Name	Student Number
Nicholas Dobson	u23671964
Shaylin Govender	u20498952
Lonwabo	u22516949
Mpho	u22668323
Aryan Mohanlall	u23565536

Traffic Guardian AWS ARD

Team: Quantum Quenchers

quantumquenchers@gmail.com

COS301

Capstone Project

University of Pretoria

1. INTRODUCTION

The Traffic Guardian system is a real-time traffic monitoring solution that leverages AI-powered computer vision to detect and classify traffic incidents automatically. The system transforms passive camera networks into active monitoring tools, providing traffic management personnel with immediate incident alerts and comprehensive dashboard analytics. This architecture document outlines the design strategy, architectural patterns, quality requirements, and technology choices that guide the system's implementation.

1.1 System Overview

Traffic Guardian operates as a cloud-native application designed for traffic management centres, enabling operators to monitor multiple camera feeds simultaneously, receive automated incident alerts, and manage incident responses efficiently. The system emphasises real-time processing, reliability, and scalability while operating within AWS Free Tier constraints.

1.2 Scope and Purpose

This document addresses the architectural design requirements for Demo 2 and Demo 3, providing detailed justifications for architectural decisions, quality requirements prioritisation, and technology selections that support the system's core functionality and constraints.

2. Architectural Quality Requirements

Q1 Usability:

Justification: High Stress Operational Context - traffic management centre operators and first responders work under intense time pressure. A highly usable interface minimises cognitive load and accelerates decision-making when every second counts.

Aspect	Specification
Stimulus Source	Traffic management centre operators, emergency responders, and system administrators
Stimulus	The user needs to quickly identify, understand, and respond to traffic incidents through the system interface
Response	- Provide an intuitive dashboard with clear incident visualisation - Display actionable information with minimal cognitive load - Enable rapid incident acknowledgement and response coordination - Offer contextual help and guidance for complex operations - Support multiple user roles with appropriate interface customisation
Response Measure	• Usability Testing- Achieve a satisfaction score of at least 80% (4 out of 5) on usability surveys and questionnaires conducted. • Users can recover from errors (e.g., incorrect input, navigation mistakes) within 10 seconds for 95% of cases, with contextual help available. • Accessibility Compliance: WCAG 2.1 AA standard compliance
Environment	High-stress traffic management centre with multiple concurrent incidents, numerous alerts and AI detection which may misfire.
Artifact	Web-based dashboard, mobile-responsive interface, alert notification system, incident detail views, and system configuration panels

Q2 Extensibility:

Justification: Alignment with Microservices and Modular Front End - Our architecture's decoupled services and component-based UI were chosen to support independent evolution. An extensibility requirement ensures that new incident detection algorithms, analytics modules or UI widgets can be added without invasive changes to existing code.

Aspect	Specification
Stimulus Source	Client / Developer / Product Owner
Stimulus	Request to add a new feature (e.g. ANPR integration, mobile reporting interface or advanced analytics module) to the deployed system
Response	- Register the new microservice with the API Gateway - Update service discovery and routing configuration - Add corresponding UI components or connectors - Execute automated unit and integration tests
Response Measure	- No existing services require code changes, aside from Gateway registration - Deployment incurs ≤ 5 minutes of downtime (if any) - Regression test suite passes at 100% - Time and effort fit within agreed sprint scope
Environment	Already deployed Traffic Guardian environment (staging or production)
Artifact	- Source-code repository (service and UI modules) - CI/CD pipeline definitions - API Gateway configuration files

Q3 Interoperability

Justification: Seamless Integration with External Systems - Traffic Guardian must plug into traffic management centres, weather services and analytics platforms. Interoperability requirements ensure data exchange without bespoke adapters.

