Software Design Document

Contents

The Software Design Document (SDD) provides a comprehensive overview of the design and architecture of Anomalib. It is intended for developers, architects, and anyone interested in the technical underpinnings of the software.

1. Introduction

Purpose

The purpose of this document is to give a detailed overview of the architecture, components, and design decisions of Anomalib. It serves as a guide for new developers and a reference for all stakeholders involved in development.

Scope

This section outlines the scope of the software, including its intended functionality, user base, and integration with other systems or technologies.

2. Overall Descripti

Back to top

Product Perspective

Anomalib is a deep learning library that aims to collect state-of-the-art anomaly detection algorithms for benchmarking on both public and private datasets. Anomalib provides several ready-to-use implementations of anomaly detection algorithms described in the recent literature, as well as a set of tools that facilitate the development and implementation of custom models. The library has a strong focus on visual anomaly detection, where the goal of the algorithm is to detect and/or localize anomalies within images or videos in a dataset.

Product Features

- Modular Design

  Offers a plug-and-play architecture that allows users to easily modify the existing anomaly detection algorithms or add new ones based on their specific requirements.

- Wide Algorithm Support

  Includes implementations of the state-of-the-art (sota) anomaly detection algorithms, covering a variety of techniques such as autoencoders, GANs, flow-based models, and more.

- Real-time Detection

  With a support of multiple inferencers such as Torch, Lightning, ONNX, OpenVINO and Gradio, Anomalib is capable of performing real-time anomaly detection on streaming data.

- Visualization Tools

  Comes with a suite of visualization tools for analyzing anomalies, understanding model decisions, and interpreting results.

- Customization and Extensibility

  Provides APIs for customization and extension, enabling researchers and practitioners to experiment with sota algorithms and techniques.

- Comprehensive Documentation a Back to top ty Support

  Features detailed documentation and a supportive community forum to assist users in implementation and troubleshooting.

User Classes and Characteristics

- Research Scientists

  Academics and researchers conducting studies in anomaly detection, requiring a flexible toolset for experimenting with different models and techniques.

- Data Scientists and Analysts

  Professionals looking to incorporate anomaly detection into their data analysis workflows without diving deeper into algorithmic complexities.

- Machine Learning Engineers

  Individuals focused on developing and deploying anomaly detection models, seeking a library that offers sota algorithms.

- Industry Practitioners

  Specialists in sectors like manufacturing, healthcare and security, who need to detect and respond to anomalies as part of risk management and quality control processes.

Operating Environment

Anomalib is designed to be environment-agnostic, capable of running on a variety of platforms including standalone desktop environments, on-premises servers, and cloud-based infrastructures. The library supports both CPU and GPU-based processing, with the latter providing significant performance improvements for large-scale anomaly detection tasks. The library also supports edge devices for real-time anomaly detection in IoT and embedded systems.

