# The Rusty Crabs Automotive Safety Plan

## Hazard And Risk Analysis

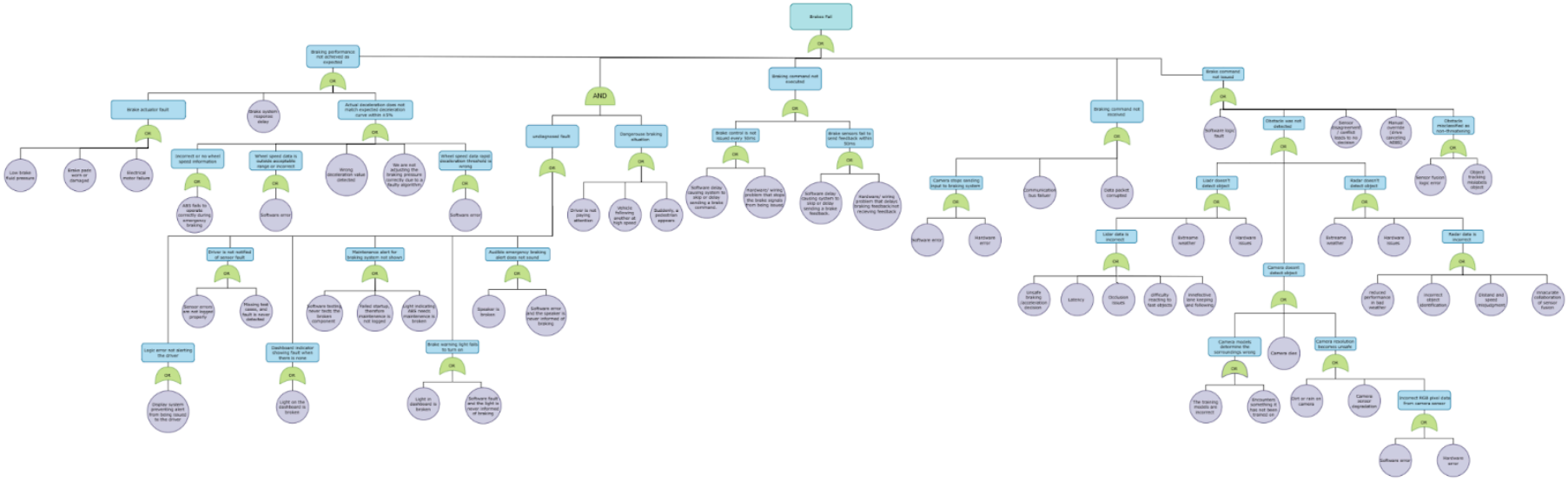
| Hazard (The thing in the system that fails) | Operational Situation ( the scenario or context where it can fail ) | Severity | Exposure | Controllability | ASIL |
|---|---|---|---|---|---|
| Failed to break in time | Vehicle following another at high speed | S3 | E4 | C2 | D |
| | Suddenly, a pedestrian appears | S3 | E3 | C2 | C |
| Wheel speed sensor senses skidding | If there is rain/mud | S2 | E3 | C2 | B |
| | If there is ice | S3 | E2 | C3 | C |
| | If there is gravel | S2 | E2 | C1 | QM |
| Incorrect or no wheel speed information | ABS fails to operate correctly during emergency braking | S2 | E4 | C2 | B |
| Radar failing | extreme weather conditions | S3 | E4 | C2 | D |
| | hardware issue | S3 | E3 | C2 | C |
| Radar data | incorrect detection of objects | S2 | E3 | C2 | B |
| | Distance and Speed Misjudgments | S2 | E4 | C2 | C |
| | Reduced Performance in Adverse Conditions: Weather | S3 | E3 | C2 | C |
| | Inaccurate calibration of Sensor Fusion | S2 | E2 | C1 | |
| Lidar Data | Occlusion issues | S2 | E3 | C2 | B |

