CISC 320 System Design Document
CleanShare

Section 3 - Group 6
David Balan
Anthony Grecu
Charlie Guarasci
Liam Harper-Mccabe
Kieron Luke
Mark Nistor
Kayetan Protas

November 14, 2025

Contents

List of Figures

List of Tables

1 Introduction

1.1 Purpose

This Software System Description (SSD) document provides a detailed technical blueprint for CleanShare, a desktop application that detects and blurs alcoholic beverages in images before users share them online. The SSD expands upon the Requirements Analysis Document (RAD) [1] by translating high-level requirements into a concrete architecture, component design, and runtime behaviour that developers can implement and maintain.

The primary purpose of this document is to:

- Describe how CleanShare's privacy-first, offline-only requirements are realized in the system architecture and deployment model.
- Specify the responsibilities and interfaces of each major subsystem (GUI, Application Core, Detection Engine, Redaction Pipeline, I/O & Privacy, Evaluation utilities).
- Provide an agreed-upon reference for developers, testers, TAs, and future maintainers so that implementation, testing, and evolution of the system all follow the same design.
- Serve as a baseline for design reviews and future change requests, allowing the team to trace any modification back to the original architectural rationale.

In short, the SSD answers *how* CleanShare is built and behaves internally, given the *what* defined in the RAD and project outline [2].

1.2 Scope

CleanShare targets Windows 10+ desktop systems and is built using C++17, Qt 6.x [3], OpenCV 4.x [4], and ONNX Runtime [5]. Within this scope, the system enables users to:

- Import a JPEG/PNG image from the local file system.
- Automatically detect alcoholic beverage containers using a YOLOv11-based CNN model deployed via ONNX Runtime.
- Blur detected alcoholic regions using a configurable Gaussian blur pipeline.
- Preview the original and redacted images side-by-side and optionally refine blur regions using manual tools (brush/rectangle with undo/redo).
- Export a finalized, "clean" image at the original resolution, entirely locally and offline.

The scope includes the full end-to-end workflow (import $\rightarrow$ detect $\rightarrow$ blur $\rightarrow$ preview $\rightarrow$ export), core error handling (invalid files, failed inference, no detections), performance targets (e.g., $\leq$ 10 s per 1080p image on CPU), and local logging sufficient for debugging while preserving privacy.

Out of scope for this version are:

- Video or real-time stream processing (only static images are supported).
- Mobile, web, or cross-platform deployments beyond Windows 10+.
- Cloud-based inference, online services, or any network-dependent features.
- Advanced extensions such as full batch processing, social media integration, or multiple redaction styles beyond the baseline Gaussian blur (these are considered potential future work).

1.3 Definitions, Acronyms, Abbreviations

This subsection defines the key terms and acronyms used throughout the SSD so that readers can interpret the design consistently.

CleanShare The Windows desktop application being designed in this project. CleanShare automatically detects and blurs alcoholic beverages in user-supplied photos so they are "safe to share" on social or professional platforms, while performing all processing locally on the user's machine.

CNN (Convolutional Neural Network) A type of deep learning model used for image-based tasks. In CleanShare, a CNN (YOLOv11-based) is used to detect alcoholic containers in images.

YOLOv11 "You Only Look Once" version 8, a modern family of real-time object detection models [6]. CleanShare uses a YOLOv11-derived model fine-tuned on the Liquor Data dataset and exported to ONNX format.

ONNX (Open Neural Network Exchange) A standard format for representing trained ML models [5]. CleanShare ships with an ONNX model file (e.g., `model.onnx`) that encapsulates the trained detector.

ONNX Runtime The inference engine used to execute the ONNX model locally [5]. CleanShare embeds ONNX Runtime libraries to run CNN inference on the CPU (and optionally GPU) without external services.

OpenCV An open-source computer vision library used for image loading, preprocessing, and blurring operations (e.g., Gaussian blur) in CleanShare [4].

GUI (Graphical User Interface) The visual front-end of CleanShare, implemented using Qt Widgets. The GUI provides buttons, menus, previews, and tools for importing, editing, and exporting images.

Qt A C++ application framework used to build the CleanShare desktop GUI (windows, dialogs, widgets) and to manage application lifecycle and events [3].

ROI (Region of Interest) A portion of the image that has been identified for special processing. In CleanShare, ROIs correspond to regions containing alcoholic beverages or manually selected blur areas.

ROI Mask A binary mask indicating which pixels belong to blur regions. The Redaction Pipeline uses the ROI mask to apply Gaussian blur only where required and leave the rest of the image untouched.

NMS (Non-Maximum Suppression) A post-processing step that removes duplicate or overlapping detection boxes, keeping only the highest-confidence bounding boxes returned by the CNN.

Detection Engine The logical subsystem responsible for image preprocessing, CNN inference via ONNX Runtime, NMS, and confidence thresholding. It outputs a list of detected boxes and associated scores.

Redaction Pipeline The subsystem that converts detection boxes and manual edits into an ROI mask and applies the selected blur strategy (e.g., Gaussian blur) while preserving the original image dimensions.

I/O & Privacy Module The subsystem responsible for reading and writing images (JPEG/PNG), validating inputs, and enforcing local-only processing (no network calls, no long-term storage of sensitive data).

ImageSession An in-memory object managed by the Application Core that encapsulates the state of the currently loaded image, including original pixels, detection results, masks, configuration, and undo/redo history.

Application Core The subsystem that orchestrates the main workflow (import → detect → blur → preview → export), manages session state, and coordinates interactions among GUI, Detection Engine, Redaction Pipeline, and I/O.

Evaluation Utilities Developer-only tools that run the detector on a labeled dataset to compute metrics such as recall, false positive rate, and processing time, ensuring that CleanShare satisfies its success criteria.

Liquor Data Dataset A labeled dataset of alcoholic containers (beer, wine, spirits) from Lamar University used to train and evaluate the CNN model integrated into CleanShare [7].

1080p An image resolution of approximately 1920×1080 pixels. CleanShare's performance requirements are specified in terms of processing time for 1080p images on a typical CPU-only Windows laptop.

Recall The fraction of actual alcohol containers in an image that are correctly detected by the model.

Precision The fraction of predicted alcohol detections that are truly alcohol containers (i.e., not false positives).

False Positive Rate The proportion of non-alcoholic objects that are incorrectly detected (and blurred) as alcohol containers.

mAP@0.5 (Mean Average Precision at IoU 0.5) A standard object detection metric summarizing detection quality across a dataset at an Intersection-over-Union (IoU) threshold of 0.5. Used to monitor overall model performance.

CPU-only Inference Running the ONNX model solely on the system's CPU without requiring GPU acceleration. CleanShare's baseline performance guarantees are defined under CPU-only inference.

Blur Strategy An interchangeable implementation of the redaction effect (e.g., Gaussian blur now, possible pixelation/mosaic later) selected by the Redaction Pipeline based on configuration or user preference.

1.4 Overview of Document

This document provides a high-level and detailed system overview of CleanShare, from architecture down to component APIs and runtime behaviour. It is organized as follows:

- **Section 2: System Overview** describes the overall context in which CleanShare operates, major subsystems at a glance, and the primary user workflow from image import to export.
- **Section 3: Architectural Design (4+1 Views)** presents the logical, process, development, physical/deployment, and use-case views that jointly describe the system's architecture.
- **Section 4: External Interface Design** specifies user interfaces, file formats, OS/hardware interfaces, and the model/artifact interfaces used by CleanShare.
- **Section 5: Data Design** defines the core data structures (images, boxes, masks, session state), configuration parameters, and persistence strategy.
- **Section 6: Component Design** refines logical modules into concrete runtime components, detailing their responsibilities, APIs, and relevant design patterns.
- **Section 7: Dynamic Behavior** explains how components interact at runtime using sequence diagrams and state machines for key workflows.
- **Section 8: Error Handling, Logging, and Telemetry** describes how the system categorizes, surfaces, and recovers from errors while maintaining privacy-preserving, local-only logging.
- **Section 9: Security & Privacy Design** presents the privacy principles, threat model, and access-control considerations that enforce a privacy-first posture.
- **Section 10: Performance, Capacity, and Quality Attributes** documents performance targets, scalability considerations, and usability/accessibility goals.

- **Section 11: Deployment & Installation** outlines how CleanShare is packaged, distributed, and configured on Windows desktops.
- **Section 12: Build, CI/CD, and Configuration Management** explains build tooling, the CI pipeline, versioning (including model versioning), and configuration management practices.
- **Section 13: Verification & Validation Strategy** details the testing strategy (unit, integration, system, regression) and dataset-based evaluation of model performance.
- **Section 14: Requirements-to-Design Traceability** maps each major functional and non-functional requirement to the design elements that satisfy it.
- **Section 15: Design Rationale and Architectural Decisions (ADRs)** summarizes key design choices, trade-offs, and why specific technologies and patterns were selected.
- **Section 16: Risks and Mitigations** identifies major technical and project risks and how they are mitigated.
- **Section 17: Maintenance & Support** describes how the system is structured for ongoing updates, bug fixes, and documentation upkeep.
- **Appendices** provide a glossary extension and references to the RAD, project outline, library documentation, and dataset/model sources.

This structure ensures that readers can start from a high-level understanding and progressively drill down to specific components, interfaces, and test plans as needed.

2 System Overview

2.1 Context

CleanShare exists as a standalone desktop application running locally on user hardware. It interacts with external systems only to load and save image files, and no network communication is used. The system's primary goals are to protect user privacy, provide effective and accurate detection of alcoholic beverages, and produce final images ready to be shared online. CleanShare integrates Qt UI, OpenCV processing, and ONNX runtime inference into a single workflow to ensure quick and accurate image blurring.

2.2 System Decomposition (at a glance)

CleanShare can be broken down into five major subsystems:

1. Presentation Layer (Qt GUI) Includes the MainWindow Class and supports the main UI operations such as image import, preview of original vs. blurred, manual editing, and export options.
2. Application Core Includes the AppControllerClass and manages state, workflow, and transitions. Also handles configuration for blur type.
3. Detection Includes the ONNX Detector and Heuristic Detector classes and controls preprocessing, ONNX Runtime CNN inference, post-processing with confidence threshold, and an optional fallback on the heuristic detector.
4. Redaction Includes the RedactionPipeline Class, which builds a mask from detection boxes and applies blurring.
5. IO and Utils Includes the ImageIO and EvaluationService classes and is responsible for loading the JPEG/PNG and writing the blurred output. This layer ensures all operations remain non-local and holds only temporary in-memory data to maintain user privacy.

2.3 Primary User Workflow

1. Import Image The user uploads a JPEG or PNG, and the system performs file existence check, pixel loading via OpenCV, dimension recording, and highlights errors with import if they exist.

2. Pre-Processing and Detection The CNN model identifies alcoholic beverage containers by running the ONNX inference, generating candidate bounding boxes, applying NMS to filter duplicates, and removing low-confidence detections

   All detections are mapped back to the original image size, and the system builds ROI masks, bounding box overlays, data for UI preview, and if no alcohol is detected, the UI informs the user and allows manual marking.

3. Blur and Post-Processing The system applies a Gaussian blur to all ROI regions in a non-destructive pipeline, so the user can undo/redo changes

4. Preview/Edit User sees a side-by-side view (original vs. blurred) and may add or remove blurred regions and adjust blur strength

5. Export User exports the final, sanitized image at original resolution.

3 Architectural Design (4+1 Views)

3.1 Logical View

CleanShare is a Windows desktop application written in C++ and built using Qt, OpenCV, and ONNX Runtime. The system is divided into several logical modules that work together to implement all user facing and internal features. These modules include the Presentation layer implemented with Qt, the Application Core that manages workflow and state, the Detection Engine that performs model inference, the Redaction Pipeline that creates and applies blur masks, the IO and Privacy module responsible for file handling and data protection, and an Evaluation Utilities module for developer testing. These components form the logical structure required to support all detection, blurring, and export features of the application.

3.1.1 Modules and Responsibilities

- **Qt GUI**: file open/save, side-by-side preview, blur strength, manual tools (brush/rectangle).

  The Presentation module (GUI) provides the user interface for importing images, triggering detection, editing redaction regions, previewing the original and blurred versions, and exporting results. It stores only UI related data such as the current screen, selected tool, zoom level, and temporary preview images. This module communicates exclusively with the Application Core, sending user commands and receiving updates to refresh the interface.

- **Application Core**: orchestration, state/config (thresholds, blur type).

  The Application Core orchestrates the entire workflow including image import, detection, mask creation, redaction, preview updates, and final export. It maintains session state such as the loaded image, detection results, blur regions, undo and redo history, and user configuration. It is responsible for coordinating all interactions with IO operations, model inference, and redaction computations.

- **Detection Engine**: preprocessing, ONNX inference, NMS, thresholds.

  The Detection Engine is responsible for preprocessing the image for inference, running ONNX Runtime using the YOLOv11 based model, applying non maximum suppression, filtering low confidence predictions, and producing detection results mapped back to the original image coordinates. It owns the loaded model weights and all preprocessing configuration.

- **Redaction Pipeline**: ROI mask creation, Gaussian/pixelate/mosaic.

The Redaction Pipeline builds region of interest masks using model detections and any manual regions added by the user. It then applies blur or another redaction style through OpenCV to produce a final modified image. It maintains internal mask representations and redaction parameters.

- **I/O & Privacy**: JPEG/PNG I/O, local-only processing, ephemeral metadata.

  The IO and Privacy module loads and validates PNG and JPEG files, saves exported results, and ensures that all processing remains local with no external transmission or persistent storage of sensitive information. It handles temporary buffers and optional configuration files.

- **Evaluation Utilities**:

  The Evaluation Utilities module supports developer level testing by measuring accuracy metrics such as recall, precision, false positives, and Mean Average Precision (MAP). It also records inference times and can output reports. It interacts with the Detection Engine and Redaction Pipeline but is not used during normal user operation.

