# Vision Inspection Pipeline: Improvement and Benchmarking Plan

## Overview and Goals

**Role & Mission:** As the Lead ML Systems Engineer and Technical Program Manager (TPM), our mission is to **enhance the Vision Inspection Pipeline (VIP)** for detecting defects in plastic containers. We will introduce a **reproducible benchmarking harness** and implement targeted improvements to the image processing pipeline, addressing current shortcomings while **avoiding any changes to the API or web interface**. This plan balances classical computer vision techniques with robust software engineering practices to ensure the pipeline meets thesis project acceptance criteria. The ultimate goal is to reduce false positives, increase detection confidence, handle background and glare issues, and provide a framework for **evaluating improvements in a reproducible manner**.

**Scope:** We focus on **classical image processing and heuristic methods** (no deep learning models) for defect detection, aligned with the pre-thesis scope. The improvements will be implemented incrementally, with thorough testing, benchmarking, and version control adherence. Code changes will be **minimal and additive**, preserving the current repository structure and avoiding disruptions to existing APIs or the front-end.

## Challenges in the Current Pipeline

The current VIP pipeline uses rule-based detectors for defects (scratches, contamination, discoloration, cracks, flash) with techniques like edge detection (Canny/Sobel), thresholding, and morphology. While this approach is fast and interpretable, several issues have been observed:

* **False Positives:** The detectors sometimes flag benign features as defects, likely due to noise, background textures, or lighting artifacts. For example, background patterns or reflections can be mistakenly identified as scratches or contamination.
* **Low Confidence Levels:** Many detections have low confidence scores, indicating the algorithms are often uncertain. This may stem from weak feature differentiation—e.g. a faint scratch edge might barely cross the Canny threshold, or a slight discoloration might hover near the color difference cutoff. Low confidence results make it hard to reliably accept detections.
* **Background Interference:** The imaging setup may include background elements (e.g. conveyor, fixtures) that confuse the detectors. A lack of explicit background removal means the pipeline might pick up **irrelevant edges or spots** in the background, contributing to false positives and inconsistent results.
* **Glare on Shiny Surfaces:** Plastic containers often have specular highlights or glare under strong lighting. These bright spots can trigger the defect detectors (especially discoloration or contamination algorithms) or obscure actual defects. Glare leads to both missed defects (true flaws masked by bright reflection) and false alarms (reflection identified as a defect region).

To fulfill the thesis acceptance criteria, we must mitigate these issues. The following sections outline **improvement strategies** to address each challenge, followed by a detailed plan to implement a benchmarking harness for evaluating these enhancements.

## Improvement Strategies for the Pipeline

To enhance defect detection while adhering to classical computer vision (no deep learning), we will implement several techniques drawn from recent research and industry practices. These strategies focus on better isolating true defect signals and filtering out noise:

* **Background Removal:** Introduce a background subtraction or masking step to eliminate static background regions from analysis. By removing background texture, the detectors can focus on the container itself, reducing spurious detections. Background subtraction has been shown to **effectively eliminate background interference and highlight defect areas**, improving detection performance[[1]](https://www.mdpi.com/2072-666X/14/5/905#:~:text=background%20subtraction%20algorithm%20measures%20the,The)[[2]](https://www.mdpi.com/2072-666X/14/5/905#:~:text=match%20at%20L935%20improved%20the,area%20defects%20such%20as%20stains). We will likely use a reference background image or a running average model to subtract the backdrop, or simply mask known non-product regions. This will directly target the false positive issue by **preventing detectors from “seeing” the irrelevant background**. Notably, adding background subtraction in a defect inspection system improved mAP by ~3.4% in a recent study[[3]](https://www.mdpi.com/2072-666X/14/5/905#:~:text=improved%20the%20mAP%20by%203.4,area%20defects%20such%20as%20stains), underlining its value in our context.
* **Adaptive Region Growing:** Enhance the detection algorithms by incorporating an **adaptive region growing method** to capture entire defect areas after initial edge or spot detection. For instance, once a potential scratch or crack edge is found, region growing can expand the region along continuous edges or similar intensity, ensuring the full defect is marked. Research has shown that combining edge detection with adaptive region growth yields a robust binary mask of defect areas[[4]](https://www.degruyterbrill.com/document/doi/10.1515/nleng-2022-0356/html?srsltid=AfmBOopm7jxPi5tdLmVj0y887hE5SgJOfrDI88IKED4BmXhx20RgVTbC#:~:text=At%20present%2C%20the%20detection%20of,the%20rapid%20screening%20method%20of). This method can reduce fragmented detections and increase confidence by aggregating more evidence of a defect. It also provides better shape information for later analysis. We will seed the region growing with detected edge pixels or thresholded defect regions, then grow until intensity or color criteria fail – capturing the defect region more completely.
* **Glare Detection and Removal:** Implement a **specular highlight removal** step to handle glare on shiny plastic. This typically involves detecting highlight pixels (often near-saturated white regions) and reducing or removing them in the image used for defect detection. A simple but effective approach is to work in a perceptual color space (e.g. HSV) and identify highlights via intensity thresholds (e.g. mean plus some standard deviation)[[5]](https://www.mdpi.com/2076-3417/14/6/2469#:~:text=In%20existing%20highlight%20removal%20methods%2C,channel%20of%20the%20HSV%20space). Those highlight regions can be downplayed or filled using interpolation so they do not trigger false alarms. By applying methods inspired by recent highlight removal research, we can significantly reduce the **visual disturbance caused by specular reflections**, leading to more accurate defect detection[[6]](https://www.mdpi.com/2076-3417/14/6/2469#:~:text=In%20metal%20images%2C%20most%20pictures,of%20later%20maintenance%20and%20replacement)[[7]](https://www.mdpi.com/2076-3417/14/6/2469#:~:text=The%20existing%20highlight%20removal%20algorithms,%28utilizing%20deep%20learning%20methods). Removing these bright spots will ensure underlying defects aren’t missed and that detectors (like for discoloration or contamination) don’t confuse glare for a defect. We will also consider **detail preservation** after removing highlights – e.g. applying adaptive filtering or sharpening in highlight areas, as suggested by Jiang et al.[[8]](https://www.mdpi.com/2076-3417/14/6/2469#:~:text=Retinex%20model%20and%20generate%20a,validate%20the%20proposed%20method%20using)[[9]](https://www.mdpi.com/2076-3417/14/6/2469#:~:text=restored%20image%2C%20this%20paper%20introduces,quality%20qualitative%20and%20quantitative%20evaluation), to avoid losing any genuine defect information that coincides with glare.
* **Gradient Histogram Analysis (HOG Features):** Introduce **Histogram of Oriented Gradients (HOG)** feature analysis or similar gradient-based descriptors to better characterize defect vs. non-defect regions. HOG is a classic feature descriptor that captures the distribution of edge orientations, which can help differentiate, for example, a true crack (which has a linear, oriented gradient pattern) from random noise or glare (which might have more scattered gradients). HOG (and related descriptors like Local Binary Patterns) have proven extremely effective for crack detection in concrete and other surfaces[[10]](https://www.mdpi.com/2079-9292/11/20/3357#:~:text=Generally%2C%20computer,3%20%2C%20120). In one study, using HOG features combined with an SVM classifier achieved over 99% accuracy distinguishing cracks from non-cracks[[11]](https://www.mdpi.com/2079-9292/11/20/3357#:~:text=dimensionality%20reduction%20techniques%20are%20applied,the%20best%20results%20as%20an). While we will not train complex models, we can leverage HOG in simpler ways: e.g. calculate a confidence score for a detected region based on how “crack-like” or “scratch-like” its gradient histogram is. This could serve as an additional filter to **boost confidence for real defects and reject false positives**. We will keep this integration modular – possibly as an optional analysis step in each detector that can flag low-confidence detections for review (or feed into a human-in-the-loop mechanism described later).
* **Improved Thresholding and Morphological Filtering:** Besides the major additions above, we will revisit the parameter tuning of each detector. This includes **adaptive thresholding** (vary thresholds based on image brightness or use Otsu’s method where applicable) and refined morphological operations to clean up detections. For example, we can use *size filtering* to ignore tiny specks that are below the minimum defect size, and *shape validation* (using aspect ratio, circularity, etc.) to ensure detections match expected defect shapes. Recent conventional defect detection approaches often define shape features or moments to classify spots vs real defects[[12]](https://www.degruyterbrill.com/document/doi/10.1515/nleng-2022-0356/html?srsltid=AfmBOopm7jxPi5tdLmVj0y887hE5SgJOfrDI88IKED4BmXhx20RgVTbC#:~:text=easy%20to%20cause%20false%20detection,of%20Support%20Vector%20Machine%20were). Incorporating some of these heuristics (e.g. a scratch should have a high length-to-width ratio, a contamination spot should be roughly circular, etc.) will further cut down false positives and improve confidence scores.