| | | | | | |
|---|---|---|---|---|---|
| | Difficulty to reacting to rapidly moving obstacles | S3 | E4 | C2 | D |
| | Unsafe braking and acceleration decisions | S3 | E4 | C2 | D |
| | Ineffective lane keeping and following | S2 | E3 | C2 | B |
| | Latency | S3 | E3 | C2 | C |
| Object not detected | Vehicle fails to detect a car in front and doesn't initiate braking | S3 | E4 | C2 | C |
| The camera resolution goes below the lowest allowed resolution | Dirt or rain on the camera | S3 | E3 | C2 | C |
| | hardware error | S2 | E2 | C2 | A |
| Camera stops sending input to braking system | software error | S3 | E2 | C3 | C |
| | hardware error | S3 | E2 | C3 | C |
| The camera models determines the surroundings wrong | majority of the models determine the surroundings wrong | S3 | E2 | C2 | B |
| | encounters something the model has never seen before | S2 | E2 | C2 | A |
| Corrupted or incorrect RGB pixel data from the camera sensor | Hardware error | S3 | E2 | C3 | C |
| | Software error | S3 | E2 | C3 | C |
| Wheel speed data rapid deceleration threshold is wrong | Software error | S3 | E4 | C3 | D |
| Wheel speed data is | Software error | S3 | E2 | C3 | C |

| | | | | | |
|---|---|---|---|---|---|
| outside acceptable range or inaccurate | | | | | |
| Brake warning light fails to turn on | The light on the dashboard is broken | S2 | E3 | C1 | A |
| | There is a software fault and the light is never informed of braking | S2 | E2 | C1 | QM |
| audible emergency braking alert does not sound | The speaker is broken | S2 | E3 | C1 | A |
| | There is a software fault and the speaker is never informed that we are braking | S2 | E2 | C1 | QM |
| Maintenance alert for braking system not shown | Failed startup, therefore maintenance is not logged | S2 | E3 | C2 | B |
| | The light indicating the ABS needs maintenance is broken | S2 | E3 | C2 | B |
| | The software testing never checks the component that is broken | S2 | E2 | C2 | A |
| Dashboard indicator is stuck showing a fault when there is none | The light on the dashboard is broken | S1 | E3 | C1 | QM |
| Driver is not notified of a sensor fault | Sensor errors are not logged properly | S3 | E3 | C2 | C |
| | We are missing test cases and the fault with the error is never discovered | S3 | E3 | C2 | C |
| Brake control is not issued every 50 ms as | Software delay causing system to skip or delay sending a brake command. | S3 | E2 | C2 | B |

| | | | | | |
|---|---|---|---|---|---|
| required | | | | | |
| | Hardware/ wiring problem that stops the brake signals from being issued | S3 | E3 | C2 | C |
| Brake sensors fail to send feedback within 50 ms | Software delay causing system to skip or delay sending braking feedback. | S3 | E2 | C2 | B |
| | Hardware/ wiring problem that delays braking feedback/not receiving feedback | S3 | E3 | C2 | C |
| Actual deceleration does not match expected deceleration curve within ±5% | Wrong deceleration value detected | S3 | E3 | C3 | D |
| | We are not adjusting the braking pressure correctly due to a faulty algorithm | S3 | E1 | C3 | B |
| Logic error not alerting the driver | Display system preventing alert from being issued to the driver | S3 | E2 | C3 | C |

## Fault Tree Analysis



## Safety goals

- The AEBS shall prevent unintended failure to apply braking in the presence of an obstacle
- The AEBS ensures accurate and timely data to enable braking control logic
- The AEBS alert driver of any changes

# Requirements Management Plan

**Collection** - How we are going to collect our requirements

Requirements will be gathered from the official requirements document and supplemented by findings from our Hazard Analysis and Risk Assessment (HARA). This ensures both predefined and safety-critical requirements are captured.

**Categorization** – categorising them based on their role in the system

Requirements will be categorised based on their role within the Automatic Braking System. Categories will include system components such as the Driver Interface, Radar, and Lidar. Additionally, we will break down requirements into:

- High-level functional requirements
- Low-level functional requirements

This will ensure that there is clear distinction between broad system goals and specific details for our implementation.

**Prioritization**

The MosCow technique (Must, Have, Should have, Could have, Won't have) will guide Prioritization based on importance and project constraints.