Assumptions and Dependencies

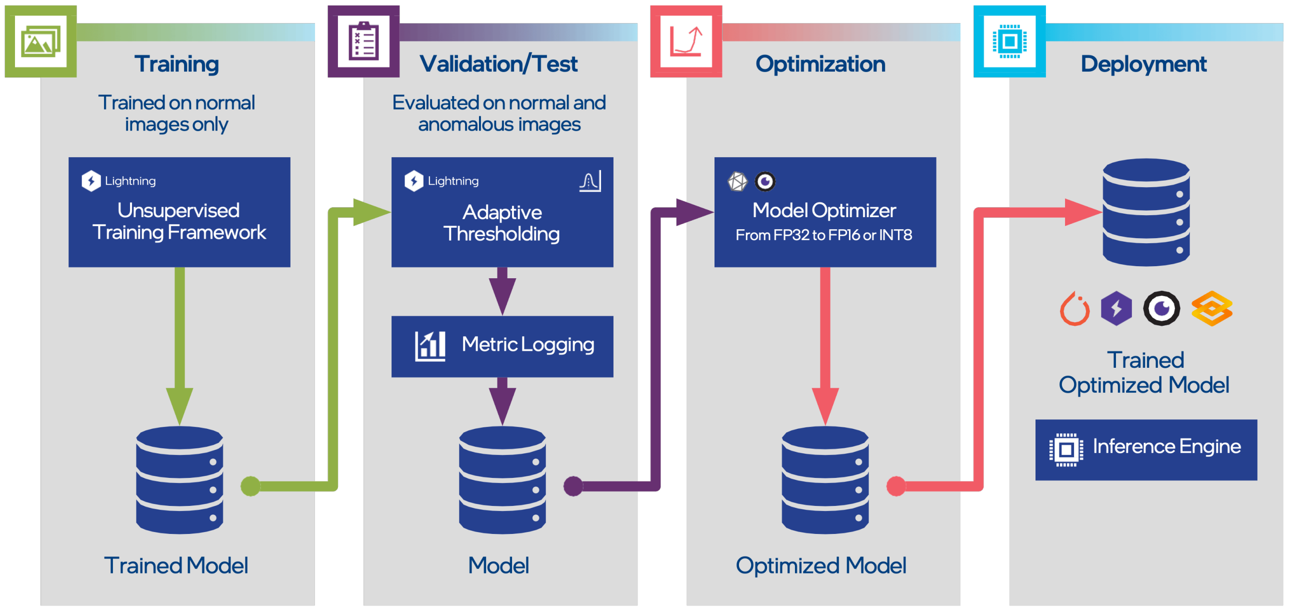
Anomalib's effectiveness and performance are dependent on the quality and granularity of the input data, as well as the appropriateness of the chosen anomaly detection algorithms for the specific use case. It assumes that users have a basic understanding of anomaly detection principles and are familiar v Back to top ɔgramming for custom implementations.

3. System Architecture

Architectural Overview

Anomalib is designed with a focus on modular, scalable, and flexible architecture to meet the diverse needs of anomaly detection across various domains.

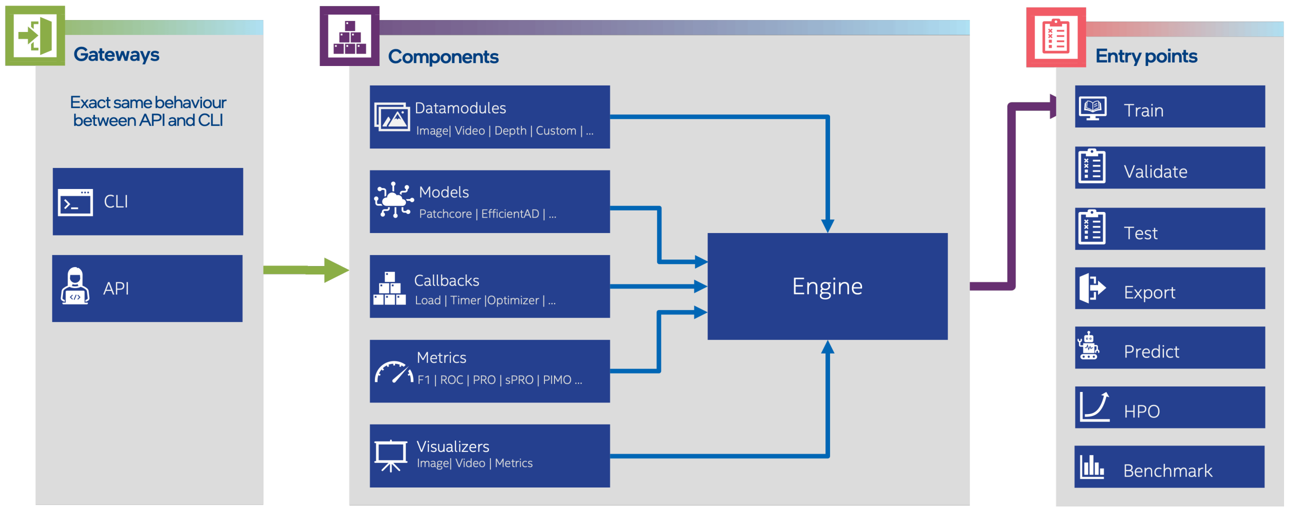
High-Level Workflow



As shown in Figure 1, Anomalib's high-level workflow consists of the following components:

- **Training [Optional]**: Involves training the anomaly detection model to learn patterns and characteristics of normal behavior. This step is optional for unsupervised algorithms. For zero-shot learning, the model is not trained on any specific dataset.
- **Validation/Test**: Validates the model by finding the threshold for anomaly detection and testing the model's performance on a validation or test dataset.
- **Optimization [Optional]**: Involves optimizing the model by converting it to a more efficient form, such as a compressed or quantized model. This step utilizes tools such as ONNX, OpenVINO for optimization.
- **Deployment**: Deploys the mode Back to top on environment for real-time or batch anomaly detection. This step uses multiple tools such as Torch, Lightning, ONNX, OpenVINO and Gradio for deployment.

