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

# Traffic Guardian AWS CSD

## Team: Quantum Quenchers

quantumquenchers@gmail.com

COS301

Capstone Project

University of Pretoria

## 1. Overview

This document outlines the coding standards and conventions for the **Traffic Guardian** system - an AI-powered traffic incident detection and management platform. These standards ensure **uniformity**, **clarity**, **flexibility**, **reliability**, and **efficiency** across our multi-component system.

### Technology Stack

- **Frontend:** React.js with TypeScript, modular CSS and Cypress for E2E
- **Backend:** Node.js with Express (Socket.IO for live updates)
- **AI/ML:** Python with YOLO/OpenCV
- **Database:** PostgreSQL
- **CI/CD:** GitHub Actions

## 2. Naming Standards

### Variables & Constants

- camelCase for variables and non-React functions.
- UPPER_SNAKE_CASE for constants and environment keys.
- Meaningful, intention-revealing names (avoid single letters except for conventional iterators).

### Types, Interfaces & Enums (TypeScript)

- PascalCase for type/interface/enum names (e.g., IncidentStats, TrafficIncident). See usage in the dashboard component's TypeScript interfaces.

### React Components & Hooks

- PascalCase for component files and component names (e.g., Dashboard.tsx, CarLoadingAnimation).
- camelCase for custom hooks (e.g., useFetchWeather).

### Files & Folders

- PascalCase for React component files (Dashboard.tsx), kebab-case or lowercase for folders (services, components, assets).
- CSS files mirror component/file names where co-located (e.g., Dashboard.css).

## 3. Code Layout & Structure

### Formatting & Indentation

- 2 spaces for indentation.
- Use Prettier for formatting and EditorConfig (if present) to harmonise indentation/line endings.
- Line length target 100–120 chars; wrap long JSX props and object literals vertically.

### Imports

- Order: Node/Polyfills → External packages → Absolute aliases → Relative (grouped, blank line between groups).
- No circular imports; prefer function parameters over cross-module singletons.

## Comments

- Keep explanations brief.
- Document side effects and non-obvious logic; avoid redundant comments.

## One Statement per Line

- Avoid chaining unrelated expressions on one line

## React/TS Conventions

- Prefer function components; declare Props/State via interfaces.
- Narrow types; avoid any. When unavoidable, type the minimum surface (see NewAlertPayload minimal typing pattern in Dashboard.tsx).
- Side effects go in useEffect with complete dependency arrays; cleanup timers/sockets in return callbacks (as done when closing the Socket.IO connection and intervals).
- Use useCallback/useMemo to stabilise handlers and computations used by children

## CSS & Theming

- Design tokens via CSS variables; define light/dark schemes under [data-theme="light"] / [data-theme="dark"]. Tokens must be used instead of hard coded colours.
- Prefer rem for spacing/typography; avoid magic numbers.
- Class names: kebab-case; avoid deep specificity; keep selectors shallow.
- Keep components presentational in CSS; stateful/interactive logic stays in TSX.

## Accessibility & Testability

- Use semantic elements (button, nav, section) and ARIA when needed.
- All interactive elements must be keyboard accessible.
- Stable test selectors: use data-cy="…", as consistently implemented throughout the dashboard UI. This is our onlyE2E selector mechanism.

## 4. Error Handling & Logging

### Principles

- Fail fast near the source; recover gracefully where user experience matters.
- Do not leak secrets in logs (tokens, API keys, PII).
- Provide user-friendly messages in the UI; keep technical detail in logs.

### Frontend

- Wrap API calls in try/catch; surface toasts/notifications for transient issues (as done when data load fails and for socket connection issues).
- Centralise error helpers in services/errors.ts (stack mapping, network vs. app errors).
- Use a React Error Boundary for rendering errors (component level crashes).

### Backend/Socket.IO

- Use structured logging (Winston/Pino). Map error codes to HTTP status/Socket events (e.g. connect_errorhandled with a critical notification).
- Propagate with context (throw new AppError('Fetching incidents failed', { cause, code })).

