# A Secure OCR-Based and Voice Detection Framework for Air-Gapped Media Systems

**By**

Jareer Ahmad Khan
Enrollment No. 01-133222-029

Nouman Ahmed
Enrollment No. 01-133222-059

**Supervised By**

Dr. Adil Ali Raja


Session 2022–2026

A Report is submitted to the Department of Electrical Engineering,
Bahria University, Islamabad.
In partial fulfillment of requirement for the degree of BS(EE).

# Certificate

We accept the work contained in this report as a confirmation to the required standard for the partial fulfillment of the degree of BS(EE).

_____.

Head of Department

_____

Supervisor

_____.

Internal Examiner

_____

External Examiner

# Dedication

We dedicate this project to our parents, who have given us their continued support, prayers, and sacrifices that have always been our greatest source of strength and motivation during this journey. In the same spirit, we are also grateful to our faculties and mentors who have helped us with their invaluable guidance and encouragement, which has shaped us academically. Also, this work is dedicated to everyone who believes in the transformation power of technology towards improving lives and innovating our everyday systems.

# Acknowledgments

We would like to express our profound gratitude to our project supervisor, Dr. Adil Ali Raja, for their unwavering support, invaluable guidance, and exceptional mentorship throughout this project. Their insight, encouragement, and constructive feedback have been instrumental in shaping our ideas and refining the outcomes of this research. We would also like to acknowledge the valuable contributions of Dr. Hassan Danish, the Head of the Department of Electrical Engineering, for his support and encouragement. His vision and leadership have been critical in providing the necessary resources and infrastructure for this research. Furthermore, We extend our sincere appreciation to the distinguished faculty members of the Electrical Engineering Department, such as Dr Junaid Imtiaz, Sir Mudasir Wahab and Dr. Asad Waqar. Their expert guidance, deep knowledge, and insightful feedback have significantly contributed to the development of my research abilities and understanding. Thank you all for your support and encouragement.

# Abstract

In an era where data privacy is a global concern, this project presents a “Visual sentry” a Secure OCR based and Voice Detection Framework for Air-Gapped System. The system automatically monitors broadcasts in real time for user defined trigger words without reliance on cloud services. Unlike other standard surveillance projects our system not only extract text from the screen but also extracts information form audio as well using speech to text framework and efficiently shows alert when trigger word detected either from the Audio or Screen. The system uses Tesseract OCR Engine to extract ticker, headlines from the screen and VOSK for speech detection. The model uses adaptive image pre-processing adaptive thresholding, region-of-interest segmentation,fuzzy logic to understand the typos and Late Fusion to combine visuals and audio. Additionally the system adds an automated forensic loop that records instantly video buffers, screenshot and time stamp for information verification instead of recording 24hr and fill up the terabyte space our project records only when the required information is detected. LIFI system is also integrated with this system where we can transmit our trigger word using laser and we receive the word using Lm393 comparator and a photoiode, after that we print that word on our LCD. The system entirely focus on combining two different modules namely Tesseract OCR and VOSK to work simultaneously using late fusion and an optimized pipeline required to get efficent results, our system GUI runs on the main thread while our heavy AI processing runs on background thread. This prevents the system from freezing and making it more robust. According to experimental results on live news broadcasts, the dual-modal approach maintains low latency on hardware.

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Project Background and Overview

Security teams across the world rely on surveillance cameras, yet very few of these systems actually understand what they are watching. Video streams stack up. Hard drives fill. In the hope of finding something important, Operators spend multiple hours staring at the screen. Each passing year, the gap between important information and raw footage has grown wider as camera resolutions climb and on-screen content becomes denser. News channels scroll tickers packed with breaking updates. Spoken announcements slip past in seconds. A conventional closed-circuit television setup sees every pixel but comprehends almost none of it.

Visual Sentry was built to close that gap. It is a desktop-based, multithreaded surveillance application that watches screen content and listens to audio streams at the same time, raising an alert the moment a user-defined trigger word appears on either channel. Detection happens in real time. Evidence gets preserved the instant something significant is spotted. A physical alert then fires on an external microcontroller to notify a human operator, and none of this requires an internet connection at any stage.

The project is designed specifically for air-gapped environments. These are machines that have been deliberately isolated from the internet and all untrusted networks. Such environments are common in defence installations, nuclear facilities, financial clearing houses, and intelligence operations. The entire point of an air-gap is to prevent sensitive data from leaking out or hostile code from entering. That same isolation, however, renders most modern artificial intelligence tools useless, because almost every commercial cloud service depends on an internet round-trip to function. Visual Sentry brings the intelligence directly to the machine where the data already lives, rather than shipping the data out to a remote server.

Two distinct recognition engines run inside the application in parallel. The first is Tesseract, a widely used open-source optical character recognition engine that reads text directly from the screen. Breaking news tickers, captions, subtitles, on-screen warnings, and stock market headlines are all extracted, cleaned, and checked against the user's watchlist in real time. The second engine is VOSK, an offline automatic speech recognition tool built on top of the Kaldi speech recognition toolkit. It transcribes spoken audio directly on the central processing unit without routing any data through a remote server. A positive detection on either channel is sufficient to trigger the full system response.

An architectural decision can shape the entire design: the threading model. Users usually abandon frozen applications within a second. In an operational setting, a frozen screen is simply unacceptable. Therefore, Visual Sentry restricts all artificial intelligence inference to background daemon threads. The main thread handles only rendering and user interaction. Results from the vision and acoustic threads flow into the main thread via thread-safe queues, keeping the display fluid and interactive even as the system processes high-resolution video frames and continuous audio buffers simultaneously.

Live broadcast results confirmed that the design works. Accuracy measured against a labeled test reached 95%, whereas recall hit a full 100% – which means not a single trigger event got missed during evaluation. Precision was 91.67%, which favors finding real alerts at the cost of occasional false positives. The resulting score of 95.65% shows a workable balance for monitoring, where missing a real event is more damaging than incorrectly flagging. All inference runs on a standard AMD Ryzen 5 7430U laptop with 8 GB RAM running Windows 11. No graphics card is required anywhere in the pipeline.

The remainder of this report covers each layer of the system in turn. Chapter 2 reviews the body of prior research that shaped the design decisions, covering optical character recognition, automatic speech recognition, multimodal fusion, air-gap security, and edge computing. Chapter 3 lays out the functional and non-functional requirements alongside a comparison with existing solutions. Chapter 4 describes the system architecture at high and low levels. Chapter 5 covers implementation, code structure, threading discipline, and deployment procedures. Chapter 6 presents the test results using full set of evaluation. The report ends with conclusions and recommendations for future work.

## 1.2 Problem Description

Visual Sentry emerged from five interlinked problems that existing surveillance solutions either ignore completely or handle poorly. Each one on its own might be manageable. Combined, they expose a significant gap in the current landscape of intelligent monitoring tools.

### 1.2.1 Semantic Blindness in Conventional CCTV

The first problem is fundamental. A camera can record everything happening on a TV screen, including the live news ticker scrolling at the bottom. But the system does not actually read or understand the words being shown. If any inappropriate or sensitive words appear on the screen, or if someone says something important during a live broadcast, the video is simply stored without being analyzed. The system depends on human operators to notice such moments themselves. In large facilities with many video feeds running simultaneously, this becomes extremely difficult because people can become tired or distracted, or miss important details.

Existing motion detection systems can identify physical movement or intrusion, but they cannot understand spoken language or text appearing in videos.

### 1.2.2 Cloud Dependency and Privacy Risk

The second problem concerns commercial products that do try to interpret content intelligently. Services such as Google Video Intelligence, Amazon Rekognition, and Azure Cognitive Services can transcribe speech and extract text from video with high accuracy. Every one of them, however, requires raw data to be uploaded to an external server before any analysis can occur. In a defence or intelligence environment that is not merely inconvenient it is a direct policy violation. Air-gapped systems exist precisely so that sensitive data never touches an untrusted network. A surveillance tool that depends on the cloud is therefore structurally incompatible with the very environments where intelligent monitoring would be most valuable. Even in less sensitive settings, transmitting raw audio and video to third-party servers creates legal and privacy liabilities that many organisations are no longer prepared to accept under current data protection regulations.

### 1.2.3 Storage Inefficiency of Continuous Recording

The third problem is storage waste. A single 1080p camera running at thirty frames per second produces three to five gigabytes of compressed video every hour. Multiply that by twenty cameras over a thirty-day retention period and the total reaches tens of terabytes. The reality is that very little of this footage has been reviewed. Forensic investigation typically focuses on a confirmed incident, which means the majority of stored material is dead. Organizations pay for storage arrays, electricity, cool-

ing infrastructure, and backup cycles just to save content that nobody will ever examine. An event-driven approach that records only when something interesting happens would eliminate 99% of that overhead, but standard systems are not built this way because they cannot recognize what "interesting" means.