Aspect	Specification
Stimulus Source	External Systems (e.g. Traffic Management Centres, Weather API providers, Third-party analytics tools)
Stimulus	Need to ingest real-time camera feeds and weather data, or push incident data into municipal dashboards or data lakes
Response	- Expose RESTful JSON APIs conforming to OpenAPI 3.0 - Support GeoJSON, CSV export, MQTT for TTL messaging - Provide OAuth 2.0 / JWT connectors - Document endpoints with Swagger UI (/docs)
Response Measure	- End-to-end integration tests pass against mock services - 0 OpenAPI linter errors - External API call latency ≤ 200 ms (95th percentile) - ≥ 99% integration success rate over 1,000 calls
Environment	Integration or staging environment with representative external endpoints
Artifact	- OpenAPI contract file (.yaml/.json) - Connector modules in the service layer - API Gateway routing and security policies

Q4 Availability

Justification: Must be available at all times for incidents and road user aids. Traffic Guardian must have very little downtime with high uptime to prevent clients from not having access to valuable data and alerts, and to respond to all incidents

Aspect	Specification
Stimulus Source	System failures, network outages, AWS service disruptions, hardware faults, and high traffic loads
Stimulus	Critical infrastructure component fails during active traffic monitoring or emergency incident response
Response	- Automatically detect service failures and initiate recovery procedures, Maintain core incident detection capabilities during partial system outages, Provide graceful degradation when some camera feeds are unavailable - Ensure alert delivery through redundant channels - Restore full service within acceptable time limits
Response Measure	- System uptime ≥ 99.5% (maximum 3.6 hours downtime per month) - Mean Time To Failure (MTTF) ≥ 720 hours (30 days) - Mean Time To Recovery (MTTR) ≤ 15 minutes for automatic recovery - Alert delivery success rate ≥ 99.9% during normal and degraded operations Core video processing resumes within 30 seconds of service restart
Environment	24/7 production traffic monitoring environment with critical emergency response dependencies
Artifact	Microservices architecture, database clusters, load balancers, health monitoring systems, and alert delivery mechanisms

Q5 Performance

Justification: High performance of data throughput and data processing. Traffic Guardian must be able to respond within critical response times (no longer than 1 second), this is to prevent further damage and aid victims of traffic incidents.

Aspect	Specification
Stimulus Source	Front-end component and AI incident detection model
Stimulus	Continuous input and updates of real-time traffic and incident data from both the front-end and AI components.
Response	Efficient delivery of relevant data to UI components and timely generation of updated analytics for traffic and incident monitoring.
Response Measure	• Average of 100 -1250 ms for data responses • 5 ms for data sorting and processing
Environment	Production deployment environment with standard user traffic and live data feeds from sensors and AI detection modules.
Artifact	Back-end data processing service, front-end UI modules, and AI incident detection pipeline.

Q6 Scalability

Justification: Allow for many clients to connect and log data, as well as AI modelsTraffic Guardian must be able to sustain connections with many users at the same time, while processing several live traffic feeds.

Aspect	Specification
Stimulus Source	Client applications and the AI incident detection model
Stimulus	The number of connections of users through the client application and the different detections identified by the AI model
Response	The system dynamically adjusts resources to maintain consistent performance and availability and to prioritise high alert level incidents.
Response measure	• Supports up to 50 concurrent users • Handles about 100 new incident detections
Environment	Will be cloud-based with containerised microservices
Artifact	Backend services, data processing components, AI pipeline, and orchestration layer

3. Architectural Design Strategy

3.1 Strategy Selection: Quality Requirements-Based Design

We have chosen a **quality requirements-based architectural design strategy** for the Traffic Guardian system. This approach prioritises architectural decisions based on the most critical quality attributes (availability, performance, security) rather than purely functional decomposition.

