## **Table of Contents**

[**Table of Contents 1**](#_h5jlfk41kn4j)

[**1. Executive Summary 2**](#_9yt8l164f0ha)

[**2. System Overview 3**](#_vb1rimzdtxuu)

[2.1 Key Functional Components 3](#_qxmlq6qtptrh)

[2.2 High-Level Architecture Diagram (Conceptual) 4](#_jv4680kj6mxj)

[2.3 Architectural Priorities 5](#_fmv1o2cnszzh)

[2.4 Core Design Principles 5](#_6amk4sv8e7ng)

[**3. Core Components & Technology Stack 6**](#_6imtyfcp8rb1)

[3.1 Load Balancer: AWS Application Load Balancer (ALB) 6](#_r3b1ti5dea0s)

[3.2 API Gateway: AWS API Gateway 7](#_z30j4v5htwzt)

[3.3 Panic Microservice: ECS with Fargate 8](#_617xpn8zdv29)

[3.4 Message Queue: Amazon SQS 8](#_zc3qvsemixq6)

[3.5 Asynchronous Workers: ECS Tasks (Fargate) 9](#_chlxniejrcxp)

[3.6 Persistent Storage: Amazon Aurora (PostgreSQL-Compatible) 10](#_77cgyaizxvq8)

[3.7 Caching Layer: Amazon ElastiCache (Redis) 11](#_bsj0eakw63ku)

[3.8 Monitoring: AWS CloudWatch 12](#_lbgdtn97uwi2)

[3.9 Alerting & Health Checks 13](#_w4xvqoqu8xz)

[**4. Data Flow & Consistency Guarantees 14**](#_kcb77nqrhx2t)

[4.1 POST /panic: Panic Creation Flow 14](#_oyosbdwdqmqu)

[4.2 GET /panic/:id: Panic Retrieval Flow 15](#_k3dh3f8w5fu8)

[4.3 Eventual Consistency Model 16](#_29kopscrvti0)

[4.4 Cache Invalidation & TTL Strategy 16](#_82j4rdngbmac)

[**5. Scalability & Fault Tolerance Mechanisms 17**](#_bems8062gkvb)

[5.1 Horizontal Scalability 17](#_v01mpj6z590k)

[5.2 Fault Tolerance & Recovery 17](#_sh1e75e147sn)

[Built-In Fault Tolerance Features: 17](#_xg94kx6cqhi6)

[Worker Resilience: 18](#_kg18fg2oyhb2)

[5.3 Graceful Degradation 18](#_nim3nkbj7shk)

[5.4 Multi-AZ & High Availability 18](#_1xe16olzvdef)

[5.5 Chaos Readiness & Future Scaling 19](#_pw34o6d404w0)

[**6. Security & Compliance Considerations 19**](#_mwfce3b7rjue)

[6.1 Authentication & Authorization 19](#_j2ksqqyyl6i1)

[6.2 Data Encryption 20](#_nogyf0s28r1q)

[6.3 Audit Logging & Traceability 20](#_gtd41evltdvq)

[6.4 Access Control & Operational Security 20](#_pbjxqrbuwv)

[6.5 Regulatory & Compliance Alignment 21](#_q0uuqlfquaon)

[**7. DevOps & Deployment Strategy 21**](#_pax3mdm9m45q)

[7.1 Continuous Integration (CI) 21](#_8ulnz3idb23h)

[7.2 Continuous Deployment (CD) 22](#_yqxu2xf5vzho)

[7.3 Infrastructure as Code (IaC) 22](#_ufc3hugoucfm)

[7.4 Environment Configuration & Secrets 23](#_y3azmf1tjmdu)

[7.5 Observability & Deployment Safety 23](#_bgoddvykq64l)

[7.6 Local Developer Experience 23](#_tt0xoy4xoad5)

[**8. Monitoring, Logging and Alerting 23**](#_vqypoi5ef8jl)

[8.1 Monitoring Strategy 24](#_gecq9fs78kj2)

[8.2 Logging Strategy 24](#_nj3jtmwewqeo)

[8.3 Alerting Strategy 25](#_s9il19chw1o9)

[8.4 Dashboards and Visualization 25](#_u0i92qp4v4xr)

[8.5 Incident Response & Recovery 26](#_oynswes1nu7g)

[**9. Cost & Resource Optimisation Strategy 26**](#_5tblw42qjjrs)

[9.1 Compute Efficiency 26](#_wvxk36mscx3a)

[9.2 Storage & Database Cost Controls 27](#_hyyemewomr7y)

[9.3 Caching Strategy for Cost Avoidance 27](#_w4eku4tqxcln)

[9.4 Reserved & Spot Instances (Future Phase) 27](#_tnadwbdd9b2a)

[9.5 Cost Monitoring & Alerts 28](#_lxfrh63q03jb)

[9.6 Dev/Test Environment Efficiency 28](#_4aid9yjwzsam)

[9.7 Architectural Principles that Drive Cost Efficiency 28](#_ynv4tisijo5l)

[**10. Future Considerations 29**](#_5fegibnjtr60)

[10.1 Multi-Region Active-Active Deployment 29](#_33ebug7a3ku4)

[10.2 WebSocket Support for Real-Time Updates 29](#_7wg0bjwd91ic)

[10.3 Streaming Architectures (Kafka / MSK) 30](#_7ypmtg4kpj2f)

[10.4 Feature Flags and Experimentation 30](#_98askl1renxb)

[10.5 Disaster Recovery & Chaos Engineering 30](#_gf0hqutuopz2)

[10.6 Machine Learning & Smart Routing 31](#_ikwkavdysgzk)

[10.7 Auditability & Legal Reporting Enhancements 31](#_1nlv22el3zad)

[**11. Appendices 31**](#_ofxjsdahtyv)

[11.1 Summary Technology Table 32](#_punjs7np87el)

[11.2 Glossary of Terms 32](#_5nzzswfwdhef)

[11.3 Useful AWS Documentation 33](#_dnb1gktj3og5)

[11.4 Contact and Presentation Notes 33](#_qy9t6271exje)

## **1. Executive Summary**

This document outlines the proposed high-level system architecture for a fault-tolerant, scalable and observable API to support AURA’s “Never miss a panic” mission. The design prioritises real-time availability, eventual consistency where appropriate and operational resilience under unpredictable load conditions.

At its core, the solution is built on **modular microservices**, **asynchronous processing** and **cloud-native AWS components** that offer both horizontal scalability and built-in fault tolerance. It supports up to 1000 requests per second, with the flexibility to scale further as demand grows.