High-Level Architecture


Gateways
Exact same behaviour between API and CLI
CLI
API
Components
Datamodules
Image| Video | Depth | Custom | ...
Models
Patchcore | EfficientAD | ...
Callbacks
Load | Timer |Optimizer | ...
Metrics
F1 | ROC | PRO | sPRO | PIMO ...
Visualizers
Image| Video | Metrics
Engine
Entry points
Train
Validate
Test
Export
Predict
HPO
Benchmark

As shown in Figure 2, Anomalib's high-level architecture consists of the following components: API/CLI, Components and Entry Points.

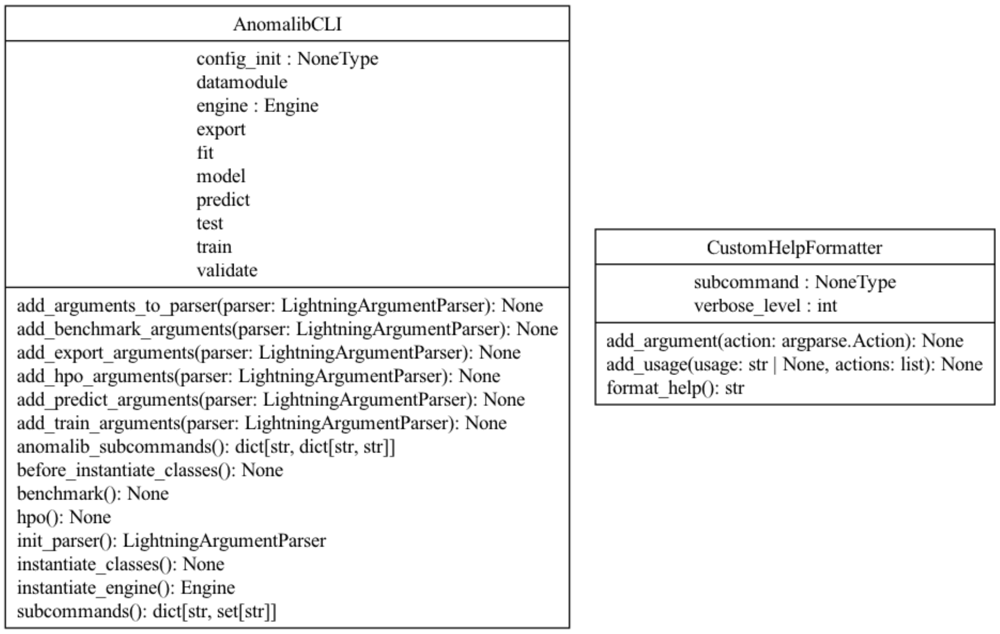
- **API/CLI**: Provides an interface for users to interact with the library, including APIs for data ingestion, configuration, and result export, as well as a command-line interface for running anomaly detection tasks. Both API and CLI provides the same functionality.
- **Components**: Consists of the core components of Anomalib, including, Datamodules, Models, Callbacks, Metrics, Visualizers and Engine.
- **Entry Points**: Includes the main entry points for running anomaly detection tasks, such as `train`, `validate`, `test`, `export`, and `predict`.

4. Detailed System Design

Here we will provide a detailed overview of the design and architecture of Anomalib, including the CLI, core components and entyrpoints.