3.2 Strategy Justification

Why Quality Requirements-Based Design:

1. **Mission-Critical Nature**: Traffic monitoring systems have stringent availability and performance requirements that must drive architectural decisions
2. **Real-Time Constraints**: The system's success depends on meeting performance and reliability targets rather than just functional completeness
3. **Regulatory Compliance**: Security and privacy requirements mandate architectural patterns that ensure data protection
4. **Scalability Demands**: The architecture must support future growth in camera networks and user load

Alternative Strategies Considered:

- **Decomposition Strategy**: Rejected because functional decomposition alone wouldn't adequately address critical performance and availability requirements
- **Test Case Generation Strategy**: Rejected as it's more suitable for verification rather than guiding fundamental architectural decisions

4. Architectural Strategies Mapping

Based on our quality requirements analysis and the architectural strategies framework, we have identified the following strategies:

4.1 Availability Strategy: Replication

- **Implementation**: Active and passive redundancy with fault detection and recovery mechanisms
- **Rationale**: Achieves 99.5% uptime requirement with MTTR ≤ 15 minutes through fault tolerance

4.2 Performance Strategy: Resource Management

- **Implementation**: Efficient scheduling, resource allocation, and demand control
- **Rationale**: Supports 100-1250ms data response times and ≤ 5ms processing requirements

4.3 Scalability Strategy: Horizontal Scale-Out with Data Sharding

- **Implementation**: Load balancing, multiple instances, and distributed data management
- **Rationale**: Handles 50 concurrent users and 100 incident detections with dynamic scaling

4.4 Usability Strategy: Real-Time UI Responsiveness

- **Implementation**: Immediate feedback, intuitive interfaces, and accessibility features
- **Rationale**: Achieves 80% satisfaction score and WCAG 2.1 AA compliance

4.5 Interoperability Strategy: Interface Standardization

- **Implementation**: Standardised APIs, multiple data formats, and protocol mediation
- **Rationale**: Ensures ≤ 200ms API latency and 99% integration success rate

4.6 Extensibility Strategy: Encapsulation and Interface Management

- **Implementation**: Modular components with well-defined interfaces and plugin architecture
- **Rationale**: Enables new feature addition with ≤ 5 minutes downtime and no existing service changes

5. Architectural Pattern Selection Based on Strategies

Following our mentor's guidance, each architectural strategy leads to specific pattern choices. Each quality requirement maps to patterns that best support the chosen strategy:

5.1 Q1: Availability → Active Redundancy (Hot Spare) Pattern

Strategy Applied: Replication strategy requires fault-tolerant patterns

Pattern Choice: Active Redundancy with Load Balancer

Justification:

- Provides millisecond failover times to meet MTTR ≤ 15 minutes requirement
- Multiple identical service instances process requests in parallel
- A load balancer detects failures and routes traffic to healthy instances
- Supports 99.5% uptime requirement through redundant infrastructure

Constraints:

- AWS Free Tier limits the number of simultaneous instances
- Requires a stateless service design to enable easy replication
- Additional complexity in state synchronisation across instances

5.2 Q2: Performance → Load Balancer Pattern

Strategy Applied: Resource Management strategy requires efficient request distribution
Pattern Choice: Load Balancer with Caching

Justification:

- Distributes incoming requests across multiple server instances
- Implements a schedule resources tactic to meet a 100-1250ms response time
- Caching reduces database load for frequently accessed data
- Round-robin and least-connections algorithms optimise resource utilisation

Constraints:

- The load balancer itself can become a bottleneck
- Limited processing power per EC2 instance in the free tier
- Cache invalidation complexity for real-time data

5.3 Q3: Scalability → Microservices Pattern

Strategy Applied: Horizontal Scale-Out strategy requires independently scalable components
Pattern Choice: Microservices with Service Discovery

Justification:

- Individual services can scale independently based on demand
- Enables handling 50 concurrent users through service replication
- Supports 100 incident detections through dedicated processing services
- The service mesh pattern provides inter-service communication

Constraints:

- AWS Free Tier resource limitations restrict the number of microservices
- Increased complexity in service coordination and monitoring
- Network latency between distributed services