Key goals of this design include:

* **High Availability:** Ensured via multi-AZ deployments, managed services with built-in failover and auto-replacement of unhealthy components.
* **Horizontal Scalability:** Achieved through stateless microservices, load-balanced containers, event-driven workers and auto-scaling database replicas.
* **Fault Tolerance:** Delivered through message queues, dead-letter queues, retries, health checks and degraded read capability.
* **Eventual Consistency:** GET /panic/:id is designed to return a panic record even if full processing is incomplete, using placeholder writes and cache layering.
* **Observability:** System performance is instrumented with native AWS tools like CloudWatch and extended with optional integrations (e.g. Prometheus/Grafana) for operational visibility.

The technology stack leverages AWS as the primary cloud platform, in alignment with AURA's existing infrastructure and team skill set. Services such as **Amazon API Gateway**, **ECS with Fargate**, **SQS**, **Aurora** and **ElastiCache** form the backbone of this architecture.

This approach ensures a resilient, maintainable and future-ready platform that aligns with AURA’s core mission of enabling rapid, reliable emergency response at scale.

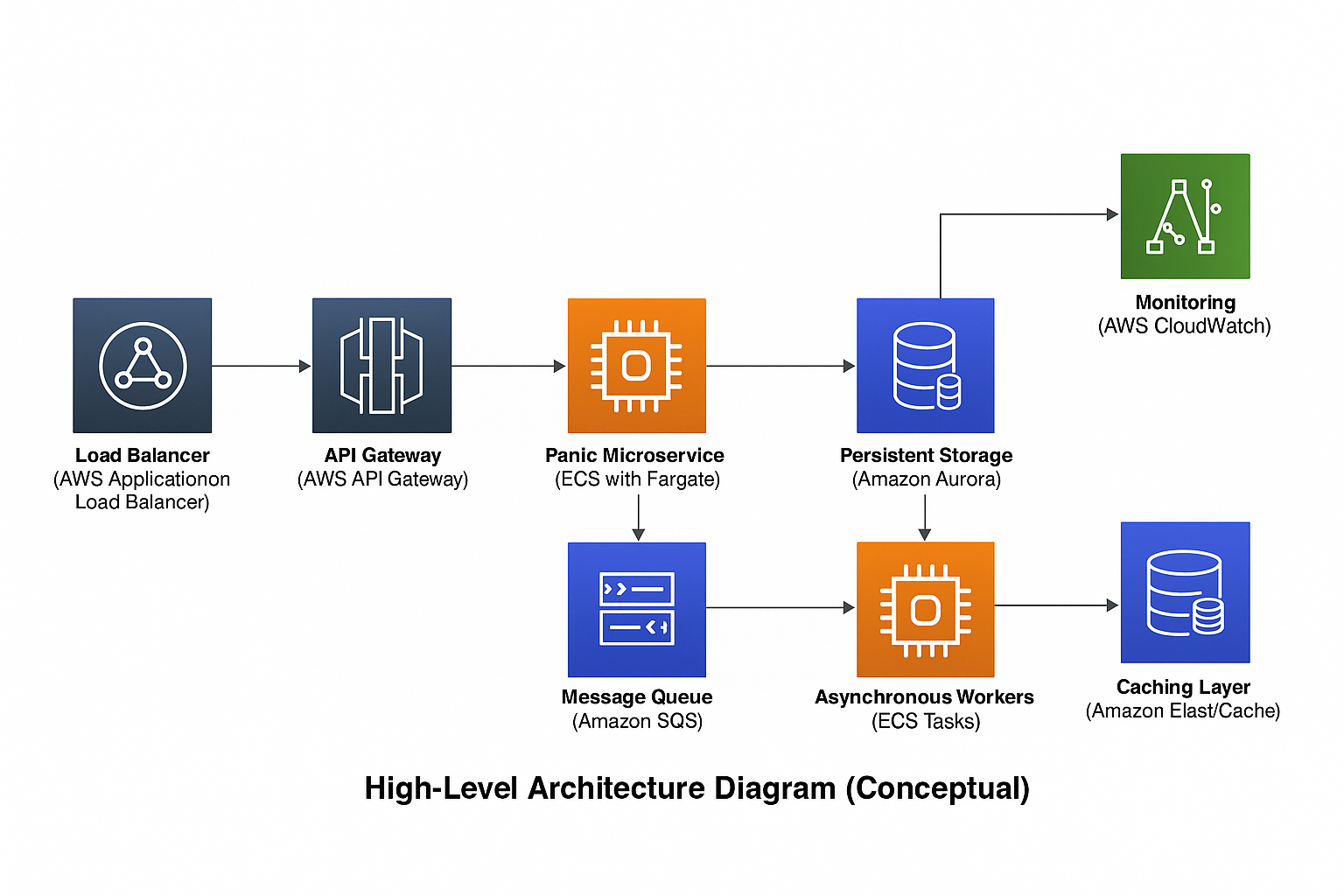
## **2. System Overview**

The proposed architecture supports AURA’s requirement for a **highly fault-tolerant Panic API** that reliably processes and serves emergency events with real-time responsiveness. The system mimics a ride-hailing-style experience where users can trigger a panic and be matched with the nearest available responder, all while ensuring the API remains performant, observable and available under stress.

### **2.1 Key Functional Components**

* **Panic API Service:**Stateless microservice exposing endpoints:  
  + POST /panic to initiate a panic event
  + GET /panic/:id to retrieve panic status or metadata  
    This service handles input validation, initial persistence and offloads downstream processing.
* **Asynchronous Worker Layer:**Workers subscribe to a queue (Amazon SQS) and perform time-consuming tasks like:  
  + Matching responders
  + Enriching metadata
  + Updating panic resolution state
* **Persistent Storage:**Amazon Aurora stores authoritative panic records, with ElastiCache (Redis) optionally caching recent lookups to improve GET latency.
* **Queueing Layer:**SQS acts as a buffer between the API and worker layer, enabling resilience, decoupling and retry-based fault tolerance.
* **Observability Layer:**AWS CloudWatch tracks system health, throughput, errors and latency. Alerts and dashboards ensure rapid detection and response to anomalies.

### **2.2 High-Level Architecture Diagram (Conceptual)**



### **2.3 Architectural Priorities**

* **Stateless & Horizontally Scalable:**All compute components (API service, workers) are stateless and can be scaled out independently. This allows seamless adaptation to increased load.
* **Eventual Consistency for Panic Reads:**POST /panic writes a lightweight “placeholder” panic immediately to ensure the record is visible to GET /panic/:id even before full backend processing completes.
* **Built for Spikes:**SQS queue absorbs surges in panic creation, while workers scale based on queue depth, ensuring downstream systems are not overwhelmed.
* **Auto-Healing:**Health checks at ALB and ECS level ensure that failed services are replaced. Aurora and ElastiCache offer built-in failover and replication.