### Validation

- Server: Validate all request bodies/params (Zod/Joi) before business logic.
- Client: Validate forms; sanitise any interpolated values used in the DOM.

## 5. Configuration & Secrets

- Environment files: Use checked-in templates (.envExample.txt, .env.development) and never commit real secrets.
- Frontend env keys are prefixed REACT_APP_…; provide safe defaults (e.g., REACT_APP_SERVER_URL || 'http://localhost:5001').
- Rotate keys; scan commits with secret scanners; document required keys in README.

## 6. General Principles

### Core Values

1. **Clarity over Cleverness** - Write code that tells a story
2. **Consistency** - Follow established patterns throughout the codebase
3. **Modularity** - Clear separation between UI, business logic, and data layers
4. **Security First** - Always consider security implications
5. **Performance Awareness** - Write efficient code that scales

### Code Style Philosophy

- Favour explicit over implicit
- Use meaningful names that describe intent
- Keep functions small and focused
- Minimise cognitive load for future development

## 7. File Structure & Organization

### Repository Structure

```
traffic-guardian/
├── .github/workflows/
├── API/
│   ├── src/
│   │   ├── config/
│   │   ├── controllers/
│   │   ├── middleware/
│   │   ├── models/
│   │   ├── routes/
│   │   ├── services/
│   │   ├── modules/
│   │   ├── app.js
│   │   └── server.js
│   ├── UnitTesting/
│   ├── IntegrationTesting/
│   └── schema.sql
├── frontend/
│   ├── src/
│   │   ├── components/
│   │   ├── pages/
│   │   ├── contexts/
│   │   ├── services/
│   │   ├── utils/
│   │   └── App.tsx
│   ├── cypress/
│   └── __tests__/
├── AI_Model_BB/
│   ├── Code/
│   └── Testing/
├── docs/
├── assets/
└── README.markdown
```

### File Naming Conventions

- **React Components:** PascalCase (`TrafficDashboard.tsx`)
- **Pages:** PascalCase (`IncidentArchives.tsx`)
- **Utilities:** camelCase (`formatDateTime.ts`)
- **CSS/SCSS:** kebab-case (`incident-list.css`)
- **Python files:** snake_case (`incident_detection.py`)

# 8. Git Repository and Strategy

## Git Flow

We use the Git Flow model to enable parallel development while keeping production code stable.

Branches Maintained:

- Main (production)
- Dev (integration)
- Feature Branches

Branch Purposes:

1. Main — Always production-ready. When Dev is stable and fully tested, merge to Main for release.
2. Dev — Central hub for adding features, experiments, reviews/refactors; prepares content for the next release to Main.
3. Feature Branch — Branch from Dev and merge back into Dev once complete; discard if the feature is scrapped.

Project Flow:

- Name branches by the feature/issue they address.
- Merge completed features into Dev.
- Promote Dev into Main for production releases.

## Git Branch Naming Conventions (Feature Branches)

- Descriptive: concise name that clearly reflects the work (e.g. API).
- Alphanumeric only; no consecutive hyphens.

## Review Process

- Start new features from Dev; merge back once fully implemented.
- CI runs automated linting first to enforce coding standards, followed by unit tests.
- After automated checks pass, changes undergo manual review.
- At least one review is required.

## Linting

- Use ESLint to maintain code quality and consistency across the project

## 9. Frontend Standards (React.js)

### Component Organization

- Use functional components with hooks
- Keep components under 200 lines
- Extract custom hooks for reusable logic
- Implement proper error boundaries
- Use React.memo() for performance-critical components

### TypeScript Standards

- Always use TypeScript for new files
- Define interfaces for all props and complex objects
- Use strict type checking
- Prefer `interface` over `type` for object shapes
- Use meaningful generic type names

### State Management

- Use useState for local state
- Use custom hooks for complex state logic
- Use Context sparingly (authentication, theme only)
- Avoid prop drilling with a proper component structure