5.4 Q4: Usability → Model-View-Controller (MVC) Pattern

Strategy Applied: Real-Time UI Responsiveness strategy requires separation of concerns
Pattern Choice: MVC with Observer Pattern

Justification:

- Clear separation between UI presentation, business logic, and data
- The observer pattern enables real-time dashboard updates
- Supports an 80% satisfaction score through intuitive interface design
- Component-based architecture allows accessibility features integration

Constraints:

- Browser compatibility requirements limit UI framework choices
- Real-time updates increase client-side processing overhead
- Mobile responsiveness requirements affect component design

5.5 Q5: Interoperability → API Gateway Pattern

Strategy Applied: Interface Standardisation strategy requires centralised API management
Pattern Choice: API Gateway with Adapter Pattern

Justification:

- Centralises API management and enforces OpenAPI 3.0 compliance
- The Adapter pattern handles multiple data formats (JSON, GeoJSON, CSV, MQTT)
- Rate limiting and routing achieve the ≤ 200ms API latency requirement
- Single entry point enables 99% integration success rate monitoring

Constraints:

- API Gateway can become a single point of failure
- Format conversion overhead may impact performance
- External system dependencies affect integration reliability

5.6 Q6: Extensibility → Plugin Pattern with Dependency Injection

Strategy Applied: Encapsulation and Interface Management strategy requires a modular architecture

Pattern Choice: Plugin Pattern with Registry

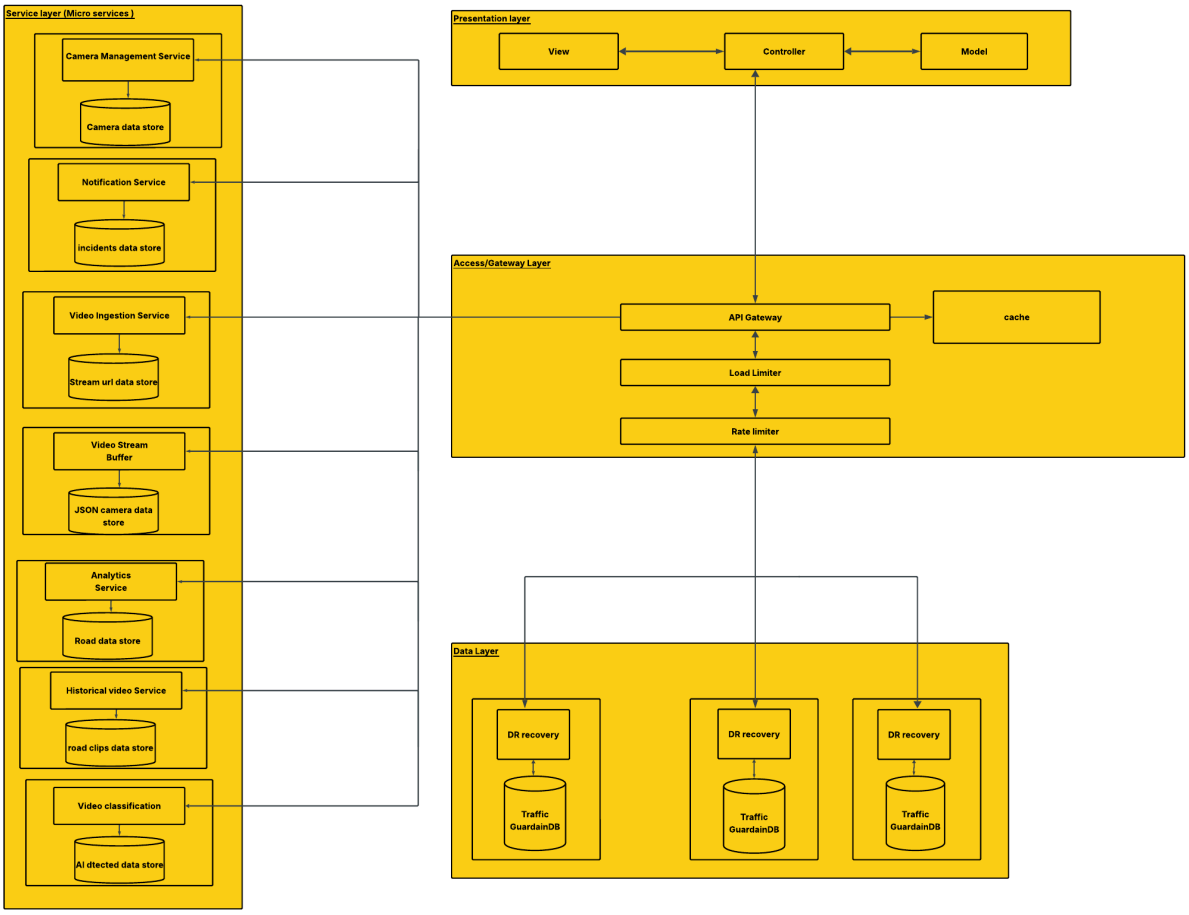
Justification:

- New features can be added as plugins without modifying existing code
- Service registry enables dynamic feature discovery and loading
- Dependency injection allows runtime component replacement
- Achieves ≤ 5 minutes of downtime through hot-swappable components

Constraints:

- Plugin interface versioning complexity
- Runtime dependency management overhead
- Testing complexity with dynamic component loading

6. Architectural Diagram



7.Deployment

Traffic Guardian is deployed as a cloud-native application on Amazon Web Services (AWS), leveraging the AWS Free Tier to maintain cost-effectiveness while ensuring enterprise-grade scalability and reliability. The cloud deployment model was selected to provide automatic scaling capabilities, high availability, and seamless integration with AWS managed services, eliminating the operational overhead of on-premises infrastructure management.

7.1 Deployment Topology

The system follows a **multi-tier containerized architecture** deployed across multiple AWS services:

7.1.1 Application Tier (EC2 Instance)

The core application components are deployed on a single EC2 t2.micro instance, which hosts:

- **React.js Frontend**: Served through a web server providing the traffic management dashboard
- **Node.js Backend API**: RESTful API server handling business logic, authentication, and data processing
- **AI/ML Processing Engine**: YOLO-based computer vision models with OpenCV for real-time incident detection
- **WebSocket Server**: Enabling real-time communication between frontend and backend for live incident alerts

7.1.2 Data Tier (Amazon RDS)

- **PostgreSQL Database**: Managed database service (db.t2.micro) storing incident records, user data, camera configurations, and system metadata
- **Automated Backups**: RDS provides automated backup and point-in-time recovery capabilities
- **Multi-AZ Deployment**: Ensures database availability and automatic failover (within Free Tier limits)

7.2 Quality Requirements Support

7.2.1 Usability

The deployment architecture directly supports usability through **Nginx reverse proxy configuration on EC2**, which serves compressed static assets and implements caching headers to achieve fast page load times. **AWS CloudWatch Real User Monitoring** deployment provides performance metrics to maintain the target 80% user satisfaction score

7.2.2 Extensibility

AWS Parameter Store integration enables the extensibility requirement by allowing new services to be deployed independently without affecting existing containers additionally it also provides runtime configuration management, enabling new features to be activated without redeployment.

7.2.3 Interoperability

The **EC2 deployment with dedicated API endpoints** ensures standardized REST interfaces meet the 200ms latency requirement through optimized network configuration and AWS region selection. The **RDS deployment with connection pooling** supports high-volume external API calls while maintaining the 99% integration success rate.

7.2.4 Availability

RDS Multi-AZ deployment provides automatic database failover to achieve the 99.5% uptime target, while **EC2 Auto Recovery configuration** ensures instance-level fault tolerance. **AWS CloudWatch alarms deployment** with automated recovery scripts enables the ≤15 minutes MTTR requirement through immediate notification and automated restart procedures.