Each of these improvements will be implemented in a **modular** way (e.g., as preprocessing steps or additional checks in the detector classes) so they can be toggled and combined. The net effect expected is a more reliable pipeline: fewer false alarms, higher confidence on true defects, and robustness to imaging conditions.

## Benchmarking Harness Design

To validate the improvements and ensure the pipeline meets its requirements, we will create a **reproducible benchmarking harness**. This framework will allow easy comparison of different techniques (the current vs improved algorithms, or any future methods) on standardized datasets and metrics, with minimal changes to the repository. The harness will generate consistent evaluation reports, making it easier to quantify gains (e.g. reduction in false positive rate, increase in detection accuracy) and to demonstrate thesis outcomes with evidence.

### Operating Principles

Our benchmarking system will follow these key principles:

* **Repository-Aware & Minimal Intrusion:** Align with the existing code structure. We will avoid large refactors; instead, we add new modules (preferably in a dedicated bench/ directory) for benchmarking. Existing detection code will be reused via adapters, not modified. We won’t shuffle existing files/folders unless absolutely necessary. This ensures current pipeline functionality remains unchanged for the API/UI, and the benchmark tools simply **plug into** the codebase.
* **Reproducible & Traceable:** Every benchmark run should be repeatable and traceable. We’ll fix random seeds for any randomness and log the environment (library versions, OS, hardware) for each run. Each experiment ties to a specific git commit and branch, so results can be traced in version control. Outputs will include references to the exact code used (via commit hash) and configuration, ensuring anyone can reproduce the results given the same code and data.
* **Plugin-Friendly (Extensibility):** The harness will treat defect detection techniques (algorithms) as **plug-ins**. We’ll define a small interface that any new technique (be it a classical method or future ML model) can implement to be benchmarked. No core changes should be required to add a new detection approach; just drop in a new technique module and a config file. This modularity will make it easy to compare the current approach with alternatives (or even simple machine learning baselines, though ML is out-of-scope for now, the harness prepares for future work).
* **Single Source of Truth Outputs:** All results from a run (metrics, per-image outcomes, logs, etc.) will be saved in a structured output folder. We’ll standardize on a format (CSVs, JSONL, etc.) so that analysis and reporting scripts can ingest them. This avoids scattering results or relying on prints/logs; everything needed for the report is in one place per run. The outputs will be timestamped and organized by run ID.

### Repository Structure and Git Flow

Before coding, we will review the repository’s current structure and branching strategy. The likely setup uses a **main** (or master) branch for stable releases and a **develop** branch for integration, with feature branches for ongoing work. We will verify this and ensure our work aligns:

* We will create a feature branch for this benchmarking project, e.g. feat/bench/<short-id> off develop. All our changes will live here until ready.
* The **main** branch remains clean (release-ready), and **develop** will accumulate finished features. We'll merge to develop when our changes are tested, then eventually to main as appropriate.
* We will tag important results or milestones with annotated git tags (e.g. bench-<date>-<run-id>) to mark the state of the code for each benchmark result. This helps track which code produced which result in the long run.

To record experiment metadata, we’ll introduce a simple **benchmark registry file** at the repo root, e.g. bench\_runs.jsonl. Each run appends a line (in JSON format) with details: { run\_id, git\_commit, branch, config, results\_path, timestamp }. This is an append-only log of all benchmark runs executed, enabling traceability of what was run when and where to find the outputs.