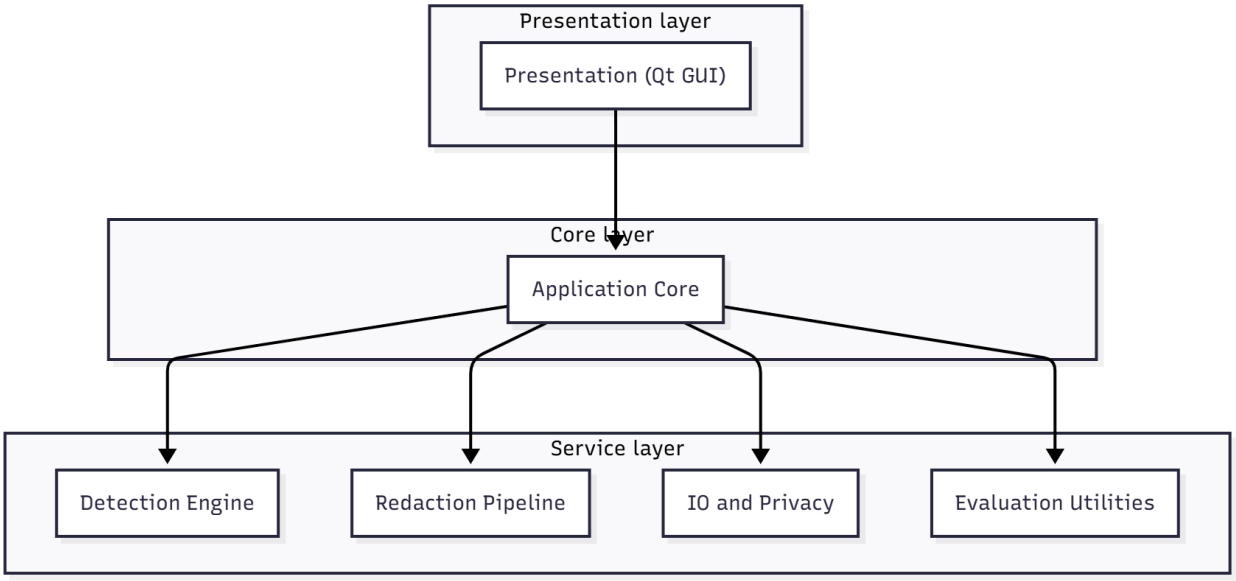

Figure 1: Logical View

3.1.2 Key Interfaces (logical)

Public interfaces between modules; data passed (images, boxes, masks). The Presentation layer communicates with the Application Core through high level commands such as ‘importImage‘, ‘requestDetection‘, ‘requestRedaction‘, ‘requestExport‘, and ‘userCancelled‘. The Application Core communicates with the Detection Engine through operations such as ‘preprocessForDetection‘, ‘runInference‘, and ‘postprocessDetections‘. It interacts with the Redaction Pipeline by invoking ‘buildMask‘ and ‘applyBlur‘. IO operations such as ‘loadAndValidateImage‘ and ‘saveImage‘ allow the Application Core to read and write image data. The Application Core may also call ‘runEvaluation‘ in the Evaluation Utilities module to produce developer level metrics. All data exchanged between modules consists of file paths, image buffers, region descriptors, and configuration values.

3.2 Process View

Thread/process model, responsiveness targets, cancellation during inference, progress reporting.

CleanShare uses a multithreaded execution model in which the GUI thread is responsible for rendering windows, handling user interaction, and updating the interface, while a worker thread executes long running

tasks such as model inference and redaction. The evaluation utilities may run in the same process or in a separate executable but are not part of the main application workflow. This approach ensures that the application remains responsive and smooth, even when processing large images.

During the main runtime flow, the user launches the application and imports an image through the Presentation layer. The Application Core validates the image through the IO module and, once successful, posts a detection job. The Detection Engine preprocesses the image, runs inference through ONNX Runtime, and postprocesses the raw model outputs. The Application Core then instructs the Redaction Pipeline to build an ROI mask and apply blur to produce a redacted version. The worker thread returns the blurred result to the Presentation layer, which updates the preview. The user can optionally adjust the blurred regions manually, and the Application Core reprocesses them through the Redaction Pipeline. Finally, the user exports the blurred image, and the Application Core saves it using the IO module.

Alternate flows cover scenarios with no detections, invalid files, or slow inference. When no detections are returned, the Application Core notifies the Presentation module, which enables manual region editing. When an invalid image is provided, the IO module returns an error message that is propagated to the user. If inference is slow, progress updates are periodically sent and the user can cancel the operation, which stops the worker thread and returns the application to a stable state. All concurrency behaviour maintains thread safety by ensuring that the Presentation layer is updated only from the GUI thread, and that session state is protected against concurrent modification.

3.3 Development View

Repo layout, module boundaries, third-party libs (Qt, OpenCV, ONNX Runtime), coding standards.

The *CleanShare* codebase is organized into directories that correspond to the system's logical modules. The ‘gui‘ directory contains Qt windows, widgets, and controllers. The core directory contains the application controller, workflow orchestration logic, and the session state. The detection directory includes ONNX Runtime integration and all preprocessing and postprocessing functions. The redaction directory implements the ROI mask creation and blur operations. The io directory handles image validation, loading, and saving. The eval directory contains the evaluation tools used in development. A ‘ third party ‘ directory stores external headers or wrappers, and a test directory includes all unit and integration tests.

The dependency rules ensure that ‘gui‘ depends only on core, and core depends on detection, redaction, and io. The detection module depends on OpenCV and ONNX Runtime, while redaction depends on OpenCV. The io module depends on Qt and OpenCV. The eval module uses detection and redaction. No module below ‘gui‘ depends upward on ‘gui‘. The system uses C++17, Qt 6 for the GUI, OpenCV 4 for image processing, ONNX Runtime for model inference, and ‘CMake‘ or ‘qmake‘ as its build system.

Configuration files such as ‘model.onnx‘ are stored in the program directory. User preferences may be stored using ‘QSettings‘ or a small JSON file. Unit tests independently verify each module, and integration tests run the entire pipeline on sample images to ensure correctness. This structure provides clarity for developers and supports modular development.

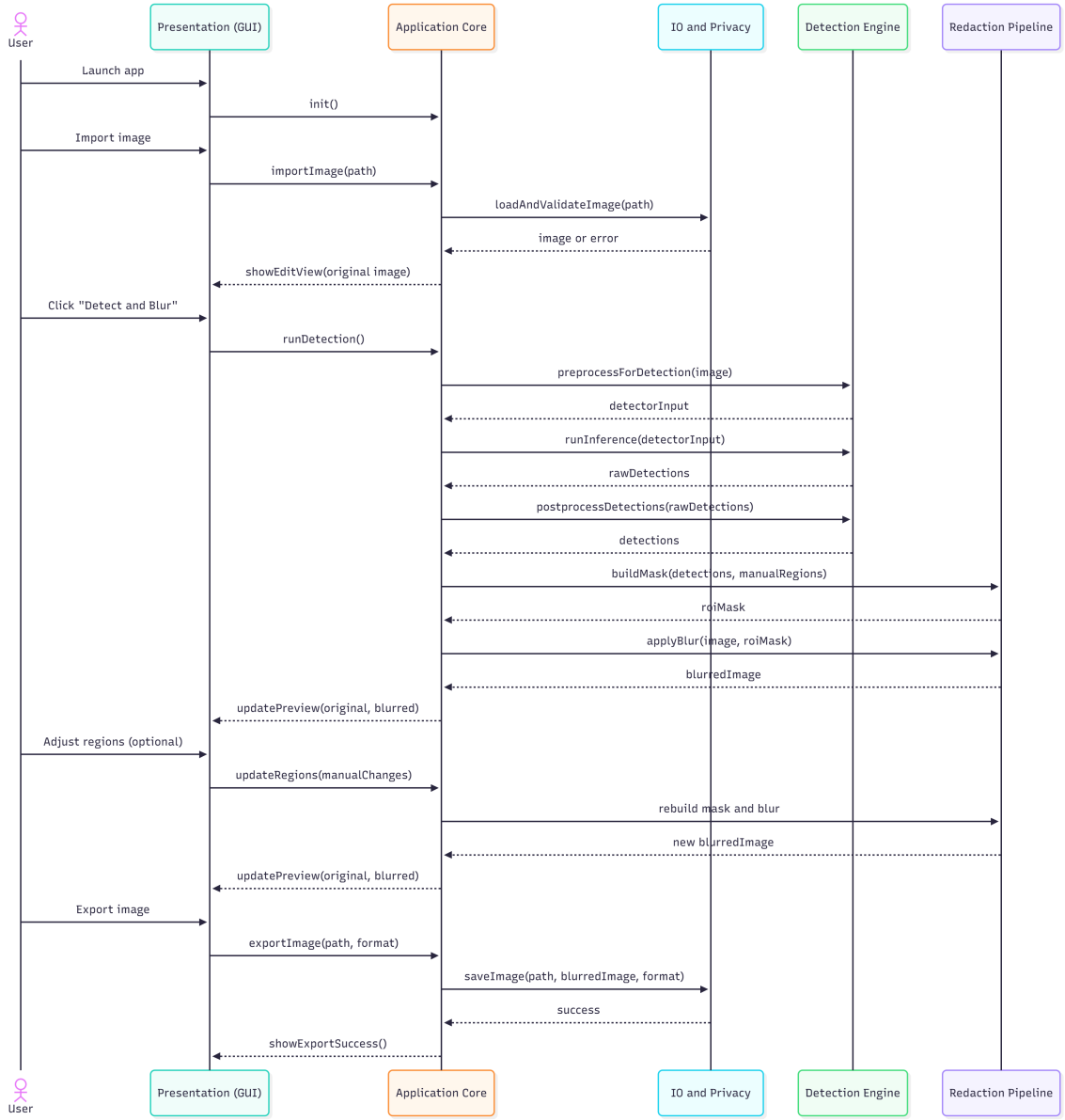

Figure 2: Process View Sequence

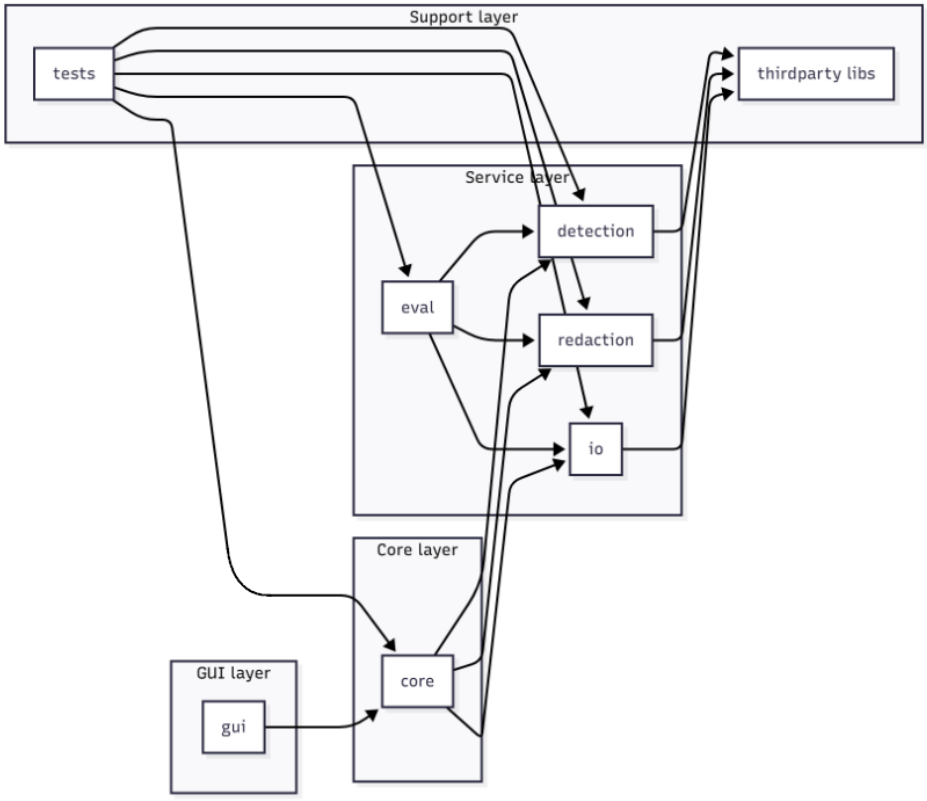

Figure 3: Development View

3.4 Physical/Deployment View

Target: Windows 10+ desktop; optional GPU. Packaging, installers, runtime dependencies.

CleanShare is deployed as a Windows desktop application running on Windows 10 or later. It requires at least 8 GB of RAM, and all inference is performed locally on the CPU, typically in less than 10 seconds for a 1080p image. GPU usage is optional. Network connectivity is not required, and the application does not transmit or store any private data externally.

The deployment artifacts include the main *CleanShare* executable, the ‘model.onnx‘ weight file, all required Qt, OpenCV, and ONNX Runtime libraries, an optional configuration file, and an optional evaluation executable. The deployment topology consists of a single user machine running a single application process with no external servers or databases. All images remain on the local machine.

The installation is performed through an MSI or EXE installer that installs the application, dependencies, and model file in appropriate program directories. Shortcuts may be registered. Uninstallation removes all application files, but leaves user exported images untouched.

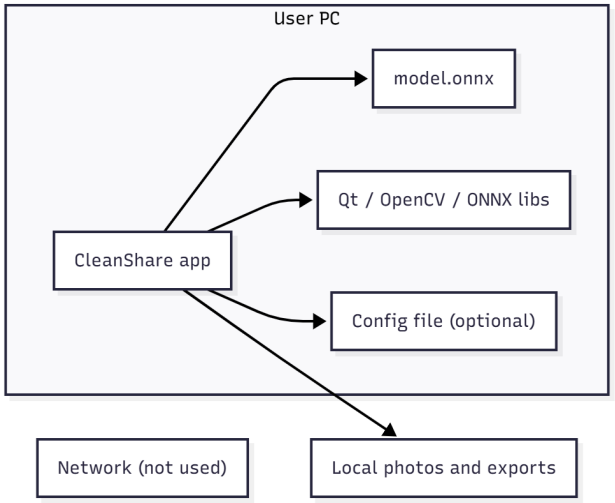

Figure 4: Physical/Deployment View

3.5 Scenarios (Use-Case View)

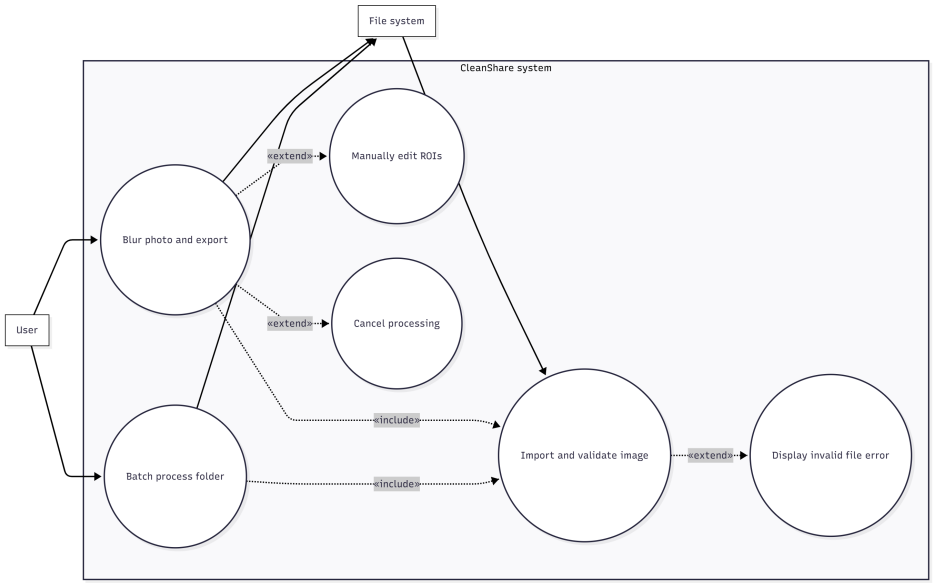

Figure 5: Use Case Diagram

- UC-1: Blur a photo and export (main path + alternate flows: no detections, invalid file, cancel).
  1. The user selects “Import image” in the GUI and chooses an image from the local file system.
  2. CleanShare validates the image format and loads it into the current session.
  3. The user starts automatic detection. The Detection Engine preprocesses the image, runs ONNX inference, and returns detections to the Application Core.
  4. The Application Core passes the detections to the Redaction Pipeline, which builds an ROI mask and applies the configured redaction style (for example blur or pixelation).
  5. The GUI displays a side-by-side preview of the original and redacted images.