### 1.2.4 Single-Modality Limitations

The fourth problem is about modality: existing intelligent systems address only one modality at a time. OCR tools read text on screen but entirely ignore spoken audio. Speech-to-text engines write out audio but miss every written update. A Real broadcast surveillance does not respect these boundaries. A news channel carries information across both channels simultaneously, and a target keyword might appear in the visual ticker, in the spoken commentary, or in both at the same moment. An OCR-only system will miss every spoken alert. An ASR-only system will miss every textual update. Reliable coverage requires that both channels be monitored simultaneously, with detection on either path sufficient to trigger the full response.

### 1.2.5 User Interface Freezing During AI Inference

The fifth problem is a practical engineering failure that prevents many prototype surveillance applications from reaching a usable state. Running neural network inference on the same thread causes the application to freeze during every inference cycle. The user sees an unresponsive window. Buttons stop working. The display stops updating. In an operational deployment this is completely unacceptable, because an operator must be able to see live feedback, adjust trigger words, switch video sources, and stop the system at any moment without waiting for the current inference

pass to finish. Solving this problem requires a disciplined multi-threaded design one where the interface and the intelligence are permanently separated and communicate only through thread-safe channels.

These five problems are interconnected. Cloud dependency forces a choice between intelligent analysis and data isolation. Continuous recording fills storage with semantically blind footage. Single-modality designs leave entire channels unmonitored. User interface freezing makes the system unusable in practice. Visual Sentry addresses all five in a single integrated architecture.

## 1.3 Project Objectives

The project was defined around seven concrete objectives, each of which addresses one or more of the problems identified above. These objectives were established at the start of the project and tracked throughout implementation and evaluation.

1. **Dual-Modal Real-Time Detection.** Develop a detection system that reads on-screen text using OCR and transcribes spoken audio using ASR simultaneously, with the entire pipeline running offline on a standard CPU-only laptop.

2. **Fuzzy Matching for OCR Error Tolerance.** Implement a text normalisation and matching stage that tolerates common OCR misrecognition patterns. A Levenshtein-based similarity ratio with a configurable threshold of 0.80 allows the system to match degraded readings such as “AL3RT” against the target word “ALERT” and correctly fire an alert.

3. **Late Decision-Level Fusion.** Combine the two modalities using

late decision-level fusion, where both pipelines operate fully independently and a positive detection on either channel constitutes a system-level alert. This approach requires no synchronisation between the vision and audio threads and remains robust even when one channel is temporarily degraded.

4. **Event-Driven Forensic Recording.** Replace twenty-four-hour continuous capture with event-driven recording that activates only when a trigger is detected. Each detection event automatically saves a five-second H.264-encoded video buffer, a JPEG screenshot of the triggering frame, and a timestamped information file. Idle time produces zero disk writes.

5. **Fully Responsive GUI via Multi-Threaded Architecture.** Keep the graphical user interface fully responsive at all times by isolating all artificial intelligence inference on background daemon threads and passing results to the main thread through thread-safe Python queue objects.

6. **LiFi-Based Optical Alert Channel.** Implement a light-fidelity alert subsystem in which the host application transmits the detected trigger word over a serial COM port to a dedicated ESP32 transmitter unit. The transmitter modulates a laser diode through a 2N2222 transistor switching circuit, encoding the trigger string bit-by-bit with a defined start-bit handshake and per character re-synchronisation. A second ESP32 on the receiver side reads the incoming optical signal through an LM393 comparator connected to a BPW34 photodiode or LDR, reconstructs the character stream, and displays the trigger word on a 16×2 LCD completing a fully wireless free, air-gap-safe optical alert chain.

## 1.4 Project Scope

Defining scope clearly establishes Visual Sentry's dos and don'ts and the specific environment it is designed for. All three boundaries are important to understand before reading the technical chapters.

### 1.4.1 In Scope

A webcam connected to an OpenCV device index, an IP camera stream, and a pre-recorded MP4 or AVI file are all video sources that the system monitors. Audio input is captured from the default system microphone at a sample rate of 16 000 Hz in 16-bit signed PCM mono format using PyAudio, with a chunk size of 4 000 bytes per processing cycle. The user enters a single trigger word or short phrase through the graphical interface and the system monitors both channels for it simultaneously.

A positive detection triggers four simultaneous outputs: a pop-up alert on the graphical interface, a JPEG screenshot of the triggering frame saved to the designated capture folder, a five-second video clip encoded using H.264 (with fallback to mp4v if the primary codec is unavailable). A cooldown window of three seconds prevents the alert cascade from firing repeatedly on a sustained trigger within a short span of time. Frame processing is rate-limited by a skip interval of five meaning the vision thread processes every fifth captured frame, reducing a 30 FPS input stream to an effective OCR processing load of 6 FPS and a minimum OCR interval of one second prevents back-to-back inference calls.

A LiFi-based optical communication channel is also within scope. The transmitter side consists of an ESP32, a laser diode, and a 2N2222 transistor switching circuit. The receiver side consists of a second ESP32, an LM393 comparator module, a BPW34 photodiode or LDR, and a 16×2

LCD. Data is transmitted bit-serially with a start-bit synchronisation sequence and per-character re-sync, keeping the entire alert chain free of any radio frequency interface.

### 1.4.2 Out of Scope

The system does not perform face recognition, object detection, activity classification, or any form of visual scene understanding beyond text extraction. It does not process handwritten text, heavily stylised fonts, or vertical text layouts. Support is limited to the English language in the current release, since both Tesseract and the VOSK small English model were used without multi-language configuration. The recorded evidence files are not cryptographically signed; they are traceable through filename timestamps only. Cloud backup, remote push notification, and mobile application integration are excluded by design, as each would violate the air-gap constraint that defines the project.

### 1.4.3 Target Deployment Environment

Visual Sentry is designed to run on a Windows 11 laptop equipped with at least 8 GB of RAM and a modern multi-core processor. An AMD Ryzen 5 7430U system was used for development and as a reference for evaluation. The code can be ported to other operating systems, such as Linux or macOS, with minor path adjustments.

### 1.4.4 Representative Use Case Scenarios

These three scenarios motivated the design throughout the development process. The command-center monitoring station is the first scenario to monitor live news broadcasts and trigger an immediate physical alert whenever a designated keyword appears in the ticker or spoken commentary.

The compliance review tool is the second scenario that scans archived broadcast recordings for restricted terms and saves timestamped evidence of each detection. The passive watchpost is the third scenario that runs unattended in a sensitive facility, notifying an operator via the hardware alarm without requiring anyone to continuously monitor a screen. All three scenarios share one requirement: detect specific content in real time, automatically preserve evidence, and alert a human without any data ever leaving the machine.

# Chapter 2

# Literature Review

## 2.1 Overview of the Literature Review

The Visual Sentry system sits at the intersection of several active research fields. Optical character recognition has evolved from rule-based feature matching to deep neural sequence models. Automatic speech recognition has moved from hidden Markov models through hybrid deep networks to end-to-end transformers. Multimodal fusion research has produced a well-established taxonomy of early, late, and hybrid strategies. Air-gapped security has grown into a subfield of its own, pushed largely by the work on covert channels. Edge AI has reframed the question of where artificial intelligence should live, shifting inference from cloud data centres to the devices that actually hold the data. Each of these strands informs one part of the Visual Sentry design, and together they define the space within which the project contributes.

This chapter surveys the relevant literature across all these directions. The review starts with the evolution of video surveillance and works outward to text recognition, speech recognition, multimodal integration, approximate string matching, air-gapped security, edge deployment, event-driven recording, and multi-threaded architectures for real-time AI. Each section identifies the state of the art, highlights the techniques most relevant to Visual Sentry, and notes where existing solutions fall short of the specific needs of offline broadcast monitoring. The chapter closes with a research-gap analysis that pulls all the threads together and a short summary of the key findings that shaped the system design presented in later chapters.

## 2.2 Evolution of Video Surveillance Systems

Video surveillance began as a purely analog discipline. Early closed-circuit television systems simply transmitted a camera feed to a bank of monitors, which were watched by security staff. Recording relied on VHS tapes, which had to be manually swapped and reviewed frame by frame during investigations. The transition to digital IP-based systems brought network-attached storage, motion-triggered recording, and remote access, but the underlying idea stayed the same: humans watched the footage and made decisions about what mattered.

The limitations of this manual approach became obvious once cameras multiplied. A mid-sized facility can now run fifty or more feeds simultaneously, and even a dedicated operator cannot maintain continuous attention across that many streams without losing critical events. This problem motivated the first wave of automated video analytics. Pixel-level motion detection, background subtraction, and object tracking appeared as lightweight analytical layers that could flag unusual activity without requiring constant human oversight [1]. These techniques worked reasonably well for physical intrusion detection but failed completely on tasks that required semantic understanding of the image content.

Recent years have brought a sharp turn toward deep learning in surveillance. Nayak et al. [2] provide a comprehensive 2023 survey of deep learning approaches to video anomaly detection, noting that convolutional and recurrent architectures have dominated the field since roughly 2018. Generative adversarial networks, autoencoders, and attention-based models have all been adapted for unusual-event detection in surveillance video. Their review cautions that most of these systems target specific anomaly classes falling, fighting, loitering and rarely address the kind of content-

level semantic analysis needed for broadcast monitoring, where the event of interest is a specific word rather than a specific action.

