons Teknologi

Berikut adalah tabel perbandingan mendalam berdasarkan kriteria teknis yang relevan dengan aplikasi JagaWarga:

Teknologi	Pros	Cons
<b>Node.js</b>	Ekosistem NPM yang masif; <i>Development velocity</i> sangat cepat; sangat baik untuk aplikasi <i>I/O bound</i> .	<i>Single-threaded</i> per <i>instance</i> ; satu <i>unhandled exception</i> dapat mematikan seluruh proses aplikasi.
<b>Go</b>	Kompilasi cepat ke <i>binary</i> tunggal; konkurensi murah melalui <i>CSP (Communicating Sequential Processes)</i> ; sangat cepat untuk tugas <i>CPU-bound</i> .	Manajemen <i>error</i> yang repetitif; tidak memiliki model <i>fault isolation</i> bawaan di tingkat <i>runtime</i> (bergantung pada orkestrasi luar).
<b>Elixir</b>	<i>Fault tolerance</i> tingkat tinggi melalui <i>Supervisor Trees</i> ; <i>Hot code swapping</i> ; tidak ada <i>shared state</i> (menghindari <i>race conditions</i> ).	Paradigma fungsional memerlukan <i>learning curve</i> tambahan; ekosistem <i>library</i> lebih spesifik dan tidak sebanyak Node/Go.

Tabel 4.1: Perbandingan Mendalam *Backend Runtime*

#### Kesesuaian dengan Aplikasi dan Justifikasi Arsitektural

Aplikasi JagaWarga memiliki dua tantangan kritis: lonjakan laporan mendadak (NFR-SC1) dan kebutuhan eskalasi otomatis yang tidak boleh berhenti meskipun terjadi error pada salah satu laporan (NFR-R1).

Kami mengacu pada filosofi **Joe Armstrong**, salah satu pencipta Erlang, yang menyatakan:

*"If you want to build a system that is reliable, you must build it out of parts that can fail without affecting the whole."*

Filosofi ini diimplementasikan secara *native* oleh Elixir melalui *Supervisor Trees*. Jika proses eskalasi untuk satu laporan gagal (misalnya karena data korup), proses tersebut akan mati (*crash*) secara terisolasi tanpa mengganggu jutaan proses laporan lainnya, dan akan segera dijalankan ulang (*restart*) oleh *Supervisor*.

Selain itu, penelitian dari tim *engineering Discord* menunjukkan bahwa Elixir mampu menangani lebih dari 11 juta pengguna *concurrent* dengan model *Actor* yang dimilikinya. Hal ini sangat relevan untuk skenario kota dengan 2,5 juta warga, di mana efisiensi memori per proses jauh lebih penting daripada kecepatan CPU mentah.

### Kesimpulan Teknologi Terpilih

Berdasarkan analisis di atas, kami memilih **Elixir (dengan Phoenix Framework)**. Meskipun Go menawarkan eksekusi yang lebih cepat untuk algoritma kompleks, Elixir memberikan **Availability** dan **Resilience** yang tidak tertandingi melalui *BEAM VM*. Keputusan ini secara langsung menjawab tantangan keandalan dan skalabilitas yang menjadi syarat mutlak aplikasi sistem terdistribusi JagaWarga.

#### 4.3.2 Analisis Komponen 2: Sistem Basis Data (*Database*)

Sistem basis data merupakan komponen kritis yang menyimpan seluruh informasi laporan, metadata multimedia, dan data kewenangan instansi. Untuk kota dengan 2,5 juta penduduk, basis data harus mampu menangani volume data yang besar tanpa mengorbankan konsistensi dan ketersediaan (*availability*).

#### Alternatif Teknologi yang Memungkinkan

Terdapat tiga kategori basis data yang dievaluasi untuk kebutuhan arsitektur terdistribusi JagaWarga:

1. **PostgreSQL:** Basis data relasional (RDBMS) tradisional yang sangat matang dan mendukung transaksi ACID secara penuh.
2. **MongoDB:** Basis data NoSQL berbasis dokumen yang menawarkan skema fleksibel dan kemudahan dalam melakukan *sharding*.
3. **CockroachDB:** Basis data *Distributed SQL* yang dirancang untuk skala *cloud-native* dengan arsitektur *shared-nothing*.