### **2.4 Core Design Principles**

| **Principle** | **Implementation Strategy** |
| --- | --- |
| High Availability | Multi-AZ deployment, managed services, auto-recovery |
| Horizontal Scalability | Stateless services, Fargate scaling, Aurora read replicas |
| Fault Tolerance | SQS buffering, retries, DLQs, health checks |
| Eventual Consistency | Write-ahead placeholder + async update + cache fallback for GET |
| Observability | Centralised metrics, logging and alarms via CloudWatch + dashboards |

## **3. Core Components & Technology Stack**

This section describes each core infrastructure component used in the system, its role and the reasoning behind the technology selection. The architecture prioritises **horizontal scalability**, **fault tolerance** and **observability**, using AWS-native and cloud-optimised tools.

### **3.1 Load Balancer: AWS Application Load Balancer (ALB)**

**Role:**Routes HTTP(S) traffic to the appropriate backend service, supporting path-based routing and health checks.

**Justification (ALB):**

* Supports Layer 7 HTTP routing, ideal for REST APIs
* Seamless integration with ECS, Lambda and API Gateway
* Built-in support for health checks and automatic scaling
* Native CloudWatch metrics and access logs
* SSL termination and path-based routing with minimal configuration

**Alternatives:**

* AWS Network Load Balancer (NLB)
* NGINX on EC2
* HAProxy

**Why not NLB:**

* Operates at Layer 4 (TCP/UDP), lacks path- or host-based routing
* No native integration with API Gateway or HTTP-specific features
* Limited visibility into request metrics (compared to ALB)

**Why not NGINX on EC2:**

* Requires manual setup, scaling, patching and configuration
* Less fault tolerant unless manually deployed in HA setup
* Monitoring, TLS cert management and logging require additional effort

**Why not HAProxy:**

* Similar concerns to NGINX regarding operational overhead
* No native AWS integration, must manage scaling and failover manually
* Requires additional services for observability and automation

### **3.2 API Gateway: AWS API Gateway**

**Role:**Exposes the POST /panic and GET /panic/:id endpoints to external consumers with rate limiting and usage control.

**Justification (API Gateway):**

* Provides request validation, throttling, quotas and authentication
* Native AWS integration with Lambda, IAM and CloudWatch
* Scales automatically to meet high load (1000+ RPS)
* Ideal for exposing public HTTP endpoints securely
* Supports REST, HTTP and WebSocket APIs

**Alternatives:**

* Kong Gateway (self-hosted or cloud)
* Amazon CloudFront + Lambda@Edge
* Express Gateway (Node.js-based open source API Gateway)

**Why not Kong:**

* Requires setup and management unless using paid cloud version
* Adds latency compared to native AWS services
* Increased operational complexity for high availability

**Why not CloudFront + Lambda@Edge:**

* Suitable for global caching and edge-based auth, but
* Not ideal for dynamic API routing or internal service calls
* Harder to debug, limited regional write capabilities

**Why not Express Gateway:**

* Community-supported, less mature for production-grade scalability
* Requires Node.js hosting and management
* Lacks out-of-the-box integrations with AWS services

### **3.3 Panic Microservice: ECS with Fargate**

**Role:**Stateless service responsible for panic creation, initial persistence and queuing async processing tasks.

**Justification (ECS Fargate):**

* Fully managed container orchestration with no server provisioning
* Integrates natively with ALB, IAM, CloudWatch, SQS
* Supports CPU/memory tuning and predictable performance
* Good balance of flexibility and abstraction
* Auto-scales based on task demand or SQS queue depth

**Alternatives:**

* AWS Lambda
* AWS EKS (Kubernetes)
* EC2 Auto Scaling Group with Docker

**Why not Lambda:**

* Cold start latency for unpredictable or bursty traffic
* Execution time limited to 15 mins (bad for long processing tasks)
* Complex deployments for non-trivial containerised applications

**Why not EKS:**

* Requires managing Kubernetes control plane (even with EKS)
* Higher operational complexity and steep learning curve
* Overkill for simpler use cases with fewer services

**Why not EC2 ASG + Docker:**

* Must manage EC2 provisioning, patching and scaling
* No built-in task isolation like Fargate offers
* Slower scaling and increased maintenance burden

### **3.4 Message Queue: Amazon SQS**

**Role:**Buffers panic events and decouples the API from downstream processing. Ensures delivery and retry in failure cases.

**Justification (SQS):**

* Fully managed, no need to provision or manage brokers
* Supports both Standard (high throughput) and FIFO (ordered, deduplicated) queues
* Native integration with Lambda, ECS and Step Functions
* Built-in Dead Letter Queues (DLQs) for error handling
* Visibility timeouts, message retention and retry policies configurable per queue

**Alternatives:**

* Apache Kafka (Amazon MSK)
* Amazon SNS (pub/sub style)
* RabbitMQ (self-hosted or AWS MQ)

**Why not Kafka (MSK):**

* Operationally complex: requires managing brokers, storage, partitions
* Not serverless, requires careful tuning for performance and fault tolerance
* Higher latency for simple queueing needs (compared to SQS)
* More appropriate for event streaming than simple decoupling

**Why not SNS:**

* Fan-out use case, no message persistence or retries per subscriber
* Not suitable if downstream systems need to consume messages at different rates
* Harder to ensure reliable processing in asynchronous worker patterns

**Why not RabbitMQ:**

* Requires broker management (even on AWS MQ)
* Limited horizontal scalability compared to cloud-native services
* Complex error handling and backpressure mechanisms

### **3.5 Asynchronous Workers: ECS Tasks (Fargate)**

**Role:**Consumes messages from SQS to handle time-consuming panic processing like responder matching or escalation logic.

**Justification (ECS Fargate):**

* Handles background tasks with full control over compute resources
* Can be triggered by SQS, API calls, or scheduled tasks
* More consistent performance than Lambda (no cold starts)
* Supports concurrency limits and retry logic within containers
* Simplified scaling and isolation per worker type

**Alternatives:**

* AWS Lambda Workers
* EC2-based background workers
* Kubernetes Jobs (EKS)
* AWS Step Functions

​​**Why not Lambda Workers:**

* Cold start delays, especially for infrequent tasks
* Limited to 15-minute max execution time
* Requires managing retry logic and DLQs separately

**Why not EC2 Workers:**

* Must manage server lifecycle, AMIs, patching
* Slower to scale in response to queue depth
* Less granular control over auto-scaling at task level

**Why not EKS Jobs:**

* Adds Kubernetes complexity for job orchestration
* Requires extra monitoring for failed job retries and re-scheduling
* Not ideal for small-scale or simple background tasks

**Why not Step Functions:**

* Better suited for orchestrating sequences of short-lived tasks
* Poor fit for high-throughput, queue-based workloads
* Execution costs can become expensive at large volumes

### **3.6 Persistent Storage: Amazon Aurora (PostgreSQL-Compatible)**

**Role:**Stores all panic records and related metadata. Supports strong consistency, relational integrity and auditability.