Broadcast surveillance is a distinct problem from physical surveillance. A news channel does not contain the kind of scene changes that motion detectors respond to, but it carries a dense and rapidly changing stream of text and speech information. Research on automated broadcast analysis has concentrated on three separate tasks. The first is text extraction from scrolling tickers and overlaid captions using optical character recognition, which was initially approached using traditional edge-based detection combined with general-purpose OCR engines. The second is speech transcription of the anchor audio, historically performed using hidden Markov models and more recently using end-to-end neural networks. The third is semantic search over the transcribed text, commonly implemented through exact string matching or simple keyword filtering.

Although intelligent surveillance has grown quickly, most commercial deployments still depend on the cloud. Products from the major technology companies Google Video Intelligence, Amazon Rekognition, Microsoft Azure Cognitive Services rely on uploading raw footage to remote servers where large models perform the analysis before returning results. This cloud-centric design is practical but clashes with the security policies of government, defense, and critical infrastructure facilities, where sensitive footage cannot cross the network boundary. Visual Sentry fills this gap with intelligent surveillance for environments that dominant cloud products cannot serve.

Broadcast surveillance also differs from traditional surveillance in the content it processes. A standard security camera produces frames dominated by static scenery with occasional motion events. A broadcast feed produces frames that change continuously, with graphics animating in and

out, cameras cutting between scenes, and text scrolling across tickers at different speeds on different channels. The implication for automated analysis is that frame-level processing cannot rely on background models or motion cues; instead, each frame must be treated as a standalone still that carries its own semantic information. This framing places broadcast surveillance closer to document analysis than to traditional video analytics, and the design of Visual Sentry reflects that observation throughout.

## 2.3 Optical Character Recognition: From Rules to Neural Networks

Optical character recognition has a long history that predates modern computer vision. The first commercial OCR systems in the 1960s relied on template matching against fixed character sets. The second generation, dominant through the 1980s and 1990s, used hand-crafted features combined with statistical classifiers. Accuracy on clean printed documents reached impressive levels, but performance degraded sharply on noisy or low-contrast inputs.

The Tesseract engine, originally developed at Hewlett-Packard Labs between 1984 and 1994 and later open-sourced by Google, became the benchmark for open-source OCR through the 2000s and 2010s. Smith [3] describes the original architecture in his 2007 overview paper, documenting the line-finding, feature extraction, and adaptive classification stages that gave Tesseract its distinctive approach. The engine later underwent a fundamental redesign. Starting with Tesseract 4.0, the classification stage was replaced with a Long Short-Term Memory recurrent neural network. This change followed the broader shift in sequence recognition toward LSTM models introduced by Hochreiter and Schmidhuber [4] and subsequently

applied to text recognition with the Connectionist Temporal Classification loss function developed by Graves et al. [5]. The combination of LSTM and CTC allowed the network to learn text transcription end-to-end, without requiring explicit character-level segmentation.

Modern OCR research has split into two broad categories. Document OCR targets scanned pages, forms, and books, where text is usually printed in standard fonts against a plain background. Scene text OCR targets text that appears naturally in photographs and videos, which often involves unusual fonts, curved baselines, and complex backgrounds. Wang et al. [6] published a detailed survey of deep learning approaches to text detection and recognition in 2023, organising the field into two core tasks: text detection, which localises text regions, and text recognition, which decodes the content within those regions. Convolutional architectures dominate the detection side, while sequence models CRNN variants, attention-based encoders, and transformer networks lead the recognition side.

The PP-OCR family released by the PaddlePaddle team exemplifies the engineering focus on lightweight deployment. Li et al. [7] introduced PP-OCRv3 in 2022, building on the earlier PP-OCRv1 and PP-OCRv2 releases with nine separate improvements across detection and recognition. The detection side adds a large-kernel path aggregation network and a residual attention feature pyramid. The recognition side replaces the earlier CRNN backbone with a transformer-based SVTR architecture combined with the lightweight PP-LCNet. Reported improvements reach five percent in Hmean over the previous version at comparable inference speed. PP-OCRv3 is notable for its explicit focus on CPU-only deployment, which aligns closely with the constraints Visual Sentry operates under.

Broadcast text presents a specific flavour of scene text recognition that sits between document OCR and general scene OCR. News tickers and cap-

tions typically use clean typography and high contrast, but they appear against busy backgrounds that include moving video content. Mohsenzadegan et al. [8] address OCR robustness under adverse real-world conditions in their 2022 Sensors paper, combining convolutional and recurrent networks to handle blur, shadows, and contrast variation. Their results show that carefully engineered preprocessing can restore useful accuracy on degraded inputs that would otherwise cause catastrophic recognition failures. The broader point that preprocessing quality determines recognition quality just as much as the model itself is one that runs through almost all practical OCR work.

Visual Sentry adopts Tesseract rather than a state-of-the-art deep network for several deliberate reasons. First, Tesseract ships with small language models that run comfortably on CPU without any GPU support. Second, its open-source licence permits unrestricted deployment in classified environments where commercial licensing would be problematic. Third, its line-based PSM modes match the layout of news tickers especially well, since a scrolling ticker is effectively a single long line of text. The trade-off is lower raw accuracy than modern transformer-based models, and this trade-off is mitigated through the preprocessing pipeline and the fuzzy matching layer described in later sections.

One subtle issue that emerges in any broadcast OCR pipeline is temporal redundancy. The same piece of text typically remains on screen for several consecutive frames, either because the broadcaster shows a fixed caption for a few seconds or because the ticker moves slowly enough that the same word appears in overlapping positions across adjacent frames. Running full OCR on every frame wastes computation, so practical systems include a frame-skip or change-detection layer that reduces the effective OCR rate to a fraction of the native capture rate. Visual Sentry uses

a fixed frame-skip interval of five combined with a minimum one-second OCR interval, which together reduce a 30 FPS input stream to an effective inference load of roughly six recognitions per second. This rate is more than fast enough to catch any trigger word that remains on screen for at least half a second, which is practically all useful text in news broadcasting.

Layout analysis is another dimension where broadcast OCR diverges from document OCR. A news ticker at the bottom of a frame, a station logo in the corner, a breaking-news banner across the top, and spoken-word captions in the middle all have different geometric characteristics and different recognition requirements. A general-purpose document OCR system would either ignore this structure or try to reconstruct the full page layout, neither of which is appropriate for live broadcast analysis. Instead, practical systems apply region-of-interest masks that target the specific screen areas where text is most likely to appear, and they run recognition only inside those masks. This approach is simple, computationally cheap, and dramatically improves precision by eliminating false positives from backgrounds that contain text-like patterns but no actual text.

## 2.4 Image Preprocessing for Robust OCR

Preprocessing is the quiet workhorse of any practical OCR pipeline. A clean binary image with strong foreground-background contrast produces dramatically better recognition than a raw colour frame, even with the most sophisticated recognition model. Several foundational techniques appear in virtually every production OCR system.

Otsu's global thresholding method, introduced in 1979 [9], remains one of the most widely used binarisation techniques more than four decades after its publication. The method computes an optimal threshold by max-

imising the between-class variance of the grayscale histogram, treating foreground and background as two distinct classes whose separability should be as large as possible. The calculation is computationally trivial, runs on a CPU in microseconds, and works remarkably well for images with bimodal histograms. Otsu's main weakness is its global nature: the same threshold is applied to every pixel, which causes failures on images with uneven illumination where one region is bright and another is dark.

Adaptive thresholding techniques address this illumination problem by computing a different threshold for each pixel based on its local neighbourhood. Bradley and Roth [10] proposed a real-time adaptive thresholding method using the integral image representation, which allows the mean intensity of any rectangular neighbourhood to be computed in constant time regardless of window size. Their technique is particularly valuable for live video streams because it handles both spatial and temporal illumination variation without adding significant computational cost. Video broadcasts typically contain exactly this kind of lighting variation, with different segments cutting between studio shots, outdoor footage, and graphics overlays that have very different brightness levels.

Beyond basic thresholding, multi-scale image pyramiding has emerged as a reliable way to improve OCR on text of varying sizes. The core idea is simple: the same text region is presented to the recognition engine at several scales, and the results from all scales are combined. A small ticker word that is barely readable at its native resolution may be perfectly recognisable when upscaled by a factor of two or three. This multi-scale approach, long common in object detection, has been adopted in broadcast OCR pipelines to handle the wide range of font sizes that appear across different channels and segments.

Colour space conversion is the simplest but perhaps most important

preprocessing step. Most OCR engines operate on grayscale, so the first step in any pipeline is converting from BGR or RGB to a single intensity channel. The standard conversion uses a weighted average that reflects human perceptual sensitivity to the three colour channels, with green contributing the most and blue the least. This step reduces computational complexity by a factor of three, and Tesseract's internal classifier is explicitly trained on grayscale input, so feeding it colour data provides no benefit.