6. The user optionally adjusts blur strength or redaction style, and the Redaction Pipeline updates the redacted result.
7. The user chooses "Export image" and selects an output path on the local file system.
8. CleanShare saves the redacted image and confirms successful export to the user.

- UC-2: Manually edit ROIs (add/remove, undo/redo).

1. The user activates the manual editing mode and selects a tool (brush or rectangle) in the GUI.
2. The user adds new ROIs by drawing on the image. Each new region is recorded by the Application Core and added to the current ROI mask.
3. The user removes or shrinks existing ROIs if detection was overly conservative.
4. The user uses undo and redo controls to refine edits until satisfied.
5. The Redaction Pipeline rebuilds the ROI mask and reapplies the chosen redaction style to produce an updated redacted image.
6. The GUI updates the preview to show the new redacted result.

- UC-3 (optional): Batch process folder.

1. The user selects "Batch process folder" and chooses a directory on the local file system.
2. CleanShare scans the folder for supported image files.
3. For each image, CleanShare performs the steps from UC-1 (import, validate, detect, redact, and export), using default redaction parameters.
4. Redacted copies are written to an output folder or written alongside the originals with modified filenames.

4 External Interface Design

4.1 User Interface (UI)

The following figures illustrate the foundational wireframes for CleanShare.

Figure 6 presents the home page, serving as the primary entry point for users. Figure 7 depicts the Terms of Service (TOS) acceptance pop-up displayed over the home page. Figure 8 shows the error message that appears when a file of an unsupported format is uploaded. Figure 9 demonstrates the preview screen, which displays both the preprocessed and processed images side by side. Figure 10 illustrates the edit screen, where users can employ a brush tool to manually apply blur to specific areas of the image.

These are wireframes only; the final implementation is expected to expand the visual design and functionality, including potential future batch processing and editing features.

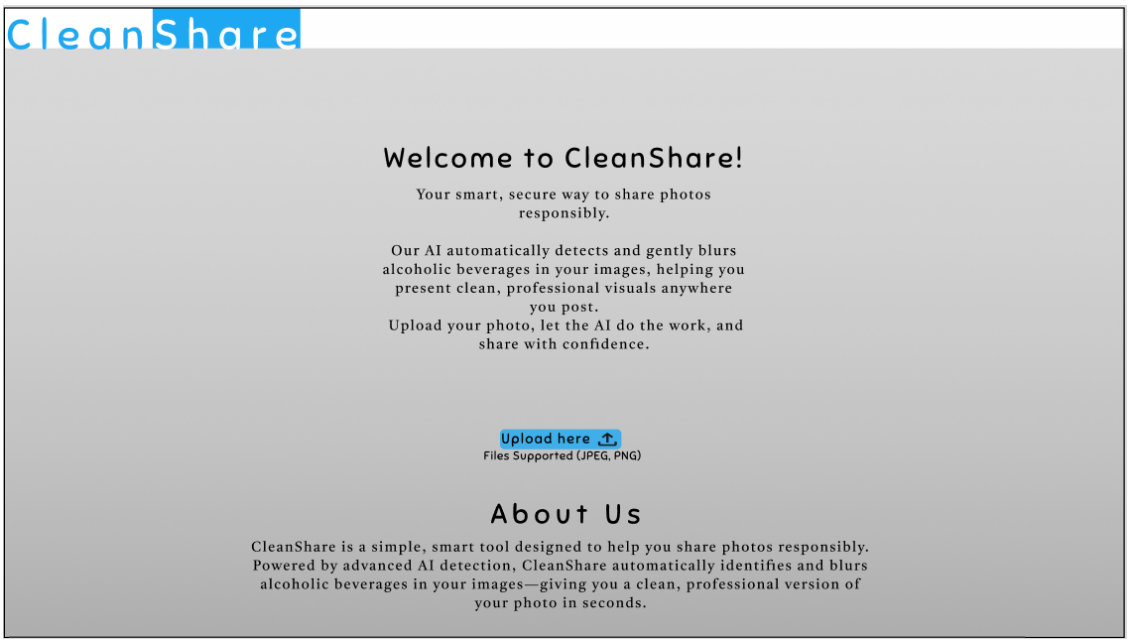

Figure 6: Home page wireframe: primary landing/upload screen for CleanShare.

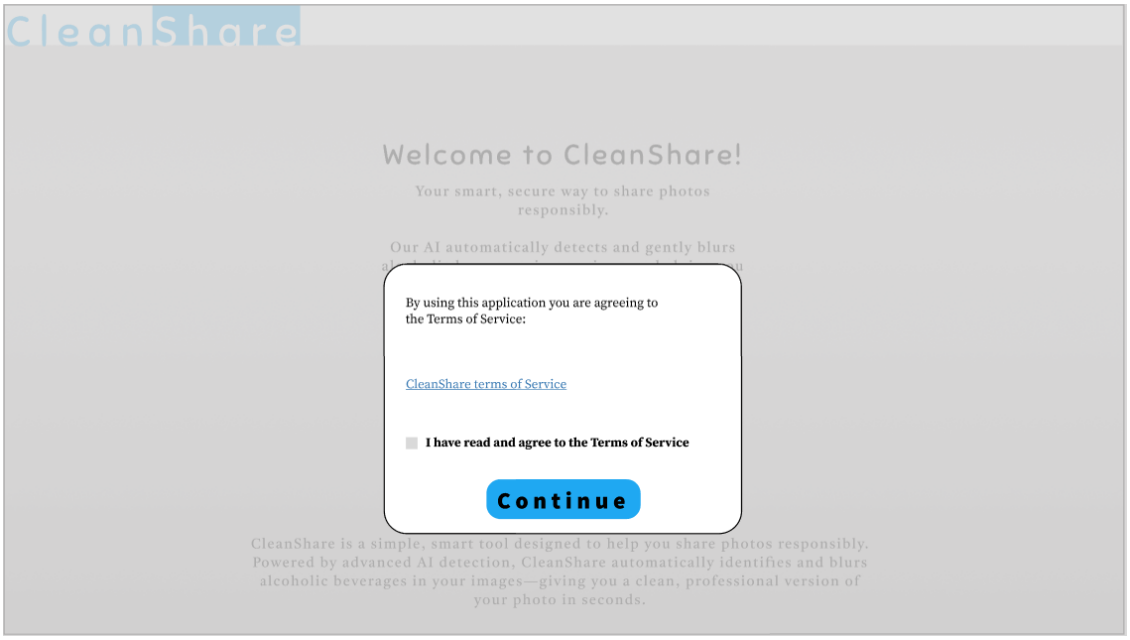

Figure 7: TOS acceptance pop-up shown over the home page.

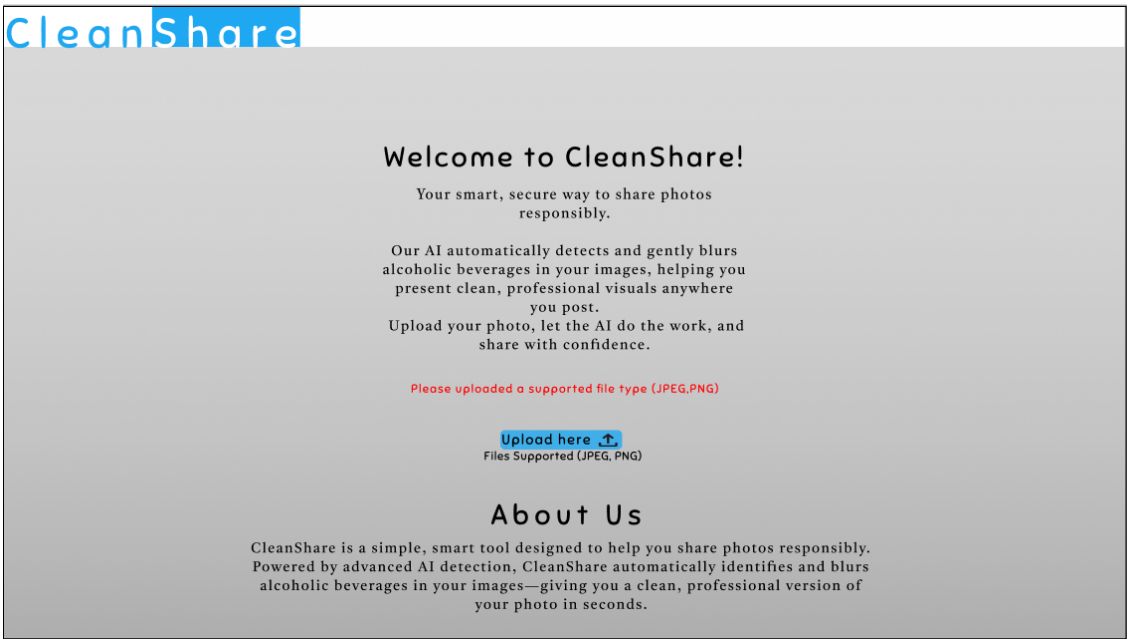

Figure 8: Unsupported file type error message.

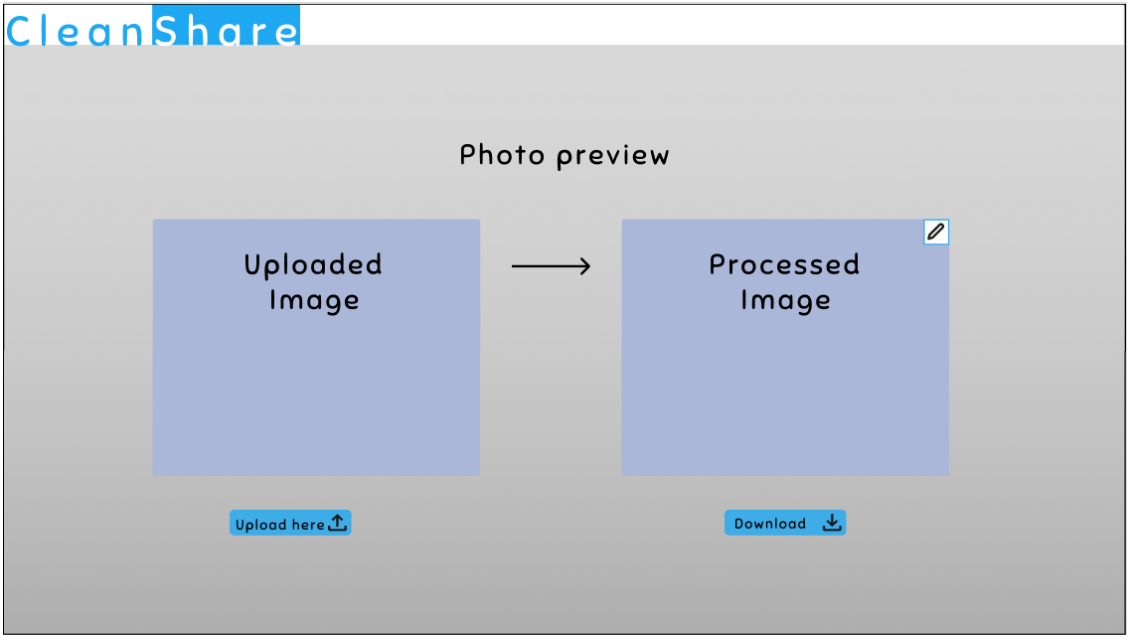

Figure 9: Preview screen showing preprocessed (original) and processed (blurred) images.

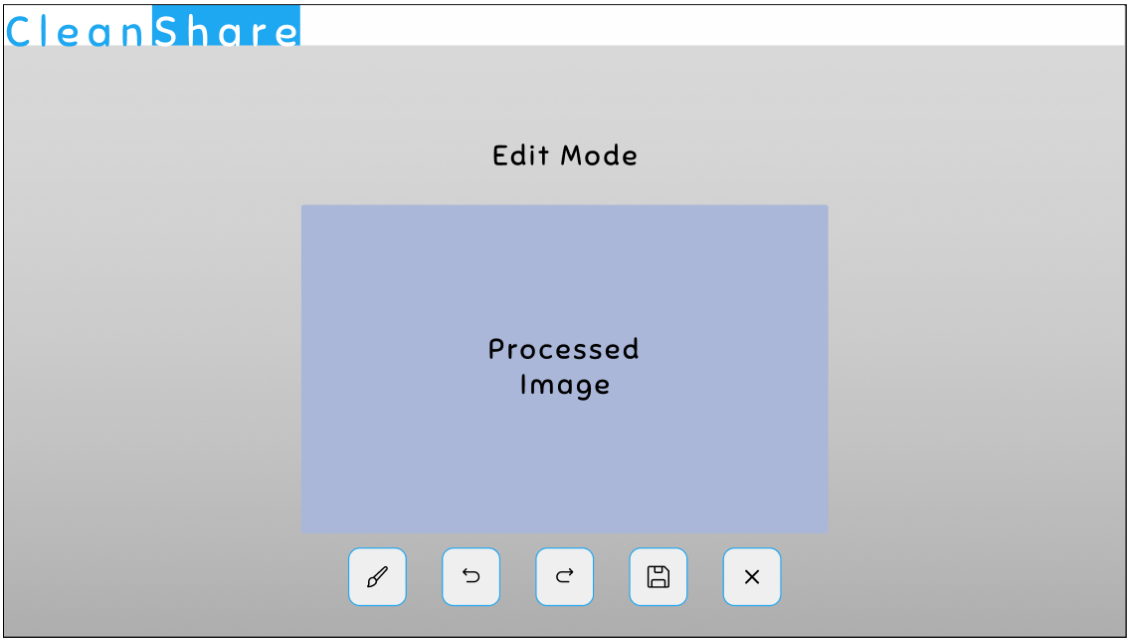

Figure 10: Edit screen with brush tool for manual blur adjustments.

4.2 File Interfaces

Supported Input/Output Types

CleanShare supports the import and export of JPEG (`.jpg/.jpeg`) and PNG (`.png`) image formats. These formats were chosen for their wide compatibility and high-quality support.

- **Input:** Users can import images from the local file system in the supported formats.
- **Output:** Processed images are exported in the same format as the input by default, with optional future support for explicit format selection.

Image Validation Rules

To ensure reliable processing and prevent errors, CleanShare validates images according to the following rules:

1. **File Type:** Only PNG and JPEG files are accepted.
2. **File Size:** Images exceeding 10 MB are rejected.
3. **Dimensions:** Minimum 200×200 pixels, maximum 8000×8000 pixels.
4. **Integrity:** Corrupted or unreadable files trigger an error message.
5. **Aspect Ratio:** Freeform aspect ratios are allowed; extremely large dimensions may be downscaled for performance.