We will also apply the ASIL(Automotive Safety Integrity Level) scale (from A to D) to help determine urgency and criticality.

**Tracing** – How and where are we implementing these requirements in our project

A bidirectional traceability matrix will be implemented to ensure that each requirement can be traced through to our testing and implementation and vice versa. Our traceability matrix will include:

- **Requirement ID:** Unique identifier that helps reference each requirement easily
- **Requirement description:** Brief explanation of what the requirement entails
- **Implementation Module:** The code module, class, or component where the requirement is implemented
- **Test case ID:** Unique identifier that helps link each requirement to the test cases which verify they have been correctly implemented
- **GitLab issue:** The issue or task number that tracks the progress or discussion related to this requirement in GitLab

Having clear traceability ensures that there is a clear correlation between our requirement definitions to implementation and testing. Tables will be keyed in both directions

**Change Management** – the process by which we will change our requirements and how we will document these changes

A structured process for managing changes will be implemented:

- Any change will be logged in a ‘Requirements Change Log, including the Requirement ID, description of the change, and justification.
- A GitLab issue will be created for each change, to be reviewed and approved by the team before implementation.
- Previous versions of changed requirements will be archived in a ‘Previous Requirements Table’, date it was changed.
- All impacted items, (e.g., test cases, implementation references) will be updated accordingly in the traceability matrix.

**Verification** – how we will verify that the requirements have been met

Verification will be achieved by mapping each requirement to their corresponding test cases via the traceability matrix. Testing will then also be performed in accordance with the Verification Plan. This will help ensure that each requirement is validated through proper test coverage.

# Configuration Management Plan

**Integrated Development Environment and Coding Language**
The chosen integrated development environment is VSCode. VSCode is lightweight and is usable for all languages. The language we have chosen for development is Rust as it is safety critical due to its explicit content.

The coding standard called "The Power of Ten" from David J. Pearce will be used for code consistency.

The static analysis tools used will be those made for Rust. Clippy will be used. It needs to be added and run in the terminal.

**Version Control System**
Git is the version control system that will be used. Git is a full version control system allowing team members to work independently on the same code base. It can be used and controlled through the command line and is fully integrated into Gitlab. Gitlab is our code base host, which creates an easy way to implement our version control system into our code base.

We will be installing the latest version of git from https://git-scm.com/downloads. This will be done through the command line.

We will then set our global user name and email to identify who made what commits. We will set up an SSH key pair to create a secure connection between each person's host device and GitLab.

**Branching Strategy**
A Gitlab issue will be made for each code feature that will be developed. A Gitlab issue contains a title and description to describe what coding feature/bug fix/test needs to be created. Our Gitlab issues will also include the requirement that it originated from to increase traceability and clarity. Issues can also be linked to branches to show the purpose of the created branch.

A branch is a branched-off section of code that can be used to develop in isolation. For every new feature/bug fix/test, a new branch will be created after writing the Gitlab issue. Each branch will use the same naming standard of feature/<feature-name>, test/<test-name> or bugfix/< bug-name>. This will help with traceability and reverting back to a previous branch if needed. We will also be committing frequently to the branches.

A commit is a checkpoint that takes a snapshot of the current code. This creates the ability to go back to a previous checkpoint/version of the branch. A commit naming convention of <type>:<description> will be used. The main types used will be feat, or fix.

A merge request is the request to merge one branch into another. We will have two main branches.

**Develop branch**: Used throughout development. Each develop branch will be named with a version number to indicate the current version of the system.

**Main branch**: Used for product-ready code.

After the feature/bug-fix/test branch has finished development for its specific purpose, a merge request can be made with the develop branch. All code through development will be added to the develop branch until it is ready to be released. The merge request will need to be reviewed by another team member to prevent unready code from going to production. The reviewer can provide feedback for them to implement before allowing the merge. After the current whole component is finished and tested, the develop branch can be merged with main to release.

# Verification Plan for Autonomous Vehicle Software

## Code-Level Verification

This category focuses on verifying the internal correctness, reliability, and maintainability of software components.