CLI

Back to top

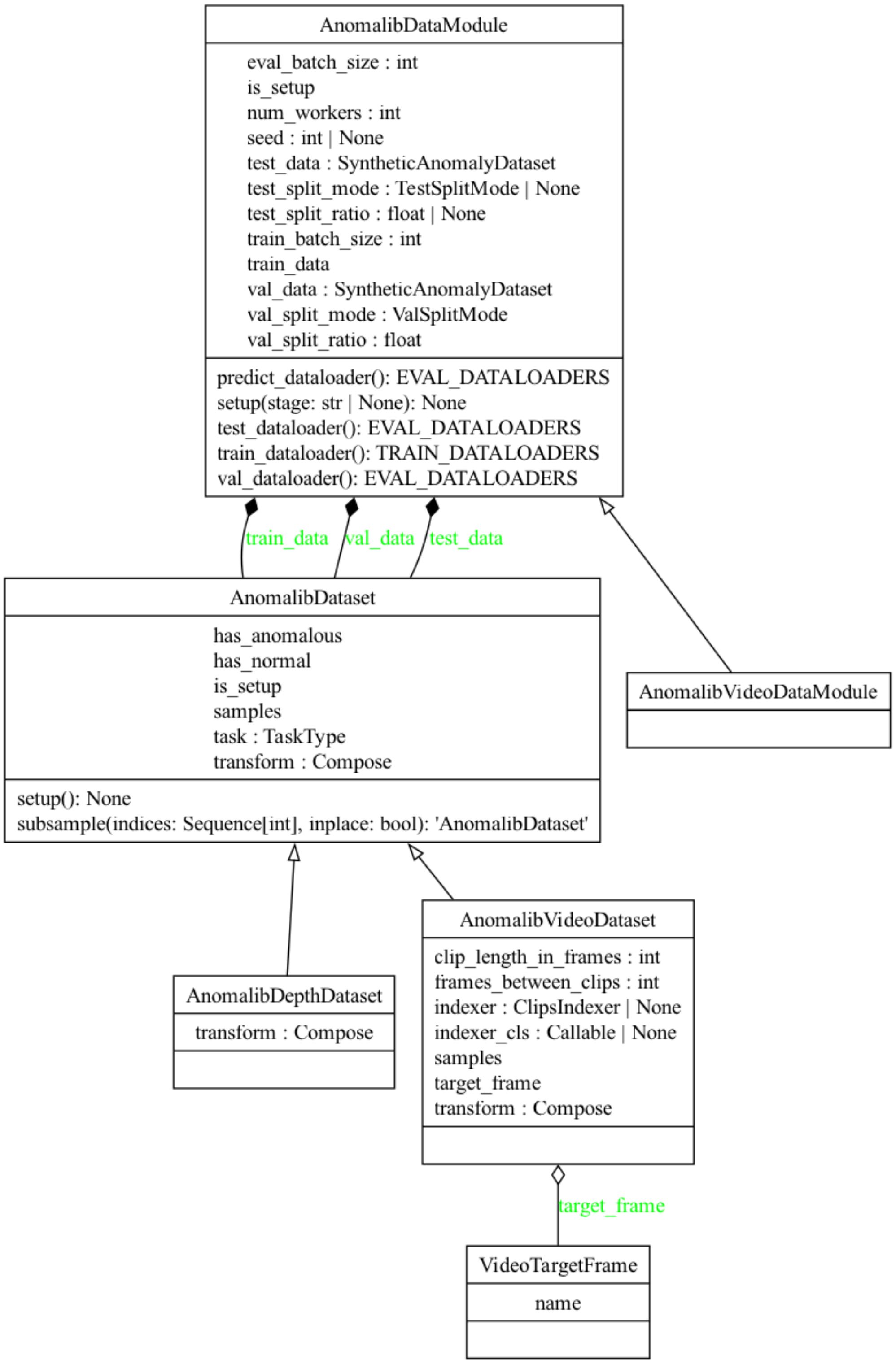

AnomalibCLI
config_init : NoneType
datamodule
engine : Engine
export
fit
model
predict
test
train
validate
add_arguments_to_parser(parser: LightningArgumentParser): None
add_benchmark_arguments(parser: LightningArgumentParser): None
add_export_arguments(parser: LightningArgumentParser): None
add_hpo_arguments(parser: LightningArgumentParser): None
add_predict_arguments(parser: LightningArgumentParser): None
add_train_arguments(parser: LightningArgumentParser): None
anomalib_subcommands(): dict[str, dict[str, str]]
before_instantiate_classes(): None
benchmark(): None
hpo(): None
init_parser(): LightningArgumentParser
instantiate_classes(): None
instantiate_engine(): Engine
subcommands(): dict[str, set[str]]
CustomHelpFormatter
subcommand : NoneType
verbose_level : int
add_argument(action: argparse.Action): None
add_usage(usage: str | None, actions: list): None
format_help(): str