Invalid images are reported immediately to the user through the GUI so that the current session state remains unaffected.

Export Naming and Versioning

To avoid file overwrites and maintain clarity, CleanShare uses the following structured naming convention for exported images:
`[original_filename]_[edit_mode]_v[version]_[YYYYMMDD_HHMMSS].[ext]`

- **original_filename:** Name of the input file.
- **edit_mode:** The redaction method applied (e.g., `brush`, `auto`).
- **version:** Incremental number for successive edits.
- **timestamp:** Exact export time.
- **ext:** File extension (matches input format by default).

This naming scheme ensures exported images are uniquely identifiable, self-descriptive, and traceable. The Application Core manages version increments and prevents filename conflicts automatically.

4.3 OS/Hardware Interfaces

CleanShare is designed for Windows 10 and later and interacts with the operating system through standard Windows APIs as well as optional GPU hardware acceleration.

Windows APIs

- **File Dialogs:** Uses native Windows file open/save dialogs (e.g., `GetOpenFileName`, `GetSaveFileName`) via Qt wrappers to allow users to select images for import and export.
- **Paths and Filesystem:** File path handling, validation, and directory scanning utilize standard Windows APIs or Qt's cross-platform `QFile`, `QDir`, and `QStandardPaths`.
- **Windowing & Events:** Qt leverages Windows message handling for GUI rendering, window management, and event loops.

Optional GPU

CleanShare performs all image processing and model inference on the CPU by default. If a compatible GPU is available, ONNX Runtime may optionally use CUDA or DirectML backends to accelerate model inference, reducing processing time for large images. GPU acceleration is intended to be transparent to the user; the application detects available hardware at runtime and selects the optimal execution provider.

Local Processing

- All operations are performed locally, with no network communication.
- Temporary buffers and ephemeral metadata remain in system memory or temporary files and are cleared upon completion.

This design ensures compatibility across standard Windows desktops while allowing optional hardware acceleration to improve performance for high-resolution images.

4.4 Model/Artifact Interfaces

CleanShare relies on a machine learning model stored as an ONNX (`.onnx`) file for automatic detection of sensitive regions in images. The system defines clear interfaces for locating, validating, and using this artifact to ensure reliable and reproducible results.

Model File Format

- The detection model is stored in ONNX format, allowing interoperability across frameworks and efficient inference with ONNX Runtime.
- The model includes all learned weights, preprocessing configurations, and metadata required for detection.

Versioning

- Each model file includes a version number embedded in its filename (e.g., `yolov11_v1.onnx`) and optionally within its metadata.
- Versioning ensures that updates to the detection model do not unintentionally overwrite previous versions, allowing reproducible results and backward compatibility with previously processed images.

Path Resolution

- The Application Core resolves the model path using relative paths within the program directory by default.
- The system may optionally allow user-specified model paths through configuration files or GUI selection for testing or evaluation purposes.
- Cross-platform path handling is abstracted through Qt's `QDir` and `QFile` APIs to ensure compatibility with Windows filesystem conventions.

Integrity Checks

- To ensure model reliability and prevent corrupted or tampered files, CleanShare performs hash-based integrity checks on the model file before loading.
- If a hash mismatch is detected, the Application Core reports an error to the user and halts automatic detection.
- This mechanism guarantees that the model used during inference matches the expected version and configuration.

5 Data Design

5.1 Core Data Structures

CleanShare's data model is intentionally small and image-centric. All core data structures are designed to (1) keep image and mask dimensions tightly aligned, and (2) avoid storing any long-lived user content beyond what is strictly needed for the active session.

Image representation. Internally, images are represented as an `Image` object, which wraps the underlying OpenCV matrix and relevant metadata:

- `cv::Mat pixels` (or equivalent): BGR 8-bit, 3-channel image buffer.
- `int width`, `int height`: pixel dimensions.
- `ImageFormat format`: logical format enum (`JPEG`, `PNG`, `Unknown`).
- `bool isPreview`: flag indicating whether this is a downscaled preview copy or the full-resolution original.

The following invariants are enforced:

- The "original" `Image` keeps the exact width, height, and format from disk.
- All masks and bounding boxes stored in the session are defined relative to the original image resolution.
- Preview images may be downscaled, but always maintain aspect ratio and a bijective mapping between preview coordinates and original coordinates.

Detection boxes and results. The Detection Engine exposes its outputs through a small set of data types:

- `BoundingBox`:
  - `float xCenter`, `float yCenter`: center of the box in normalized coordinates $[0, 1]$.
  - `float width`, `float height`: box size in normalized coordinates $[0, 1]$.
  - `float confidence`: model confidence score $[0, 1]$ after thresholding.
  - `int classId`: numeric class identifier (for now, typically a single "alcohol" class).
- `DetectionResult`:
  - `std::vector<BoundingBox> boxes`: final boxes after NMS and confidence filtering.
  - `std::string modelVersion`: semantic version tag of the ONNX model used.
  - `double inferenceMs`: measured inference time in milliseconds (used for performance evaluation).

Boxes are stored in normalized coordinates so they remain valid if the working image is temporarily resized for pre-processing; the Application Core is responsible for mapping them back to original-pixel coordinates when building masks and overlays.

Masks and manual edits. Redaction is implemented with binary masks aligned to the original image resolution:

- `Mask`:
  - Internally represented as a single-channel 8-bit `cv::Mat` with size (`height`, `width`).
  - Pixel value `0`: not masked (no blur applied).
  - Pixel value `255`: masked (blur applied).
- `Stroke` (for brush tools):
  - `std::vector<Point> points`: sampled cursor positions in original-image coordinates.
  - `int radius`: brush radius in pixels.
  - `StrokeMode mode`: enum (`Add`, `Erase`) indicating whether to add to or remove from the mask.
- `RectEdit` (for rectangle tools):
  - `Rect bounds`: top-left + bottom-right in original-image coordinates.
  - `RectMode mode`: enum (`Add`, `Erase`).

The Redaction Pipeline maintains at most three masks per session:

- `autoMask`: pixels derived solely from model detections.
- `manualMask`: pixels introduced or removed via manual tools.
- `combinedMask`: the effective ROI used for blur, derived by combining auto and manual masks.

Session state. All state needed for a single image is encapsulated in an `ImageSession` object owned by the `SessionController`. A typical definition is:

- `Image originalImage`;
- `Image previewImage`;
- `Image redactedImage`;
- `DetectionResult detections`;
- `Mask autoMask`;
- `Mask manualMask`;
- `Mask combinedMask`;
- `DetectionConfig detectionConfig`;
- `BlurConfig blurConfig`;
- `UndoStack<Command> undoStack`, `RedoStack<Command> redoStack`;
- `SessionState state`;

Key invariants:

- All masks and the `redactedImage` share the same resolution as `originalImage`.
- The undo/redo stacks only contain reversible operations on `manualMask`; they do not persist image data.
- When `state` is `Exported`, the in-memory data still represent the last previewed result, so a user can re-export without re-running detection.

5.2 Configuration

Configuration values are split between detection behaviour, redaction behaviour, and UI preferences. The Application Core treats these as structured objects so that they can be validated and overridden consistently.

Detection configuration (`DetectionConfig`). This structure controls how the Detection Engine filters model outputs:

- `float confidenceThreshold`: minimum score for a box to be kept (e.g., default 0.5).
- `float nmsIoUThreshold`: Intersection-over-Union threshold for NMS suppression (e.g., default 0.45).
- `int maxDetections`: optional cap on number of detections per image to bound runtime.
- `IntSize inputSize`: logical model input resolution (e.g., 640×640).

These values are typically initialized from the global configuration file and may be adjusted per session (for example, a "High Sensitivity" mode may lower `confidenceThreshold`). Any changes made via the GUI are reflected in the in-memory `DetectionConfig` but need not be persisted unless explicitly supported by the settings UI.

Blur configuration (`BlurConfig`). This structure encapsulates redaction parameters:

- `BlurType type`: enum selecting the active `BlurStrategy` (e.g., `Gaussian`, future `Pixelation`).
- `int kernelSize`: odd-valued kernel size in pixels for Gaussian blur (e.g., default 21).
- `double sigma`: standard deviation parameter for Gaussian blur (may be derived from kernel size).
- `float previewScale`: factor $(0, 1]$ used for generating preview images to balance speed and fidelity.

The Redaction Pipeline reads `BlurConfig` on every call to `apply` so changes to blur strength or type take effect immediately.

UI preferences (`UiPreferences`). Lightweight, non-critical preferences are grouped separately so they can be persisted without touching core algorithm configuration:

- `QString lastOpenDir`, `QString lastExportDir`;
- `bool showGridOverlay`;
- `bool syncZoomBetweenPanes`;
- `Theme theme`: enum (`Light`, `Dark`, `SystemDefault`).

Defaults and override order. CleanShare uses the following precedence when resolving configuration values:

1. **Compile-time defaults**: safe baseline values hard-coded in the application.
2. **Config file**: if present and valid, overrides the compile-time defaults.
3. **Session overrides**: changes made by the user in the running session (e.g., slider for blur strength).

Session overrides live only in memory; they are discarded when the application closes unless explicitly written back to the config file by a future "Save Settings" feature.

5.3 Persistence Strategy

The persistence strategy is deliberately minimal to maintain the privacy-first requirement. Only three kinds of artifacts are ever written to disk:

- **Exported images**:
  - Written to a user-selected path via `IOService::saveImage`.
  - Preserve original resolution and, by default, original format (JPEG in → JPEG out, PNG in → PNG out) unless the user explicitly chooses a different extension.
  - Never overwritten without an explicit user confirmation handled by the standard file-save dialog.
- **Configuration file**:
  - A small file (e.g., `config.json` or `settings.ini`) located in the application directory or a standard per-user settings location.
  - Stores non-sensitive data: detection thresholds, blur defaults, and UI preferences.
  - If the file is missing or corrupted, CleanShare falls back to compile-time defaults and may regenerate a fresh configuration on next launch.
- **Local logs** (optional):
  - A rotating text log (e.g., `logs/cleanshare.log`) for debugging; capped in size or age.

  - Contains only high-level events and timing (e.g., "image loaded", "inference took 820 ms").
  - Never stores raw pixels, masks, bounding boxes, or user file paths beyond what is unavoidable for diagnosing errors (and even then, file paths may be truncated or hashed).

The system *does not* persist the following:

- In-memory images, masks, or detection results once the session is closed.
- Any temporary files in the application's own directories; short-lived intermediates are kept in RAM.
- User identifiers, usage analytics, or telemetry of any kind.

Clean-up policy is simple by design:

- Exported images are considered user-owned and are not modified or deleted by CleanShare.
- Log files may be trimmed at startup (for example, deleting logs older than a fixed retention window or truncating to a maximum size).
- If a future evaluation tool writes metrics (e.g., CSV reports) for developers, it does so in a clearly separate directory and is disabled in end-user builds.

5.4 Internationalization/Localization (if any)

For the current course deliverable, CleanShare targets an English-only audience, so internationalization is kept intentionally minimal. The data design, however, avoids making localization impossible later:

- All user-facing strings are exposed through Qt mechanisms (e.g., `QString` and resource files), rather than hard-coding narrow character types; this allows future extraction into `.ts` translation files.
- File paths and image names are stored as `QString` to support Unicode filenames on Windows, independent of locale.
- Configuration keys are ASCII-only, but their values (such as last-used directories) can contain arbitrary Unicode as supported by the OS.
- The system does not store or present any locale-specific date/time or number formats in persistent data; all timing information used for performance evaluation is internal and, if logged, is written in a simple numeric form.

As a result, the current build can reasonably assume an English UI while leaving a straightforward path to future localization: UI text can be externalized into translation catalogs without changing the underlying data structures or file formats.

6 Component Design

This section refines the logical modules from the RAD into concrete runtime components and their interfaces. CleanShare is implemented as a Qt-based Windows desktop application with a modular architecture: a Presentation layer (GUI), an Application Core that orchestrates workflow and session state, a Detection Engine (OpenCV + ONNX Runtime), a Redaction Pipeline, and an I/O & Privacy component for file handling and guarantees of local processing.

6.1 Class/Component Diagrams

Figure 11 shows the high-level component/class diagram for CleanShare. The main components are:

- **MainWindow / Qt GUI** (Presentation): handles user interaction (open image, run detection, adjust blur regions, export image), displays the side-by-side preview, and exposes toolbar actions (undo/redo, brush/rectangle tools, blur strength).
- **SessionController** (Application Core): coordinates the end-to-end workflow *import* → *detect* → *blur* → *preview* → *export*. It maintains a single active `ImageSession` object with all transient state (original image, scaled working copy, detection results, masks, configuration).
- **DetectionEngine** (Detection Engine): encapsulates image pre-processing, CNN inference via ONNX Runtime, optional heuristic detection, non-maximum suppression (NMS), and confidence thresholding. It returns a list of normalized bounding boxes and class scores.
- **RedactionPipeline** (Redaction Pipeline): converts detection boxes and user-edited regions into a final ROI mask and applies the chosen blur strategy while preserving the original resolution.
- **BlurStrategy** and concrete strategies (e.g., `GaussianBlurStrategy`, future `PixelationBlurStrategy`): plug-in components that implement the actual redaction effect for a given ROI mask.
- **IOService** (I/O & Privacy): responsible for loading/saving JPEG/PNG files, validating formats, and ensuring that all image data stays local and ephemeral.
- **EvaluationRunner** (Evaluation utility, dev-only): runs the detection engine on a labeled dataset to compute recall, false positives, and timing against the success criteria; not part of the end-user GUI but included for maintainability and verification.