### White-Box Testing

### Unit Testing

- Validate each newly implemented function or module in isolation.
- Inputs initialized with representative values.
- Assertions used to verify correctness of outputs.
- Unused logic and redundant variables identified and removed.

#### CI/CD Integration

- GitLab (or similar) pipelines to:
    - Automatically run unit tests on each commit/pull request.
    - Prevent broken or non-compiling code from being merged.

#### Code Coverage Analysis

- Ensure high **statement** and **branch coverage**.
- Minimum thresholds enforced through CI checks.

#### Mutation Testing

- Inject small logic changes (mutants) to verify test suite robustness.
- Ensure tests detect deviations and maintain fault sensitivity.

#### Code Quality Standards

- Follow **Power of Ten rules**:
    - Limit control complexity.
    - Avoid dynamic memory and deep nesting.
    - Maintain predictable, simple flow.

## Component and Subsystem-Level Functional Testing

This layer tests core vehicle modules such as perception, decision-making, and control.

#### Functional Testing

#### Purpose

- Validate the implementation of system-level and user-facing functionalities.

#### Scope

- Derived from:
    - System requirements.
    - Use cases.

    - HARA safety goals.
    - Scenario libraries (OpenSCENARIO, Euro NCAP).

**Component-Level Functional Testing**

- Test perception, planning, control modules independently.
    - Example: Feed labeled sensor data to object detection; verify expected classifications.

**System-Level Functional Testing**

- Evaluate real-world driving behavior in both **simulated** and **controlled** environments.
    - Example: Emergency stop when obstacle appears.

**Execution Environments**

- **SIL** (Software-in-the-loop): Early algorithm validation.
- **HIL** (Hardware-in-the-loop): Integration with actual ECUs and simulated sensors.
- **Simulations**: CARLA, LGSVL, PreScan for repeatable virtual scenarios.
- **Closed-Track Testing**: Real vehicle behavior validation under safe but realistic conditions.

**Traceability**

- Every functional test maps to a requirement or safety goal.
- Ensures ISO 26262 compliance and functional completeness.

## Safety Verification via HARA-Derived Tests

Focused on validating safety goals against identified hazards.

## Hazard-Based Test Derivation

**Step 1: Hazard Analysis (HARA)**

- Identify scenarios like sensor failure, unexpected behavior.
- Rate hazards by Severity (S), Exposure (E), Controllability (C).
- Assign **ASIL** levels and derive **safety goals**.

**Step 2: Safety Requirements**

- Translate safety goals into technical safety requirements.
- Allocate mitigation strategies to hardware/software.

**Step 3: Test Generation**

- Ensure the system:
    - Prevents or mitigates hazard.
    - Enters safe state on fault (fail-safe or fail-operational).
    - Triggers alerts, fallback behavior (e.g., automatic braking).

## System-Wide Black-Box and Non-Functional Testing

These methods assess external behavior, user experience, and robustness of the entire system.

## Black-Box Testing

### Independent Validation

- Tests developed or reviewed by team members uninvolved in implementation.

### Use Case Testing

- Based on operational scenarios and HARA safety assessments.

### Equivalence Partitioning

- Test representative inputs from valid and invalid partitions.

### Boundary Value Analysis

- Test inputs near valid range limits to uncover edge-case failures.

## Non-Functional Testing

### Performance Testing

- Test real-time responsiveness and latency under normal loads.

### Stress Testing

- Simulate resource starvation, sensor failure, or communication lag.

#### Load Testing

- Validate behavior under concurrent usage (e.g., multi-sensor input).

#### Usability Testing

- Evaluate Human-Machine Interface (HMI) for clarity and ease of use.

#### Security Testing

- Assess communication/data integrity and resistance to tampering.

## Verification Priorities

The verification plan will prioritize the following aspects:

1. **Unit Testing** – Foundation for reliability; validates individual components.
2. **Code Coverage** – Ensures thorough testing and minimizes untested logic paths.
3. **CI/CD Integration** – Guarantees only tested and compilable code is merged.
4. **Performance Testing** – Critical for real-time responsiveness and passenger safety.
5. **Stress Testing** – Validates robustness under extreme or unexpected conditions.