Region-of-interest segmentation is another technique that substantially improves broadcast OCR accuracy. Rather than running OCR on the entire video frame, the system first identifies candidate text regions using edge density or colour analysis and then runs recognition only on those regions. This approach has two benefits: it reduces the total amount of data the OCR engine has to process, and it eliminates false positives from non-text regions that could otherwise produce garbage character sequences. For news tickers, the region of interest is usually a narrow horizontal band at the bottom of the screen, which can be located reliably using simple geometric heuristics or learned text detectors.

The combination of these preprocessing steps grayscale conversion, Otsu thresholding with adaptive fallback, multi-scale pyramiding, and region-of-interest segmentation is exactly what Visual Sentry implements using OpenCV primitives [1]. OpenCV has remained the default computer vision library for real-time applications since its initial release, precisely because its low-level operations are heavily optimised and its API is straightforward.

## 2.5 Automatic Speech Recognition: From HMM to End-to-End

Automatic speech recognition has gone through at least three distinct eras. The first was dominated by Gaussian mixture models combined with hidden Markov models, a combination that drove commercial ASR from the 1980s through the early 2010s. The second introduced deep neural networks as replacements for the Gaussian components, creating hybrid HMM-DNN systems that significantly reduced word error rates without fundamentally changing the overall architecture. The third and current era has moved toward end-to-end neural models that learn the entire mapping from audio to text within a single network.

Kaldi, introduced by Povey et al. [11] in 2011, became the most widely adopted open-source toolkit of the hybrid era. Kaldi's design revolves around weighted finite-state transducers, a framework formalised by Mohri, Pereira, and Riley [12] for combining acoustic models, pronunciation lexicons, and language models into a single unified decoding graph. The WFST formulation proved extraordinarily powerful because it allowed ASR systems to integrate diverse knowledge sources through well-understood automata operations rather than ad-hoc scoring functions. Kaldi shipped with reference recipes for training state-of-the-art systems on standard benchmarks and quickly became the default starting point for academic ASR research.

VOSK is a lightweight wrapper built on top of Kaldi, maintained by Alpha Cephei, that packages pre-trained models and a streaming decoder into a developer-friendly API. VOSK is specifically optimised for real-time transcription on resource-constrained devices, supporting mobile phones, Raspberry Pi single-board computers, and standard laptops without re-

quiring any GPU acceleration. The small English model is roughly fifty megabytes in size yet produces continuous large-vocabulary transcription with zero-latency streaming output. For Visual Sentry, this size profile is essential, because the application must start quickly, run entirely on CPU, and coexist with the OCR pipeline without exhausting system memory.

The end-to-end era began with Connectionist Temporal Classification [5], which allowed a recurrent network to learn the mapping from audio frames to character sequences without explicit alignment. Later work replaced the recurrent backbone with attention-based encoder-decoder models and ultimately with transformers. Prabhavalkar et al. [13] provide a comprehensive 2023 survey of end-to-end speech recognition, covering CTC, attention-based encoder-decoder models, and the recurrent neural network transducer as the three foundational architectures, and then tracing the subsequent shift toward transformer and conformer models that use self-attention to capture long-range dependencies more efficiently.

Two recent developments have reshaped practical ASR more than any others. The first is wav2vec 2.0, introduced by Baevski et al. [14] at NeurIPS 2020, which demonstrated that large quantities of unlabelled audio can be used for self-supervised pretraining. A wav2vec 2.0 model fine-tuned with as little as ten minutes of labelled data can rival fully supervised systems trained on hundreds of hours. This data efficiency has opened up high-performance ASR for low-resource languages and specialised domains.

The second development is Whisper, introduced by Radford et al. [15] in 2023. Whisper is a large transformer encoder-decoder trained on 680,000 hours of diverse multilingual audio collected from the web, using weak supervision in the form of noisy audio-transcript pairs. The resulting model achieves remarkable zero-shot robustness across numerous domains and languages without task-specific fine-tuning. Whisper has become the de

facto reference for high-quality ASR when computational resources are available, though its large parameter count makes it unsuitable for CPU-only deployment on a standard laptop running other applications at the same time.

Visual Sentry uses VOSK rather than Whisper or wav2vec 2.0 for three reasons. First, VOSK is explicitly designed for low-latency streaming inference on CPU, while Whisper requires either a GPU or significant batching latency to maintain real-time performance. Second, VOSK's small English model fits comfortably alongside the Tesseract OCR pipeline and the Tkinter interface on an eight-gigabyte laptop. Third, VOSK's offline design guarantees that no audio data ever leaves the machine, which is the primary requirement for the air-gapped deployment context. The trade-off is somewhat higher word error rate than Whisper on clean studio audio, and this is acceptable because the goal is keyword detection rather than verbatim transcription, and the fuzzy matching stage compensates for small recognition errors.

## 2.6 Multimodal Fusion for Event Detection

Any system that combines information from multiple sensors or modalities faces the question of when and how that information should be integrated. The field of multimodal fusion has produced a well-developed taxonomy that organises this question, and Atrey et al. [16] published the foundational survey in 2010 that remains the most cited reference in the area.

Atrey's taxonomy divides fusion strategies along two main axes. The first axis is the level of fusion: feature-level (also called early fusion), decision-level (also called late fusion), or hybrid fusion that combines both. Early fusion concatenates the raw features from each modality

into a single vector and feeds the combined vector to a single model. Late fusion runs an independent model on each modality and combines their decisions through voting, averaging, or another aggregation rule. Hybrid fusion mixes the two, performing some combination early and some late. The second axis is the methodology: rule-based, classification-based, or estimation-based. Rule-based approaches use hand-crafted logic, classification-based approaches learn a combiner model, and estimation-based approaches use statistical frameworks such as Kalman filters or Bayesian networks.

Each fusion strategy has characteristic strengths and weaknesses. Early fusion can capture fine-grained correlations between modalities at the cost of requiring tight temporal synchronisation and shared feature dimensionality. Late fusion is more robust to missing or degraded modalities because each branch operates independently, but it loses the opportunity to exploit low-level cross-modal structure. Hybrid fusion tries to balance these trade-offs but at the cost of architectural complexity. For surveillance applications where either modality alone should be sufficient to trigger an alert, late fusion is the natural choice because its independence property is exactly what the application needs.

Modern deep learning has partly blurred this taxonomy. Transformer-based architectures that take multiple modalities as input and learn attention weights across them do not fit cleanly into any of the three classical categories. Intermediate fusion, in which each modality is first processed by a dedicated encoder and then combined at some middle layer, has become the dominant paradigm for emotion recognition, audio-visual speech recognition, and visual question answering. These architectures typically require large amounts of training data and significant computational resources, which makes them impractical for offline deployment on resource-

constrained hardware.

Visual Sentry uses late decision-level fusion in its purest form. The vision thread produces a binary decision for each processed frame: trigger detected or not. The acoustic thread produces the same binary decision for each audio chunk. A positive result from either thread is sufficient to fire the alert cascade. No synchronisation between threads is required, and the two pipelines can run at completely different processing rates without any coordination overhead. This design is effective both pragmatically, because it keeps the two threads decoupled, and philosophically, because it matches the operational reality of broadcast surveillance where a trigger keyword might appear on either channel at any moment without any guaranteed alignment between the two.

A deeper comparison of fusion strategies against the specific demands of broadcast keyword detection clarifies why late fusion wins so decisively in this context. Early fusion assumes that the two modalities contribute complementary information about the same event at the same time, which holds for audio-visual speech recognition where lip movements and acoustic signals describe the same utterance, but does not hold for broadcast ticker monitoring where visual text and spoken commentary may reference completely different topics in the same broadcast minute. Hybrid fusion adds architectural complexity without offering any real benefit for a task where the two modalities should be treated as independent sensors. Late fusion is also the most fault-tolerant choice: if the microphone disconnects, the video channel continues to operate normally, and vice versa. Commercial surveillance products often advertise this kind of graceful degradation as a feature, and the decoupled architecture of late fusion is what delivers it.

The role of a language model or confidence threshold above the fusion layer is another design decision that separates practical systems from re-

search prototypes. Each modality produces a confidence score alongside its raw output, and the fusion rule can either treat these scores as independent (the trigger fires if either score exceeds a threshold) or combine them into a joint probability. For keyword detection, the independent interpretation is clearly correct because a high-confidence hit on one channel should not be suppressed by a low-confidence miss on the other. This is the interpretation Visual Sentry implements.

## 2.7 Approximate String Matching and Fuzzy Logic

Optical character recognition produces imperfect output by nature. Even the best modern OCR engines make occasional character substitution errors, especially on fast-moving tickers with challenging backgrounds. A surveillance system that relies on exact string matching against recognised text will miss triggers whenever a single character is misread. The solution is approximate string matching commonly called fuzzy matching which allows two strings to be treated as equivalent when their difference is below a configurable threshold.