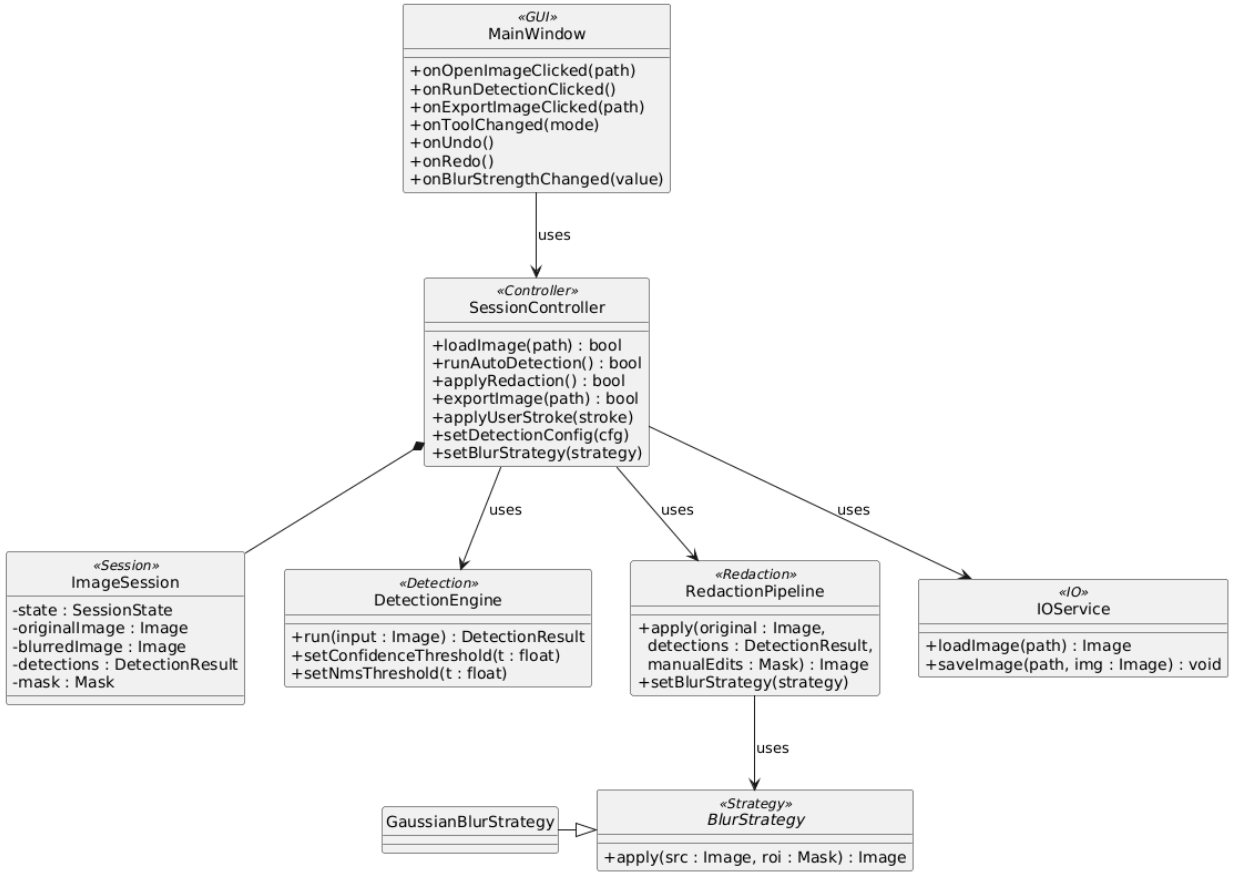

Figure 11: High-level component/class diagram for CleanShare.

6.2 Component Responsibilities & APIs

For each major component, we document its purpose, key responsibilities, main public methods, and important invariants.

MainWindow / Qt GUI (Presentation)

Purpose: Provide an intuitive interface so a first-time user can import, blur, preview, and export an image in under 30 seconds.

Key responsibilities:

- Render landing page (drag-and-drop area, “Upload” button) and edit/view page (before/after preview, controls).
- Dispatch user actions (open, detect, export, undo/redo, tool selection) to `SessionController`.
- Display progress indicators and error messages (invalid file, slow inference, no detections).

Representative API (Qt signals/slots simplified):

- `slot onOpenImageClicked(QString filePath);`
- `slot onRunDetectionClicked();`
- `slot onExportImageClicked(QString filePath);`
- `slot onToolChanged(ToolMode mode);`

- `slot onUndo();  slot onRedo();`
- `slot onBlurStrengthChanged(int value);`

Invariants:

- At most one active `ImageSession` is bound to the GUI at a time.
- All visible preview images correspond to the currently active session state.

SessionController (Application Core)

Purpose: Central coordinator that enforces the main use case "Blur a photo and export" and maintains consistent session state.

Key responsibilities:

- Create and own the `ImageSession` object when a new image is loaded.
- Orchestrate pre-processing, detection, redaction, and export by delegating to `DetectionEngine`, `RedactionPipeline`, and `IOService`.
- Manage configuration (detection threshold, blur type) and propagate changes to the relevant components.
- Integrate manual edits (brush/rectangle add/remove) into the current ROI mask and manage an undo/redo stack.

Representative API:

- `bool loadImage(const QString& path);`
- `bool runAutoDetection();`
- `bool applyRedaction();`
- `bool exportImage(const QString& path);`
- `void applyUserStroke(const Stroke& stroke);`
- `void setDetectionConfig(const DetectionConfig& cfg);`
- `void setBlurStrategy(std::shared_ptr<BlurStrategy> strategy);`

Invariants:

- `ImageSession` is either in state *Idle*, *Loaded*, *Detected*, *Edited*, or *Exported* (see state machine).
- All detection boxes and masks stored in the session are aligned with the current working image resolution.

DetectionEngine

Purpose: Encapsulate all model-related work: image pre-processing, CNN inference (ONNX Runtime), optional heuristic detection, and post-processing.

Key responsibilities:

- Convert the loaded image into the model's expected input format (resize, normalize, channel order).
- Run the YOLOv11-derived ONNX model on CPU and measure latency.
- Apply confidence threshold and NMS to filter boxes.
- Optionally merge heuristic OpenCV detections as a fallback.

Representative API:

- `DetectionResult run(const Image& input);`
- `void setConfidenceThreshold(float t);`
- `void setNmsThreshold(float t);`

Invariants:

- Each `DetectionResult` is tied to exactly one input image and stores boxes in normalized coordinates.
- The engine is stateless between calls except for configuration parameters.

RedactionPipeline

Purpose: Convert detections + user edits into a final blurred image while preserving the original resolution.

Key responsibilities:

- Build an ROI mask from detection boxes and user-applied strokes (add/remove regions).
- Invoke the selected `BlurStrategy` to apply the blur to the ROI.
- Ensure that the output image has identical dimensions and format to the original.

Representative API:

- `Image apply(const Image& original, const DetectionResult& detections, const Mask& manualEdits);`
- `void setBlurStrategy(std::shared_ptr<BlurStrategy> strategy);`

Invariants:

- The redacted image always matches the original resolution.
- ROI masks are binary (masked vs not masked); no pixels outside the mask are modified.

BlurStrategy and Concrete Strategies

Purpose: Separate "what gets blurred" (mask) from "how it is blurred" (effect), allowing future effects without changing the pipeline.

Key responsibilities:

- Provide a common interface for different blur implementations.
- Implement specific algorithms (Gaussian blur now; pixelation/mosaic later).

Representative API:

- `virtual Image apply(const Image& src, const Mask& roi) = 0;`

Invariants:

- Derived strategies must not modify pixels outside the ROI mask.

IOService (I/O & Privacy)

Purpose: Handle all disk access while respecting the "local-only, no cloud" constraint.

Key responsibilities:

- Validate file type and size (JPEG/PNG only).
- Read images into the internal representation used by the application.
- Write out blurred images at the original resolution and format.

Representative API:

- `Image loadImage(const QString& path) throw(InvalidFileException);`
- `void saveImage(const QString& path, const Image& img);`

Invariants:

- No images or metadata are persisted beyond the exported file the user explicitly saves.
- Temporary files (if any) are deleted at the end of the session.

EvaluationRunner (Dev Utility)

Purpose: Support offline evaluation against the success criteria (recall, false positives, processing time) without modifying production code.

Key responsibilities:

- Load a labeled dataset of images and ground-truth boxes.
- Call `DetectionEngine` and compute recall, false positives, and mAP.
- Record per-image processing times.

6.3 Design Patterns

Several design patterns are used to keep the system modular, testable, and easy to extend:

- **Model–View–Presenter (MVP) for the GUI:**
  - *View*: `MainWindow`, preview widgets.
  - *Presenter/Controller*: `SessionController` translates user actions into operations on the model.
  - *Model*: `ImageSession` and underlying components (DetectionEngine, RedactionPipeline).

  This keeps GUI code thin and localizes workflow logic in the core.

- **Strategy for blur effects:**
  - Abstract `BlurStrategy` defines the interface.
  - Concrete classes (e.g., `GaussianBlurStrategy`, future `PixelationBlurStrategy`) implement different visual effects.
  - The pipeline chooses a strategy at runtime based on user settings, enabling future extensions without changing callers.

- **Strategy for detection backends (optional extension):**
  - Interface `IDetectionEngine` with implementations such as `OnnxDetectionEngine` and `HeuristicDetectionEngin`
  - Allows switching between pure CNN inference and a lighter heuristic mode if performance or model availability changes.

- **Adapter / Facade for ONNX Runtime:**
  - A thin wrapper (e.g., `OnnxRuntimeAdapter`) hides the low-level ONNX API and exposes a simple `run(Image)` call.
  - This isolates external library details and simplifies testing and replacement of the model.
- **Command for undo/redo:**
  - Each manual operation (brush add, brush erase, rectangle add/remove) is represented as a command (`ApplyStrokeCommand`, `EraseRegionCommand`).
  - Commands know how to `execute()` and `undo()`, enabling robust undo/redo as required by the UX constraints.

7 Dynamic Behavior

This section describes how the components collaborate at runtime. The focus is on the main use case "Blur a photo and export" and on the internal state transitions of the image session, tool selection, and inference jobs.

7.1 Sequence Diagrams

Figure 12 shows the sequence diagram for the primary flow "Blur a photo and export". The main lifelines are *User*, *MainWindow*, *SessionController*, *IOService*, *DetectionEngine*, *RedactionPipeline*, and *FileSystem*.

The typical interaction is:

1. **Import:** The User selects an image file. `MainWindow` calls `SessionController::loadImage()`, which delegates to `IOService::loadImage()`. On success, a new `ImageSession` is created in state *Loaded* and the original image is shown in the "before" pane.
2. **Detect:** The User clicks "Auto Detect & Blur". `MainWindow` calls `SessionController::runAutoDetection()`, which invokes `DetectionEngine::run()` on the current image. Detection results are stored in the session and the state moves to *Detected*.
3. **Redact:** `SessionController` immediately calls `RedactionPipeline::apply()` with the original image, detection results, and any existing manual mask. The selected `BlurStrategy` is applied, producing the blurred image. The session state becomes *Edited*.
4. **Preview & Adjust:** The blurred image is shown in the "after" pane. If the User uses the brush/rectangle tools, `MainWindow` reports strokes to `SessionController::applyUserStroke()`, which updates the mask, re-runs the pipeline, and pushes commands onto the undo stack.
5. **Export:** When satisfied, the User clicks "Export". `MainWindow` calls `SessionController::exportImage()`, which calls `IOService::saveImage()` with the current blurred image. On success, the session may move to state *Exported*.

Alternate flows (e.g., invalid file, no detections, inference error) follow the same structure but include error messages and, for "no detections", skip the detection results while still allowing manual marking.

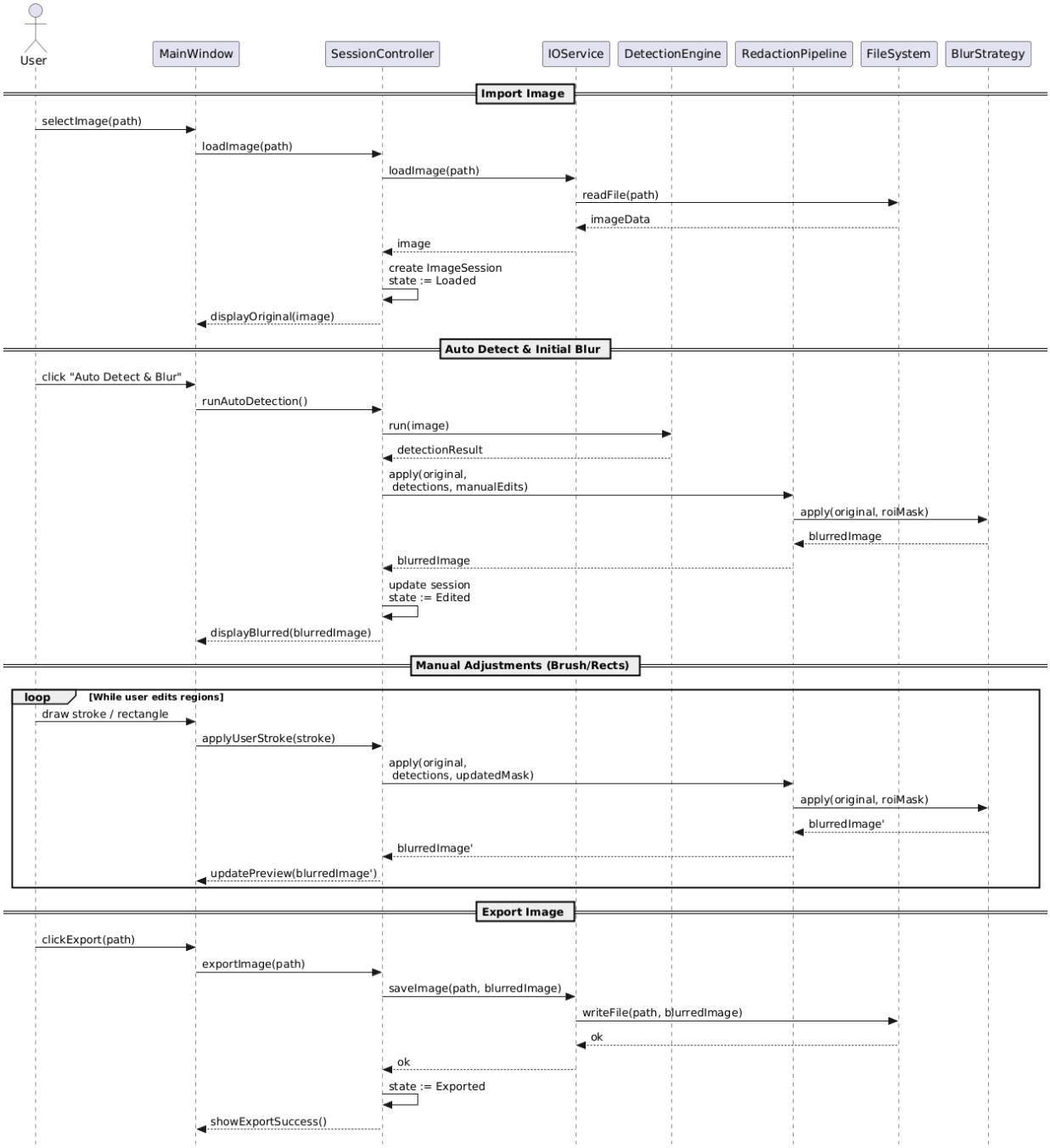

Figure 12: Main flow for importing, detecting, blurring, previewing, and exporting an image.

7.2 State Machines

CleanShare has several important stateful behaviours. We model them with state machines to clarify legal transitions and simplify error handling.

ImageSession Lifecycle

The `ImageSession` object tracks the end-to-end state of the current image:

- **Idle:** No image loaded yet (initial state after app launch or after closing a session).

- **Loaded:** A valid image has been loaded; no detections have been run.
- **Detected:** Automatic detection has completed successfully; detection results are stored.
- **Edited:** At least one redaction pass has been applied (automatic or manual). Manual brush/rectangle operations keep the session in this state while updating the mask.
- **Exported:** The current blurred image has been successfully saved to disk for this session.
- **Error** (transient): An error occurred (invalid file, inference failure); the controller reports the error and typically resets the session to *Idle* or *Loaded* depending on the scenario.