As shown in Figure 3, Anomalib's command-line interface (CLI) provides a user-friendly way to interact with the library, allowing users to run anomaly detection tasks, configure algorithms, and visualize results. The CLI is implemented using LightningCLI, a command-line interface framework for PyTorch Lightning that uses jsonargparse for argument parsing. This allows users to sync their CLI arguments with their Python code, making it easier to maintain and update.

Datamodules

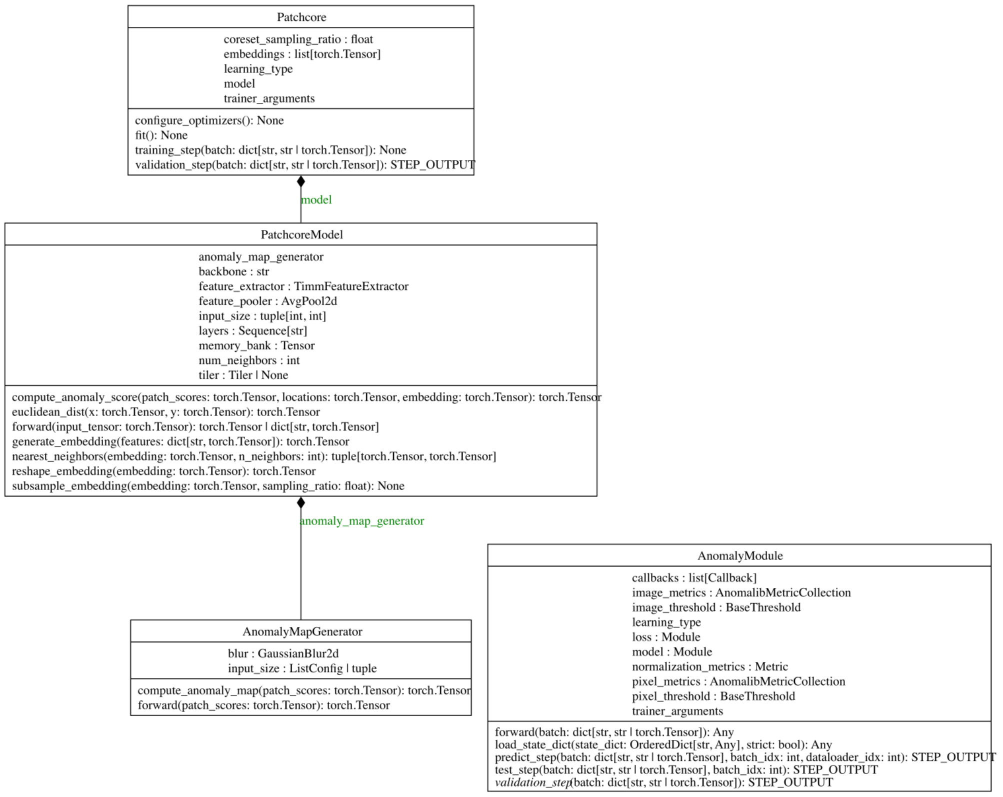
Datamodules in anomalib are responsible for preparing the data for training, validation, and testing. They are implemented using PyTorch Lightning's `LightningDataModule` class, which provides a set of APIs for loading, preprocessing, and splitting the data. The datamodules are designed to be modular and extensible, allowing users to easily modify the data loading and preprocessing steps based on their specific requirements. Anomalib has a base datamodule, which is sub-classes by image, video and depth datamodules to provide the necessary functionality for loading and preprocessing image, video and depth data.