**Justification (Aurora):**

* Fully managed, highly available with failover and read replicas
* Strong consistency with ACID compliance
* Supports SQL queries, joins, foreign keys and stored procedures
* Multi-AZ and auto-scaling read replicas available
* Compatible with standard PostgreSQL tooling

**Alternatives:**

* Amazon DynamoDB
* Amazon RDS (PostgreSQL/MySQL)
* CockroachDB (distributed SQL)

**Why not DynamoDB:**

* Eventually consistent by default (strong consistency costs more)
* Limited support for complex relational queries
* Requires careful key design and lacks built-in joins

**Why not Amazon RDS:**

* Aurora offers faster failover and better replication performance
* RDS has slower auto-scaling and more manual maintenance needs
* Aurora storage is distributed and fault-tolerant, unlike traditional RDS

**Why not CockroachDB:**

* Less mature AWS-native integration and ecosystem
* May introduce latency penalties for geo-distributed writes
* Requires a learning curve and extra setup for orchestration

### **3.7 Caching Layer: Amazon ElastiCache (Redis)**

**Role:**Speeds up read operations (GET /panic/:id) by caching recent panic records with short TTLs.

**Justification (Redis):**

* Low-latency, high-throughput in-memory store
* TTL-based caching for panic responses (GET /panic/:id)
* Supports pub/sub, sorted sets, atomic counters
* Managed Redis clusters with replication and backups
* Integrates well with ECS, Lambda and API Gateway

**Alternatives:**

* Memcached via ElastiCache
* Application-layer in-memory cache (e.g. Node.js LRU)
* Amazon DynamoDB Accelerator (DAX)
* CloudFront (CDN-level cache)

**Why not Memcached:**

* No persistence or replication
* Lacks advanced data types (sorted sets, lists, pub/sub)
* Not suitable for cache warm-up or recovery after node failure

**Why not App-level cache:**

* Non-shared across instances (problematic for horizontal scaling)
* Susceptible to instance restarts or crashes
* Inconsistent caching and difficult to invalidate globally

**Why not DAX:**

* Only works with DynamoDB, can’t cache relational DBs or API payloads
* Adds operational complexity and coupling to a specific DB strategy
* Not suitable for arbitrary key/value or computed data caching

**Why not CloudFront:**

* Edge-level cache best for static content, not dynamic panic records
* Doesn’t handle real-time data expiration or custom invalidation logic
* Harder to tune for personalised or short-lived content

### **3.8 Monitoring: AWS CloudWatch**

**Role:**Tracks logs, metrics and custom alerts for all components.

**Justification (CloudWatch):**

* Native integration with AWS services (ECS, Lambda, SQS, ALB)
* Centralised logs, metrics, dashboards and alarms
* Enables anomaly detection and composite alarms
* No infrastructure to manage, fully managed and scalable
* Integrates with EventBridge for custom responses to events

**Alternatives:**

* Prometheus/Grafana (self-hosted or Amazon Managed Service)
* DataDog
* New Relic
* ELK stack (Elasticsearch, Logstash, Kibana)

**Why not Prometheus/Grafana:**

* Requires operational management (storage, retention, scaling)
* Prometheus not ideal for distributed cloud-native metrics without federation
* Grafana is excellent for visualisation but requires setup and cost planning

**Why not DataDog:**

* Expensive at scale
* Proprietary pricing model based on hosts, metrics and retention
* Requires additional agents and integration work

**Why not New Relic:**

* Similar pricing/performance drawbacks to DataDog
* May duplicate functionality already available in CloudWatch
* Less granular integration with SQS, API Gateway, etc.

**Why not ELK Stack:**

* Requires managing ingestion pipeline (e.g., Logstash or Beats)
* Elasticsearch clusters need tuning and scaling
* More suited for log analytics than proactive monitoring

### **3.9 Alerting & Health Checks**

**Role:**Provides operational insight and automatic recovery for unhealthy components.

**Justification (Built-in AWS tools):**

* ALB health checks replace unhealthy containers automatically
* CloudWatch Alarms enable alerting on thresholds (latency, 5xx, queue depth)
* Dead Letter Queues (DLQs) catch failed tasks for retry/investigation
* EventBridge or SNS can be used to trigger auto-remediation
* Fully integrated with the rest of the AWS ecosystem

**Alternatives:**

* External uptime monitors (e.g., Pingdom, UptimeRobot)
* Custom health check endpoints
* PagerDuty or Opsgenie for on-call and incident escalation

**Why not external monitors:**

* Only detect surface-level (HTTP 200/timeout) failures
* No insight into internal queue depth, error rates, or business metrics
* Do not trigger AWS-native auto-recovery actions

**Why not custom endpoints:**

* Require dev effort and additional monitoring logic
* ALB-native checks are easier to manage and standardised
* Custom checks often lack the failover automation of AWS health checks

**Why not PagerDuty/Opsgenie:**

* Great for alerting and escalation, but adds cost
* Not a replacement for built-in health detection or failover
* Better suited as a downstream integration with CloudWatch alarms

## **4. Data Flow & Consistency Guarantees**

This section outlines the request lifecycle for both panic creation and retrieval and describes how consistency is managed in a distributed, asynchronous system. The design balances responsiveness with reliability, ensuring that panic records are accessible even if backend processing is still underway.

### **4.1 POST /panic: Panic Creation Flow**

**Sequence Overview:**

1. **Client Request** A client sends a POST /panic request to API Gateway.
2. **Validation and Acknowledgement** The request is routed through ALB to the ECS-based Panic API Service, where input is validated.
3. **Write-Ahead Insert** A placeholder panic record is synchronously written to:  
   * **Aurora DB**: Ensures visibility to subsequent reads
   * **ElastiCache (Redis)**: Optionally cached with TTL for quick access
4. **Event Queuing** The Panic API Service publishes a message to **Amazon SQS**, containing the panic ID and metadata for further processing.
5. **Asynchronous Handling** One or more **ECS-based Workers** consume the message and:  
   * Perform responder matching or escalation logic
   * Update the panic record in Aurora with enriched details
   * Optionally publish events (e.g. to SNS for responder notification)
6. **Finalisation** The panic record’s status is updated to “processed” or “active,” completing the flow.