Typical transitions:

- *Idle* $\xrightarrow{\texttt{loadImage}}$ *Loaded*
- *Loaded* $\xrightarrow{\texttt{runAutoDetection}}$ *Detected*
- *Detected* $\xrightarrow{\texttt{applyRedaction}}$ *Edited*
- *Edited* $\xrightarrow{\texttt{exportImage}}$ *Exported*
- Any non-*Idle* state $\xrightarrow{\texttt{closeSession}}$ *Idle*
- Any state $\xrightarrow{\texttt{error}}$ *Error* $\rightarrow$ recovery (*Idle* or *Loaded*)

Tool Mode (Manual Editing)

A separate state machine models the current tool mode for manual blur region editing:

- **NoneSelected**: No manual tool is active (default).
- **BrushAdd**: Painting over regions to add them to the blur mask.
- **BrushErase**: Painting to remove regions from the blur mask.
- **RectAdd**: Drawing rectangles to add regions to the blur mask.
- **RectErase**: Drawing rectangles to remove regions from the blur mask.

Transitions are triggered by toolbar button clicks:

- *NoneSelected* $\xrightarrow{\texttt{selectBrushAdd}}$ *BrushAdd*, etc.
- Selecting an already-active tool either keeps the state or returns to *NoneSelected* (toggle behaviour).
- Selecting a different tool transitions directly between tool states (e.g., *BrushAdd* $\xrightarrow{\texttt{selectRectErase}}$ *RectErase*).

Inference Job State

Finally, the detection process itself is modelled as a simple job state machine:

- **Idle**: No inference is in progress.
- **Running**: The ONNX model is currently executing on the image.
- **CancelRequested**: The user has requested cancellation (if supported); the engine should stop at the next safe opportunity.
- **Completed**: Inference finished successfully and results were delivered to the session.
- **Failed**: An error occurred (e.g., missing weights, invalid model); an error message is shown.

Transitions:

- *Idle* $\xrightarrow{\texttt{run}}$ *Running*
- *Running* $\xrightarrow{\texttt{cancel}}$ *CancelRequested* $\rightarrow$ *Idle*
- *Running* $\xrightarrow{\texttt{success}}$ *Completed* $\rightarrow$ *Idle*
- *Running* $\xrightarrow{\texttt{error}}$ *Failed* $\rightarrow$ *Idle*

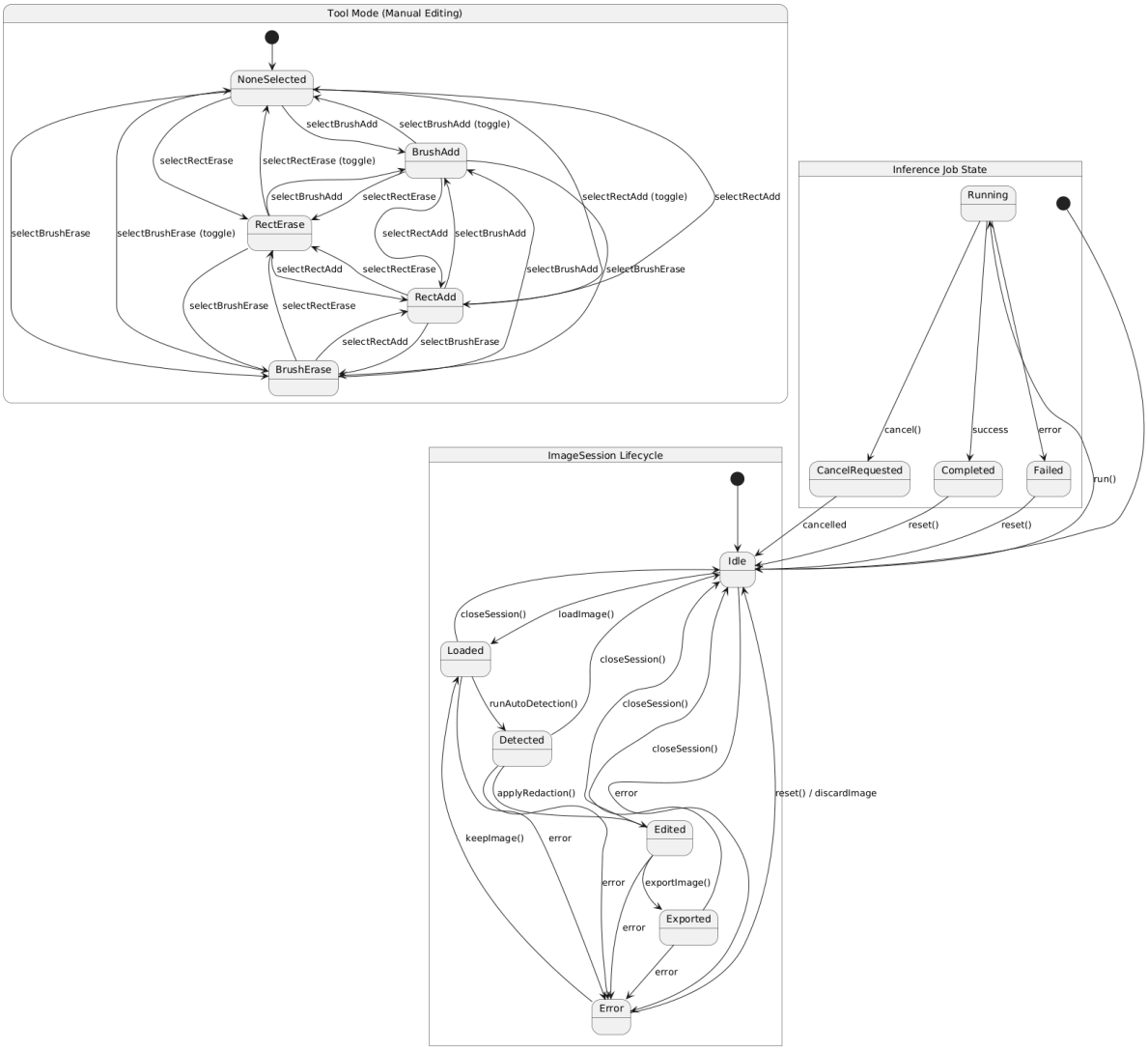

Figure 13: Statecharts for the image session, tool mode selection, and inference job.

8 Error Handling, Logging, and Telemetry

8.1 Error Classes and Recovery

CleanShare operates entirely locally, with all processing executed on the user's machine. Error handling follows a strict fail-fast model: any invalid input or failed detection step is surfaced immediately to the user with a deterministic recovery path. Errors are categorized into four classes: input errors (unsupported

or corrupted image types), processing errors (OpenCV preprocessing or ONNX inference faults), export errors (disk write failures), and UI errors (invalid user actions in manual editing mode). Input errors terminate the pipeline before preprocessing begins. Processing errors trigger an internal safe-state rollback, clearing intermediate buffers and reverting the UI to the import state. Long-running detections that exceed a configured timeout or encounter out-of-memory (OOM) conditions are treated as processing errors: the operation is aborted, the user is notified, and the session is rolled back to a known-good state. User-facing messages are concise, actionable, and tied to a clear next step (e.g., "Unsupported file format – please select a PNG or JPEG image" or "Detection failed – you can continue in manual blur mode"). Export errors preserve the in-memory processed image and prompt user re-selection of output path. UI errors reset only the active tool state, ensuring manual controls never corrupt session data. When automatic detection is unavailable or fails repeatedly, the system degrades gracefully to a "manual-only" fallback mode where users can still draw and adjust blur regions and safely export the result.

8.2 Logging

Logging is strictly local and minimal by design. A rotating text-based log tracks operational events: image load, preprocessing start, inference start and duration, blur operations, failed detections, and export completion. Log entries are tagged with standard levels (INFO for normal operations, WARNING for recoverable issues, and ERROR for failures that trigger rollback) so that developers and advanced users can filter noise when debugging. Sensitive data—including pixel arrays, bounding boxes, or user images—is never written to disk or transmitted externally. Telemetry is deliberately excluded: the system collects no analytics, no model-usage statistics, and no user identifiers. Timing data used for performance debugging is computed in-memory and discarded when the application closes. No log entry persists beyond the local session unless the user manually chooses to retain diagnostic output.

9 Security & Privacy Design

9.1 Privacy Principles

The system's security posture follows a privacy-first model consistent with the RAD: all computations execute offline, no data is transmitted, and no external services are invoked. Information disclosure is mitigated by avoiding temporary disk writes entirely; image buffers and CNN outputs remain in volatile memory. No persistence layer exists beyond the user-requested export, and the system deletes all intermediate arrays immediately after export or cancellation. The architecture prohibits cloud fallback; any network request from the application is categorically disabled. This design ensures compliance with institutional privacy expectations described in the RAD.

9.2 Threat Model (STRIDE-lite)

The threat model is scoped to a workstation-local environment using a STRIDE-lite analysis. Spoofing is irrelevant because no authentication or accounts exist. Tampering concerns arise only in filesystem interactions; the application validates image metadata and dimensions before use, rejecting malformed or oversized files. Repudiation risks are null because the application stores no user actions or long-term logs. Denial of service threats are restricted to pathological inputs (extremely large images or malformed files), managed by hard caps on resolution and memory allocation. Elevation of privilege is out of scope due to the absence of privileged operations or plugins. The primary assets considered in this model are user-supplied images, redacted outputs, the on-disk detection model, and any in-memory intermediate representations (feature maps, masks, and configuration values). Trust boundaries are defined between the user and the desktop OS, between the OS and the CleanShare process, and internally between the Qt UI, application core, and detection/redaction subsystems; untrusted input is validated before crossing each boundary. Model integrity is protected by validating the ONNX model artifact at startup (for example, checking its expected path, size, and version tag) and refusing to execute inference if the file appears missing, corrupted, or tampered.

9.3 Access Control

Access control is simplified: the sole user has implicit full execution rights, but the application enforces internal boundaries between UI, application core, and detection pipeline. The detection engine and redaction pipeline never expose raw model output externally. File system access is further constrained by the underlying operating system; CleanShare relies on standard OS-level file permissions (and any platform sandboxing) to determine which images can be opened or written and does not attempt to bypass these controls.

10 Performance, Capacity, and Quality Attributes

10.1 Targets and Budgets

CleanShare's performance and quality targets are derived directly from the RAD and are treated as hard engineering constraints rather than best-effort goals:

- $\leq$10 s per 1080p image (CPU baseline) on a typical Windows 10+ laptop with $\geq$8 GB RAM.
- $\geq$80% recall on held-out validation images, with a low false positive rate (target $\leq$15%).
- GUI responsiveness $\leq$300 ms for user actions (button presses, tool changes, slider moves).
- First-time user can complete the core workflow (import $\rightarrow$ detect $\rightarrow$ blur $\rightarrow$ export) in $\leq$30 s.

These targets are measured using a repeatable evaluation setup rather than ad hoc "feel tests":

- **Hardware test rig.** A representative "baseline" machine is defined as a Windows 10+ laptop with a mid-range CPU (for example, a 4-core Intel i5 or Ryzen 5), 8 GB RAM, and no discrete GPU. All timing numbers and capacity claims in this document assume this baseline; GPU acceleration is treated as a bonus, not a requirement.
- **Workload.** For performance tests, a fixed corpus of 1080p images is used. This corpus is drawn from the held-out Liquor Data subset plus additional distractor images (non-alcoholic scenes, soda cans, cluttered backgrounds) as described in the RAD's validation section. Each candidate build processes the entire corpus end-to-end (import $\rightarrow$ detect $\rightarrow$ blur $\rightarrow$ export).
- **Metrics and collection.** The EvaluationRunner (Sec. 6.2) records per-image:
  - Detection metrics (recall, precision, false positive rate, mAP@0.5) against ground truth labels.
  - End-to-end processing time, split into pre-processing, inference, redaction, and export.
  - Peak resident memory for the process (where available from OS tooling).

  For runtime, both median and 95th-percentile latencies must be below the 10 s target for 1080p images (Sec. 13.2).
- **Pass/fail criteria.** A build fails the performance/quality gate if any of the following hold:
  - Dataset recall $<$ 80% or false positive rate $>$ 15%.
  - Median 1080p processing time $>$ 10 s on the baseline rig.
  - GUI latency for simple actions (opening dialogs, toggling tools, updating sliders) exceeds 300 ms in UX smoke tests.

  These thresholds match the acceptance criteria in Sec. 13.3 and the RAD's non-functional requirements.

This approach ensures that performance and detection quality are continuously verified in an automated, dataset-backed way rather than only during manual demos.

10.2 Scalability Considerations

CleanShare is a single-user, single-desktop application. "Scalability" therefore focuses on how gracefully the system handles larger images, heavier workloads, and potential future batch mode, rather than multi-user throughput.

Image size and memory bounds.

- **Supported resolutions.** The design targets 1080p as the baseline and supports higher resolutions (e.g., 1440p, 4K) subject to available RAM. All internal masks and redacted images are kept at the original resolution to preserve export fidelity (Sec. 5.1).
- **Pre-processing strategy.** The DetectionEngine resizes a working copy of the image to the model's fixed input size (e.g., 640×640) and stores detections in normalized coordinates. This keeps model inference cost approximately constant per image, regardless of original resolution; only pre/post-processing time scales with pixel count.
- **Hard caps.** To avoid pathological inputs causing out-of-memory conditions, the IOService enforces a maximum allowable resolution (for example, rejecting images above a configured megapixel limit or prompting the user to downscale). Extremely large images are either rejected with a clear error or automatically downsampled for processing while maintaining correct mapping back to original coordinates for masks and overlays.

Runtime behaviour under heavier workloads.

- **Single-image pipeline.** For the current release, the application processes one image at a time on a single worker thread (Sec. 3.2). This guarantees that CPU and memory use remain bounded and predictable.
- **Batch mode (future extension).** The optional future batch feature (Sec. 3.5, UC-3) is designed as a thin loop over the existing single-image pipeline. Images are processed sequentially, reusing the same DetectionEngine and RedactionPipeline instances to avoid repeated model loading. This provides coarse-grained scalability to folders of images without changing the core complexity or memory footprint per image.
- **Complexity.** The dominant costs scale approximately linearly with the number of pixels in the working image for OpenCV operations (blur, mask composition) and with the number of detections for NMS. Since YOLO-style detectors typically produce a bounded number of high-confidence boxes per class, NMS overhead remains small even on busy scenes.

Capacity and degradation.

- **Graceful degradation.** If memory allocation for an operation fails (for example, attempting to allocate a blur kernel for an oversized image), the system aborts the current operation, surfaces an error, and encourages the user to use a smaller image. The session is rolled back to a safe state (Sec. 8.1).
- **No background queue.** CleanShare does not maintain a long-running job queue or background tasks beyond the active worker thread; this avoids hidden memory growth over time. Capacity concerns are therefore limited to "one large image at a time" plus minor GUI overhead.

Overall, the design ensures that CleanShare scales predictably from modest laptop hardware up to more powerful desktops, with clear boundaries on what image sizes are supported and how future batch features will reuse the same core pipeline.