Back to top


AnomalibDataModule
eval_batch_size : int
is_setup
num_workers : int
seed : int | None
test_data : SyntheticAnomalyDataset
test_split_mode : TestSplitMode | None
test_split_ratio : float | None
train_batch_size : int
train_data
val_data : SyntheticAnomalyDataset
val_split_mode : ValSplitMode
val_split_ratio : float
predict_dataloader(): EVAL_DATALOADERS
setup(stage: str | None): None
test_dataloader(): EVAL_DATALOADERS
train_dataloader(): TRAIN_DATALOADERS
val_dataloader(): EVAL_DATALOADERS
train_data
val_data
test_data
AnomalibDataset
has_anomalous
has_normal
is_setup
samples
task : TaskType
transform : Compose
setup(): None
subsample(indices: Sequence[int], inplace: bool): 'AnomalibDataset'
AnomalibVideoDataModule
AnomalibDepthDataset
transform : Compose
AnomalibVideoDataset
clip_length_in_frames : int
frames_between_clips : int
indexer : ClipsIndexer | None
indexer_cls : Callable | None
samples
target_frame
transform : Compose
target_frame
VideoTargetFrame
name

Models

Anomalib provides a collection of anc Back to top within the image and video domains. The models are implemented sub-classing PyTorch Lightning’s `LightningModule` class, which is called `AnomalyModule`, which provides a set of APIs for defining the model architecture, loss function, and optimization algorithm. The models are designed to be modular and extensible, allowing users to easily modify the model architecture and training workflow based on their specific requirements. Anomalib has a base model, which is sub-classes by algorithm implementations such as autoencoders, GANs, flow-based models, and more. The figure below demonstrates the high-level overview of model architecture, and the implementation of Patchcore model.


Patchcore
coreset_sampling_ratio : float
embeddings : list[torch.Tensor]
learning_type
model
trainer_arguments
configure_optimizers(): None
fit(): None
training_step(batch: dict[str, str | torch.Tensor]): None
validation_step(batch: dict[str, str | torch.Tensor]): STEP_OUTPUT
model
PatchcoreModel
anomaly_map_generator
backbone : str
feature_extractor : TimmFeatureExtractor
feature_pooler : AvgPool2d
input_size : tuple[int, int]
layers : Sequence[str]
memory_bank : Tensor
num_neighbors : int
tiler : Tiler | None
compute_anomaly_score(patch_scores: torch.Tensor, locations: torch.Tensor, embedding: torch.Tensor): torch.Tensor
euclidean_dist(x: torch.Tensor, y: torch.Tensor): torch.Tensor
forward(input_tensor: torch.Tensor): torch.Tensor | dict[str, torch.Tensor]
generate_embedding(features: dict[str, torch.Tensor]): torch.Tensor
nearest_neighbors(embedding: torch.Tensor, n_neighbors: int): tuple[torch.Tensor, torch.Tensor]
reshape_embedding(embedding: torch.Tensor): torch.Tensor
subsample_embedding(embedding: torch.Tensor, sampling_ratio: float): None
anomaly_map_generator
AnomalyMapGenerator
blur : GaussianBlur2d
input_size : ListConfig | tuple
compute_anomaly_map(patch_scores: torch.Tensor): torch.Tensor
forward(patch_scores: torch.Tensor): torch.Tensor
AnomalyModule
callbacks : list[Callback]
image_metrics : AnomalibMetricCollection
image_threshold : BaseThreshold
learning_type
loss : Module
model : Module
normalization_metrics : Metric
pixel_metrics : AnomalibMetricCollection
pixel_threshold : BaseThreshold
trainer_arguments
forward(batch: dict[str, str | torch.Tensor]): Any
load_state_dict(state_dict: OrderedDict[str, Any], strict: bool): Any
predict_step(batch: dict[str, str | torch.Tensor], batch_idx: int, dataloader_idx: int): STEP_OUTPUT
test_step(batch: dict[str, str | torch.Tensor], batch_idx: int): STEP_OUTPUT
validation_step(batch: dict[str, str | torch.Tensor]): STEP_OUTPUT

Callbacks

We use LightningCallbacks to add functionality to the training loop, such as logging, early stopping, and model checkpointing. For more details, refer to the [Lightning Callbacks documentation](). In addition to the built-in callbacks, we have implemented custom callbacks for anomaly detection tasks, which could be accessed through the `callbacks` module.

