System Verification and Validation Plan for ESF

Alaap Grandhi

February 27, 2025

Revision History

Date	Version	Notes
February 23	1.0	Initial version of the VnV Plan

Contents

This document will

1 General Information

1.1 Summary

The ESF project will involve constructing a library that enables users to both train and test multi-sensor 3D object detection networks on public autonomous driving datasets. Expanding upon the capabilities of the OpenPCDet repository, it will serve as a foundation for further research into the field by presenting a method for LiDAR-Camera fusion-based object detection that better addresses alignment issues faced by prior methods.

The training and inference (testing) portions of ESF correspond to two different use-cases that require different considerations. During training, the library is intended to allow an experienced machine learning and perception engineer to train a LiDAR-Camera fusion network on an autonomous driving dataset of their choosing. During inference, the library is intended to allow a user or a greater system to predict the set of dynamic objects in an autonomous vehicle's surroundings using a trained model and sensor input.

1.2 Objectives

One of the two primary objectives is to build confidence in the correctness of the software. As a routinely used and validated open-source repository, the base functionality of the OpenPCDet repository is considered to be error-free and thus its correctness is outside the scope of this document/project. The installation of the software and the extensions made to it are the components for which correctness will be verified.

Alongside correctness, verifying the accuracy of the solution relative to existing baselines implemented in OpenPCDet is the second primary objective. Since the software is intended to serve as a foundation for future research, its relative accuracy compared to state-of-the-art baseline methods will serve as a metric to judge how much it contributes to research in the field.

A secondary objective is ensuring the usability of the software. Since the library will be extended and used for research purposes, it is important to minimize the effort needed to understand and modify the code as needed.

1.3 Relevant Documentation

The other relevant design documents as well as their relations to this document are listed as follows:

- SRS (Grandhi (2025)): Describes the problem the software solves, the software's goals, the relevant theoretical models, and the requirements for potential solutions. This document thereby establishes the setting and mathematical tools necessary for the validation and verification methods described in this document.
- More documents will be added as they are completed.

2 Plan

This section describes the approaches that will be taken to verify and validate the different documents and steps involved in the design process. Namely, it will state the members of the VnV team and will describe the steps that will be taken to verify the SRS, design, VnV plan, implementation. It will also describe the automating testing tools that will be used.

2.1 Verification and Validation Team

Name	Role	Description
Alaap Grandhi	Author	Write the VnV plan, carry out the testing detailed in it, fill in the VnV report, and validate the software and documents against the requirements during development.
Spencer Smith	Project Supervisor/Course Instructor	Review and provide feedback on the VnV plan and report. Additionally, provide insight into better testing methods.
Matthew Giamou	Masters Supervisor	Provide advice on the project in general.
Bo Liang	Domain Expert Reviewer	Review and provide feedback on the VnV plan and report.

2.2 SRS Verification Plan

The SRS will be reviewed in two stages. First, the Project Supervisor and the Domain Expert Reviewer will evaluate the document using the SRS checklist from lecture (cite here). Since they are familiar with the document structure, they will be able to provide feedback on how well the document adheres to the template and the required structure (in addition to providing feedback on document content).

Following the creation of a series of github issues corresponding to feedback on the document, the second stage of SRS reviewing will begin. At that point, a meeting will be set up with my Masters supervisor to go over the scientific principles discussed in the SRS document alongside the suggested changes and questions from the issues on github. Since he is familiar with my research he can provide additional feedback on any technical inconsistencies or ambiguities in the document.

2.3 Design Verification Plan

To verify the design, I have constructed the following checklist detailing the aspects of the design that the Domain Expert Reviewer and optionally the Masters and Project Supervisors (if they have time) will evaluate during regular discussions:

- Does the design explicitly or implicitly address all requirements described in the SRS document?
- Are the interfaces between modules of the design clear and in-line with the modular structure of the OpenPCDet repository?
- Given the overall design description, can the Domain Expert Reviewer accurately roughly predict the downstream code structure?
- Are any aspects of the design ambiguous?
- Does the design leverage existing functionality from the OpenPCDet repository where applicable?
- Do all of the design components contribute towards meeting a requirement?
    - If not, is this due to a redundancy in the design or something missing in the SRS document?

2.4 Verification and Validation Plan Verification Plan

The VnV plan will be verified in two parts corresponding to the two different types of tests it contains.

Where possible, tests governed by automated test suites like pytest will be tested using mutation testing to ensure that these tests sufficiently cover coding mistakes that are expected to happen. The specific 'mutations' to be inserted into the code will be decided after the implementation is done.