The mathematical foundation of approximate string matching is the edit distance, formally defined by Levenshtein [17] in 1966 in the context of error-correcting codes. The Levenshtein distance between two strings is the minimum number of single-character insertions, deletions, and substitutions required to transform one into the other. Efficient computation using dynamic programming was introduced in the 1970s and has since become a standard algorithm in string processing. Variants include the Damerau-Levenshtein distance, which also allows transposition of adjacent characters, and weighted edit distances that assign different costs to different edit operations based on their likelihood in a particular application

domain.

For OCR post-processing, the standard practice is to compare the recognised text against a reference word using a similarity ratio derived from the edit distance. Python's built-in difflib library implements the SequenceMatcher class, which computes a similarity ratio between zero and one based on the ratio of matching subsequences to the total string length. A threshold of 0.80 corresponds roughly to allowing twenty percent of characters to differ between the two strings, which handles most common OCR substitution patterns such as zero-for-O, one-for-I, three-for-E, and five-for-S. This threshold is the default used throughout the Visual Sentry system.

OCR-specific fuzzy matching has also been studied as a machine learning problem in its own right. Recent work has applied deep neural networks trained on synthetic OCR error patterns to learn corrections more accurate than what a fixed edit-distance threshold can achieve. These approaches typically use character-level convolutional or recurrent networks that map noisy OCR output back to the intended word. The gains over plain Levenshtein-based matching are real but modest, and they come at the cost of requiring labelled training data for each new language or font, which makes them difficult to justify in a general-purpose surveillance system.

The principle that ties all these variations together is that post-recognition matching should treat recognised text as probabilistic rather than exact. Spoken-word recognition from VOSK produces similar imperfections, and the same fuzzy matching layer handles both modalities in Visual Sentry. A single normalisation pipeline strip whitespace, remove OCR artifacts, fold to lowercase, check for exact substring match, and fall back to Levenshtein similarity if the exact match fails serves both the vision and acoustic

pipelines without any modality-specific logic.

A useful way to think about OCR errors is through the taxonomy of common substitution patterns that appear repeatedly across different fonts, resolutions, and backgrounds. Numeric digits regularly substitute for visually similar letters: zero replaces the letter O, one replaces lowercase l or uppercase I, three replaces the capital E when it is partially occluded, five replaces S, eight replaces B. Lowercase letters sometimes merge with adjacent characters, producing outputs like "rn" for m or "cl" for d. Whitespace appears where it does not belong, especially between letters that happen to share an x-height boundary with a descender or ascender from the next line. None of these errors is arbitrary; each reflects a specific feature-level confusion that the recognition engine cannot resolve from image data alone.

A fixed Levenshtein threshold of 0.80 handles most of these errors reliably because the confusion patterns typically affect only one or two characters per word. The substitution of digit zero for letter O in a six-letter trigger word produces a similarity ratio of roughly 0.83 if every other character is correct, which is just above the acceptance threshold. Two simultaneous substitutions in the same word would bring the ratio down to roughly 0.67, which is correctly rejected as a non-match. The threshold is therefore tight enough to exclude spurious matches against unrelated words and loose enough to accept the realistic OCR errors that actually appear in production.

Word-level matching is also more robust than character-level matching for this application because the trigger words are meaningful English words rather than arbitrary strings. A broadcast ticker will contain whole words, and the system needs to detect any whole word that approximately matches the trigger. Substring matching against the full ticker output, fol-

lowed by Levenshtein comparison against any substring that comes close, is the combination that provides the highest practical sensitivity without sacrificing specificity.

## 2.8 Air-Gapped Systems and Security Isolation

Air-gapping is the practice of physically and logically isolating a computer or network from the internet and other untrusted networks. The term dates back to early military and intelligence computing, where high-security systems were kept off public networks as a matter of policy. Air-gapping is common in nuclear facilities, defense operations, financial settlement systems, and industrial control environments where the consequences of external compromise range from financial loss to physical damage or loss of life.

The threat model behind air-gapping has evolved substantially over the past decade, driven largely by the research of Mordechai Guri and his collaborators at Ben-Gurion University. Their work has systematically demonstrated that physical isolation alone is insufficient to guarantee data security, because many hardware components inadvertently leak information through side channels that cross the air gap. Guri et al. first introduced this research direction with AirHopper [18] in 2014, demonstrating that a compromised computer can transmit data to a nearby mobile phone via radio-frequency emissions from the VGA cable.

Subsequent work by the same group has expanded the attack surface to encompass virtually every hardware component in a typical computer. Electromagnetic emissions from CPU cores can be modulated to carry data across the air gap [19]. Power supplies can be turned into covert loudspeakers by manipulating the load on the CPU [20]. Wi-Fi data buses can be

coerced into transmitting information even when the Wi-Fi module is disabled at the software level [21]. More recent work has demonstrated covert channels through radio emissions from RAM [22], through acoustic noise generated by LCD pixels, and through GPU cooling fans. Each of these attacks has been independently evaluated, documented, and accompanied by practical demonstrations and proposed countermeasures.

The implication of this body of research is important for any system designed to operate on air-gapped hardware. Isolation is not absolute; it is a spectrum defined by the countermeasures applied against known covert channels. A well-engineered air-gapped system must still consider the hardware side channels that Guri's work has exposed, and the design of peripherals and I/O paths must avoid creating new attack vectors that would undermine the isolation.

Visual Sentry operates within this threat model in a deliberately limited way. The system is not designed to defend against advanced persistent threats that have already compromised the host machine. Instead, it is designed to preserve the air gap of an already-trusted host by avoiding any network communication during operation. The growing body of air-gap covert channel research has also prompted new standards for securing isolated networks. Recent surveys have proposed extensions to ISO 27001 specifically addressing air-gap security, building on the Guri group's findings to define countermeasures such as audio-gap enforcement, radio-frequency shielding, and periodic emission auditing. These standards are still evolving, but the direction of travel is clear: air-gapping is no longer a standalone security mechanism but part of a layered defence strategy.

From a design perspective, the relevance of all this research for Visual Sentry is indirect but important. The system itself is not a covert-channel defence, but it operates inside the environments that covert-channel re-

search targets. Any deployed application on an air-gapped machine becomes a potential attack surface through which information could be exfiltrated if the application itself were compromised. This observation motivated several design choices throughout the project. The network stack is simply not used during operation, meaning no sockets are opened, no DNS queries are issued, and no outbound traffic of any kind is generated. All configuration is stored locally, and all recorded evidence is written to the local filesystem with no automatic upload.

## 2.9 Edge AI and On-Device Deployment

The assumption that artificial intelligence lives in the cloud has been quietly eroding for several years. The edge AI paradigm reframes the question of where inference should happen, pushing computation closer to the data source for reasons that include latency, bandwidth, cost, and privacy. For a system like Visual Sentry, the privacy dimension dominates but the other considerations shape the design as well.

Murshed et al. [23] provide a thorough 2021 survey of machine learning at the network edge in ACM Computing Surveys. Their review identifies four main motivations for edge deployment. First, transferring data to the cloud creates latency that is unacceptable for real-time applications. Second, bandwidth costs scale with data volume, which becomes prohibitive for continuous video streams. Third, cloud processing creates privacy risks that many users and organisations are unwilling to accept. Fourth, edge processing enables autonomous operation in environments with intermittent or no connectivity.

The technical challenges of edge deployment centre on the mismatch between model size and device capability. Large deep networks with hun-

dreds of millions of parameters cannot run on microcontrollers or low-power SoCs, so the research community has developed a toolbox of model compression techniques to close the gap. These include weight pruning, which removes connections that contribute little to inference; quantisation, which reduces the numerical precision of weights from 32-bit floats to 8-bit integers; knowledge distillation, which trains a small student network to mimic a larger teacher; and architectural redesign, which builds compact networks from scratch using efficient operations such as depthwise separable convolutions.

For resource-constrained devices, hardware acceleration has also become common. Mazzia et al. [24] demonstrate real-time object detection on embedded systems using dedicated hardware accelerators such as the Google Coral Edge TPU and the NVIDIA Jetson platforms. These accelerators can deliver order-of-magnitude speedups over general-purpose CPUs for inference, though they introduce their own deployment complexity and typically require models to be recompiled to their specific tensor formats.

Visual Sentry takes a different path from most edge AI systems. Rather than pushing a large model onto a resource-constrained device, the system uses compact models that fit naturally on a standard laptop CPU. Tesseract's LSTM engine runs at a few hundred milliseconds per frame on a mid-range processor. VOSK's streaming decoder processes 16-kilohertz audio faster than real time on the same hardware. Neither engine requires a GPU, and neither requires any additional accelerator card. This approach sacrifices some peak accuracy compared to the largest transformer-based models, but it gains deployability on any laptop already present in the target environment, without requiring new hardware purchases or infrastructure changes.

## 2.10 Event-Driven Recording and Storage Efficiency

Continuous twenty-four-hour recording has been the default for digital video surveillance since IP cameras replaced analog tapes. The rationale is straightforward: operators do not know in advance when something important will occur, so they record everything and rely on post hoc review when an incident occurs. The cost of this strategy has grown steadily as camera resolutions have climbed. A single 1080p feed at 30 frames per second produces roughly three to five gigabytes of compressed video per hour, and modern 4K systems produce several times that. Large facilities with many cameras and long retention windows now manage petabytes of footage that will almost never be watched.