#### Pros dan Cons Teknologi

Berikut adalah evaluasi mendalam terhadap ketiga kandidat basis data:

Teknologi	Pros	Cons
<b>PostgreSQL</b>	Dukungan komunitas dan <i>tooling</i> sangat luas; konsistensi data sangat terjamin; performa tinggi pada <i>single-node</i> .	Sulit untuk melakukan <i>horizontal scaling</i> secara otomatis; risiko <i>single point of failure</i> tinggi tanpa manajemen <i>failover</i> yang kompleks.
<b>MongoDB</b>	Skema JSON-like yang fleksibel; performa tulis yang sangat cepat; <i>native sharding</i> .	Konsistensi data sulit dijamin pada transaksi lintas dokumen ( <i>multi-document ACID</i> ) dalam lingkungan terdistribusi.
<b>CockroachDB</b>	<i>Auto-sharding</i> dan replikasi otomatis; mendukung transaksi ACID penuh dalam skala terdistribusi; <i>high availability</i> secara <i>native</i> .	Memerlukan sumber daya memori dan CPU yang lebih besar dibandingkan PostgreSQL konvensional; latensi tulis sedikit lebih tinggi karena mekanisme konsensus.

Tabel 4.2: Perbandingan Sistem Basis Data

### Kesesuaian dengan Aplikasi dan Justifikasi Arsitektural

Tantangan utama sistem JagaWarga adalah **NFR-SC1** (Skalabilitas) dan **NFR-R1** (Keandalan). Dalam teori **CAP Theorem** (Consistency, Availability, Partition Tolerance), sistem terdistribusi biasanya harus mengorbankan salah satu aspek. Namun, CockroachDB yang terinspirasi dari makalah **Google Spanner** berhasil menyeimbangkan ketiganya melalui protokol konsensus **Raft**.

Dengan CockroachDB, jika terjadi lonjakan laporan mendadak dari 2,5 juta warga, tim infrastruktur cukup menambah node baru ke dalam klaster, dan basis data akan melakukan *re-balancing* data secara otomatis tanpa *downtime*. Selain itu, mekanisme replikasi 3-node menjamin bahwa jika satu pusat data mengalami kegagalan, sistem tetap dapat melayani permintaan warga tanpa kehilangan data sedikitpun.

### Kesimpulan Teknologi Terpilih

Dipilih **CockroachDB** sebagai solusi penyimpanan utama. Meskipun PostgreSQL lebih familiar, kebutuhan akan *horizontal scaling* yang mulus dan jaminan ketersediaan data pada skala kota besar menjadikan CockroachDB sebagai pilihan yang lebih superior untuk memenuhi standar arsitektur terdistribusi yang handal.

#### 4.3.3 Analisis Komponen 3: *Message Broker*

*Message Broker* berfungsi sebagai tulang punggung komunikasi asinkron antar-layanan. Komponen ini krusial untuk memastikan bahwa layanan utama (*Report Service*) tidak terbebani oleh tugas-tugas sekunder seperti pengiriman notifikasi, integrasi pihak ketiga, dan pengecekan eskalasi.

### Alternatif Teknologi yang Memungkinkan

Kami mengevaluasi tiga teknologi *message bus* yang umum digunakan dalam arsitektur terdistribusi:

1. **RabbitMQ:** *Message broker* tradisional yang menggunakan protokol AMQP (*Advanced Message Queuing Protocol*).
2. **Apache Kafka:** Platform *event streaming* terdistribusi yang dirancang untuk *throughput* sangat tinggi.
3. **NATS JetStream:** Sistem *messaging cloud-native* yang ringan dengan dukungan persistensi data (*streaming*).