### CSS Standards

- Use CSS Modules or styled-components
- Follow BEM methodology for class names
- Use CSS custom properties for theming
- Mobile-first responsive design
- Consistent spacing using an 8px grid system

## 10. Backend Standards (Node.js)

### API Structure

- Organise code into controllers, services, and middleware
- Use consistent error handling across all endpoints
- Implement proper input validation
- Follow RESTful conventions for endpoints
- Use meaningful HTTP status codes

### Error Handling

- Create custom error classes for different error types
- Use global error handler middleware
- Log errors with appropriate detail levels
- Return a consistent error response format
- Never expose internal error details to clients

### Security Standards

- Use environment variables for sensitive data
- Implement API key validation
- Use parameterised queries to prevent SQL injection
- Validate and sanitise all user inputs
- Implement rate limiting on public endpoints

## 11. AI/ML Standards (Python)

### Code Organization

- Use type hints for all function parameters and returns
- Implement proper logging throughout the system
- Use descriptive variable and function names
- Keep functions under 50 lines when possible

### Model Management

- Store model weights in designated directories
- Version control model configurations
- Implement fallback mechanisms for model failures
- Use consistent confidence thresholds across models
- Document model performance metrics

### Data Processing

- Validate input data before processing
- Handle missing or corrupt data gracefully
- Use appropriate data types for performance
- Implement proper memory management for large datasets
- Log processing steps for debugging

## 12. API Design Guidelines

### RESTful Conventions

- Use standard HTTP methods (GET, POST, PUT, DELETE)
- Structure URLs hierarchically and consistently
- Use plural nouns for resource endpoints
- Include version numbers in API paths
- Implement proper pagination for list endpoints

### Response Standards

- Use a consistent JSON response format
- Include success/error status in responses
- Provide meaningful error messages
- Add metadata (timestamps, pagination info)
- Use appropriate HTTP status codes

## 13. Testing Standards

### Test Organization

- **Frontend**: Use Cypress for E2E testing, Jest for unit and integration tests
- **Backend**: Separate integration and unit testing directories
- **AI/ML**: Use pytest with coverage reporting

### Coverage Requirements

- Maintain **≥80% unit test coverage** for all modules
- Test critical paths: authentication, incident CRUD, video processing
- Include integration tests for API endpoints
- Run tests automatically on every commit

### Test Quality

- Write descriptive test names that explain the scenario
- Use setup and teardown methods appropriately
- Mock external dependencies in unit tests
- Include both positive and negative test cases
- Test edge cases and error conditions

## 14. Code Review Process

### Review Requirements

- All pull requests require at least one approval
- Reviews must check code quality and standards compliance
- Security considerations must be addressed
- Performance implications should be evaluated
- Tests must be included and passing

### Review Guidelines

- Focus on code clarity and maintainability
- Suggest improvements constructively
- Verify that standards are followed
- Check for potential security issues
- Ensure documentation is adequate

## 15. CI/CD Pipeline Standards

### GitHub Actions Workflow

- Run linting and formatting checks on all commits
- Execute the full test suite before merging
- Generate and upload coverage reports
- Perform security scans on dependencies
- Deploy automatically to the staging environment

### Deployment Standards

- Use Docker containers for consistent environments
- Implement a blue-green deployment strategy
- Monitor deployment health and performance
- Maintain rollback capabilities
- Use environment-specific configurations

## 16. Configuration Management

### Environment Variables

- Never commit secrets to version control
- Use meaningful variable names with consistent prefixes
- Provide example configuration files
- Validate configuration on application startup
- Document all required environment variables

### Best Practices

- Use environment-specific configuration files
- Implement graceful fallbacks for missing configs
- Centralised configuration validation
- Use secure methods for sensitive data storage
- Keep configuration separate from application code

## Conclusion

These coding standards ensure consistency, quality, and maintainability across the Traffic Guardian project. All team members are expected to follow these guidelines to maintain code quality and facilitate collaboration.

For questions about these standards, contact the team leads or create an issue in the project repository.