Metrics

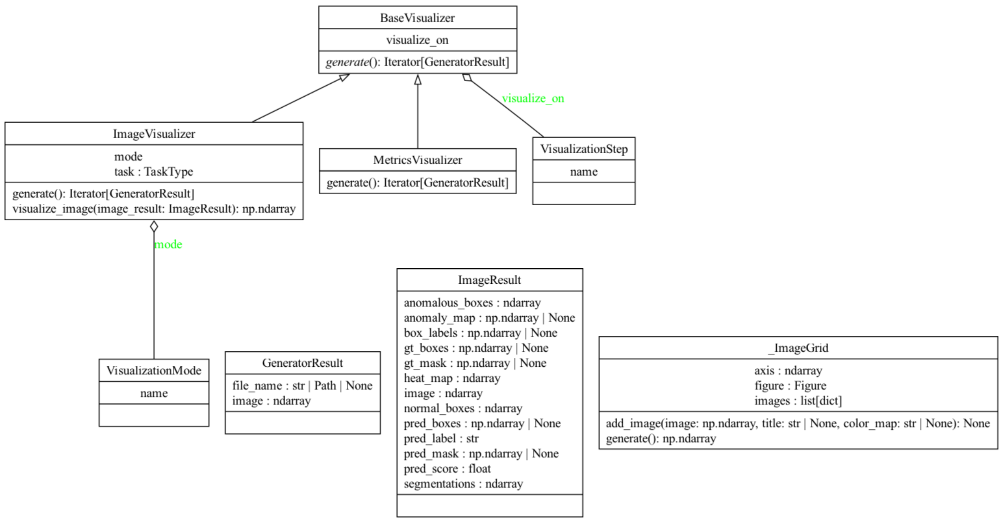
Back to top

Similar to callbacks, we make use of Lightning Torchmetrics package, which provides a collection of metrics for evaluating model performance. For more details, refer to the

Torchmetrics documentation. We have also implemented custom metrics such as PRO, sPRO and PIMO for anomaly detection tasks, which could be accessed through the `metrics` module.

Visualizers

Anomalib provides a suite of visualization tools for analyzing anomalies, understanding model decisions, and interpreting results. Currently we have implemented visualizers for image and metrics, which could be accessed through `ImageVisualizer` and `MetricsVisualizer` classes, respectively. The figure below demonstrates the high-level overview of visualizers.


BaseVisualizer
visualize_on
generate(): Iterator[GeneratorResult]
visualize_on
ImageVisualizer
mode
task : TaskType
generate(): Iterator[GeneratorResult]
visualize_image(image_result: ImageResult): np.ndarray
MetricsVisualizer
generate(): Iterator[GeneratorResult]
VisualizationStep
name
mode
VisualizationMode
name
GeneratorResult
file_name : str | Path | None
image : ndarray
ImageResult
anomalous_boxes : ndarray
anomaly_map : np.ndarray | None
box_labels : np.ndarray | None
gt_boxes : np.ndarray | None
gt_mask : np.ndarray | None
heat_map : ndarray
image : ndarray
normal_boxes : ndarray
pred_boxes : np.ndarray | None
pred_label : str
pred_mask : np.ndarray | None
pred_score : float
segmentations : ndarray
_ImageGrid
axis : ndarray
figure : Figure
images : list[dict]
add_image(image: np.ndarray, title: str | None, color_map: str | None): None
generate(): np.ndarray