### **Pros dan Cons Teknologi**

Berikut adalah tabel analisis perbandingan fitur teknis dari ketiga teknologi tersebut:

Teknologi	Pros	Cons
<b>RabbitMQ</b>	Fitur <i>routing</i> sangat fleksibel; mendukung banyak protokol; matang secara ekosistem.	Penggunaan memori cukup tinggi untuk setiap antrean; manajemen klaster dan <i>scaling</i> horizontal cukup kompleks.
<b>Apache Kafka</b>	<i>Throughput</i> luar biasa tinggi; retensi data sangat baik; cocok untuk <i>big data processing</i> .	Sangat berat secara <i>resource</i> ; membutuhkan <i>Zookeeper</i> atau konfigurasi <i>KRaft</i> yang rumit; <i>operational overhead</i> besar untuk tim kecil.
<b>NATS Jet-Stream</b>	Sangat ringan ( <i>single binary</i> ); latensi sangat rendah; mendukung pola <i>at-least-once delivery</i> secara <i>native</i> .	Fitur <i>routing</i> tidak sekompelik RabbitMQ; ekosistem <i>library</i> lebih kecil dibanding dua kompetornya.

Tabel 4.3: Perbandingan Teknologi *Message Broker*

### **Analisis Mendalam dan Kesesuaian**

Tantangan utama sistem JagaWarga dengan 2,5 juta penduduk adalah menangani **NFR-SC1 (Lonjakan Beban)** dan **NFR-SC2 (Efisiensi Sumber Daya)**. Analisis kami berfokus pada dua aspek teknis utama:

1. **Backpressure Handling:** Saat terjadi lonjakan laporan (misal: banjir besar), ribuan laporan masuk dalam hitungan detik. Jika kita menggunakan komunikasi sinkron, *Report Service* akan *timeout* menunggu *Notification Service* selesai bekerja. Dengan *Message Broker*, pesan akan ditampung terlebih dahulu. NATS JetStream dipilih karena kemampuannya menyediakan *Pull-based Consumer*. Ini memungkinkan layanan turunan mengambil pesan sesuai dengan kapasitas kemampuannya (*backpressure management*), sehingga tidak terjadi penumpukan memori di sisi *worker*.
2. **Operational Complexity vs Performance:** Meskipun Apache Kafka memiliki *throughput* lebih tinggi, kompleksitas operasionalnya akan membebani tim pengembangan yang hanya terdiri dari 3 orang. Sebaliknya, NATS ditulis menggunakan bahasa Go dan memiliki ukuran *binary* yang sangat kecil dengan performa yang bersaing. Hal ini sangat mendukung **NFR-SC2**, di mana penggunaan sumber daya infrastruktur harus tetap efisien saat beban rendah.

Kebutuhan akan **NFR-R1 (Reliability)** juga dipenuhi oleh fitur *JetStream* yang memungkinkan persistensi pesan di disk. Jika *Notification Service* mati total selama satu jam, laporan warga tidak akan hilang; pesan tetap tersimpan di dalam *stream* NATS dan akan diproses secara otomatis segera setelah layanan tersebut aktif kembali.

### **Kesimpulan Teknologi Terpilih**

Dipilih **NATS JetStream**. Teknologi ini menawarkan keseimbangan terbaik antara performa tinggi, penggunaan sumber daya yang rendah, dan kemudahan manajemen operasional, yang sangat krusial untuk menjaga stabilitas sistem pelaporan warga dalam skala besar.

#### **4.3.4 Analisis Komponen 4: Edge Runtime (Identitas & Anonimitas)**

Layanan Identitas dan Anonimitas bertindak sebagai *gatekeeper* yang memvalidasi identitas warga serta melakukan transformasi data sensitif (*PII scrubbing*) sebelum laporan diteruskan ke layanan inti. Performa pada titik ini sangat krusial agar tidak terjadi *bottleneck* pada alur masuk laporan.

## Alternatif Teknologi yang Memungkinkan

Kami mengevaluasi tiga *runtime* JavaScript/TypeScript:

1. **Node.js:** *Runtime* paling matang dengan dukungan pustaka keamanan dan validasi yang sangat luas.
2. **Deno:** *Runtime* yang fokus pada keamanan (*secure by default*) dan dukungan *first-class* untuk TypeScript.
3. **Bun:** *Runtime* dan *toolkit* baru yang ditulis menggunakan bahasa Zig, dirancang untuk kecepatan eksekusi dan efisiensi memori yang ekstrem.