**Note:** We will be careful **not to commit large data or output files**. Raw image data and heavy results will stay out of git (we’ll use .gitignore for the results folder). We may commit small derivative artifacts (like tiny sample images, or a summary CSV or markdown report for demonstration) if needed for documentation, but not big binaries. If large outputs need to be preserved, we might integrate with a remote storage bucket or simply leave them in the results directory on the system.

### Benchmark Harness Directory Layout

We will add a new top-level directory **bench/** (if one does not exist) to house all benchmarking-related code. Inside this, the structure will be organized as follows (keeping it lightweight and additive):

/bench/  
 benchcore/ # Core benchmarking framework code  
 adapters/ # Adapters tying pipeline outputs to common format (per task type)  
 techniques/ # Each detection technique as a plugin module  
 eval/ # Evaluation metrics and statistics utilities  
 viz/ # Visualization and reporting helpers  
 utils/ # Utilities (logging, timing, seeding, environment capture, etc.)  
 hitl/ # (Optional) human-in-the-loop hooks for interactive review  
 configs/  
 datasets/ # Dataset definitions (paths, splits, labels)  
 techniques/ # Technique-specific configs (parameters, thresholds, etc.)  
 experiments/ # Experiment configs combining datasets & techniques to run

Additionally, we will create or use existing directories for data and results:

/data/  
 raw/<dataset\_name>/ # Raw datasets (images and annotations as provided)  
 processed/<dataset\_name>/  
 train/, val/, test/ # Processed data splits (if we generate any standardized format or crops)  
 manifest.jsonl # (optional) A combined manifest of all images/annotations for easy loading  
  
/results/ # Benchmark run outputs (this will be git-ignored)  
 <run\_id>/  
 summary.csv # Aggregate metrics for the run  
 per\_image.jsonl # Per-image detailed results and predictions  
 metrics.json # Structured metrics (perhaps similar info as summary, in JSON)  
 figs/ # Any generated figures (plots, etc.)  
 report.md / report.html # Generated report for the run  
 logs/ # Logs or env info if needed (env.txt, pip\_freeze.txt, etc.)

This layout ensures we clearly separate the benchmarking evaluation code from the main pipeline code. We will likely place *thin wrappers* in bench/benchcore/adapters/ to call the detectors from src/vip/detect/\* (the existing detectors) so that we don’t modify those directly. Each new technique we add (e.g. a variant with region growing, or a different algorithm) can live under benchcore/techniques/ in its own subfolder.

We will search the repository for any existing utility code for metrics or plotting (perhaps some might exist for the web visualization). If found, we will **reuse rather than reinvent** – potentially writing wrapper functions to fit the harness interface where needed.

### Data Normalization and Manifest

Consistent evaluation requires a standardized dataset interface. We will implement a **data manifest generator** that consolidates our datasets (and their defect annotations) into a common format:

* A script (e.g. scripts/check\_dataset.py) will read the raw data and annotations for a given dataset and produce a unified **manifest file** (JSONL or CSV). Each entry in the manifest represents one image and includes: an id, split (train/val/test), image\_path, task (type of defect detection or classification), ground truth labels (defect present or not, defect type), and if applicable bounding boxes or masks for defect locations, plus any metadata (e.g. lighting conditions).
* We will not move or alter the raw images (to avoid breaking anything); processed data (like masks or any precomputed results) can be stored under data/processed/<dataset>.

By having a manifest, our benchmarking code can easily load images and ground truth, regardless of original format (it abstracts differences between, say, one dataset that provides annotations in XML vs another in CSV, etc.). This also facilitates checking dataset integrity (our script can validate that files exist, counts match expected, etc.).

### Technique Plugin Interface

In the benchcore/techniques module, we will define a **base class or interface** that all techniques must implement. For example, a BaseTechnique class with methods like:

* setup(device: str): to load any models or perform any initialization (if needed) on a given device (CPU/GPU).
* predict\_batch(images: List[np.ndarray]) -> List[Detections]: to run the defect detection on a batch of images and return results in a standard format.
* teardown(): to free resources if needed (close files, release GPU memory, etc.).

Since our current detectors are all CPU-based and not heavy, setup/teardown may be trivial, but we include them for completeness and future extensibility.

We will also implement **task-specific adapters** in benchcore/adapters/ because the format of a detection can vary: - Classification (e.g., defect present vs not) will produce class labels and maybe confidences. - Object detection (if we treat each defect occurrence with a bounding box) will produce boxes, labels, scores. - Segmentation (if we produce masks for defects) will produce masks.

Our detectors right now output a list of Detection dataclass instances (with fields like label, score, bbox, mask). The adapter will convert these to a **standard dict** format for evaluation, for example:

{"image\_id": ..., "predictions": [ {"label": "crack", "score": 0.95, "bbox": [x,y,w,h], "mask": <mask\_path\_or\_bitmap> }, ... ]}

Similarly, ground truth from the manifest will be standardized:

{"image\_id": ..., "gt\_defects": [ {"label": "crack", "bbox": [...], "mask": ...}, ... ], "gt\_no\_defect": False}

(for an image with no defect, gt\_no\_defect=True or an empty list might indicate that).

By standardizing outputs and ground truth, the evaluation metrics can be computed in a unified way regardless of technique.

Each new technique will live in its own subfolder under benchcore/techniques/. For instance, we might have: - benchcore/techniques/baseline/technique.py for the current rule-based pipeline as one “technique”. - benchcore/techniques/regiongrow/technique.py for an improved version using region growing. - benchcore/techniques/hog\_svm/technique.py if we experiment with an SVM classifier on HOG features (though full ML training is out of scope, we could include a stub technique for classification as a comparison).

In each subfolder, we can also include a config.yaml defining parameters (like thresholds or model file paths) and a README.md citing any relevant paper or source for that technique. This helps maintain clarity on what each technique is and where it came from (important for the thesis documentation). For example, our region growing technique’s README might cite the paper that inspired it[[4]](https://www.degruyterbrill.com/document/doi/10.1515/nleng-2022-0356/html?srsltid=AfmBOopm7jxPi5tdLmVj0y887hE5SgJOfrDI88IKED4BmXhx20RgVTbC#:~:text=At%20present%2C%20the%20detection%20of,the%20rapid%20screening%20method%20of).