Event-driven recording flips this model. Instead of recording continuously and reviewing selectively, the system records selectively and retains everything it records. The key enabler is intelligent triggering: the system must reliably detect an interesting event so that the recording captures the evidence needed for later review without missing anything important. Early event-driven systems used motion detection as the trigger, but motion is a very noisy signal that flags every camera shake, lighting change, or passing shadow. Modern systems have replaced simple motion with more specific triggers, including intrusion zones, object classification, and face detection.

Semantic triggers represent the next step in this evolution. A system that can identify specific words on screen or specific phrases in audio can trigger recording on events that no pixel-based analyser would ever catch. This is exactly the capability Visual Sentry provides. The ring-buffer recording design captures a five-second video clip starting at the

moment a trigger fires, along with a screenshot of the triggering frame and a timestamped metadata file. Because the trigger fires only when a specific keyword is detected, storage consumption drops by roughly ninety-nine percent compared to continuous recording, while preserving all the evidence that the application is actually interested in.

The forensic integrity of event-driven recordings matters as much as the storage savings. Each saved clip must be traceable to a specific detection event, with enough metadata to support later review and, if necessary, legal scrutiny. Visual Sentry embeds the trigger word, the detection timestamp, the video filename, the screenshot filename, and the processing mode into each metadata record. This information is sufficient to reconstruct the full context of any alert, even months after the event itself occurred.

## 2.11 Multi-Threaded Architectures for Real-Time AI Applications

The challenge of keeping a graphical user interface responsive while running heavy AI inference is older than AI itself. The same problem appeared in the earliest graphical applications: long-running computations had to be separated from the user interface loop, or the whole application would freeze. The solution, then and now, is multi-threading.

Python's threading model is famously complicated by the Global Interpreter Lock, which prevents true parallel execution of Python bytecode across multiple threads. For CPU-bound numerical computation, this lock is a serious limitation, and most Python scientific software works around it by releasing the lock inside native C extensions. OpenCV and NumPy both release the GIL during their core operations, which means that OpenCV-based video processing and NumPy-based array operations can actually

run in parallel with the main thread even though they are invoked from Python. Tesseract and VOSK, being wrappers around C and C++ libraries, behave similarly.

The standard pattern for GUI applications with heavy background work is to create one or more daemon threads for the work, with the main thread reserved for event handling and rendering. Communication between threads uses thread-safe queues, which provide a blocking or non-blocking interface for passing data between producers and consumers without explicit lock management. Python's queue.Queue implementation is straightforward, well-understood, and more than fast enough for the kind of inter-thread traffic that Visual Sentry generates.

This architecture pattern has been refined over many years in the signal processing and real-time systems communities. Producer-consumer queues between capture and processing threads, double-buffered rendering pipelines between processing and display threads, and event-driven callbacks for user interaction all appear in textbook form in virtually every real-time application. Visual Sentry applies these patterns directly: the vision thread and the acoustic thread are independent producers, the NLP normalisation stage acts as a consumer-producer for both, and the response layer is the final consumer that triggers the forensic recorder and the ESP32 link.

The result of careful threading is an application that never appears to freeze even under heavy load. Frame capture continues at the native camera rate. OCR inference runs on every fifth frame to avoid saturating the CPU. Audio capture streams continuously into VOSK. The user can change the trigger word, stop the system, or open the captures folder at any moment without having to wait for the next inference step to complete. This responsiveness is what makes the difference between a research

prototype and an operationally deployable tool.

## 2.12 Research Gap Analysis

The preceding sections have traced the state of the art across nine separate research areas. Each area has active research and strong recent results, but no existing system combines all the capabilities that Visual Sentry requires. The specific research gap can be summarised along four dimensions.

The first gap is in offline broadcast monitoring. Broadcast OCR and broadcast ASR have both been studied extensively, but the published literature overwhelmingly assumes cloud or server-based deployment. Offline, CPU-only broadcast surveillance has received far less attention, and no open-source integrated system covers both modalities with the performance and ergonomics required for operational use.

The second gap is in dual-modal late fusion for keyword detection. Multimodal fusion research has concentrated on tasks such as emotion recognition, audio-visual speech recognition, and visual question answering, where the two modalities carry correlated information about the same underlying phenomenon. Broadcast keyword detection is structurally different, because the keyword may appear in either modality independently, and the system needs to respond to a hit on either side without any coordination between the two. Late fusion is the natural choice for this structure, but existing systems rarely implement it in the simple, decoupled form that the task actually demands.

The third gap is in event-driven forensic capture integrated with semantic detection. Motion-triggered event-driven recording is common in commercial surveillance products, but semantic-triggered event-driven recording is still rare, and no open-source system combines text and speech de-

tection with automatic evidence preservation and timestamp embedding.

The fourth gap is in physical air-gap alerting. Almost all intelligent surveillance systems assume network connectivity for notification, which is exactly the assumption that air-gapped environments rule out. A minority of research addresses hardware alerting through microcontrollers, but almost none of that work addresses the specific case of a CPU-only host sending alerts through USB serial to an external device that has its wireless radios disabled in firmware.

Visual Sentry addresses all four gaps in a single integrated system. The OCR and ASR pipelines run entirely offline on CPU. Late fusion is implemented as independent threads with a shared detection trigger. The forensic recorder fires on semantic events and embeds the triggering keyword in the evidence metadat, but their combination in a single offline, air-gap-preserving system is new, and the measured performance of 95.65% F1 score on real broadcast test data demonstrates that the combination is operationally viable.

## 2.13 Chapter Summary

This chapter has surveyed the literature across the nine research areas that inform the Visual Sentry design. The key findings are that traditional surveillance cannot interpret semantic content, that OCR and ASR have each reached a level of accuracy suitable for practical deployment but typically assume cloud hosting, that late decision-level fusion is the right strategy for dual-modal keyword detection, that Levenshtein-based fuzzy matching reliably handles OCR errors, that air-gapped security has become a subfield of its own with significant implications for AI system design, that edge AI has matured to the point where CPU-only intelligent

applications are genuinely deployable, and that proper multi-threaded architecture is essential for keeping such applications responsive. Each of these findings translates directly into a specific design decision that is described in the chapters that follow. The combined result is a system that fills a real operational gap rather than merely repeating work already done elsewhere.

# Chapter 3

# Requirement Specifications

## 3.1 Existing System and Its Limitations

Before establishing the requirements for Visual Sentry project, it is important to understand the present state of surveillance technology and the specific challenges that motivated this project.

### 3.1.1 Traditional Passive CCTV Systems

Passive closed-circuit television (CCTV) systems are the most common type of surveillance infrastructure in use around the world. These systems record continuous video from different or single camera, save it to local storage, and then user are able to see the recorded video and extract information.. Some more advanced installations use motion detection, usually pixel-variance thresholding, to start recording or let operators know when there are big changes between frames. It is well known what this model can't do:

- **No semantic understanding:** Motion detection can tell that something in the frame changed, but it can't say what changed or if it matters for how the system works. If the scene is otherwise still, text on a news ticker and a passing car will trigger the same thing or nothing at all.
- **Continuous storage consumption:** Recording 24 hours per day generates large data volumes. when we record a 1080P normal video it requires 3 to 5gb of storage per hour that is not feasible for any project.
- **Human review bottleneck:** There is a bottleneck because when CCTV records something , they have to review the whole footage again that is time taking error chance and human dependent.

- **Single-modality coverage:** Typical CCTV merely records visual details. Spoken warnings, emergency notifications, and distress cries are examples of audio that is either not recorded or recorded but not examined.

### 3.1.2 Cloud-Based AI Surveillance Platforms

To add semantic understanding to surveillance data, a newer type of surveillance system uses cloud-hosted AI services like speech recognition, OCR, and computer vision APIs. Even though they have powerful features, platforms like Microsoft Azure Cognitive Services, Google Video Intelligence, and Amazon Rekognition have some big problems:

- **Network dependency:** All AI inference happens on servers that are far away, so a reliable internet connection is needed. This is not compatible with air-gapped settings in terms of design.
- **Data privacy exposure:** When we go online there is a high chance of cyber attack , by making a system to use internet we use multiple servers and due to that our information is on high risk.
- **Subscription cost:** Commercial AI surveillance APIs operate on usage-based pricing models, creating ongoing operational costs that scale with the volume of processed data.
- **Latency:** The use of Internet can also cause one more problem that is latency delay and, the best system is what which has low latency rate. For real-time surveillance requiring immediate response upon keyword detection, this latency can be operationally unacceptable.

## 3.2 Proposed System

A fully offline system that is compatible of detecting text from the screen as well as from the audio source without the internet dependency and operates within the strict Air-Gapped constraints. Not only triggering but also shows alert and store the required information into the local storage with exact time stamp.