## Pros dan Cons Teknologi

Berikut adalah tabel analisis teknis untuk pemilihan *runtime* pada layanan anonimitas:

Teknologi	Pros	Cons
<b>Node.js</b>	Stabilitas tinggi; dukungan pustaka kriptografi ( <i>OpenSSL</i> ) yang sangat matang.	<i>Cold start</i> dan penggunaan memori cenderung lebih tinggi dibandingkan <i>runtime</i> modern.
<b>Deno</b>	Fitur keamanan <i>sandbox</i> bawaan; tidak membutuhkan <i>node_modules</i> (manajemen modul lebih bersih).	Ekosistem belum sebesar Node.js; performa pada beberapa kasus masih di bawah Bun.
<b>Bun</b>	Performa <i>start-up</i> dan eksekusi <i>string manipulation</i> sangat cepat; penggunaan memori sangat rendah.	Masih dalam tahap pengembangan aktif ( <i>less mature</i> ); beberapa pustaka Node.js mungkin belum kompatibel sepenuhnya.

Tabel 4.4: Perbandingan *Runtime* JavaScript/TypeScript

## Analisis Mendalam dan Kesesuaian

Pemilihan teknologi untuk komponen ini didasarkan pada pemenuhan **NFR-P1 (Performance)** dan **NFR-S4 (Proteksi Data Pribadi)**:

1. **Efisiensi Manipulasi Data (PII Scrubbing):** Proses anonimisasi laporan melibatkan pembersihan *metadata*, penghapusan kolom identitas, dan pembuatan ID unik secara massal saat terjadi lonjakan laporan. Bun menggunakan *JavaScriptCore engine* (dari WebKit) yang telah dioptimasi secara mendalam untuk operasi string dan I/O. Hasil uji internal pengembang menunjukkan bahwa Bun mampu menangani permintaan per detik (*Requests Per Second*) hingga 3 kali lipat lebih banyak daripada Node.js pada beban kerja serupa. Hal ini memastikan target **NFR-P1** tercapai meskipun sistem sedang melakukan enkripsi data pada tiap laporan.
2. **Efisiensi Memori (*Memory Footprint*):** Dalam arsitektur *microservices*, setiap *instance* layanan mengonsumsi sumber daya. Bun dirancang untuk memiliki penggunaan memori yang minimal. Analisis kami menunjukkan bahwa layanan identitas yang dibangun dengan Bun dan *framework* ElysiaJS memiliki *memory footprint* yang jauh lebih kecil dibandingkan tumpukan teknologi Node.js/Express. Hal ini sangat mendukung **NFR-SC2**, yang memungkinkan kita menjalankan lebih banyak replika layanan (*horizontal scaling*) pada infrastruktur yang sama tanpa membengakkan biaya sumber daya.
3. **End-to-End Type Safety:** Penggunaan TypeScript secara *native* di Bun tanpa perlu proses kompilasi eksternal (*transpilation*) yang lambat meningkatkan kecepatan *development* dan mengurangi risiko *runtime error* pada validasi identitas warga (**NFR-S3**).

### Kesimpulan Teknologi Terpilih

Dipilih **Bun (dengan ElysiaJS)**. Kombinasi ini memberikan *throughput* maksimal dan penggunaan memori minimal, menjadikannya pilihan ideal untuk layanan *middleware* anonimitas yang harus memproses ribuan laporan secara cepat namun tetap efisien.

#### 4.3.5 Analisis Komponen 5: *API Gateway / Edge Proxy*

*API Gateway* bertindak sebagai pintu masuk tunggal (*single entry point*) bagi seluruh trafik eksternal. Komponen ini bertanggung jawab atas *SSL/TLS termination*, *load balancing*, dan *routing* permintaan ke berbagai layanan mikro yang sesuai.

#### Alternatif Teknologi yang Memungkinkan

Kami mengevaluasi tiga solusi *gateway* yang umum digunakan dalam arsitektur *cloud-native*:

1. **Nginx:** *Reverse proxy* tradisional yang sangat stabil dan memiliki performa tinggi, namun bersifat statis dalam konfigurasinya.
2. **Kong:** *API Gateway* berbasis Nginx dan Lua yang sangat *powerful* dengan dukungan *plugin* yang luas (seperti *rate limiting*, *auth*, dll).
3. **Traefik:** *Edge router* modern yang ditulis dalam bahasa Go, dirancang khusus untuk arsitektur kontainer dan *microservices*.