**Guarantees:**

* **Durability:** The panic is persisted immediately and independently of downstream workers.
* **Asynchronous Processing:** Reduces latency at the API layer.
* **Resilience:** SQS decouples systems, enabling retry/recovery if processing fails.

### **4.2 GET /panic/:id: Panic Retrieval Flow**

**Sequence Overview:**

1. **Client Request** The client issues a GET /panic/:id request.
2. **Cache-First Lookup** The system checks **Redis** for a cached copy of the panic record.
3. **Database Fallback** If not found in cache, the service queries **Aurora** for the panic ID.
4. **Placeholder Support** Even if asynchronous processing is still ongoing, a partial placeholder record (created during POST) is available to return with status like "processing".
5. **Response Composition** A complete or in-progress panic record is returned, depending on backend completion state.

**Guarantees:**

* **Availability:** Users can query panics even before enrichment is complete.
* **Eventual Consistency:** GET reflects partial or complete state, depending on processing status.
* **Low Latency:** Redis cache improves response time for recently created or frequently accessed panics.

### **4.3 Eventual Consistency Model**

This architecture embraces **eventual consistency** between write and read paths to optimise latency without sacrificing data integrity.

| **Scenario** | **Consistency Strategy** |
| --- | --- |
| Panic just created | Immediate placeholder inserted in DB/cache |
| Asynchronous worker delayed | GET returns partial data with status = "processing" |
| Worker retries after failure | Final record updated once processing succeeds |
| Cache expiry or cache miss | DB fallback ensures consistency |
| Record update by worker | Overwrites placeholder, optionally re-cached |

**Benefits:**

* Supports degraded read modes with progressive data enrichment
* Improves responsiveness under high load
* Fault-tolerant to worker lag or message retries

### **4.4 Cache Invalidation & TTL Strategy**

To avoid stale reads while preserving performance:

* Panic records cached in Redis are assigned a **short TTL** (e.g. 30–60 seconds)
* Workers **overwrite cached entries** upon finalisation, ensuring consistency
* Infrequently accessed records are not cached, relying on DB reads

This hybrid model offers **freshness for active records** and **consistency for all records**.

## **5. Scalability & Fault Tolerance Mechanisms**

This section details the strategies used to ensure that the system remains responsive, resilient and operational during high load, failures and unexpected behaviour. The architecture leverages **horizontal scaling**, **stateless components** and **AWS-native failover mechanisms** to maximise uptime and performance.

### **5.1 Horizontal Scalability**

The system is built to scale each component independently, allowing fine-grained control over throughput and cost.

| **Component** | **Scalability Mechanism** |
| --- | --- |
| **API Gateway** | Auto-scales natively to handle thousands of RPS |
| **Application Load Balancer** | Distributes traffic across multiple ECS tasks |
| **Panic API Service (ECS)** | Scales ECS tasks horizontally based on CPU/memory or custom metrics |
| **Async Workers** | Scaled by SQS queue depth or scheduled batch sizes |
| **Aurora DB** | Read replicas handle read-heavy load; write scaling via clustering |
| **Redis Cache** | Supports clustering and read replicas |

**Auto Scaling Triggers:**

* ECS services use **CPU/memory thresholds** or **CloudWatch custom metrics**
* Workers use **queue length metrics** (e.g. ApproximateNumberOfMessagesVisible)

### **5.2 Fault Tolerance & Recovery**

Each layer of the system is designed to detect, isolate and recover from failure without human intervention.

#### **Built-In Fault Tolerance Features:**

| **Layer** | **Mechanism** |
| --- | --- |
| **Load Balancer (ALB)** | Health checks to remove unhealthy targets |
| **ECS Tasks (API & Workers)** | Auto-replacement of failed tasks, multi-AZ deployments |
| **SQS Queue** | Retains messages until processed; supports retries and DLQs |
| **Aurora DB** | Automatic failover to standby in case of node failure |
| **ElastiCache (Redis)** | Automatic node replacement; backup/restore support |
| **CloudWatch Alarms** | Alerts on failure thresholds, triggers recovery workflows (via EventBridge) |

#### **Worker Resilience:**

* **Retry Logic:** Built into SQS processing, with exponential backoff
* **Dead Letter Queues (DLQs):** Unprocessed messages are isolated for inspection
* **Idempotency:** Worker operations are designed to be safely retried (e.g., match responder only once)

### **5.3 Graceful Degradation**

When parts of the system are slow or degraded, the system still delivers partial functionality:

| **Situation** | **Degraded Behaviour** |
| --- | --- |
| DB temporarily unavailable | Redis may still serve cached panic records |
| Worker lag / backlog | GET /panic/:id returns placeholder record |
| API under high load | API Gateway throttles requests gracefully |
| Cache miss | System falls back to DB query |
| Partial failure of ECS tasks | ALB routes to healthy tasks only |

### **5.4 Multi-AZ & High Availability**

All services are deployed across **multiple Availability Zones** to ensure resilience against infrastructure-level failures:

* **ALB**, **ECS**, **Aurora** and **Redis** are configured for **multi-AZ redundancy**
* No component is tied to a single instance or AZ, enabling seamless failover
* Aurora supports fast <30s failover times for writes and instant read scaling

### **5.5 Chaos Readiness & Future Scaling**

To validate resilience under pressure:

* The architecture supports **chaos testing** using injected failures (e.g. stop tasks, block queues)
* Future improvements may include:  
  + **Multi-region active-active deployments**
  + **Global SQS with FIFO deduplication**
  + **Sharded database models for geographic segmentation**

## **6. Security & Compliance Considerations**

This section outlines the mechanisms implemented to ensure that data handled by the system remains secure, private and compliant with industry standards. Given AURA’s role in handling sensitive, potentially life-saving data, the architecture prioritises **data protection**, **access control** and **secure-by-default principles** across all layers.