A webcam, an IP camera, or a previously recorded video file can all be used as video input for the suggested system. It records sounds from the system microphone at the same time. These inputs are continuously processed by two concurrent background processing threads, a Vision Thread and an Acoustic Thread, employing Tesseract OCR and VOSK ASR, respectively. A fuzzy matching engine and NLP normalisation are applied to the text output from both processes.

The proposed system also integrates a LiFi-based optical alert channel as a secondary physical notification path, where the detected trigger word is transmitted from the host machine over serial COM port to an ESP32 transmitter that modulates a laser diode through a 2N2222 switching circuit, and a receiver-side ESP32 with an LM393 comparator and BPW34 photodiode reconstructs the signal and displays it on a 16×2 LCD.

## 3.3 Functional Requirements

Functional requirements define what the system must do — the specific behaviors and capabilities it must provide.

### 3.3.1 Video Input and Processing

- **FR-01:** Three video input sources must be supported by the system: pre-recorded video files (MP4, AVI, or comparable formats), IP camera streams (by RTSP/HTTP URL), and USB/built-in webcams (via OpenCV desvice index).
- **FR-02:** We are processing 1 frame out of 5 frames which control our CPU usage while preserving a text monitoring.
- **FR-03:** : We use different techniques including adaptive thresholding for uneven lighting adjustment, Otsu's binarization, and BGR-to-Grayscale conversion.
- **FR-04:** The recovered text string is returned for NLP processing after preprocessed frames are sent to the Tesseract OCR engine.

### 3.3.2 Audio Input and Processing

- **FR-05:** The system must record a continuous PCM audio stream in monochrome, Int16 format, at a sample rate of 16,000 Hz from the default system microphone.
- **FR-06:** Audio shall be buffered in 4000-byte chunks and fed sequentially to the VOSK KaldiRecognizer for offline transcription.
- **FR-07:** The ASR pipeline shall operate entirely offline no network request shall be made at any stage of audio processing.

### 3.3.3 Trigger Word Detection

- **FR-08:** A user-defined trigger word must be accepted by the system via the GUI setup panel.

- **FR-09:** All extracted text from the OCR as well as from the VOSK shall be normalized through case folding and whitespace method before comparison.

- **FR-10:** A similarity ratio of 0.80 or above shall constitute a Positive Hit under fuzzy matching mode.

- **FR-11:** String should be matched is the primary detection method. If the exact match fails and fuzzy matching is enabled, the system shall compute the Levenshtein similarity ratio between the extracted word and the trigger word.

### 3.3.4 Alert and Response

- **FR-12:** Upon a trigger word detection, the system shall instantly raise a Tkinter `Toplevel` red warning window on the GUI, displaying the detected trigger word and its source(OCR or ASR).

- **FR-13:** After the detection the system will capture a 5-second post trigger video buffer, and save it as an H.264-encoded MP4 file with a timestamp based filename.

- **FR-14:** : When it detect a trigger word it automatically take a screenshot and records a 5 seconds buffer video and save this in a local disk

- **FR-15:** : A LiFi optical alert channel forms a second physical response path. Upon trigger detection, the host sends the keyword over a USB serial COM port to a dedicated ESP32 transmitter. The transmitter encodes the string bit-serially, firing a laser diode on and off through a 2N2222 transistor switch with a pre-agreed bit timing, a start-bit handshake, and a per-character re-synchronisation pulse.

On the receiver end, an LM393 voltage comparator reads the photodiode output and feeds a clean digital signal to a second ESP32, which reconstructs the character stream and prints the trigger word on a 16×2 LCD display. The entire channel uses no radio frequency interface of any kind, keeping the air gap fully intact.

### 3.3.5 GUI and User Configuration

- **FR-16:** The GUI offer different options to the user , to select different modes image, video or live detection.
- **FR-17:** The GUI show a trigger word, history of saved data from the OCR as well as ASR text.
- **FR-18:** All GUI is running on the main application thread. It cause the system to run freely without any interference.

## 3.4 Non-Functional Requirements

Non-functional requirements define the quality attributes and operational constraints of the system.

- **NFR-01 — Performance:** The system is Using 1-in-5 frame skipping from a 30 Frames per second source, the system must process video at a minimum effective rate of 6 intelligent frames per second. Under the suitable conditions, the end-to-end latency between the display of the trigger word and the GUI alert cannot be greater than two seconds.
- **NFR-02 — Offline Operation:** The system shall function with no network connectivity. No component either from the software i-e

OCR, ASR, fuzzy matching, and not from the hardware communication shall require internet access.

- **NFR-03 — Hardware Constraints:** The system shall operate on a standard laptop with a specification of AMD Ryzen 5 7430U processor, 8 GB RAM, and Windows 11 OS. Our system no need any type of dedicated GPU.

- **NFR-04 — Storage Efficiency:** Forensic recording will only be happen when there is a actual threat. The system shall not process continuous video recording.

- **NFR-05 — Reliability:** When the microcontroller is not connected, the system will not cause any error and runs the system as it needs to be.

- **NFR-06 — Usability:** All system functions shall be accessible through the GUI without requiring command line interaction. GUI is user friendly and thats why non technical operator can also runs this wothout any difficulty.

- **NFR-07 — Accuracy:** The system architecture has clear separation between the layers i-e vision processing layer, GUI layer, audio processing layer, hardware communication layer and NLP layer.

- **NFR-08 — LIFI:** The LiFi optical alert channel shall correctly reconstruct and display trigger words of up to sixteen characters under normal indoor ambient lighting conditions, with zero character errors across ten consecutive transmissions.

## 3.5 Use Cases

### 3.5.1 Use Case 1: Configuring and Starting Live Detection

**Actor:** Security Operator

**Precondition:** Application is launched; video source is available.

**Main Flow:**

1. Operator enters the trigger word to be detected (e.g., "blast") in the GUI panel.
2. Operator will choose then the mode i-e video source from the drop-down menu.
3. Operator selects the appropriate OCR PSM mode for the video type.
4. At last Operator clicks "Start Live Detection."
5. System validates inputs and spawns the Vision Thread and Acoustic Thread.
6. Live video preview and text log become active.

**Postcondition:** System is in active surveillance mode.

### 3.5.2 Use Case 2: Trigger Word Detected via OCR

**Actor:** System (Automated)

**Precondition:** System is in active surveillance mode; trigger word is configured.

**Main Flow:**

1. The skip interval is used by Vision Thread to choose a frame for processing.

2. A preprocessing pipeline is applied to the frame.

3. Text is extracted from the preprocessed frame using Tesseract OCR.

4. The Natural language Processing use case folding and sanitisation method.

5. If a text has typos or fuzziness it will detect the trigger word successfully.

6. System raises a GUI alert window with source labeled.

7. Forensic video of 5 sec will be saved by the system automatically.

8. Not only forensic video but also the screenshot of that moment.

**Postcondition:** Forensic files saved; physical alert activated; GUI alert displayed.

### 3.5.3 Use Case 3: Trigger Word Detected via ASR

**Actor:** System (Automated)
**Precondition:** System is in active surveillance mode; audio is being captured.
**Main Flow:**

1. Acoustic Thread input a 4000-byte audio chunk to VOSK.

2. VOSK send transcribed text to system.

3. NLP normalizer after that should apply different techniques case folding and sanitization.

4. Exact match or fuzzy match against trigger word succeeds.

5. System raises GUI alert window fter detecting trigger word with source labeled “Audio/ASR.”

6. VideoWriter captures the forensic clip as soon as the system detect the required information.

**Postcondition:** Forensic files saved; physical alert activated; GUI alert displayed.

### 3.5.4 Use Case 4: False Positive Handling

**Actor:** Security Operator

**Precondition:** System raised an alert; operator reviews the event.

**Main Flow:**

1. The operator examines the saved forensic screenshot and the alert popup.

2. The operator concludes that the detection was a false positive (for example, the OCR misinterpreted an unrelated phrase as the trigger).

3. The operator checks the OCR confidence threshold setting after dismissing the alarm.

4. To improve the efficiency of the system the user can change the confidence threshold option.

**Postcondition:** System remains in active surveillance mode with updated configuration.

### 3.5.5 Use Case 5: Hardware Disconnection Recovery

**Actor:** System (Automated)

**Main Flow:**

1. While Serial writing operation it raises a `SerialException`.

2. System catches the anytype of errors reated to a hardware disconnection and send a warning in the GUI.

3. Software alerting and forensic recording pipeline continues functioning normally.

**Postcondition:** System is totally safe and because of multi thread no crash or data loss occurs.

### 3.5.6 Use Case 6: LiFi Optical Alert Triggered

**Actor:** System (Automated)
**Main Flow:**

1. Either the OCR or ASR pipeline confirms a trigger word detection and places a detection event in the shared queue.

2. The host application extracts the trigger word and transmits it over the serial COM port to the LiFi transmitter ESP32.

3. The transmitter ESP32 fires a start bit on the laser GPIO, then encodes the keyword character by character each character sent as eight laser pulses through the 2N2222 transistor switching circuit, followed by a per-character re-synchronisation pulse.