Manual tests and the overall structure of the VnV plan will be reviewed by the Project Supervisor and the Domain Expert Reviewer. This will be done to both ensure that the document adheres to the template and to ensure that the tests presented sufficiently cover the requirements described in the SRS document.

2.5 Implementation Verification Plan

The implementation will simply be verified through semi-regular code walk-throughs with my Domain Expert. Additionally, the tests described in Section (insert section) will serve as a basis for verifying that the implementation meets the requirements detailed in the SRS document.

2.6 Automated Testing and Verification Tools

There are a few different automated tools that will be used for testing. Since the library will be written in Python (the OpenPCDet library is written in Python), PyFlakes will be used as an error linter to discover simple logical errors. Linters like flake8 that enforce style convention will not be used as the base OpenPCDet repository does not adhere to any particular style convention (it is designed with research and quick development as the primary objective) and it would be out of scope to refactor that. Additionally, the automated unit tests below will be implemented in pytest and will be set to automatically run on push using github actions. Code coverage will not be considered for this project since by nature different components are meant to be called according to the training configuration passed in and it would overly tedious to routinely test over all possible input configurations.

2.7 Software Validation Plan

Since requirement (insert requirement) specifically involves comparing the accuracy of the solution to a baseline on a publicly available dataset, validation is already implicitly conducted for the library. Considering that prior works in the field typically use mAP as a measure of solution quality, further validation is out of scope for this project due to the limited timeline. Validation of the requirements themselves will only briefly be covered during discussions with my Masters Supervisor.

3 System Tests

This section details a set of tests that shall be sufficient to ensure that the proposed solution meets the requirements detailed in the SRS document. Additional tests may be added as the SRS document is refined.

3.1 Tests for Functional Requirements

The subsections below detail the tests that will be used to ensure that the solution meets the functional requirements.

3.1.1 Input Format Testing

These tests will ensure that in inference, the system is able to process all the necessary file formats for the camera images and LiDAR pointclouds.

Camera Image Input Testing

1. test-camera-valid1→4

   Control: Automatic

   Initial State: The inference pipeline initialized with a pre-defined trained model and configuration loaded. A clean, conforming LiDAR pointcloud in the TFRecord format is also expected to have already been loaded.

   Input: A valid (subject to input constraints) multiview camera image set stored in one of the file formats specified in requirement (insert requirement) (one test case for each).

Output: The inference pipeline should run as expected without errors and output a set of predicted bounding boxes.

Test Case Derivation: For all the file formats specified in requirement (insert requirement), the inference pipeline should run without issues. All other aspects are kept constant between the tests to ensure that the test can detect any issues with specific image file formats.

How test will be performed: This test will be automatically run through pytest. The configuration, pretrained model, LiDAR pointcloud will all be manually generated beforehand and then automatically accessed by pytest. The same set of valid multiview camera images will simply be converted to the different file types beforehand to be automatically accessed by pytest on github push.

2. test-camera-invalid

   Control: Automatic

   Initial State: The inference pipeline initialized with a pre-defined trained model and configuration loaded. A clean, conforming LiDAR pointcloud in the TFRecord format is also expected to have already been loaded.

   Input: A valid (subject to input constraints) multiview camera image set stored in a file format that is not expected to be handled by the program (a binary image).

   Output: A invalid-input error is expected to be thrown by the program.

   Test Case Derivation: The inference pipeline should not even try to run for any file format not specified in requirement (insert requirement). This is to test for cases where images are input in unforeseen or invalid formats, and makes sure that the system's response to such input types is well-defined.

   How test will be performed: This test will be automatically run through pytest on github push in much the same way as the valid tests. The only difference is that it will load a set of multiview camera images stored in a binary format.

LiDAR Pointcloud Input Testing

1. test-lidar-valid1→4

   Control: Automatic

   Initial State: The inference pipeline initialized with a pre-defined trained model and configuration loaded. A clean, conforming set of multiview camera images in the PNG format is also expected to have already been loaded.

   Input: A valid (subject to input constraints) LiDAR pointcloud stored in one of the file formats specified in requirement (insert requirement) (one test case for each).

   Output: The inference pipeline should run as expected without errors and output a set of predicted bounding boxes.

   Test Case Derivation: For all the file formats specified in requirement (insert requirement), the inference pipeline should run without issues. All other aspects are kept constant between the tests to ensure that the test can detect any issues with specific image file formats.

   How test will be performed: This test will be automatically run through pytest. The configuration, pretrained model, multiview camera image set will all be manually generated beforehand and then automatically accessed by pytest. The same LiDAR pointcloud will simply be converted to the different file types beforehand to be automatically accessed by pytest on github push.