### **6.1 Authentication & Authorization**

| **Component** | **Control Strategy** |
| --- | --- |
| **API Gateway** | Enforced via **IAM-based roles**, **API Keys**, or **JWT tokens** for integrators |
| **Internal Services** | Use **IAM roles for tasks** to enforce least-privilege access via AWS IAM policies |
| **Database Access** | ECS tasks access Aurora via **IAM + encrypted connection strings** |

**Best Practices:**

* Only authenticated integrators can invoke panic endpoints
* Service-to-service auth is scoped to minimal privilege levels (e.g., ECS task role limited to panic table)
* All credentials (tokens, keys, secrets) managed via **AWS Secrets Manager**

### **6.2 Data Encryption**

**In Transit:**

* All traffic between client and API Gateway uses **HTTPS (TLS 1.2 or higher)**
* Internal service calls (API → DB, API → SQS) use **TLS-secured endpoints**

**At Rest:**

* **Aurora DB** is encrypted using AWS-managed KMS keys (or customer-managed keys if required)
* **ElastiCache**, **SQS** and **CloudWatch Logs** are also encrypted at rest
* Panic record metadata, user data and timestamps are all protected via field-level encryption if needed

### **6.3 Audit Logging & Traceability**

All interactions are traceable via a layered logging and observability approach:

| **Activity** | **Logged To** | **Retention & Audit Strategy** |
| --- | --- | --- |
| API invocations | CloudWatch Logs + API Gateway logs | Includes request ID, source IP, status code |
| DB queries / writes (Aurora) | Aurora Performance Insights / Enhanced Monitoring | Used for audit trails and anomaly detection |
| Panic lifecycle events | Custom application logs | Correlated by panic ID |
| Worker executions & failures | CloudWatch Logs | Includes queue event ID, status, duration |

**Additional Features:**

* Request IDs and panic IDs are passed through headers for distributed tracing
* Optional integration with **AWS X-Ray** for tracing across microservices

### **6.4 Access Control & Operational Security**

* **IAM Policies** enforce **least privilege** across all AWS resources
* **Role-based access** is used within CI/CD pipelines and deployment systems
* **CloudTrail** is enabled to monitor all IAM access and infrastructure API calls
* **Secrets Manager** or **Parameter Store** is used for managing all service credentials

### **6.5 Regulatory & Compliance Alignment**

While not tied to a specific regulatory framework, the system design aligns with common compliance expectations such as:

| **Requirement** | **Supporting Mechanism** |
| --- | --- |
| **Data minimisation** | Only essential metadata is stored (location, time, responder ID, etc.) |
| **Encryption & secure transport** | TLS, KMS encryption, Secrets Manager |
| **Retention & deletion** | Configurable TTLs and policies in DB, SQS and cache |
| **User data access logging** | Comprehensive logging and IAM tracing |
| **Disaster recovery support** | Multi-AZ deployment + backup & restore capability |

If required, the system can be extended to support:

* GDPR/POPIA data subject request workflows (e.g., right to access/delete)
* ISO 27001-style audit controls
* Enhanced monitoring for regulatory reporting (e.g., incident response SLAs)

## **7. DevOps & Deployment Strategy**

This section outlines how infrastructure and application code are built, tested and deployed to ensure reliability, fast iteration and operational consistency. The system embraces **CI/CD**, **Infrastructure as Code (IaC)** and **canary deployment principles** to support safe and repeatable delivery of critical emergency response functionality.

### **7.1 Continuous Integration (CI)**

| **Stage** | **Tooling / Approach** |
| --- | --- |
| **Source Control** | GitHub |
| **CI Pipeline** | GitHub Actions or AWS CodePipeline |
| **Build Step** | Docker builds for ECS services; static analysis for TypeScript |
| **Linting & Formatting** | ESLint, Prettier |
| **Unit Testing** | Jest (for TypeScript services); executed in isolated CI job |
| **Secrets Handling** | GitHub Actions Secrets or AWS Secrets Manager (for staging) |

**Benefits:**

* Ensures all code meets quality and security standards before deployment
* Detects regressions early in the development lifecycle
* Simple integration with GitHub PR workflows for automation

### **7.2 Continuous Deployment (CD)**

| **Stage** | **Tooling / Approach** |
| --- | --- |
| **Infrastructure Provisioning** | AWS CDK or Terraform (modular IaC) |
| **Service Deployment** | AWS CodeDeploy (for ECS) or GitHub Actions runner |
| **Versioning** | Container image tags + Git commit SHA tracking |
| **Canary/Blue-Green** | ECS + ALB target group switching |
| **Rollback Strategy** | Immediate rollback to last stable image upon alarm |

**Deployment Flow:**

1. Merge to main triggers image build and push to Amazon ECR
2. Deployment plan is applied via IaC (CDK/Terraform)
3. ECS task definition is updated and deployed
4. Health checks + canary ALB routing used for zero-downtime rollout
5. On failure, alarms trigger rollback or manual intervention

### **7.3 Infrastructure as Code (IaC)**

All cloud infrastructure is defined as code, enabling consistent and auditable environments.

| **Tooling** | **Notes** |
| --- | --- |
| **AWS CDK (preferred)** | Strong support for TypeScript, familiar to fullstack engineers |
| **Terraform** | Viable alternative for broader multi-cloud portability |
| **State Management** | Stored securely (e.g., in S3 or Terraform Cloud) |
| **Environments** | Dev, staging and production defined as separate stacks or workspaces |

### **7.4 Environment Configuration & Secrets**

| **Concern** | **Implementation** |
| --- | --- |
| **Secrets & credentials** | Managed with AWS Secrets Manager or SSM Parameter Store |
| **Environment variables** | Injected at deploy time via task definition or Lambda configuration |
| **Sensitive key rotation** | Scheduled via Secrets Manager lifecycle policies |

### **7.5 Observability & Deployment Safety**

To protect against regressions or unnoticed failures:

* **CloudWatch Alarms** are tied to error rate, latency and task health metrics
* **Post-deploy Smoke Tests** run against staging and production environments
* **ALB Health Checks** ensure that only healthy instances receive traffic
* **Manual Approval Step** (optional) for high-risk deploys (e.g., responder matching logic)

### **7.6 Local Developer Experience**

To support fast iteration and maintain team velocity:

* Local development uses Docker Compose or AWS SAM for service emulation
* Unit + integration test suites support offline validation
* Preconfigured mocks for dependent services (e.g., SQS, Redis, Aurora via test containers)

This DevOps foundation enables rapid delivery with high confidence, aligning with the real-time demands of emergency dispatch technology.

## **8. Monitoring, Logging and Alerting**

Effective observability is critical in a system where uptime, latency and response reliability directly impact user safety. This section outlines how the system will be monitored and instrumented and how operators will be alerted to degradation before users are impacted.

The solution adopts an AWS-native first approach, enhanced with optional open-source and third-party observability tools.