### Evaluation Metrics and Statistics

We will incorporate comprehensive metrics to evaluate the defect detection performance. Different tasks require different metrics:

* **Classification Metrics:** If evaluating on a per-image basis (defect present or not, or defect type classification), use accuracy, precision, recall, F1-score (both macro and weighted if multiple classes), as well as more nuanced ones like ROC-AUC (for binary classification of defect vs no defect) and log loss if we have probabilistic output. We will also calculate **Top-K accuracy** (with K=3) if relevant, although likely our use-case is binary presence or multi-class one-vs-all rather than multi-class classification.
* **Detection Metrics:** For defect localization (object detection style), use **COCO-style mean Average Precision (mAP)**. Specifically, mAP at IoU 0.50:0.95 (the average over multiple IoU thresholds) is a strong overall indicator[[2]](https://www.mdpi.com/2072-666X/14/5/905#:~:text=match%20at%20L935%20improved%20the,area%20defects%20such%20as%20stains). We will also report mAP at 50% IoU (sometimes easier to interpret) and at 75% (stricter), as well as Average Recall (AR) and per-class AP. If our defects have size variability, COCO metrics for small/medium/large object AP can be considered (though in our case defects are probably all small/medium). These metrics will quantify how well the detectors are locating the defects and not missing or hallucinating too many.
* **Segmentation Metrics:** If we evaluate segmentation masks (for precise defect shape), use **Intersection over Union (IoU)** for each class and the mean IoU (mIoU) across classes. Also the **Dice coefficient** (equivalent to F1-score for the segmentation pixels) and possibly a boundary F1-score (which focuses on alignment of edges, useful for crack shapes).
* **System Performance Metrics:** We will measure throughput (images processed per second) and latency (per-image processing time, mean and median) to ensure any new techniques are within acceptable performance. We will also log memory usage (peak RAM usage, etc.) and potentially algorithm complexity (though rough, maybe count number of parameters if ML model or just record that classical methods have none). The pipeline currently is real-time enough (~2-5 seconds per image as per documentation); we should ensure our additions (like region growing or HOG computation) don’t degrade performance beyond acceptable limits. By logging these metrics, we can include them in the report to discuss the trade-offs of accuracy vs speed.
* **Statistical Analysis:** To strengthen the evaluation, we will apply statistical tests. For example, use **bootstrap confidence intervals (95%)** for key metrics (like overall accuracy or mAP) to gauge uncertainty[[1]](https://www.mdpi.com/2072-666X/14/5/905#:~:text=background%20subtraction%20algorithm%20measures%20the,The). We can also do a **paired test** on per-image outcomes (e.g., if comparing two techniques, use a Wilcoxon signed-rank test on their per-image error differences) to see if improvements are significant. For metrics like AUC, Delong’s test could assess significance between two ROC curves. These analyses will be included especially if we compare multiple techniques.

We will implement these metrics possibly using existing libraries (e.g. scikit-learn for classification metrics, pycocotools for detection mAP if available) or write our own lightweight versions for full control. The benchcore/eval/ module will house these calculations, possibly with submodules like classification.py, detection.py, segmentation.py for clarity.

### Output Schema and Artifacts

For each benchmark run (an experiment which could include one or multiple techniques on a dataset), the harness will produce standardized outputs:

* **Summary CSV (summary.csv):** This CSV will have one row per combination of dataset split and technique evaluated (for example, if we run 2 techniques on both validation and test sets, it might have 4 rows). Each row will contain:
* Identification: run\_id, timestamp, git commit, branch, dataset name, split name, task type (e.g. classification/detection).
* Technique info: technique name (and version or variant if applicable).
* Dataset statistics: number of images processed, etc.
* Performance metrics: images per second, average latency (ms) per image, median latency, peak memory usage (MB).
* **Classification metrics** (if applicable): accuracy, precision\_macro, recall\_macro, F1\_macro, (and maybe precision\_weighted, recall\_weighted, but macro gives overall picture), top-3 accuracy (if multi-class), ROC-AUC (for binary cases), log loss, ECE (expected calibration error) if we have probability output.
* **Detection metrics** (if applicable): mAP (0.50:0.95), AP@50, AP@75, AR, and possibly AP\_small/medium/large if relevant.
* **Segmentation metrics** (if applicable): mIoU, mean Dice score, boundary F1.
* **Custom defect metrics:** We might include some domain-specific metrics, e.g. number of detections per image on average, or percentage of images with defects found. For example, columns like defects\_count (total defects detected), defects\_pct (percentage of images where defects were detected vs should have), etc., broken down by type (if multiple defect categories).
* Reproducibility info: random seed used, device (CPU/GPU), whether CUDA/cuDNN deterministic mode was enabled (ensuring reproducibility in ML if it were used).
* **Per-Image Results (per\_image.jsonl):** A JSON Lines file with one entry per image. Each entry can store the image ID, maybe the file name, the ground truth labels, and the predictions made by each technique. This is invaluable for debugging (we can later filter this to see which images each technique failed on, etc.). We will also record per-image inference time and maybe confidence scores of predictions. Example entry (conceptual):
* {  
   "image": "data/raw/test/img\_123.jpg",  
   "gt\_defects": [{ "label": "crack", "bbox": [100,50,20,5] }],  
   "predictions": [  
   { "technique": "baseline", "label": "crack", "score": 0.60, "bbox": [95,48,22,6] },  
   { "technique": "improved\_region", "label": "crack", "score": 0.92, "bbox": [98,50,21,5] }  
   ],  
   "latency\_ms": { "baseline": 50.2, "improved\_region": 55.8 }  
  }
* If multiple techniques are run, we either include all in one entry as above or have separate JSONL per technique. The exact schema can be refined, but the goal is to have **detailed data to analyze false positives/negatives on a per-image basis**.
* **Metrics JSON (metrics.json):** This could be a machine-readable summary of the key metrics (similar to the CSV but in JSON form). Possibly not strictly needed if CSV exists, but JSON might be easier to load in some analysis scripts.
* **Figures and Visualizations:** Under results/<run\_id>/figs/ and .../curves/, etc., we will save any plots generated. For example:
* PR curves or ROC curves for each technique (for classification or detection confidence).
* Histograms of latency distribution.
* Confusion matrices (for classification, or for detection errors maybe a matrix of defect type confusions).
* Bar charts for per-class AP or per-defect-type F1, etc.
* "Error galleries": e.g. a collage of images where technique A failed but B succeeded, etc. (This might be more manual to assemble, but we can automate selecting a few examples and save them as combined images).
* **Report (Markdown/HTML):** We will programmatically generate a Markdown report that compiles key results and visualizations. This will be saved as report.md (and optionally converted to HTML for a nicer view, e.g. using a Python markdown library or just letting GitHub render the MD). The report will contain:
* A **summary table** ranking the techniques by key metrics (with confidence intervals if computed). For example, a table of mAP or accuracy with ± confidence bounds.
* Plots (embedded images from the figs folder) such as PR/ROC curves side by side, latency distribution comparisons.
* Confusion matrices images if applicable.
* Sample images or error case illustrations (embedded thumbnails).
* A brief narrative analysis of results, possibly auto-filled with some template text plus dynamic values (like "Technique A achieved X% higher F1 than Technique B on scratches, indicating ...").
* It will also include citations for datasets or methods as appropriate (we can maintain a list of references in the report).