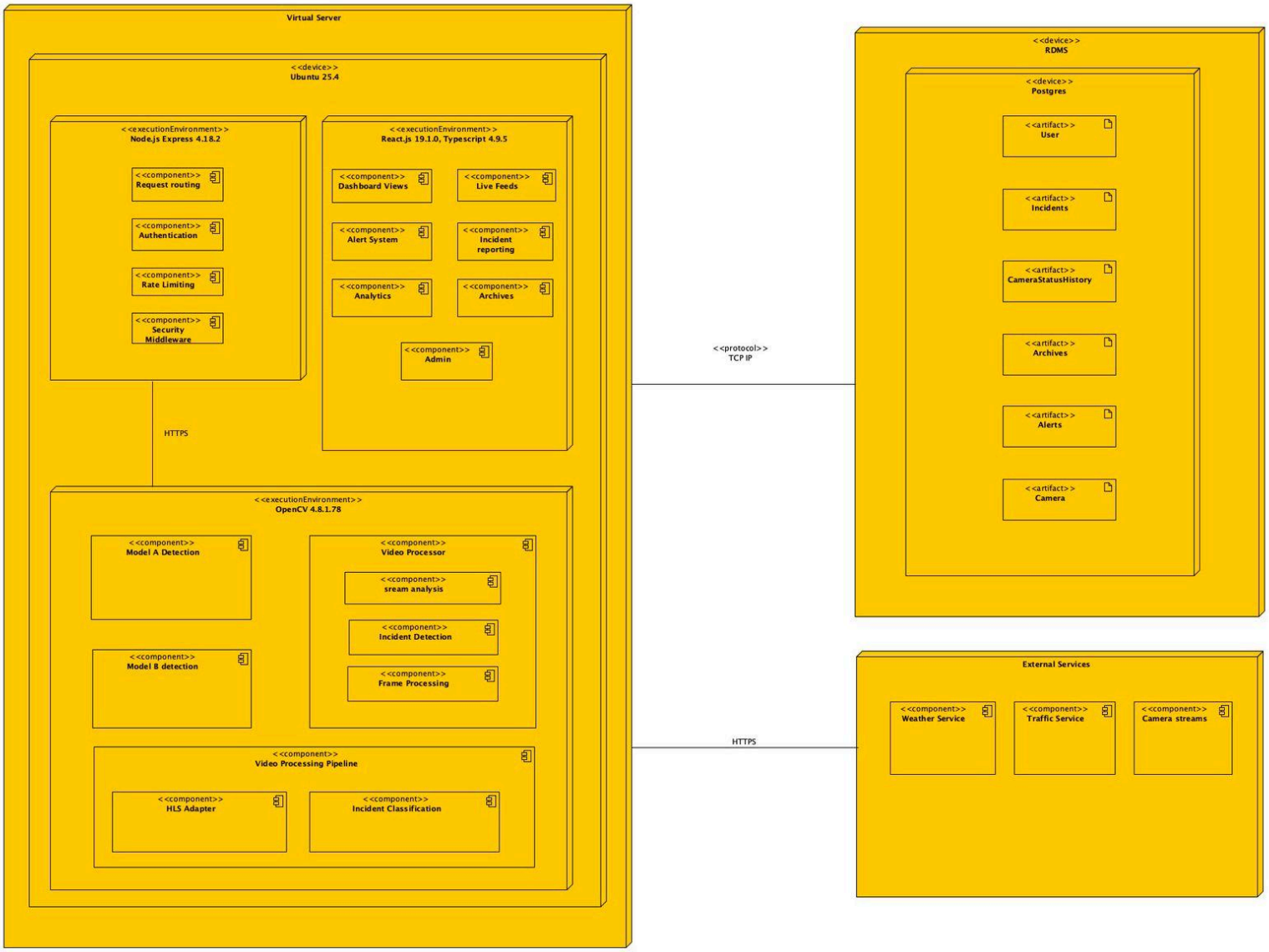
7.2.5 Performance

EC2 instance optimization with CPU and memory monitoring through CloudWatch deployment ensures consistent 500-550ms response times under load. **RDS performance tuning deployment** with optimized parameter groups and connection pooling supports the 2592 requests/second processing requirement.