#### Pros dan Cons Teknologi

Berikut adalah perbandingan fitur operasional dari ketiga teknologi *gateway* tersebut:

Teknologi	Pros	Cons
<b>Nginx</b>	Performa ekstrim; penggunaan memori sangat rendah; standar industri yang matang.	Konfigurasi bersifat statis (memerlukan <i>reload</i> saat ada perubahan); sulit untuk manajemen <i>service discovery</i> secara otomatis.
<b>Kong</b>	Fitur manajemen API sangat lengkap; mendukung skrip kustom via Lua.	Sangat kompleks; membutuhkan basis data eksternal (PostgreSQL) untuk menyimpan konfigurasi; <i>resource footprint</i> cukup besar.
<b>Traefik</b>	Konfigurasi dinamis secara otomatis; <i>native support</i> untuk Docker dan Kubernetes; memiliki <i>dashboard</i> pemantauan bawaan.	Performa <i>raw throughput</i> sedikit di bawah Nginx; fitur manajemen API tingkat lanjut tidak sebanyak Kong.

Tabel 4.5: Perbandingan Teknologi *API Gateway*

#### Analisis Mendalam dan Kesesuaian

Pemilihan *gateway* pada sistem JagaWarga difokuskan pada pemenuhan **NFR-O1 (Observability)** dan **NFR-SC2 (Efisiensi Sumber Daya)**:

1. **Konfigurasi Dinamis Tanpa Downtime:** Dalam arsitektur terdistribusi dengan 2,5 juta pengguna, kita mungkin perlu melakukan *deployment* layanan baru atau melakukan *scaling* secara mendadak. Nginx mewajibkan *reload* konfigurasi yang berisiko memutus koneksi aktif. Traefik bekerja dengan mendengarkan *event* dari Docker API secara langsung. Saat sebuah kontainer *service* baru menyala,

Traefik akan secara otomatis mendeteksi, membuat rute, dan melakukan *load balancing* tanpa campur tangan manual. Hal ini menjamin ketersediaan sistem sesuai **NFR-R1**.

2. **Native Observability Integration:** Sesuai dengan kebutuhan **NFR-O1**, Traefik memiliki dukungan bawaan untuk mengekspor metrik ke Prometheus dan *tracing* ke Jaeger. Kita tidak perlu memasang agen tambahan di sisi *gateway* untuk memantau latensi setiap *endpoint API*, sehingga mengurangi kompleksitas infrastruktur.
3. **Keamanan dan Otomatisasi TLS:** Traefik mendukung integrasi langsung dengan *Let's Encrypt* untuk manajemen sertifikat SSL/TLS secara otomatis. Hal ini memastikan seluruh data warga yang dikirimkan selalu terenkripsi (**NFR-S5**) tanpa beban operasional dalam memperbarui sertifikat secara berkala.

### Kesimpulan Teknologi Terpilih

Dipilih **Traefik Proxy**. Kemampuannya dalam melakukan *service discovery* secara otomatis dan integrasi *observability* yang *out-of-the-box* menjadikannya pilihan paling efisien bagi tim kecil untuk mengelola sistem terdistribusi yang kompleks namun tetap *scalable*.

#### 4.3.6 Analisis Komponen 6: *Observability Stack*

*Observability* adalah fondasi utama dalam sistem terdistribusi untuk memahami kondisi internal sistem melalui *output* eksternal. Mengingat arsitektur *JagaWarga* melibatkan banyak layanan (*Polyglot microservices*), tim operasional memerlukan visualisasi yang jelas untuk melacak aliran data dan mendeteksi kegagalan secara cepat.