We will ensure the report generation code in benchcore/viz/report.py is flexible and can handle missing pieces (e.g., if a certain metric is not applicable, skip that section).

### Configuration Files

To run experiments easily, we will use YAML configuration files:

* **Dataset configs (configs/datasets/\*.yaml):** Each file defines a dataset, including paths to images, annotation files if any, how to split into train/val/test, what task type (classification or detection), and any class label mappings (e.g. defect type names). For example, configs/datasets/plastic\_containers.yaml might list the directories of good vs defective images and define classes like scratch, crack, contamination, .... This allows our code to load the appropriate data for benchmarking.
* **Technique configs (configs/techniques/\*.yaml):** Each technique (especially if it has tunable parameters) gets a config. For a classical method, this might include threshold values, morphological kernel sizes, etc., which can be tweaked. For consistency, these can mirror what is currently in the code (the current pipeline likely has hardcoded or global config for each detector – we can refactor those into these YAMLs so that the benchmarking harness can instantiate a detector with given params easily). If a technique uses a model, this config would include the model file path or URL. For example, a baseline.yaml might list all current defect detectors and their parameters; an improved.yaml might switch on region\_growing=True or different threshold values.
* **Experiment configs (configs/experiments/\*.yaml):** These tie together one or more techniques with one or more datasets and specify what metrics or analysis to run. For instance, an experiment config might look like:
* dataset: plastic\_containers  
  techniques:  
   - baseline  
   - improved  
  output: compare\_baseline\_vs\_improved  
  metrics: [classification, detection]
* This tells the harness to run both techniques on the plastic\_containers dataset, and produce outputs under a run ID that might incorporate compare\_baseline\_vs\_improved. We can have options here like enabling the human-in-loop or not, how many images to run (for quick tests vs full dataset), etc.

This configuration-driven approach makes the benchmarking highly flexible without needing to hardcode details in code. It also makes it easier for others to replicate or adjust experiments by editing YAMLs rather than code.

### Command-Line Interface (CLI) Scripts

We will provide entry-point scripts (likely in a scripts/ directory to keep them separate from core library code) for running benchmarks and generating reports:

* **scripts/bench\_run.py:** This will be the main driver to execute an experiment. Example usage:
* python scripts/bench\_run.py --exp configs/experiments/baseline\_vs\_improved.yaml --run-id auto
* The script will parse the experiment config, load the specified dataset(s) and technique(s), run each technique on the dataset, collect metrics, and save all outputs under a new results/<run\_id>/ folder. If --run-id is "auto", it can generate a unique ID (e.g., a timestamp or incremental counter). The script should handle seeding and environment logging as well. After completion, it will append a record to bench\_runs.jsonl with the run info. We will ensure it catches and logs errors gracefully (one misbehaving technique should not crash the entire run, similar to how the pipeline orchestrator handles detector errors).
* **scripts/bench\_report.py:** Generates a report for a completed run. Example:
* python scripts/bench\_report.py --run results/2025-09-20\_123456/
* (assuming run folder named by datetime and ID). This will read the summary and per-image files, produce the report.md and report.html, and possibly open it or print the path. If the bench\_run already created the report, this might not be needed, but it's useful if we want to regenerate or tweak the report layout without rerunning everything.
* **scripts/bench\_compare.py:** This could take multiple result folders or summary.csv files and produce a consolidated comparison (maybe an aggregate markdown). Example:
* python scripts/bench\_compare.py --glob "results/\*/summary.csv" --out results/compare.md
* It would read all summary CSVs matching the pattern and create a comparative table or charts. This is more of a nice-to-have for analyzing multiple runs (like comparing many techniques across runs). We may implement this if time permits or if multiple runs need to be synthesized in one view (e.g. for a summary in the thesis document).

### Environment Capture and Seeding

Each run will capture the environment details: - Python version, library versions (we can dump pip freeze to env.txt in the results folder). - If GPU is available, the CUDA/cuDNN versions, GPU model, etc. - OS info, CPU info, memory info (maybe using platform and psutil if available).

We will also enforce a seeding strategy at the start of each run:

random.seed(42)  
np.random.seed(42)  
if torch\_available:  
 torch.manual\_seed(42); torch.use\_deterministic\_algorithms(True)

(for example, using a fixed seed and enabling deterministic mode for frameworks). This ensures that any randomness in the pipeline (though minimal in classical CV, could be in train/test splits or augmentations if used) is consistent run-to-run.

### Human-in-the-Loop (Optional Integration)

While not a core requirement, we plan for potential **human-in-the-loop (HITL) hooks** to allow interactive refinement. In benchcore/hitl/hooks.py, we can define logic such as: after a run, identify the images or cases where the detectors are most uncertain or where techniques disagree. For instance, any detection with confidence below a threshold or any image where one technique finds a defect and another doesn’t could be flagged. These could be written to a review\_queue.jsonl.