10.3 Usability & Accessibility

Usability and accessibility are treated as explicit quality attributes and are aligned with the RAD's UX requirements (Sec. 9.2) and the primary use-case scenarios (Sec. 3.5):

Interaction model and responsiveness.

- **Simple primary workflow.** The main workflow is linear and visible: "Import image", "Auto Detect & Blur", optional manual adjustments, and "Export". Each step is reachable from the main window with at most one click, satisfying the "all major operations in ≤ 3 actions" requirement.
- **Non-blocking UI.** All long-running work (model inference, full-resolution redaction) runs on a background worker thread; the Qt GUI thread remains free to repaint and respond to input (Sec. 3.2). While a job is running, the user sees a progress indicator and can cancel if needed, keeping perceived responsiveness under the 300 ms target for all simple interactions.
- **Undo/redo safety.** Manual edits (brush/rectangle) are implemented via a Command-based undo stack (Sec. 6.3). Commands only manipulate masks and never touch the original pixels, ensuring that undo/redo is fast, predictable, and cannot corrupt the base image.

Discoverability and error feedback.

- **High-contrast, labeled controls.** Buttons and tool icons use clear text labels or tooltips (e.g., "Import image", "Auto Detect & Blur", "Export") and are placed in consistent locations across states (landing vs. edit view), matching the RAD's requirement for labeled controls and high-contrast actionable buttons.
- **Inline error messages.** All errors (unsupported file types, failed inference, no detections) are surfaced as inline banners or dialogs near the relevant UI element, with actionable wording ("Unsupported file format – please select a PNG or JPEG image"). The user is never stranded in an ambiguous state; each error path leads back to a stable screen (Sec. 8.1).
- **Tool states.** The current manual tool (none/brush/rectangle, add/erase) is always visible via a highlighted toolbar button. This reduces mode confusion and makes it easy for users to understand what their next click will do.

Accessibility considerations.

- **Keyboard support.** Core commands (open, run detection, export, undo/redo, tool switching) are reachable via standard keyboard shortcuts and tab-based focus navigation, so users who cannot rely exclusively on a mouse can still operate the main workflow.
- **Visual clarity.** The preview panes maintain a minimum size and support synchronized zooming/panning, ensuring that blurred regions are easy to inspect. Guided defaults (e.g., a sensible initial zoom level and blur strength) are chosen to reduce cognitive load for first-time users.
- **Out-of-scope aspects.** Full screen-reader support and advanced accessibility features (for example, ARIA roles, high-contrast OS themes integration) are not explicitly implemented for this course deliverable, but the UI layout and use of standard Qt widgets keep the path open for future accessibility improvements.

Combined, these usability and accessibility decisions support the non-functional goal that a new user can successfully blur and export a photo quickly, with clear feedback, minimal confusion, and low risk of making irreversible mistakes.

11 Deployment & Installation

11.1 Packaging

CleanShare is distributed as a standalone Windows application, packaged with all required runtime dependencies - ensuring that there are no external installation steps beyond the main executable. The delivered distribution folder contains the Qt GUI assets, ONNX model weights which were trained from the YOLOv11

fine-tuned Liquor Data dataset, and all OpenCV libraries required for preprocessing and blurring. Users can launch the program directly by opening the primary executable.The distribution also bundles the ONNX Runtime shared libraries (for example, `onnxruntime.dll`) required for local model inference, so no separate machine-learning runtime installation is needed.

To maintain a simple and offline installation process, the application does not use an installer. The application is provided as a compressed folder that the user extracts and runs directly on their computer. This approach avoids the need for admin permissions, keeps all system files self contained, and ensures compatibility across Windows systems without ever modifying the host environment. All processing stays local, and the application only creates new files when the user exports a processed image. The extracted folder follows a simple layout (for example: CleanShare.exe in the root, a models/ directory for ONNX files, a qt_assets/ directory for UI resources, a config/ or settings file, and the required DLLs in the same tree), so deploying to a new machine is equivalent to copying or unpacking this single directory without any additional installation steps.

11.2 Runtime Configuration

CleanShare maintains the system's lightweight and ease of use across various machines by supporting a minimal runtime configuration model. A configuration file in the application folder contains basic user settings such as blur strength, confidence threshold, and whether or not the previews are displayed at lower resolution for performance. Although this file can be altered if necessary, the default settings are adequate for everyday use and call for extra user input. The configuration file (for example, config.json or settings.ini in the application root) is the single source of truth for runtime options, and no environment variables or global OS settings are read or required by CleanShare. In practice, the default configuration is intended to work out of the box for typical workflows without requiring any additional user input or tuning from the user.

After the user-initiated export step, no user data or intermediate images are saved; all processing takes place locally. By maintaining the original image resolution when exporting, the application maintains a consistent output format. CleanShare is self-contained, simple to reset, and easy to use on any Windows supported system. If the configuration file becomes corrupted or is deleted, CleanShare can regenerate it with the default values on the next launch, allowing users to quickly restore a known-good baseline configuration.

12 Build, CI/CD, and Configuration Management

12.1 Build Tooling

CleanShare is built using C++17 with CMake as the main build system. CMake manages the compiler configuration, dependency resolution, and the integration with Qt [3], OpenCV [4], and ONNX Runtime [5]. This ensures that the builds are consistent across different development environments and avoids manual dependency configuration on each machine. The project includes separate debug and release build profiles - Debug contains additional loggina and insertions to support development, while Release applies full optimization for faster image processing and model inference. All third party libraries required for runtime execution are bundled in the build process to produce an all encompassing output folder.

Typical compiler flags include explicitly enabling the C++17 standard, turning on high optimization for Release builds (for example /O2 or -O2) and strict warning levels (such as /W4 or -Wall -Wextra), with warnings treated as errors in the CI environment to prevent unsafe or non-portable changes from entering the main branch.

12.2 CI Pipeline

A continuous integration pipeline, implemented using GitHub Actions, is used to maintain the code quality and ensure that any new changes to the code do not produce any errors. Upon any pull or push to the repository, the pipeline will execute on these following stages:

- **Build:** Configure and compile the project using CMake to verify that the codebase builds cleanly.

- **Static Analysis:** Run tools such as clang-tidy and cppcheck to detect potential issues early on the run.
- **Unit Tests:** Execute automated tests covering each step of the interactions, image preprocessing, ROI generation, and blur operations.
- **Model Evaluation:** Validate the ONNX model against a small test subset to verify detection accuracy and output format consistency
- **Packaging:** For the main branch, generate a Release build and bundle the executable, file model, and dependencies into a folder for distribution.

The pipeline prevents merging code that fails to build or violates quality constraints, making sure that the project stays stable, predictable, and deployable during development. For successful builds on the main branch, the pipeline also tags releases in version control (for example v1.2.0) and attaches the packaged artifacts to that tag, and can optionally run a code-signing step using an organization certificate so that downloaded binaries can be traced back to a specific, authenticated source revision.

12.3 Semantic Versioning & Model Versioning

CleanShare follows a semantic versioning structure for its releases, with each release being labeled as

```
MAJOR.MINOR.PATCH
```

A change in the MAJOR version indicates a significant architectural change, MINOR updates introduce new features, and PATCH additions apply bug fixes or minor improvements.

The ONNX model used for the detection is versioned separately from the application. Model files include both a semantic version number and a unique hash to guarantee reproducibility (e.g. liquor-model-v.1-hash.onnx). When the model is updated, the CI pipeline evaluates the new version against a small valid dataset to make sure accuracy and performance stay within the expected bounds before release.

A simple compatibility scheme is maintained (for example, all 1.x application versions are guaranteed to work with any 1.y model version, while a 2.x application may require a 2.y or newer model), and incompatible app–model pairings are detected at startup so that the user receives a clear error instead of undefined behaviour.

12.4 Configuration Management

All configuration files; model file, default settings, and build scripts, are stored in version control. Changes to configuration are reviewed along with code to maintain consistency across versions. The application does not require system level configuration and does not modify registry values or program files outside of its own directory. Doing this allows for a self contained system which is reproducible and easy to deploy.

Configuration and build changes are also summarized in a changelog or VERSION file so that each deployment can quickly determine which configuration, model and build scripts correspond to a particular tagged release.

13 Verification & Validation Strategy

13.1 Test Levels

CleanShare use a structured, multi-level testing approach to account for correctness, stability, and predictable behavior spanning all parts of the system. The testing is divided into the following levels:

- Unit Testing

  Unit tests validate the behaviour of individual components such as image loading, preprocessing, ROI mask construction, blur application, and basic I/O file operations. These tests confirm that core algorithms behave consistently, handle edge cases, and do not introduce regressions when the codebase changes.

- Integration Testing

  Integration tests confirm that the main subsystems function as intended. This covers the communication between the redaction pipeline, the ONNX detection model, the application core, and the Qt GUI. Tests verify that blur regions are applied to the expected coordinates, that detection results are accurately mapped back to the original image dimensions, and that the transition from detection → preview → export is dependable.

- System Testing

  System Testing System-level tests use full workflows, including loading an image, inferring, applying blur, previewing, and exporting the finished product, to assess CleanShare end-to-end. Consistency, usability, response times, and appropriately handling no-detection scenarios are the main objectives of these tests. All user-initiated blur regions must be preserved, and the output must always match the original resolution.

- Regression Testing

  Regression tests compare outputs to earlier iterations of the detection model or new features. This guarantees that updates won't alter current behaviour, decrease accuracy, or create unanticipated visual artifacts. A consistent collection of sample photos from the repository is used for regression testing.

Dedicated performance tests run representative 1080p and higher-resolution images to measure end-to-end processing time and memory usage, ensuring that newly introduced code paths do not violate latency or resource targets. Lightweight UX smoke tests are also executed on fresh builds to quickly exercise the main user flows (open, detect, adjust, export) and confirm that the interface remains responsive, discoverable, and free of obvious usability regressions.

13.2 Dataset-Based Evaluation

Because CleanShare relies on an ONNX model trained on the Liquor Data dataset [7], a portion of that data set is used for validation and performance verification. The subset consists of a mix of various beers, wine, and spirit containers, along with some non-alcoholic distractor objects.

Evaluation metrics:

Metric	**Description**	**Target**
Recall	Measures how many alcohol containers are detected correctly.	$\geq 80\%$
False Positive Rate	Ensures the system does not incorrectly blur non-alcoholic objects.	$\leq 15\%$
mAP@0.5	Validates overall detection quality across the validation set.	Stable across model updates
Processing Time	Measures time to detect and blur a 1080p image on a typical Windows machine.	≤ 10 s

Table 1: CleanShare detection and performance evaluation metrics.

These evaluations help ensure that updates to the detection model maintain consistent accuracy and speed across releases. The CI pipeline automatically runs a small subset of these checks when a new version is proposed.

Precision is also monitored alongside recall so that the model maintains a good balance between correctly detecting alcohol containers and avoiding unnecessary blurring of non-alcoholic objects. Evaluation results and metric trends are exported as machine-readable reports (for example JSON or CSV artifacts produced by the CI pipeline), allowing regressions to be detected, audited, and compared across multiple model and application versions.

13.3 Acceptance Criteria

CleanShare is ready for release when the following conditions are met:

Accuracy Requirements: The detection model constantly meets or exceeds the required performance metrics on the validation subset.

Runtime Requirements: A standard 1080p image can be processed within 10 seconds on a typical Windows machine without GPU acceleration.

Functional Stability: The complete workflow (import, detect, blur, preview, and export) must operate without crashes, visual corruption, or loss of user-selected regions.

Local Processing Guarantee: No user data is transmitted, logged externally, or written to disk except for explicit exports.

Output Fidelity: Exported images must have the same quality as the original pre-processed image, and maintain all blur regions as displayed in the preview.

Error Handling: Invalid file types, missing dependencies, and failed detection runs must be handled gracefully, with clear user feedback.

Meeting these criteria ensures that CleanShare is stable, accurate, and aligned with its intended purpose of providing a reliable offline image blurring tool. A candidate build fails acceptance if any quantitative thresholds are not satisfied (for example, recall dropping below 80%, false positive rate exceeding 15%, or median 1080p processing time exceeding 10 seconds) or if qualitative UX goals are violated by frequent crashes, corrupted previews, or unclear error messages, giving a clear pass/fail basis for promoting a build to release.

14 Design Rationale and Architectural Decisions (ADRs)

CleanShare is a reliable, privacy tool that automatically detects and blurs alcoholic beverages in images. The application must be able to run offline, be easy to use for non technical users, and still achieve strong detection accuracy, every major part of the design was driven by these constraints.

In order to be able to run the tool offline we decided to develop a local desktop application, All image processing and machine learning is done locally, completely avoid the use of external services like APIs or remote databases. This ensures the tool functions fully on its own and meets the requirement for offline operation.

To keep the system easy to use for non technical users, we made a straight forward workflow where the user imports an image, runs the detection model, makes any optional adjustments, and exports the final blurred version. The Qt interface provides clear buttons and a simple layout, which supports the ease of use requirement.

For the detection system, we decided to use a CNN-based model, because alcoholic beverages vary in shape, size, and lighting. A CNN model is able to generalize across these variations and produce better and more accurate detections than rule-based image processing. The model is ran through ONNX to keep inference efficient and work well with our desktop application.

Finally, Gaussian blur is used for redaction, fulfilling the requirement to clearly obscure detected alcohol while keeping the rest of the image intact. Key architectural choices were evaluated against alternatives: for example, Qt Widgets was selected over QML or a web or browser-based UI to reduce runtime dependencies and simplify offline deployment on standard Windows desktops, and ONNX Runtime was chosen instead of a heavier framework backend to keep inference fast and portable across machines. Gaussian blur was preferred to pixelation or solid blocking because it strongly obscures labels while preserving overall scene context and remains inexpensive to compute on CPU-only systems.

15 Risks and Mitigations

The development of CleanShare includes a few technical risks, that we must be aware of.

1. Low detection accuracy

   Because alcoholic beverages appear in many different shapes, sizes, or lighting conditions, the model may miss some bottles or cans. To avoid this we use a CNN-based model with proper post processing and allow users to manually adjust blur regions to fix missed detections.

2. False positives

   The machine learning model may occasionally detect objects that resemble alcohol bottles or cans, leading to unnecessary blurring. To fix this we will work on our model accuracy and also provide a way for users to remove or adjust blur areas.

3. Unsupported or invalid image formats

   Users may upload images that the system does not support. Currently, we only plan to allow standard formats like JPEG and PNG, so uploading other file types may lead to errors. To work around this, our application will validate the file type and size during import and display a clear message if the image cannot be processed. This prevents the system from failing and guides the user toward supported formats.

Additional project risks include dataset bias in the Liquor Data training set (for example, over-representation of certain container types or lighting conditions), schedule slippage, and third-party library incompatibilities between Qt, OpenCV, ONNX Runtime, and Windows updates. These are mitigated by augmenting and periodically re-evaluating the dataset, prioritizing a minimal end-to-end workflow in early milestones, pinning library versions in CMake, and validating the build on a small matrix of target Windows configurations in CI.