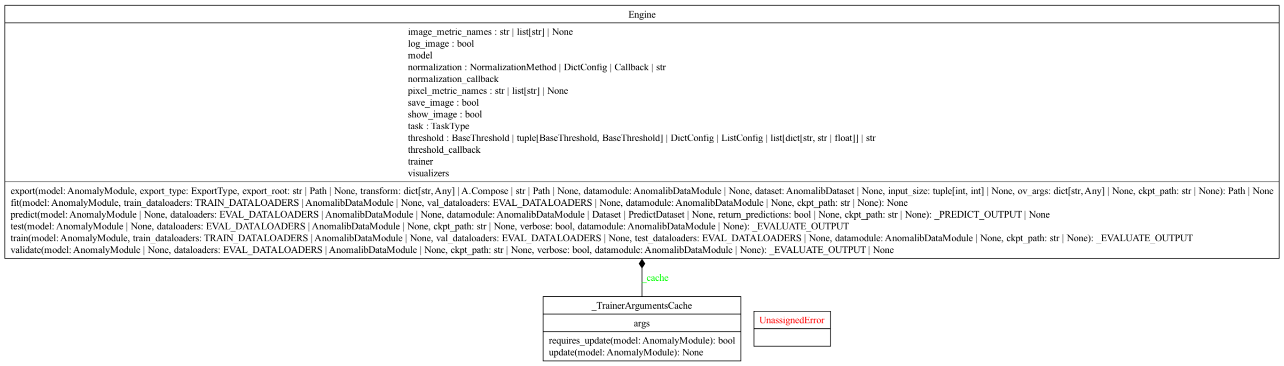
Engine

The figure below demonstrates the high-level architecture of the engine, which is responsible for orchestrating the anomaly detection workflow, including data preparation, model training, validation, testing, export and predict.

Under the hood, `Engine` uses Lightning's `Trainer` to manage the training workflow. One of the main distinguishing features of the `Engine` is its ability of caching arguments to create the Trainer class after the datamodule and model are created or modified. This allows the user to modify the datamodule and n Back to top having to re-create the Trainer class.

Another important feature of the `Engine` is its ability to export the trained model to

OpenVINO format for deployment on edge devices. This is achieved by using the `openvino` module, which provides a set of tools for converting PyTorch models to OpenVINO format.


Engine
image_metric_names : str | list[str] | None
log_image : bool
model
normalization : NormalizationMethod | DictConfig | Callback | str
normalization_callback
pixel_metric_names : str | list[str] | None
save_image : bool
show_image : bool
task : TaskType
threshold : BaseThreshold | tuple[BaseThreshold, BaseThreshold] | DictConfig | ListConfig | list[dict[str, str | float]] | str
threshold_callback
trainer
visualizers
export(model: AnomalyModule, export_type: ExportType, export_root: str | Path | None, transform: dict[str, Any] | A.Compose | str | Path | None, datamodule: AnomalibDataModule | None, dataset: AnomalibDataset | None, input_size: tuple[int, int] | None, ov_args: dict[str, Any] | None, ckpt_path: str | None): Path | None
fit(model: AnomalyModule, train_dataloaders: TRAIN_DATALOADERS | AnomalibDataModule | None, val_dataloaders: EVAL_DATALOADERS | None, datamodule: AnomalibDataModule | None, ckpt_path: str | None): None
predict(model: AnomalyModule | None, dataloaders: EVAL_DATALOADERS | AnomalibDataModule | None, datamodule: AnomalibDataModule | Dataset | PredictDataset | None, return_predictions: bool | None, ckpt_path: str | None): _PREDICT_OUTPUT | None
test(model: AnomalyModule | None, dataloaders: EVAL_DATALOADERS | AnomalibDataModule | None, ckpt_path: str | None, verbose: bool, datamodule: AnomalibDataModule | None): _EVALUATE_OUTPUT
train(model: AnomalyModule, train_dataloaders: TRAIN_DATALOADERS | AnomalibDataModule | None, val_dataloaders: EVAL_DATALOADERS | None, test_dataloaders: EVAL_DATALOADERS | None, datamodule: AnomalibDataModule | None, ckpt_path: str | None): _EVALUATE_OUTPUT
validate(model: AnomalyModule | None, dataloaders: EVAL_DATALOADERS | AnomalibDataModule | None, ckpt_path: str | None, verbose: bool, datamodule: AnomalibDataModule | None): _EVALUATE_OUTPUT | None
_cache
_TrainerArgumentsCache
args
requires_update(model: AnomalyModule): bool
update(model: AnomalyModule): None
UnassignedError

Overall, Anomalib is designed with a focus on modular, scalable, and flexible architecture to meet the diverse needs of anomaly detection across various domains. The library provides a comprehensive set of tools for data preparation, model training, validation, testing, export, and predict, as well as a suite of visualization tools for analyzing anomalies, understanding model decisions, and interpreting results.

Previous
Developer Guide

Next
Contribution Guidelines

Back to top