In future, a small UI (possibly a Streamlit app) could load this review queue and allow an expert to verify or correct the results, feeding those corrections back to improve the algorithms (either by adjusting thresholds or retraining a model in an ML scenario). This is beyond the current implementation scope, but our design leaves room for it. If we include this now, it will be simply to log the potential review cases, which can be manually looked at as needed.

### Testing and Continuous Integration

We will maintain code quality and reliability by writing **unit tests** for the new modules: - Tests for metrics (e.g. does our mAP calculation match a known result on a small dummy dataset?). - Tests for the adapter (given a known Detection output, does it convert to the expected dict format?). - Tests for the technique interface (perhaps using a dummy technique that returns a preset output, ensuring the harness can integrate it).

These tests will be added to the repository’s test suite (if it exists). We will also run linting (using **Ruff** and formatting with **Black**) on all new code. Part of our done criteria is that the code passes lint checks and any existing CI pipelines. We will not break current tests or functionality; all new additions will be compatible.

If the repository has CI (Continuous Integration) set up (like GitHub Actions or similar), we will add steps to run the new tests. Possibly, we might include a separate job that runs a quick benchmark on a small sample (to ensure the harness doesn’t crash), though that might be too heavy for CI if it requires image data. At minimum, lint and unit tests will run.

### Implementation Plan (Step-by-Step)

1. **Repository Scan & Plan:** First, thoroughly review the current repo structure and identify locations of relevant code:
   * Confirm where the detectors live (src/vip/detect/\*.py as per docs).
   * Check for any existing evaluation or testing scripts.
   * Note the current config management (likely src/vip/config.py holds some settings). Based on this, refine the plan of where to add files (as per the structure above). *(*No code changes yet, just mapping things out and ensuring no conflicts with existing modules names.*)*
2. **Introduce Benchmarking Scaffold:** Create the /bench/ directory and subdirectories (benchcore, configs, etc.) as outlined. Add an empty \_\_init\_\_.py where needed to make them packages. This step establishes the skeleton in the feature branch.
3. **Baseline Adapter & Technique:** Implement a basic adapter and register the **current pipeline** as a baseline technique:
   * In benchcore/techniques/baseline/technique.py, create a class (e.g. BaselineTechnique) that uses the existing Pipeline orchestrator or individual detectors. It might instantiate the Pipeline and call its process(image) method to get results. Alternatively, we can call each detector class on the image – but using the Pipeline ensures it mirrors actual usage (the Pipeline orchestrator may handle running all detectors and merging results).
   * Ensure this returns detections in the standard format.
   * Adapters: Write adapter functions to convert the Pipeline’s output (the Detection objects) into the standardized dict for evaluation. For example, a scratch detection might have a mask or bbox – we include that. A contamination detection might just be a mask region – we might derive a bbox from it if needed for detection metrics.
4. **Metrics Implementation:** Implement the evaluation metrics in benchcore/eval/. Start with simpler ones (accuracy, precision, etc.) and ensure we can compute on outputs. For detection metrics, we might integrate an existing library or use a known implementation for mAP to avoid bugs (pycocotools or a custom IoU calculator + AP calculation).
   * Write unit tests for these with small fabricated data (e.g., a known set of predictions vs ground truth for which we can calculate metrics by hand).
5. **Benchmark Run Script:** Develop bench\_run.py to orchestrate everything:
   * Parse arguments for experiment config path and run ID.
   * Load the experiment YAML: get dataset and techniques.
   * Load dataset: either through manifest or directly reading images into a list (for first version, direct loading is fine, but manifest allows consistency).
   * For each technique, initialize it (call setup), loop through images (possibly in batches if supported) and collect predictions. Measure time around the prediction calls to log latency.
   * Accumulate predictions and then compute metrics comparing to ground truth.
   * Save summary CSV and per-image JSONL. Populate all fields as planned (if something isn’t applicable for a technique, e.g. segmentation metrics for a technique that doesn’t produce masks, just put N/A or leave blank).
   * Call benchcore.viz.report.generate\_report(results\_path) to create the report artifacts (or this can be a separate step).
   * Append the run info to bench\_runs.jsonl.
   * Print a completion message with where results are.
6. **Test a Demo Run:** After implementation, run a small test (maybe on a handful of images or a synthetic mini dataset) to ensure the pipeline works end-to-end. This could involve creating a dummy dataset config that points to 5 images in data/raw/demo with known “defects” and verifying the output files are correct. This step ensures that all parts (technique loading, metric calc, saving files, report generation) connect properly.
7. **Implement Improvements (Techniques):** Now that the infrastructure is in place, implement the *improved techniques*:
   * Add a new technique module, e.g. benchcore/techniques/regiongrow/technique.py, which extends the baseline by applying **background removal and region growing** before or after using the base detectors. For example, it could call the baseline detector, then post-process the detections: remove any detections on background regions (using a mask of the background), and extend detected regions via region growing (possibly using OpenCV’s floodFill or a custom BFS based on gradient continuity). This technique’s predict\_batch will output refined detections.
   * Similarly, a glare\_filter technique could pre-process images to remove highlights (e.g., apply a bright pixel mask and inpaint or reduce brightness in those areas) then run the baseline detector.
   * We might combine all improvements into one “improved” technique or separate them to test individual impact. For instance, we could have:
     + techniques/baseline (current methods as is),
     + techniques/bg\_removed (baseline + background subtraction),
     + techniques/glare\_removed (baseline + glare removal),
     + techniques/regiongrow (baseline + region growing),
     + techniques/all\_improvements (everything combined: bg removal + glare removal + region growing + HOG-based filtering). Each with its config toggling certain steps. This allows benchmarking each addition’s effect.
   * Ensure these techniques are plugged into the harness by adding their entry in technique configs and experiment configs for comparison.
8. **Benchmark and Iterate:** Run the benchmark on a validation dataset to compare the baseline and improved techniques. Analyze the report to see if false positives indeed dropped, confidence scores rose, etc. We might need to iterate on parameters (for example, the threshold for glare detection, or region growing stopping criteria) to get optimal results. Thanks to the harness, each iteration is traceable and results are recorded.
9. **Visualization Enhancements:** As we gather results, add any useful visual aids to the report:
   * Generate example images with detections drawn (the existing pipeline likely has some overlay visualization code we can reuse). For the report, we can include a few example images as thumbnails: e.g., one where the baseline missed a defect but improved caught it, one with a false positive that got eliminated, etc. These will strengthen the qualitative analysis.
   * Include a table or bullet points in the report highlighting key findings (e.g., “Background removal reduced false positive count by X%[[1]](https://www.mdpi.com/2072-666X/14/5/905#:~:text=background%20subtraction%20algorithm%20measures%20the,The)”, “Region growing increased detection area overlap with ground truth by Y%”).
   * Cite any papers or sources for techniques in the report text (the plan is to include such citations, which we have gathered, into the report for completeness and academic rigor).
10. **Finalize and Document:** Verify all acceptance criteria (below) are met. Write or update documentation (maybe a README\_benchmark.md) explaining how to use the new benchmarking tool. Summarize the improvements in the thesis write-up, referencing the evidence from these results. Once everything is satisfactory:
    * Run Black and Ruff to auto-format and lint; fix any warnings.
    * Ensure all tests (including new ones) pass.
    * Commit changes to the feature branch, push to remote, and create a Pull Request into develop.
    * After review, merge into develop. Tag the commit of the final run (e.g. bench-2025-09-20-final) for future reference.