16 Maintenance & Support

The system is designed to be easy to maintain, update, and support throughout its lifecycle. To ensure this, the application is setup into separate modules, each with a clear responsibility. This makes it easier for us to change one part of the system without affecting the others to much. We also plan to add test cases for core functionalities that way we can easily verify if a change or an update breaks edge cases, or even the system. These test cases will cover important areas such as image importing, model loading, detection output, blur application, and exporting. By keeping the project modular and supported by testing, we will be able to identify issues quickly and make targeted fixes without needing to rework the entire application. Ongoing maintenance follows a lightweight process: issues are reported and triaged in the project repository using labelled templates for bug reports and feature requests, and fixes are batched into minor releases on a roughly monthly cadence, with critical regressions shipped as ad hoc patch releases when needed. Documentation ownership is assigned by subsystem (GUI, application core, detection engine, redaction pipeline), and the responsible maintainer ensures that user guides, developer notes, and model/versioning documentation stay in sync with each tagged release.

17 Requirements-to-Design Traceability

Req ID	Requirement (short)	Satisfied by (Design element/section)

FR-01	Detect and blur alcoholic beverages automatically	• Detection Engine & Redaction Pipeline • Sec. 2.2 (System Decomposition) • Sec. 3.1 (Logical View) • Sec. 6.2 (Component Responsibilities & APIs) • Sec. 7.1 (Sequence Diagrams)
FR-02	Side-by-side before/after preview	• Qt GUI / MainWindow (Presentation layer) • Sec. 2.2 (Presentation Layer) • Sec. 3.1.1 (Modules and Responsibilities) • Sec. 4.1 (User Interface) • Sec. 6.2 (MainWindow / Qt GUI) • Sec. 7.1 (Main flow sequence)
FR-03	Manual edit tools (brush/rectangle, undo/redo)	• Qt GUI + SessionController + RedactionPipeline • Sec. 2.2 (Presentation & Application Core) • Sec. 3.1.1 (Qt GUI, Application Core, Redaction Pipeline) • Sec. 3.5 (UC-2: Manually edit ROIs) • Sec. 6.2 (SessionController, RedactionPipeline) • Sec. 6.3 (Command pattern for undo/redo) • Sec. 7.2 (Tool Mode state machine)
FR-04	Upload images in JPEG/PNG formats	• I/O & Privacy module / IOService • Sec. 2.2 (I/O & Utils) • Sec. 4.2 (File Interfaces) • Sec. 5.1 (Core Data Structures – image representation) • Sec. 6.2 (IOService) • Sec. 7.1 (Import step in main flow)

FR-05	Verify file type and size before processing	• IOService validation + error handling • Sec. 2.2 (I/O & Privacy responsibilities) • Sec. 4.2 (File Interfaces – validation rules) • Sec. 8.1 (Error Classes and Recovery) • Sec. 9.2 (Threat Model – tampering checks)
FR-06	Apply NMS and confidence thresholding to detections	• DetectionEngine • Sec. 3.1.1 (Detection Engine responsibilities) • Sec. 6.2 (DetectionEngine API) • Sec. 13.2 (Dataset-Based Evaluation – detection metrics)
FR-07	Identify bounding boxes/ROIs and apply Gaussian blur	• RedactionPipeline & BlurStrategy • Sec. 3.1.1 (Redaction Pipeline responsibilities) • Sec. 5.1 (Boxes/masks data structures) • Sec. 6.1 (Component/Class Diagram) • Sec. 6.2 (RedactionPipeline & BlurStrategy APIs) • Sec. 6.3 (Strategy pattern for blur effects) • Sec. 7.1 (Redaction steps in sequence)
FR-08	Export blurred image at original resolution	• RedactionPipeline (resolution-preserving) & IOService • Sec. 2.3 (Primary User Workflow – Export) • Sec. 3.5 (UC-1: Blur a photo and export) • Sec. 5.1 (Image/mask alignment) • Sec. 6.2 (RedactionPipeline invariants, IOService::saveImage) • Sec. 11 (Deployment & Installation – export behaviour)

FR-09	Handle failed uploads and model inference errors gracefully	• Error handling and recovery • Sec. 2.3 (Alternate flows: invalid file, slow inference) • Sec. 3.2 (Process View – cancellation, rollback) • Sec. 4.1 (UI error banners) • Sec. 8.1 (Error Classes and Recovery) • Sec. 8.2 (Logging – local diagnostics)
FR-10	Support "no detections" flow with manual-only blur mode	• Qt GUI + SessionController alternate path • Sec. 2.3 (Primary Workflow – "no alcohol detected" message) • Sec. 3.2 (Alternate flows) • Sec. 3.5 (UC-1 & UC-2 interactions) • Sec. 7.1 (Sequence diagram – no-detections branch)
NFR-PERF	$\leq$ 10 s per 1080p image; GUI actions $\leq$ 300 ms	• Worker-thread process model & performance targets • Sec. 3.2 (Process View – background worker, responsiveness) • Sec. 10.1 (Targets and Budgets) • Sec. 13.2 (Processing Time metric) • Sec. 13.3 (Acceptance Criteria – runtime requirements)
NFR-ACC	$\geq$ 80% recall, $\leq$ 15% false positives, stable mAP@0.5	• Evaluation utilities & dataset-based testing • Sec. 2.2 (Evaluation Utilities module) • Sec. 6.2 (EvaluationRunner) • Sec. 13.2 (Dataset-Based Evaluation) • Sec. 13.3 (Acceptance Criteria – accuracy requirements)

NFR-STAB	Stable import → detect → blur → preview → export workflow (no crashes/corruption)	• Lifecycle state machines, error recovery, and tests • Sec. 3.2 (Process View) • Sec. 7 (Dynamic Behavior – ImageSession lifecycle) • Sec. 8.1 (Error Classes and Recovery) • Sec. 13.1 (Test Levels – integration/system tests) • Sec. 13.3 (Acceptance Criteria – functional stability)
NFR-FID	Exported image preserves original resolution/quality and all blur regions as previewed	• Data & redaction design • Sec. 2.3 (Primary User Workflow – export guarantees) • Sec. 5.1 (Core Data Structures – image + masks) • Sec. 6.2 (RedactionPipeline invariants, IOService) • Sec. 10.1 (Quality attributes) • Sec. 13.3 (Output Fidelity acceptance criteria)
NFR-UX	First-time user can blur & export in ≤ 30 s via intuitive workflow	• GUI layout, primary workflow, scenarios • Sec. 2.3 (Primary User Workflow) • Sec. 3.5 (Scenarios / Use-Case View) • Sec. 4.1 (User Interface – landing and edit/view pages) • Sec. 10.3 (Usability & Accessibility)

NFR-UX-FB	Clear, discoverable controls; visual feedback; undo/redo without corruption	• Qt GUI behaviour, tool state, command pattern • Sec. 3.1.1 (Qt GUI responsibilities) • Sec. 3.5 (UC-2: Manual edit) • Sec. 6.2 (MainWindow, SessionController APIs) • Sec. 6.3 (Command pattern for undo/redo) • Sec. 7.2 (Tool Mode state machine) • Sec. 10.3 (Usability & Accessibility)
NFR-PRIV	Local-only processing; no images or metadata leave the device	• I/O & Privacy module, security & privacy design • Sec. 2.1 (Context – standalone, no network) • Sec. 2.2 (I/O & Privacy subsystem) • Sec. 5.3 (Persistence Strategy – no long-term storage) • Sec. 8.2 (Logging – no telemetry) • Sec. 9.1 (Privacy Principles) • Sec. 9.2 (Threat Model) • Sec. 11 (Deployment & Installation – offline packaging)
NFR-LOG	Minimal, local-only logging; no telemetry or analytics	• Logging subsystem • Sec. 8.2 (Logging – rotating local logs, no sensitive data) • Sec. 9.1 (Privacy Principles – no external transmission)
NFR-PLAT	Target platform: Windows 10+ desktop, $\geq$ 8 GB RAM, CPU-only baseline (GPU optional)	• Scope and deployment design • Sec. 1.2 (Scope) • Sec. 2.1 (Context) • Sec. 3.4 (Physical/Deployment View) • Sec. 10.1 (Capacity assumptions) • Sec. 11 (Deployment & Installation)

NFR-DOC	User & developer documentation; versioned model/config for reproducibility	• Reference material & configuration management • Sec. 1.3 (Definitions, Acronyms, Abbreviations) • Appendix B (References) • Sec. 12.1 (Build Tooling) • Sec. 12.3 (Semantic Versioning & Model Versioning) • Sec. 12.4 (Configuration Management) • Sec. 13.2 (Machine-readable evaluation reports)
NFR-MAINT	Modular, testable architecture with CI; easy to evolve and fix	• Architectural decomposition, components, CI/CD, maintenance • Sec. 3 (Architectural Design – 4+1 views) • Sec. 6 (Component Design) • Sec. 12.2 (CI Pipeline) • Sec. 13.1 (Test Levels) • Sec. 16 (Maintenance & Support)
NFR-ETH	Evaluate model bias; communicate limitations/false positives; safe manual fallback	• Ethical / responsible ML use • Sec. 9.1 (Privacy Principles – institutional expectations) • Sec. 13.2 (Dataset-Based Evaluation – monitoring FP/FN) • Sec. 2.3 & Sec. 3.2 (Alternate flows: false positives, manual correction) • Sec. 8.1 (Graceful degradation to manual mode)

Table 2: Requirements-to-design traceability matrix mapping CleanShare's functional (FR) and non-functional (NFR) requirements to SDD design elements and sections.

The design of CleanShare shows the requirements outlined for the project, each major requirement maps to a specific part of the system. To start off the requirement for offline functionality is met by making the tool a local desktop application where all processing happens in house. The requirement of accepting standard image formats is demonstrated by the image processor which accepts both PNG and JPEG file types. The requirement for alcoholic beverage cans / bottles is fulfilled through the CNN machine learning model which we will integrate into the Detection Engine, thus allowing the system to identify bottles and cans in any image. Once the ML model detects a specific area, our blur pipeline applies Gaussian blur to the detected

regions, meeting the requirement to blur out alcoholic content. Finally the requirement for a user friendly, non technical interface is supported through a simple user interface, where users can import an image, run the detection, make optional manual adjustments, and export the final result. This straightforward process ensures that the system is easy to navigate and use, even for individuals without technical experience.

A Glossary

This glossary summarizes key terms used throughout the document. For extended definitions, see Section 1.3.

CleanShare The CISC 320 group project desktop application that detects alcoholic beverages in still images and applies blur redaction so they can be shared safely while keeping all processing local.

Application Core
The middle layer of the architecture that orchestrates image loading, model inference, redaction, and exporting, insulated from the GUI and low-level libraries.

GUI (Graphical User Interface)
The front-end of CleanShare implemented with Qt, providing windows, buttons, menus, image previews, and interaction with the user.

Qt A cross-platform application framework used for the CleanShare desktop GUI, handling windows, widgets, events, and image display.

OpenCV An open-source computer vision library used for reading/writing images, resizing, color conversions, drawing blur masks, and handling image matrices.

ONNX (Open Neural Network Exchange) A standard format for representing trained neural network models, allowing CleanShare to run the detector independently of the original training framework.

ONNX Runtime
A high-performance inference engine for ONNX models used by CleanShare to run the YOLO detector efficiently on the user's machine.

CNN (Convolutional Neural Network)
A type of deep neural network specialized for processing images, used by CleanShare's detector to recognize alcoholic beverages.

YOLO (*You Only Look Once*) A family of real-time object detection architectures used as the basis for CleanShare's bottle/can detector (e.g., Ultralytics YOLOv8/YOLOv11 variants).

Liquor Data Dataset
An annotated dataset of alcoholic beverage images (bottles, cans, cartons, etc.) used to fine-tune the YOLO detector for CleanShare's specific task.

Bounding Box
An axis-aligned rectangle predicted by the detector that encloses a candidate alcoholic item in the image.

ROI (Region of Interest)
Any region in the image that is considered relevant for processing; in CleanShare this typically refers to detected alcohol boxes or user-drawn corrections.

ROI Mask A binary (or multi-channel) image that marks which pixels are inside redaction regions; used by the Redaction Pipeline to apply blur only where needed.

NMS (Non-Maximum Suppression)
A post-processing step that filters overlapping detection boxes by keeping the highest-confidence box and discarding redundant ones, reducing duplicate detections.

Detection Engine
The core service responsible for loading the ONNX model, running inference on images, and producing raw detection boxes, classes, and confidence scores.

Redaction Pipeline
The component that converts detections (plus manual edits) into final blur masks and applies the selected blur strategy to the image.

ImageSession
An in-memory data structure representing the state for a single loaded image, including the original pixels, preview image, detection results, user overrides, and export status.

Session State The overall state managed by the Application Core, including the current `ImageSession`, configuration, undo stack, and progress flags used for the GUI.

Blur Strategy
A configurable policy that defines how redaction is applied (e.g., Gaussian blur with a given kernel size, pixelation strength, or solid fill) to all active masks.

Configuration
Structured parameters that influence detection thresholds, NMS settings, blur strength, performance limits (e.g., max resolution), and UI preferences.

Evaluation Utilities
Internal tooling for measuring model performance (e.g., precision, recall, mAP, latency) on validation data to ensure CleanShare meets its accuracy and runtime targets.

Precision The proportion of predicted alcohol detections that are actually correct (true positives divided by all predicted positives).

Recall The proportion of real alcohol instances in the images that are successfully detected (true positives divided by all ground-truth positives).

False Positive Rate
The fraction of predictions that incorrectly flag non-alcohol content as alcohol, relative to all predicted detections.

mAP@0.5 Mean Average Precision computed at an Intersection-over-Union (IoU) threshold of 0.5, summarizing detection quality across classes.

1080p A common image resolution of 1920×1080 pixels; CleanShare's performance targets (e.g., $\leq 10\,\text{s}$ per image) are specified relative to this resolution.

Local-only Processing
An architectural guarantee that all inference, redaction, and previews run entirely on the user's machine, with no network requests or remote logging.

B References

References

[1] CleanShare Team, *CISC 320 CleanShare Requirements Analysis Document (RAD)*. Queen's University, School of Computing, 2025.

[2] CISC 320 Instructional Staff, *CISC 320 Project Outline: CleanShare Alcohol-Redaction Tool*. Queen's University, School of Computing, 2025.

[3] The Qt Company, *Qt 6 Documentation*. Available at: `https://doc.qt.io/qt-6/index.html`. Accessed Nov. 2025.

[4] OpenCV Team, *OpenCV: Open Source Computer Vision Library Documentation*. Available at: `https://docs.opencv.org/`. Accessed Nov. 2025.

[5] Microsoft and ONNX Community, *ONNX Runtime Documentation*. Available at: `https://onnxruntime.ai/docs/`. Accessed Nov. 2025.

[6] Ultralytics, *Ultralytics YOLOv8/YOLOv11 Documentation*. Available at: `https://docs.ultralytics.com/`. Accessed Nov. 2025.

[7] Lamar University, *Liquor-data: Alcoholic Beverage Object Detection Dataset*. Roboflow Universe, Available at: `https://universe.roboflow.com`. Accessed Nov. 2025.