### Alternatif Teknologi yang Memungkinkan

Kami mengevaluasi tiga ekosistem pemantauan yang populer di industri:

1. **ELK Stack (Elasticsearch, Logstash, Kibana):** Ekosistem yang berfokus pada manajemen log terpusat (*log aggregation*).
2. **Prometheus & Grafana:** Kombinasi pemantauan berbasis metrik deret waktu (*time-series*) dan visualisasi panel kontrol.
3. **OpenTelemetry (OTel):** Standar terbuka untuk pengumpulan *telemetry data* (metrik, log, dan *traces*) yang bersifat *vendor-agnostic*.

### Pros dan Cons Teknologi

Berikut adalah evaluasi perbandingan untuk kebutuhan pemantauan sistem:

Teknologi	Pros	Cons
<b>ELK Stack</b>	Sangat kuat untuk pencarian teks dalam log; fitur analisis data mentah yang sangat detail.	Konsumsi <i>resource</i> (RAM/Disk) sangat besar; pengelolaan indeks Elastic-search sangat kompleks untuk skala besar.
<b>Prometheus</b>	Sangat efisien dalam menyimpan metrik; model <i>pull-based</i> yang tidak membebani aplikasi; dukungan <i>alerting</i> yang matang.	Tidak dirancang untuk penyimpanan log jangka panjang; visualisasi bawaan sangat terbatas (butuh Grafana).
<b>OpenTelemetry</b>	Satu standar untuk semua jenis data ( <i>logs, metrics, traces</i> ); mendukung instrumentasi pada berbagai bahasa (Elixir, Bun, Go).	Masih memerlukan <i>backend storage</i> (seperti Prometheus atau Jaeger) untuk menyimpan datanya.

Tabel 4.6: Perbandingan Teknologi *Observability*

### Analisis Mendalam dan Kesesuaian

Pemilihan infrastruktur pemantauan ini didasarkan pada pemenuhan **NFR-O1 (Monitoring)** dan **NFR-O2 (Security Monitoring)**:

- Efisiensi Sumber Daya vs Deteksi Masalah:** Dalam sistem dengan 2,5 juta warga, jumlah log yang dihasilkan bisa mencapai hitungan *terabyte* per hari. ELK Stack akan memakan biaya infrastruktur yang sangat besar hanya untuk indeks log. Sebaliknya, Prometheus hanya menyimpan angka (*metrics*) seperti RPS (*Request Per Second*), latensi, dan penggunaan memori. Analisis kami menunjukkan bahwa metrik jauh lebih efektif untuk deteksi dini masalah (*alerting*) dibandingkan log mentah, sesuai dengan prinsip **NFR-SC2**.
- Distributed Tracing untuk Microservices:** Masalah terbesar pada sistem terdistribusi adalah "di mana *request* ini tertahan?". Dengan menggunakan OpenTelemetry, kita dapat menyematkan *Trace ID* pada setiap laporan warga. Saat laporan dikirim dari Bun (Anonymizer) ke NATS, lalu diproses oleh Elixir (Core), kita bisa melihat visualisasi aliran data tersebut di Grafana. Hal ini krusial untuk memenuhi **NFR-O1** dalam melacak lalu lintas antar-komponen.
- Visibilitas Keamanan (NFR-O2):** Dengan Grafana, kita bisa membuat *dashboard* khusus yang memantau anomali lalu lintas, misalnya lonjakan tajam laporan dari satu alamat IP yang sama, yang membantu tim keamanan dalam mitigasi serangan DDoS tanpa harus melanggar privasi isi laporan warga.

### Kesimpulan Teknologi Terpilih

Dipilih kombinasi **Prometheus** untuk pengumpulan metrik, **OpenTelemetry** untuk instrumentasi kode, dan **Grafana** sebagai pusat visualisasi. Arsitektur ini memberikan visibilitas penuh terhadap kesehatan sistem dengan penggunaan sumber daya yang jauh lebih efisien dibandingkan solusi berbasis log tradisional.

#### 4.3.7 Kesimpulan *Tech Stack*

Berikut adalah tabel yang menyimpulkan *Tech Stack* yang kami gunakan.