Throughout this process, we maintain frequent commits and descriptive messages (and possibly use draft PRs) so that the integration of this feature is smooth.

### Acceptance Criteria

We will consider this initiative successful when the following are true (these serve as our “definition of done” checkpoints):

* ✅ **Benchmark Run Artifacts:** Running python scripts/bench\_run.py with a sample experiment config successfully produces a results folder with **summary.csv** containing all the specified columns and metrics, and these values make sense (no placeholders or zeros unless justified). The per\_image.jsonl is generated and correctly logs predictions.
* ✅ **Automated Report Generation:** bench\_report.py (or the bench\_run itself) produces a **report.md** and **report.html** with the content described. The report includes at least one table of results, one plot, and one example image (with proper citations for any images or data samples). All links or image references in the report are valid (e.g., if we include an image thumbnail, it should load).
* ✅ **Extensibility Test:** Adding a new dummy technique (for example, duplicating baseline technique under a new name) should be possible by creating a new folder and config **without modifying the core harness code**. This means our design truly supports plug-and-play extensibility. We will test this by simulating the addition of a trivial technique and ensuring the harness picks it up via config.
* ✅ **Data Integrity:** The bench\_runs.jsonl registry is updated on each run, and contains correct information (the commit hash matches git rev-parse HEAD, the config names are correct, etc.). No large data files are accidentally committed to the repo; the .gitignore covers results/ and any dataset files.
* ✅ **No Regression:** The existing pipeline (API and web app) continues to function exactly as before. We explicitly did not modify the API or frontend code. We can run the web app and ensure it still processes an image and returns detections. All enhancements are in the benchmarking and in new optional pipeline code paths, so the production usage is unaffected unless we choose to replace the pipeline with an improved technique after benchmarking.
* ✅ **Code Quality:** All new code adheres to the project’s style (PEP8 via Black, linted via Ruff). Type hints are used appropriately. Tests are added and passing. CI (if configured) shows green for our branch. Documentation is updated for any new scripts or modules.

By meeting these criteria, we ensure the pipeline improvements are not only developed but also validated and maintainable. The thesis project will then have a solid foundation to demonstrate improved defect detection results with empirical evidence.

## Data Sources and Example Datasets

In developing and testing the pipeline, we will utilize or reference several standard datasets relevant to surface defect detection. These datasets can also provide example images for our reports (either via locally included samples or external links), enriching the analysis with real-world scenarios:

* **SDNET2018 (Concrete Crack Dataset):** A large collection of 56,000+ images of concrete surfaces (bridges, walls, pavements) with annotations for cracks[[13]](https://www.kaggle.com/datasets/aniruddhsharma/structural-defects-network-concrete-crack-images#:~:text=SDNET2018%20contains%20over%2056%2C000%20images,06%20mm). It includes images of both cracked and non-cracked concrete, which is useful for binary classification and crack segmentation tasks. *Potential use:* We can demonstrate our crack detection algorithm on a subset of this data (though our focus is plastic, it’s a good benchmark for crack algorithms).
* **NEU Surface Defect Database:** A dataset of steel surface defects (6 classes of typical surface imperfections like scratches, patches, rolled-in scale, etc.). It contains 1,800 grayscale images (300 per class) of steel strips with defects. This is a classic for testing detection and classification of surface anomalies on a uniform material.
* **DAGM 2007 Dataset:** An older but widely used dataset for industrial optical inspection, containing ten classes of textured surfaces with and without defects (manufacturing flaws). Each class has a set of images with labeled defect regions. This can test our segmentation capabilities and also serve as a baseline for classical methods (many algorithms have been benchmarked on DAGM).
* **KolektorSDD (Surface Defect Dataset):** A collection of images of **electric commutator surfaces** with tiny cracks. The Kolektor SDD dataset is known for its challenge in detecting small, subtle defects on a curved metallic surface. If available, using this dataset will test our glare removal and fine crack detection under tough conditions.
* **Severstal Steel Defect Data:** From a public Kaggle competition on steel defect detection, it provides a large set of steel coil images with defects segmented (four defect types). This could be used to evaluate our segmentation performance and how the pipeline scales to higher resolution images.
* **Magnetic Tile Defect Dataset:** Images of magnet tiles with defects (cracks, holes, frays) – a common academic dataset for surface anomaly detection using traditional methods. Good for validating algorithms on a different texture and material.
* **DeepPCB Dataset:** A PCB (Printed Circuit Board) defect dataset where each image has a reference (defect-free) counterpart and a test image with defects. The task is to find differences (defects) between the two – a slightly different approach involving image comparison. While our pipeline doesn’t currently do template matching, this dataset can still be used by simulating a single-image detection (treat any discrepancy as a “defect” region). It’s useful to see how our methods perform on high-resolution, fine-detail defect detection.