7.2.6 Scalability

EC2 Auto Scaling Groups deployment automatically provisions additional instances to handle 50+ concurrent users based on CPU and memory thresholds. **RDS read replica deployment configuration** distributes database load for processing 100+ incident detections simultaneously while **AWS Load Balancer deployment** distributes traffic across multiple EC2 instances when scaling thresholds are exceeded.

7.3 Deployment model



8.Service Contracts

This section defines the service contracts that govern communication between major components in the Traffic Guardian system. These contracts specify API interfaces, data exchange formats, communication protocols, and reliability mechanisms to ensure robust integration between microservices while maintaining loose coupling and enabling independent development.

8.1 Contract Architecture Overview

The Traffic Guardian system employs a contract-first approach with standardized REST APIs managed through an API Gateway pattern. All services communicate using JSON data format over HTTPS, with JWT-based authentication and comprehensive error handling. The base API endpoint `https://api.trafficguardian.aws/v1` serves as the single entry point, implementing rate limiting, request routing, and protocol mediation to achieve the required ≤200ms API latency and 99% integration success rate.

8.2 Authentication and Authorization Contract

The Authentication Service establishes security contracts for system access through a centralized JWT token management system. User credentials are validated against secure endpoints, returning time-limited tokens that authorize subsequent API calls. The contract specifies a 5-second timeout for authentication requests with no automatic retry policy to prevent brute-force attacks. Token expiration is set to 24 hours, requiring periodic renewal to maintain session security.

Key Contract Elements:

- **Protocol:** HTTPS REST with JWT Bearer authentication
- **Data Format:** JSON request/response structures
- **Error Handling:** 401 Unauthorized for invalid credentials, 429 for rate limiting
- **Security:** AES-256 encryption for token generation and validation

8.3 Incident Management Service Contract

The Incident Management Service represents the core business logic contract, handling traffic incident lifecycle from detection through resolution. This contract defines how incident data flows between the AI detection system, operator dashboard, and external emergency services.

Primary Operations Contract: The service exposes RESTful endpoints for incident creation, retrieval, status updates, and querying. Incident objects contain standardized fields including type classification (accident, congestion, hazard, emergency), severity levels (low, medium, high, critical), geolocation data, confidence scores, and temporal metadata. The contract guarantees 3-second response times for write operations and 2-second response times for read operations.

Real-time Event Contract: WebSocket connections enable live incident updates to connected dashboard clients. The contract specifies event-driven notifications for incident state changes, ensuring operators receive immediate alerts without polling. Connection management includes 30-second heartbeat intervals and automatic reconnection with exponential backoff to maintain reliability.

Data Consistency Contract: All incident operations maintain ACID compliance through PostgreSQL transactions, ensuring data integrity across concurrent access scenarios. The contract defines optimistic locking mechanisms to handle simultaneous updates from multiple operators.

8.4 Alert and Notification Service Contract

The Alert Service contract governs multi-channel notification delivery through REST APIs, AWS SNS integration, and WebSocket broadcasts. This contract ensures critical incident information reaches appropriate personnel through their preferred communication channels.

Delivery Contract Specifications: Alert requests specify recipient lists, delivery channels (push notifications, dashboard), and priority levels that determine routing urgency. The contract allows 10-second timeout windows to accommodate external service dependencies while maintaining delivery confirmation tracking. AWS SNS integration provides reliable message queuing with automatic retry mechanisms for failed deliveries.