### **8.1 Monitoring Strategy**

All key infrastructure and services are instrumented to capture metrics across the **Four Golden Signals** of observability: **Latency**, **Traffic**, **Errors** and **Saturation**.

| **Layer** | **Metrics Captured** | **Tooling** |
| --- | --- | --- |
| **API Gateway** | Latency, 4xx/5xx error rates, request volume | CloudWatch Metrics |
| **ALB** | Target response times, target health, request count | CloudWatch + ALB Access Logs |
| **Panic API Service** | Container CPU/memory, request duration, panic processing time | CloudWatch + Custom Metrics |
| **SQS Queue** | Message backlog, oldest message age, DLQ count | CloudWatch Alarms |
| **Async Workers** | Processing time, success/failure rates, concurrency levels | CloudWatch Logs + Metrics |
| **Aurora DB** | Query latency, connection count, CPU usage, read/write throughput | Enhanced Monitoring + Insights |
| **Redis (ElastiCache)** | Cache hit/miss ratio, memory usage, evictions | CloudWatch |

### **8.2 Logging Strategy**

Structured, context-rich logs are essential for debugging and forensics. All logs are sent to **AWS CloudWatch Logs**, with context-aware grouping by service and environment.

| **Source** | **Log Content** | **Format/Notes** |
| --- | --- | --- |
| **API Gateway** | HTTP method, path, request ID, response code, caller IP | JSON / Access Logs |
| **Panic API Service** | Lifecycle logs with panic ID, validation status, DB/queue outcome | JSON / Winston or pino (TS) |
| **Async Workers** | Queue ID, processing result, retry attempts, error stack traces | Grouped by message ID |
| **Database Logs** | Slow queries, connection pool exhaustion | Aurora Performance Insights |

**Best Practices:**

* All logs include a **request ID** and **panic ID** for distributed tracing
* Sensitive data is redacted or omitted (e.g., personal location or device info)
* Logs are retained based on environment-specific policies (e.g. 7d dev, 30–90d prod)

### **8.3 Alerting Strategy**

Alerts are tuned to catch real system issues while avoiding noise. Alert channels are routed via **CloudWatch Alarms** to **Amazon SNS**, with optional integration into tools like PagerDuty or Slack.

| **Alert Type** | **Trigger Condition** | **Response Strategy** |
| --- | --- | --- |
| **API Latency Spike** | P95 latency exceeds 500ms for 1+ minute | Investigate downstream services |
| **Error Rate Spike** | 5xx or failed requests exceed 2% of traffic | Triage logs; auto-retry or rollback |
| **SQS Queue Backlog** | Queue length exceeds 1000 or age > 2 minutes | Scale workers or debug stuck jobs |
| **Worker Failures** | DLQ receives > 10 messages in 5 minutes | Alert and investigate retry cause |
| **Aurora CPU Saturation** | > 80% CPU usage for 5 minutes | Scale read replicas or optimise queries |
| **Redis Evictions** | Memory hits capacity + eviction rate increases | Increase node size or shard count |

### **8.4 Dashboards and Visualization**

AWS CloudWatch dashboards consolidate real-time operational views across services, including:

* Panic creation and processing throughput
* System health (task counts, SQS depth, DB connections)
* Error vs. success rate trends over time
* Latency heatmaps per endpoint

**Optional Enhancements:**

* **Grafana (via Amazon Managed Grafana):** For richer dashboards and team-friendly views
* **Prometheus + Exporters:** If open-source metrics pipelines are preferred
* **X-Ray / OpenTelemetry:** For distributed tracing (pending implementation scope)

### **8.5 Incident Response & Recovery**

* **Alarms trigger automated remediation** (e.g., scale out workers, restart tasks) or notify on-call engineers
* **Runbooks** define action plans for common incidents (e.g., “High queue age” → scale ECS worker count)
* **Postmortem templates** are used after major incidents to review, document and prevent recurrence

## **9. Cost & Resource Optimisation Strategy**

This section outlines how the system design proactively reduces unnecessary cloud spend without compromising performance, scalability, or reliability. Cost controls are embedded in both infrastructure choices and operational workflows, using a mix of automation, right-sizing and service-level optimisation.

### **9.1 Compute Efficiency**

**Primary Focus:** Minimise waste from over-provisioning while maintaining rapid scalability.

| **Component** | **Optimisation Strategy** |
| --- | --- |
| **ECS (Fargate)** | Use **resource-based auto-scaling** (CPU/memory) to run only what’s needed |
| **Async Workers** | Scale based on **SQS queue depth** and idle task timeout |
| **Development Environments** | Use **lower-capacity tasks**, shorter TTLs and off-hours task suspension |
| **CI/CD Runners** | Self-host on ephemeral compute (e.g., spot EC2) if needed for longer builds |

### **9.2 Storage & Database Cost Controls**

| **Service** | **Optimisation Actions** |
| --- | --- |
| **Aurora** | Use **Aurora Serverless v2** if traffic is bursty or inconsistent  Archive old records to S3 + Athena after X days (e.g., closed panics > 90 days old) |
| **Redis (ElastiCache)** | Start with **single-node or replication-disabled cluster** for early-stage environments |
| **SQS** | Tune retention period to business needs (e.g., 1–3 days), clean up dead-letter queues regularly |

### 

### **9.3 Caching Strategy for Cost Avoidance**

* Reduce unnecessary DB reads via Redis cache for frequently accessed panic records (GET /panic/:id)
* Use **short TTLs** (e.g., 30–60 seconds) to balance freshness with read volume
* Avoid caching non-critical or low-frequency data

This approach not only improves latency but also **reduces Aurora read I/O and Redis memory costs**.

### **9.4 Reserved & Spot Instances (Future Phase)**

As traffic patterns stabilise, the system can be further cost-optimised:

| **Strategy** | **Benefit** | **Applicable Components** |
| --- | --- | --- |
| **Reserved Fargate Capacity** | Up to 30–50% savings for predictable workloads | Panic API service, workers |
| **Spot ECS Tasks** | Ideal for non-critical background jobs | Low-priority async workers |
| **Savings Plans** | Global discount across ECS + Lambda usage | All compute-bound services |

### **9.5 Cost Monitoring & Alerts**

| **Tool** | **Use Case** |
| --- | --- |
| **AWS Cost Explorer** | Visualise daily/weekly spend trends |
| **AWS Budgets** | Set hard and soft budgets per environment (e.g., staging vs prod) |
| **CloudWatch Metrics** | Alert if cost-driving metrics spike unexpectedly |
| **Tag-based Cost Allocation** | Track cost per service/environment via naming conventions and tags |

### **9.6 Dev/Test Environment Efficiency**

* Use **smaller instance sizes** for test workloads (e.g., t4g for ECS)
* Reduce log retention and dashboard granularity in non-production environments
* Run non-critical dev environments **only during business hours** using scheduled start/stop rules

### **9.7 Architectural Principles that Drive Cost Efficiency**

| **Principle** | **Cost Benefit** |
| --- | --- |
| **Serverless or managed-first** | Avoids VM management overhead and reduces idle infrastructure |
| **Asynchronous decoupling** | Allows deferred processing and cost-friendly scaling per workload |
| **Stateless design** | Enables scale-in and instance recycling without data loss |
| **Observability with purpose** | Avoids overspending on high-frequency logs or underused dashboards |

## **10. Future Considerations**

While the current architecture is designed for immediate scalability, fault tolerance and performance, several areas have been identified for future evolution. These enhancements are intended to increase flexibility, improve user experience, support geographic growth and prepare the platform for more advanced product capabilities.