For the purpose of our thesis demonstration, we will likely focus on the **plastic containers dataset** we gather (with our own images of good vs defective containers). However, referencing these public datasets serves to validate our approach against known benchmarks and to ensure our design is not narrowly tailored to one case. The final report will include a section with sample images (or thumbnails) from some of these datasets, with proper citations, to illustrate different defect types and to show that our pipeline improvements are grounded in broadly applicable techniques. For example, we will show a concrete crack image from SDNET2018 next to a detected crack in a plastic container to emphasize the generality of edge-based crack detection[[10]](https://www.mdpi.com/2079-9292/11/20/3357#:~:text=Generally%2C%20computer,3%20%2C%20120), or display a steel surface with glare where highlight removal would be crucial[[7]](https://www.mdpi.com/2076-3417/14/6/2469#:~:text=The%20existing%20highlight%20removal%20algorithms,%28utilizing%20deep%20learning%20methods).

**Citations:** Our report and documentation will cite sources for algorithms and datasets used. Key references include Yan et al. (2025) for the edge + region growing method[[4]](https://www.degruyterbrill.com/document/doi/10.1515/nleng-2022-0356/html?srsltid=AfmBOopm7jxPi5tdLmVj0y887hE5SgJOfrDI88IKED4BmXhx20RgVTbC#:~:text=At%20present%2C%20the%20detection%20of,the%20rapid%20screening%20method%20of), Zoubir et al. (2022) for HOG features in crack detection[[11]](https://www.mdpi.com/2079-9292/11/20/3357#:~:text=dimensionality%20reduction%20techniques%20are%20applied,the%20best%20results%20as%20an), Jiang et al. (2023) for highlight removal techniques[[5]](https://www.mdpi.com/2076-3417/14/6/2469#:~:text=In%20existing%20highlight%20removal%20methods%2C,channel%20of%20the%20HSV%20space), and Zheng & Zhang (2023) for background subtraction in defect detection[[1]](https://www.mdpi.com/2072-666X/14/5/905#:~:text=background%20subtraction%20algorithm%20measures%20the,The), among others. These connections to literature not only support the choices of methods but also demonstrate academic rigor in the thesis.

By executing this improvement and benchmarking plan, we expect to deliver a **robust, tested, and well-documented enhancement** to the Vision Inspection Pipeline. This will directly address the current false positive and low-confidence issues through better image processing (background removal, glare handling, region growing, etc.), and it will provide quantitative evidence of performance gains. Moreover, the introduction of a reproducible evaluation harness ensures that future work (or potential integration of machine learning methods post-thesis) can be assessed consistently. Overall, this plan strengthens the pipeline’s reliability for semi-automated inspection of plastic containers and establishes a framework for continuous improvement.

[[1]](https://www.mdpi.com/2072-666X/14/5/905#:~:text=background%20subtraction%20algorithm%20measures%20the,The) [[2]](https://www.mdpi.com/2072-666X/14/5/905#:~:text=match%20at%20L935%20improved%20the,area%20defects%20such%20as%20stains) [[3]](https://www.mdpi.com/2072-666X/14/5/905#:~:text=improved%20the%20mAP%20by%203.4,area%20defects%20such%20as%20stains) Wafer Surface Defect Detection Based on Background Subtraction and Faster R-CNN

<https://www.mdpi.com/2072-666X/14/5/905>

[[4]](https://www.degruyterbrill.com/document/doi/10.1515/nleng-2022-0356/html?srsltid=AfmBOopm7jxPi5tdLmVj0y887hE5SgJOfrDI88IKED4BmXhx20RgVTbC#:~:text=At%20present%2C%20the%20detection%20of,the%20rapid%20screening%20method%20of) [[12]](https://www.degruyterbrill.com/document/doi/10.1515/nleng-2022-0356/html?srsltid=AfmBOopm7jxPi5tdLmVj0y887hE5SgJOfrDI88IKED4BmXhx20RgVTbC#:~:text=easy%20to%20cause%20false%20detection,of%20Support%20Vector%20Machine%20were) Research on surface defect detection method and optimization of paper-plastic composite bag based on improved combined segmentation algorithm

<https://www.degruyterbrill.com/document/doi/10.1515/nleng-2022-0356/html?srsltid=AfmBOopm7jxPi5tdLmVj0y887hE5SgJOfrDI88IKED4BmXhx20RgVTbC>

[[5]](https://www.mdpi.com/2076-3417/14/6/2469#:~:text=In%20existing%20highlight%20removal%20methods%2C,channel%20of%20the%20HSV%20space) [[6]](https://www.mdpi.com/2076-3417/14/6/2469#:~:text=In%20metal%20images%2C%20most%20pictures,of%20later%20maintenance%20and%20replacement) [[7]](https://www.mdpi.com/2076-3417/14/6/2469#:~:text=The%20existing%20highlight%20removal%20algorithms,%28utilizing%20deep%20learning%20methods) [[8]](https://www.mdpi.com/2076-3417/14/6/2469#:~:text=Retinex%20model%20and%20generate%20a,validate%20the%20proposed%20method%20using) [[9]](https://www.mdpi.com/2076-3417/14/6/2469#:~:text=restored%20image%2C%20this%20paper%20introduces,quality%20qualitative%20and%20quantitative%20evaluation) Highlight Removal Emphasizing Detail Restoration

<https://www.mdpi.com/2076-3417/14/6/2469>

[[10]](https://www.mdpi.com/2079-9292/11/20/3357#:~:text=Generally%2C%20computer,3%20%2C%20120) [[11]](https://www.mdpi.com/2079-9292/11/20/3357#:~:text=dimensionality%20reduction%20techniques%20are%20applied,the%20best%20results%20as%20an) Concrete Bridge Crack Image Classification Using Histograms of Oriented Gradients, Uniform Local Binary Patterns, and Kernel Principal Component Analysis

<https://www.mdpi.com/2079-9292/11/20/3357>

[[13]](https://www.kaggle.com/datasets/aniruddhsharma/structural-defects-network-concrete-crack-images#:~:text=SDNET2018%20contains%20over%2056%2C000%20images,06%20mm) Structural Defects Network (SDNET) 2018 - Kaggle

<https://www.kaggle.com/datasets/aniruddhsharma/structural-defects-network-concrete-crack-images>