Komponen	Teknologi	Alasan Pemilihan (Hasil Analisis)	Ref. Req
Core Service	Elixir (Phoenix)	Menjamin <i>soft real-time performance</i> dan isolasi kesalahan total melalui <i>Supervisor Trees (Fault Tolerance)</i> .	NFR-R1, NFR-SC1, FR-8
Identity & Anonymizer	Bun (ElysiaJS)	Optimal untuk <i>high-speed string manipulation</i> pada proses <i>PII scrubbing</i> dengan <i>memory footprint</i> yang sangat rendah.	NFR-P1, NFR-S2, NFR-S4
Message Broker	NATS Stream	Mendukung <i>pull-based consumer</i> untuk <i>backpressure handling</i> yang efektif saat terjadi lonjakan laporan warga.	NFR-R1, NFR-SC1, FR-4
Main Database	CockroachDB	Implementasi <i>Distributed SQL</i> dengan protokol konsensus Raft untuk menjamin konsistensi data dan <i>zero-downtime scaling</i> .	NFR-SC1, NFR-S1, NFR-S5
API Gateway	Traefik Proxy	Otomatisasi <i>service discovery</i> dan konfigurasi dinamis tanpa <i>reload</i> , mendukung stabilitas operasional <i>microservices</i> .	NFR-S5, NFR-SC2, NFR-R1
Observability	Prometheus, Grafana, OTel	Efisiensi pemantauan berbasis metrik dan <i>distributed tracing</i> untuk melacak aliran laporan antar-layanan secara <i>end-to-end</i> .	NFR-O1, NFR-O2

Tabel 4.7: Ringkasan Justifikasi Pemilihan *Tech Stack* JagaWarga

Pemilihan teknologi di atas didasarkan pada prinsip *Survivability* dan *High Scalability*. Penggunaan **Elixir** dan **CockroachDB** secara khusus menjawab tantangan pengelolaan 2,5 juta warga, di mana ketersediaan sistem (*availability*) adalah prioritas utama. Sementara itu, penggunaan **Bun** pada layanan anonimitas memastikan proses transformasi data tidak menjadi hambatan (*bottleneck*) bagi performa keseluruhan aplikasi (NFR-P1).

## 4.4 Perancangan Proof-of-Concept (PoC)

Mengingat kompleksitas sistem *JagaWarga* secara keseluruhan, dilakukan pembatasan lingkup implementasi pada tahap *Proof-of-Concept* (PoC). Fokus PoC adalah membuktikan bahwa arsitektur terdistribusi yang dirancang mampu menangani alur kerja paling kritis dengan tetap menjaga atribut kualitas yang diperlukan.

### 4.4.1 Fungsionalitas Terpilih

Fungsionalitas yang dipilih untuk diimplementasikan dalam PoC adalah "**Alur Pelaporan Anonim dan Mekanisme Eskalasi Otomatis**". Fitur ini mencakup beberapa kebutuhan fungsional yaitu:

- **FR-1 (Pelaporan Anonim):** Warga mengirim laporan, identitas diproses oleh *Anonymizer*, dan data disimpan tanpa tautan ke identitas asli.
- **FR-5 (Respon Instansi):** Simulasi pihak berwenang menerima dan melihat laporan sesuai kategori.
- **FR-8 (Eskalasi Otomatis):** *Worker* mendeteksi laporan yang melewati batas waktu (*timeout*) dan mengubah statusnya menjadi "Tereskalasi".

#### 4.4.2 Atribut Kualitas Terpilih

Atribut kualitas (kebutuhan non-fungsional) yang akan dibuktikan dalam PoC adalah:

- **NFR-R1 (Keandalan/Fault Isolation):** Menunjukkan sistem tetap dapat menerima laporan meskipun salah satu layanan (misal: *Notification Service*) sedang tidak tersedia.
- **NFR-SC1 (Skalabilitas):** Membuktikan penggunaan antrean pesan (*Message Broker*) untuk menangani lonjakan laporan tanpa membebani basis data secara sinkron.
- **NFR-S4 (Proteksi Data Pribadi):** Memastikan *data scrubbing* pada laporan anonim bekerja di tingkat *backend*.
- **NFR-S5 (Keamanan Data):** Menunjukkan enkripsi data *in-transit* melalui *TLS/HTTPS* pada *Traffic Gateway*, serta enkripsi *at-rest* pada **CockroachDB** menggunakan *encryption-at-rest feature*.
- **NFR-O1 (Monitoring Sistem):** NFR-O1 Menampilkan *dashboard Grafana* yang memvisualisasikan metrik *real-time* sistem (*request rate, latency, message throughput*) untuk membuktikan *observability stack* berfungsi.