2. test-lidar-invalid

   Control: Automatic

   Initial State: The inference pipeline initialized with a pre-defined trained model and configuration loaded. A clean, conforming set of multiview camera images in the PNG format is also expected to have already been loaded.

   Input: A valid (subject to input constraints) LiDAR pointcloud stored in the file format specified in table x for the given test case.

   Output: A invalid-input error is expected to be thrown by the program.

   Test Case Derivation: The inference pipeline should not even try to run for any file format not specified in requirement (insert requirement).

This is to test for cases where pointclouds are input in unforeseen or invalid formats, and makes sure that the system's response to such input types is well-defined.

How test will be performed: This test will be automatically run through pytest on github push in much the same way as the valid tests. The only difference is that it will load a LiDAR pointcloud stored in a binary format.

3.1.2 Tests for Correct Optimization

The test below will ensure that the proposed solution is being optimized correctly.

Model Training Consistency Testing

1. test-consistent

   Control: Manual

   Initial State: A set of pre-configured baseline methods will be set up for training. Additionally, the proposed method built on the OpenPCDet platform will be set up for training.

   Input: A sufficiently small training dataset that has been predetermined to produce similar training accuracies across the baseline methods (corresponding to an mAP standard deviation below a threshold that will be decided on with my Domain Expert).

   Output: When all the models are trained and subsequently evaluated on the same small training dataset, the proposed method shall not fall below the mean mAP of the baseline methods by more than a given absolute threshold that will be decided on with my Domain Expert.

   Test Case Derivation: There is no true way to test if the combined bounding box is exactly minimized considering the randomness inherent in machine learning techniques and floating point computation. Rather, since the baselines are assumed to roughly minimize the bounding box loss on the training dataset, consistency with their results should be sufficient for verifying this. Additionally, by evaluating on

the same dataset that was trained on, we can ensure that the ADAM optimization is minimizing the loss for the training subset.

How test will be performed: This test will be manually run due to the time-intensive nature of model training. Moreover, it will not be regularly run but rather only run when major changes to the proposed solution are made. It will also not be run until the solution is in a trainable state as a whole and the other tests have all passed.

Note that since ADAM will be directly imported from pytorch's implementation, its validity will not be verified. Rather, the code walkthroughs mentioned above in section (insert section) will aid in verifying that it has been used correctly.

3.1.3 Tests for Visualization

The below test will verify that the visualization produced during the inference phase is correct.

Visualization Testing

1. test-viz-correct

   Control: Automatic

   Initial State: Any components aside from the inputs that are needed for the visualization pipeline to run.

   Input: A set of pre-configured bounding boxes will be loaded alongside a predetermine LiDAR pointcloud. Additionally, a set of viewing positions and resolutions to view the produced pointclouds from.

   Output: The output pointcloud visualizations with bounding boxes should look identical to pre-generated verified visualizations of the same scene and bounding boxes. These pre-generated visualizations will be created in the future and carefully examined to ensure correctness before setting up the test.

   Test Case Derivation: This is really just using the method of manufactured solutions to verify that the visualization pipeline is working as expected without bugs. A correct and matching visualization here

does not guarantee a correctly functioning visualization pipeline but it is a necessary condition.

How test will be performed: This test will automatically be run through pytest with the LiDAR pointcloud, bounding boxes, and verified visualizations being generated beforehand. Matplotlib.testing.compare.compare_images will be run to compare each generated visualization with the corresponding verified visualization.

3.2 Tests for Nonfunctional Requirements

The subsections below will detail the tests that will ensure that the proposed solution meets the nonfunctional requirements.

3.2.1 Tests for Accuracy

The test below will ensure that the proposed solution is at least relatively as accurate as the BEVFusion baseline.

Title for Test

1. test-mAP

   Type: Manual

   Initial State: The BEVFusion model will be loaded with the BEVFusion configuration currently present in the OpenPCDet repository. Additionally, the proposed solution will be loaded with some chosen configuration.

   Input/Condition: A training and validation set from the nuScenes dataset.

   Output/Result: After both models have been trained on the nuScenes training set, their mAP metrics on the validation set will be returned. The proposed solution's validation mAP should not fall below the BEVFusion mAP by more than a threshold that will be determined with my Domain Expert.

   How test will be performed: Since model training is time-intensive, this test will mainly be conducted whenever major changes to the proposed

solution are made. The threshold will only serve as a guideline and after each run of this test, a discussion will be done with my Domain Expert and Masters Supervisor to determine if the result is satisfactory.