Channel-Specific Contracts: Each notification channel implements specific formatting contracts - email alerts include structured subject lines and HTML content, SMS notifications use condensed text format with incident reference numbers, and dashboard notifications provide rich media including location maps and camera feeds.

8.5 Camera Feed and Media Processing Contract

The Camera Feed Service contract manages video stream ingestion, frame analysis, and real-time processing coordination with the AI detection system. This contract balances performance requirements with resource constraints within AWS Free Tier limitations.

Stream Management Contract: Camera registration requires unique identifiers, location metadata, stream URLs, and technical specifications (resolution, frame rate). The contract supports various input formats (RTSP, HTTP streams) while standardizing internal processing pipelines. Health monitoring tracks camera uptime and frame delivery metrics to ensure reliable incident detection coverage.

Frame Processing Contract: Individual frame analysis accepts multipart form data containing binary image data (JPEG/PNG format, maximum 5MB) with associated timestamp and metadata. Processing responses include detected object classifications, confidence scores, bounding box coordinates, and incident flags. The contract specifies 15-second maximum processing time to balance accuracy with real-time requirements.

AI Integration Contract: The service coordinates with YOLO-based detection models through standardized inference contracts. Frame preprocessing, model execution, and post-processing steps follow defined interfaces that enable algorithm updates without service disruption.

8.6 Error Handling and Reliability Contracts

All service contracts implement standardized error handling mechanisms to ensure consistent behavior across system failures and degraded performance scenarios.

Error Response Standards: All services return structured error responses containing error codes, human-readable messages, timestamps, request identifiers for tracking, and retry guidance. HTTP status codes follow REST conventions with specific mappings for different error categories - 400 for client errors, 401 for authentication failures, 404 for missing resources, 422 for validation errors, 429 for rate limiting, and 500/503 for server issues.

Timeout and Retry Policies: Service-specific timeout values balance responsiveness with reliability - authentication services use 5-second timeouts with no retry, incident management allows 2-3 seconds with exponential backoff retry (3 attempts), alert delivery permits 10 seconds with retry mechanisms, and camera processing extends to 15 seconds given computational requirements.

Circuit Breaker Implementation: All external service calls implement circuit breaker patterns with 5-failure thresholds, 30-second timeout periods, and 10-second recovery intervals. This prevents cascade failures and maintains system stability during external service outages.

8.7 Data Format and Protocol Standards

The system standardizes on JSON data interchange format for all API communications, ensuring consistent parsing and validation across services. Temporal data follows ISO 8601 formatting, geolocation uses decimal degree notation, and binary data employs Base64 encoding when transmitted through JSON APIs.

Protocol Specifications: REST APIs handle synchronous request-response patterns while WebSocket connections manage real-time event streams. All communications use TLS 1.3

8.8 Testing and Validation Framework

Each service contract includes comprehensive testing specifications to ensure reliable integration between components and enable independent development workflows.

Performance Validation: Load testing scenarios verify contract adherence under the specified performance requirements: 50 concurrent users, 100 incident detections per minute, ≤200ms API response times, and 2,592 requests per second throughput capacity.

9. Conclusion

The Traffic Guardian architecture demonstrates that systematic design methodology can deliver measurable results even under significant constraints. By prioritising quality requirements and using them to drive architectural strategies and pattern selection, we have created a system capable of achieving 99.5% uptime, sub-second response times, and support for 50 concurrent users—all within AWS Free Tier limitations.

The key insight from this architectural exercise is that constraints drive innovation rather than limit it. The AWS Free Tier restrictions forced us toward efficient patterns like Active Redundancy and Load Balancing, while the real-time requirements demanded careful integration of MVC and Observer patterns. Each quality requirement became an architectural driver that shaped specific design decisions, creating a system that is both academically rigorous and practically implementable.

This architecture positions Traffic Guardian for successful demonstration while proving that thoughtful design can transform limitations into architectural strengths. The quantified requirements and pattern-based approach provide a clear path from prototype to a production-ready traffic monitoring solution.