### **10.1 Multi-Region Active-Active Deployment**

**Rationale:** Improve global resilience and reduce latency for users across different regions.

| **Enhancement** | **Impact** |
| --- | --- |
| Deploy ECS, Aurora and Redis in multiple regions | Enables failover and geo-local response times |
| Use **Route 53 + latency-based routing** | Directs traffic to nearest healthy region |
| SQS queues replicated or sharded regionally | Supports localised processing while maintaining decoupling |

**Challenges to Address:**

* Data consistency and replication between regions (Aurora Global Database, S3 replication)
* Eventual consistency trade-offs and active-active write reconciliation
* Regional responder availability integration

### **10.2 WebSocket Support for Real-Time Updates**

**Rationale:** Improve agent and integrator experience with instant panic status updates, responder ETA, etc.

| **Enhancement** | **Impact** |
| --- | --- |
| Add **WebSocket API** via API Gateway or AppSync | Enables push-based status updates |
| Maintain live connection during active panic | Reduces need for polling and improves UX |

**Use Case:** Agent dashboards, client mobile apps, or integrator systems can receive live updates on panic lifecycle changes without polling.

### **10.3 Streaming Architectures (Kafka / MSK)**

**Rationale:** Prepare for richer, high-frequency event processing (e.g. analytics, location telemetry, panic clustering).

| **Enhancement** | **Impact** |
| --- | --- |
| Use **Amazon MSK** (Kafka) | Enable replayable streams and partitioned event flow |
| Integrate with analytics layer | Build near-real-time dashboards and panic heatmaps |

**Use Cases:**

* Multiple consumers (analytics, machine learning, audit trail)
* Replay panic events to backfill or test new worker logic
* Enrich responder allocation with behavioural models

### **10.4 Feature Flags and Experimentation**

**Rationale:** Improve deployment safety and enable product iteration via controlled rollouts.

| **Enhancement** | **Impact** |
| --- | --- |
| Integrate feature flag platform (e.g. LaunchDarkly, AWS AppConfig) | Toggle features without redeploying |
| Support A/B testing of new panic routing logic or responder ranking | Validate improvements with confidence |

### **10.5 Disaster Recovery & Chaos Engineering**

**Rationale:** Validate assumptions about resilience and ensure recovery procedures are testable.

| **Enhancement** | **Impact** |
| --- | --- |
| Define **Recovery Time Objectives (RTO)** and **Recovery Point Objectives (RPO)** | Improve clarity on failover readiness |
| Introduce **chaos experiments** (e.g., SQS throttling, DB failover drills) | Builds confidence in resilience strategy |

### **10.6 Machine Learning & Smart Routing**

**Rationale:** Optimise panic resolution through smarter responder dispatch and predictive escalation.

| **Enhancement** | **Impact** |
| --- | --- |
| Apply ML models to location, historical response times, responder status | Improve accuracy of matching logic |
| Feed model data through **streaming (Kafka) or batch (Glue/S3)** pipelines | Scale model training with production data |

### **10.7 Auditability & Legal Reporting Enhancements**

**Rationale:** Support future compliance needs (e.g., GDPR, POPIA, ISO 27001).

| **Enhancement** | **Impact** |
| --- | --- |
| Add **field-level encryption** or **data minimisation rules** | Protect sensitive PII or geolocation data |
| Build **audit trail microservice** | Track lifecycle of every panic event with full traceability |
| Define **data retention policies** per regulation | Simplify compliance and reduce long-term cost |

These considerations ensure the system can evolve with AURA’s growing user base, data needs and reliability expectations, without requiring fundamental redesign. Most can be integrated incrementally through decoupled components and strategic infrastructure investments.

## **11. Appendices**

This section provides supporting materials referenced throughout the document, including a summary of selected technologies, diagrams, definitions of key concepts and links to additional reading. These resources are intended to assist reviewers in evaluating the design's technical soundness and alignment with AURA’s platform needs.

### **11.1 Summary Technology Table**

| **Component** | **Primary Technology** | **Purpose** |
| --- | --- | --- |
| Load Balancer | AWS ALB | Route HTTP(S) traffic and perform health checks |
| API Gateway | AWS API Gateway | Secure external API access and request throttling |
| Panic Microservice | ECS with Fargate | Stateless panic creation and queuing |
| Async Processing | ECS Workers (Fargate) | Decouple long-running responder-matching workflows |
| Queueing | Amazon SQS (with DLQ) | Buffer events and ensure reliable delivery |
| Persistent Storage | Amazon Aurora (PostgreSQL) | Store panic metadata with ACID guarantees |
| Caching Layer | ElastiCache (Redis) | Accelerate panic reads and reduce DB load |
| Monitoring | CloudWatch + Logs | Track metrics, logs and system alarms |
| Alerting | CloudWatch Alarms + SNS | Notify operators of critical issues |
| IaC | AWS CDK / Terraform | Define infrastructure in code |
| CI/CD | GitHub Actions, CodePipeline | Automate builds, tests and safe deployments |

### **11.2 Glossary of Terms**

| **Term** | **Definition** |
| --- | --- |
| ECS | Elastic Container Service - managed container orchestration by AWS |
| Fargate | Serverless compute engine for running ECS tasks without managing EC2 |
| SQS | Simple Queue Service - decouples components using message queues |
| ALB | Application Load Balancer - routes HTTP(S) requests at Layer 7 |
| DLQ | Dead Letter Queue - isolates failed messages for inspection and recovery |
| Aurora | Amazon’s high-performance managed relational database engine |
| TTL | Time To Live - determines how long a cache entry remains valid |
| IaC | Infrastructure as Code - define and provision infrastructure programmatically |
| Chaos Engineering | Discipline of injecting failures to test system resilience |

### **11.3 Useful AWS Documentation**

| **Topic** | **Link** |
| --- | --- |
| ECS with Fargate | https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html |
| Amazon API Gateway | https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html |
| Amazon Aurora (PostgreSQL) | https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-postgresql.html |
| Amazon SQS | https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/welcome.html |
| CloudWatch Metrics & Alarms | https://docs.aws.amazon.com/cloudwatch/index.html |
| ElastiCache for Redis | https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/WhatIs.html |

### **11.4 Contact and Presentation Notes**

* **Document Author:** Wikus van der Westhuizen (wait.wat.huh@gmail.com)
* **Assessment Type:** AURA Technical Architecture Assessment
* **Submission Format:** Live walkthrough + document handover
* **Source Code Link:** <https://github.com/waitwathuh/aura-panic-api>