While Waymo could be tested for as well, it is a much larger dataset that would take quite long to train on. Moreover, evaluating this test for the nuScenes dataset alone should provide almost as much insight as evaluating on both (since in existing research, better performing models tend to do better across both datasets).

3.2.2 Tests for Understandability

Rather than use a fixed test for understandability, it will be tested closer to the end of the project through code walkthroughs. Specifically, I will provide a demonstration of how to use the library to my Domain Expert and optionally Project and Masters supervisors (if they have time). I will ask them the following questions afterwords to gauge how easy to understand the code is.

1. How do you think the code works at a high level?
2. If you had to describe what x (to be determined after implementation) class of submodules does, how would you do so?
3. If you had to learn how to use this code with nothing but the documentation present in the repository and the code itself, how long do you think it would take you to do so?

Afterwords, I will test further test their understanding of the code by letting them play around with it and getting them to perform the following tasks with assistance as necessary.

1. Train a model using x (to be determined after implementation) configuration.
2. Evaluate x (to be determined after implementation) pretrained model on the nuScenes validation dataset.
3. Modify the configuration of x (to be determined after implementation) model to have x (to be determined after implementation) effect.

Through the code walkthrough and their subsequent interaction with it, I hope to get at least a rough idea of how understandable the code is.

3.3 Traceability Between Test Cases and Requirements

[Provide a table that shows which test cases are supporting which requirements. —SS]

4 Unit Test Description

[This section should not be filled in until after the MIS (detailed design document) has been completed. —SS]

[Reference your MIS (detailed design document) and explain your overall philosophy for test case selection. —SS]

[To save space and time, it may be an option to provide less detail in this section. For the unit tests you can potentially layout your testing strategy here. That is, you can explain how tests will be selected for each module. For instance, your test building approach could be test cases for each access program, including one test for normal behaviour and as many tests as needed for edge cases. Rather than create the details of the input and output here, you could point to the unit testing code. For this to work, you code needs to be well-documented, with meaningful names for all of the tests. —SS]

4.1 Unit Testing Scope

[What modules are outside of the scope. If there are modules that are developed by someone else, then you would say here if you aren't planning on verifying them. There may also be modules that are part of your software, but have a lower priority for verification than others. If this is the case, explain your rationale for the ranking of module importance. —SS]

4.2 Tests for Functional Requirements

[Most of the verification will be through automated unit testing. If appropriate specific modules can be verified by a non-testing based technique. That can also be documented in this section. —SS]

4.2.1 Module 1

[Include a blurb here to explain why the subsections below cover the module. References to the MIS would be good. You will want tests from a black box

perspective and from a white box perspective. Explain to the reader how the tests were selected. —SS]

1. test-id1

   Type: [Functional, Dynamic, Manual, Automatic, Static etc. Most will be automatic —SS]

   Initial State:

   Input:

   Output: [The expected result for the given inputs —SS]

   Test Case Derivation: [Justify the expected value given in the Output field —SS]

   How test will be performed:

2. test-id2

   Type: [Functional, Dynamic, Manual, Automatic, Static etc. Most will be automatic —SS]

   Initial State:

   Input:

   Output: [The expected result for the given inputs —SS]

   Test Case Derivation: [Justify the expected value given in the Output field —SS]

   How test will be performed:

3. ...

4.2.2 Module 2

...

4.3 Tests for Nonfunctional Requirements

[If there is a module that needs to be independently assessed for performance, those test cases can go here. In some projects, planning for nonfunctional tests of units will not be that relevant. —SS]

[These tests may involve collecting performance data from previously mentioned functional tests. —SS]

4.3.1 Module ?

1. test-id1

   Type: [Functional, Dynamic, Manual, Automatic, Static etc. Most will be automatic —SS]

   Initial State:

   Input/Condition:

   Output/Result:

   How test will be performed:

2. test-id2

   Type: Functional, Dynamic, Manual, Static etc.

   Initial State:

   Input:

   Output:

   How test will be performed:

4.3.2 Module ?

...

4.4 Traceability Between Test Cases and Modules

[Provide evidence that all of the modules have been considered. —SS]

References

Alaap Grandhi. System requirements specification. https://github.com/alaapgrandhi/equivariant-sensor-fusion/tree/main/docs/SRS/SRS.pdf, 2025.

5 Appendix

This is where you can place additional information.

5.1 Symbolic Parameters

The definition of the test cases will call for SYMBOLIC_CONSTANTS. Their values are defined in this section for easy maintenance.

5.2 Usability Survey Questions?

[This is a section that would be appropriate for some projects. —SS]