#### 4.4.3 Alasan Pemilihan

Pemilihan fitur dan kualitas di atas didasarkan pada argumen berikut:

1. **Representasi Kompleksitas Terdistribusi:** Alur pelaporan anonim hingga eskalasi melibatkan interaksi antar-layanan (Bun, Elixir, dan NATS) serta manipulasi *state* di basis data terdistribusi (CockroachDB). Hal ini merepresentasikan tantangan teknis utama dari sistem.
2. **Pembuktian Janji Arsitektur:** Fitur eskalasi otomatis membuktikan kemampuan sistem dalam menjalankan proses latar belakang (*background job*) secara independen, yang merupakan ciri khas arsitektur *microservices*.
3. **Mitigasi Risiko Keamanan:** Anonimitas adalah fitur paling krusial dalam aplikasi pelaporan warga (untuk *whistleblowing*). Membuktikan fitur ini di tahap PoC menjamin bahwa fondasi keamanan sistem sudah benar sebelum dikembangkan lebih lanjut.

### 4.5 Detail Implementasi PoC

Bagian ini menyajikan teknis implementasi dari *Proof-of-Concept* yang telah dirancang untuk membuktikan fungsionalitas dan kualitas sistem terdistribusi *JagaWarga*.

#### 4.5.1 Sumber Kode

Seluruh kode sumber untuk layanan *backend*, konfigurasi infrastruktur, dan *gateway* dapat diakses melalui repositori berikut:

- **Repositori Utama:** <https://github.com/anomali-nangor/jagawarga-poc>

#### 4.5.2 Panduan Menjalankan Sistem

Sistem telah dikontainerisasi menggunakan Docker untuk memastikan konsistensi lingkungan pengembangan. Berikut adalah langkah-langkah untuk menjalankan PoC:

```

1 # 1. Clone repositori
2 git clone https://github.com/anomali-nangor/jagawarga-poc.git
3 cd jagawarga-poc
4
5 # 2. Menjalankan seluruh layanan dan infrastruktur
6 # Termasuk CockroachDB Cluster, NATS, Bun Services, dan Elixir Services
7 docker-compose up -d

```

```
8  
9 # 3. Verifikasi status container  
10 docker ps
```

Listing 4.1: Instruksi Menjalankan PoC

#### 4.5.3 Video (Tautan Demonstrasi di Youtube)

Dapat diakses di <https://youtu.be/wf9yZBnSmU>

## 4.6 Asumsi-Asumsi Perancangan

Dalam menyusun rancangan arsitektur dan implementasi PoC ini, diambil beberapa asumsi sebagai batasan ruang lingkup:

1. **Validitas Identitas:** Sistem mengasumsikan bahwa data kependudukan (NIK/KTP) yang dikirimkan warga sudah divalidasi oleh layanan pihak ketiga eksternal yang terpercaya. PoC hanya fokus pada cara sistem menangani data tersebut setelah divalidasi.
2. **Penyimpanan Multimedia:** Dokumen multimedia (foto/video) dalam PoC diasumsikan disimpan dalam *local volume* yang terikat pada kontainer *Report Service*, bukan pada *Object Storage* (seperti S3) yang terdistribusi secara geografis.
3. **Waktu Eskalasi:** Untuk kebutuhan demonstrasi, waktu tunggu eskalasi otomatis (FR-8) dipercepat dari satuan hari menjadi satuan detik/menit agar perubahan status dapat terlihat secara *real-time*.
4. **Integrasi Eksternal:** Sistem eksternal yang disebutkan pada FR-7 disimulasikan sebagai sebuah *mock webhook* yang mencatat penerimaan data laporan tanpa melakukan pemrosesan lebih lanjut.
5. **Keamanan Jaringan:** Seluruh komunikasi antar-layanan di dalam *private network* Docker diasumsikan aman, namun komunikasi publik melalui *Gateway* tetap diwajibkan menggunakan protokol HTTPS/TLS.