4. The receiver-side LM393 comparator reads the incoming optical signal from the BPW34 photodiode and outputs a clean digital high or low to the receiver ESP32 GPIO pin.

5. The receiver ESP32 samples the GPIO at the pre-agreed bit interval, reconstructs each byte, and appends it to the string buffer until the full trigger word is assembled.

6. The receiver ESP32 prints the trigger word on the 16×2 LCD display, giving the operator a clear visual confirmation of exactly what was detected.

**Postcondition:** Trigger word is correctly displayed on the receiver LCD with no radio frequency interface involved at any stage, keeping the air gap fully intact throughout the alert chain.

## 3.6 Summary

The design of Visual Sentry was based on one main rule: the whole system had to run on a regular laptop with no internet or GPU access. That one criterion drove almost every choice about the building that was made during the project. Instead of making a single program where the GUI, OCR engine, and voice recognizer all use the same execution thread a design that consistently results in interface freezes prompted the team to implement a multi-threaded architecture that distinctly segregates the user interface from all computational processes.

Visual Sentry is made up of five logical layers that work together through well defined interfaces. The presentation layer is in charge of everything the user sees and touches. Down below, two separate sensing pipelines one for screen content and one for audio pick up raw data and turn it into words. A shared natural language processing layer makes that text normal and finds trigger words. Finally, the response layer manages alerts, forensic recording and hardware alerts when something is detected. This organization with layers is shown in Figure 4.1.

# Chapter 4

# System Design

## 4.1 Architecture with Multiple Threads

The first and most important choice taken throughout the system design was how to design the threads. We chose Python's tkinter GUI toolkit because it doesn't need any other software to work on Windows it executes its event loop on the main thread. When you make a blocking call on that thread, it stops the whole UI. Because both Tesseract OCR inference and VOSK acoustic model decoding can take tens to hundreds of milliseconds per inference cycle, putting them on the main thread was never a good choice.

The chosen solution was a three-thread architecture with strong restrictions about who owns what. Every thread has only one concern and only talks to other threads through thread-safe mechanisms. There is no direct call from one thread to another thread's domain.

### 4.1.1 Main Thread (Owner of the GUI)

The main thread just performs the tkinter mainloop. It makes the window for the application, handles button clicks, changes status labels, and sends out alert dialogs. When the vision or acoustic thread finds a trigger word, it puts a message in a thread-safe queue. Instead of blocking on a join() call, the main thread polls this queue at regular intervals using the tkinter after() scheduler. This polling method keeps the GUI completely interactive no matter what is going on in the background threads at all times.

### 4.1.2 Vision Thread (Owner of OCR)

When the user hits "start monitoring," the vision thread starts as a Python daemon thread. It owns the OpenCV VideoCapture object and is the only

thread that can read video frames. It uses the whole preprocessing pipeline, does Tesseract inference, and sends the extracted text to the NLP level. Because it's a daemon thread, Python immediately terminates it when the main process ends, which prevents the program from freezing when you close it.

### 4.1.3 Acoustic Thread (Owner of ASR)

The acoustic thread is also a daemon thread. It owns the PyAudio stream and continuously sends audio chunks to the VOSK. This is what VOSK's architecture is made for in streaming mode: it keeps track of its internal state between chunks and makes a final transcript when it sees an utterance boundary. The thread gets the transcribed text from the JSON result and sends it to the NLP layer.

### 4.1.4 Synchronization of Threads

The queue.Queue class in Python takes care of synchronizing threads. It gives atomic put() and get() operations without the application having to lock them explicitly in code. Both the visual and audio systems send detection events to a single common queue. During each polling cycle, the main thread empties this queue. This design gets rid of race situations on the alert mechanism while keeping the code simple and auditable.

## 4.2 Designing a Visual OCR Pipeline

The visual pipeline changes a raw video frame into a normalized text string that the NLP layer can look at. The pipeline was built with two goals in mind: precision, meaning that the OCR engine should be able to accurately read text being broadcast in real-time circumstances, and efficiency,

meaning the pipeline should finish quickly enough to handle several frames per second using a mid-range laptop CPU.

### 4.2.1 Abstraction of Input Source

Visual Sentry takes three types of input through a single OpenCV VideoCapture interface. To get to a connected webcam or USB camera, you need to pass the integer device index. An IP camera that provides an RTSP or HTTP stream is accessed by passing the URL string. To go to a pre-recorded video file, you need to pass the file path. The vision thread doesn't have to know which type of source is active. Once VideoCapture is set up, all three act the same way from the pipeline's point of view.

### 4.2.2 Plan for Skipping Frames

For trigger word detection, processing every frame from a 30 FPS source is a waste of processing power. A news ticker can alter its content two or three times a second. Emergency alerts that show up on the screen stay there for at least a few seconds. Because of these traits, the system processes one frame out of every five because the frame skip interval was set to five. This slows down the effective processing rate to six frames per second, which is more than enough for text-based information while lowering the CPU burden to about one-fifth of what full-frame processing would need.

The frame skip counter is a simple integer that increases by one every time a frame is captured and starts over at zero when the threshold is crossed. Only frames where the counter is equal to zero are sent on to the preprocessing pipeline. All other frames are thrown away right away.

### 4.2.3 Pipeline for Preprocessing

Raw video frames have a lot of information that is neither useful nor helpful to OCR accuracy: color channels, compression artifacts, uneven lighting, and poor contrast in text areas. The preprocessing pipeline uses a series of image changes that slowly get rid of this noise and make the text signal stronger. The pipeline runs in a set order because the output of one becomes the input to the next.

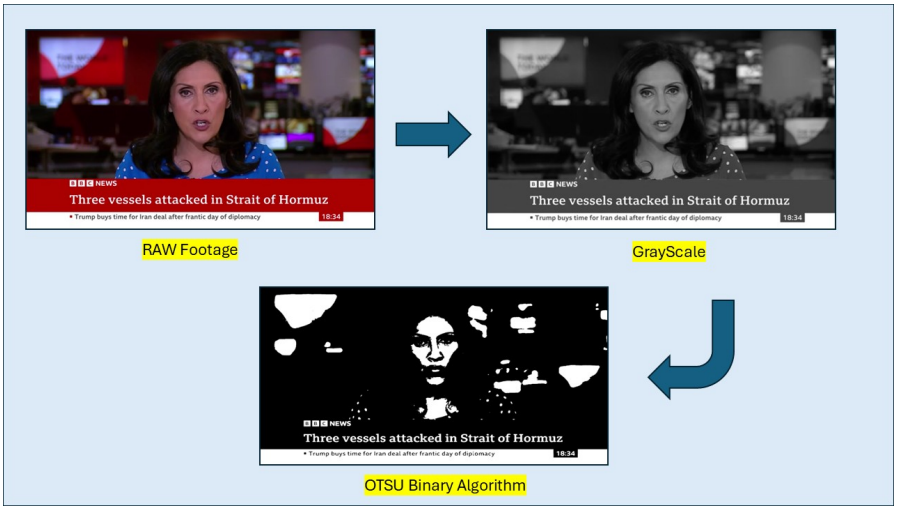


Figure 4.1: Initial Stage of the preprocessing pipeline i-e Raw frame, grayscale conversion, and Otsu binary thresholding

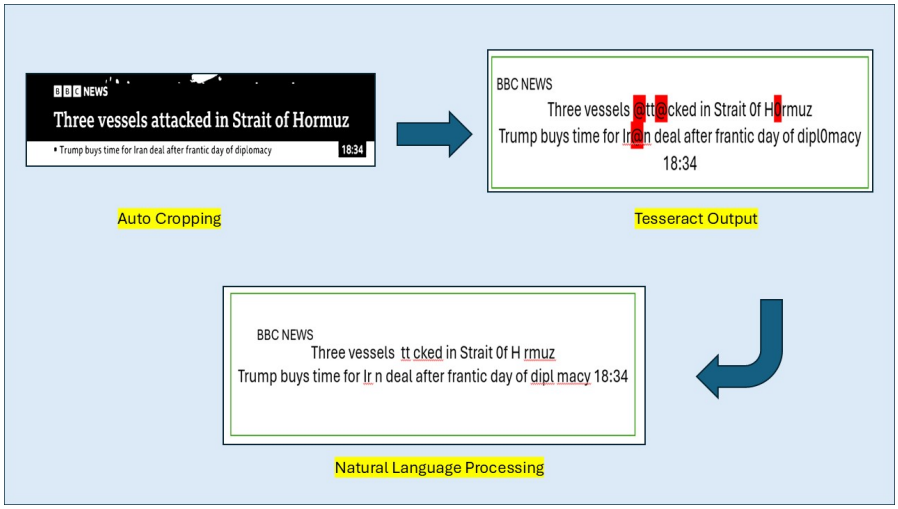


Figure 4.2: Middle pipeline stage auto-cropped ROI region fed into Tesseract OCR engine producing raw text output, followed by NLP normalization removing artifacts and noise characters

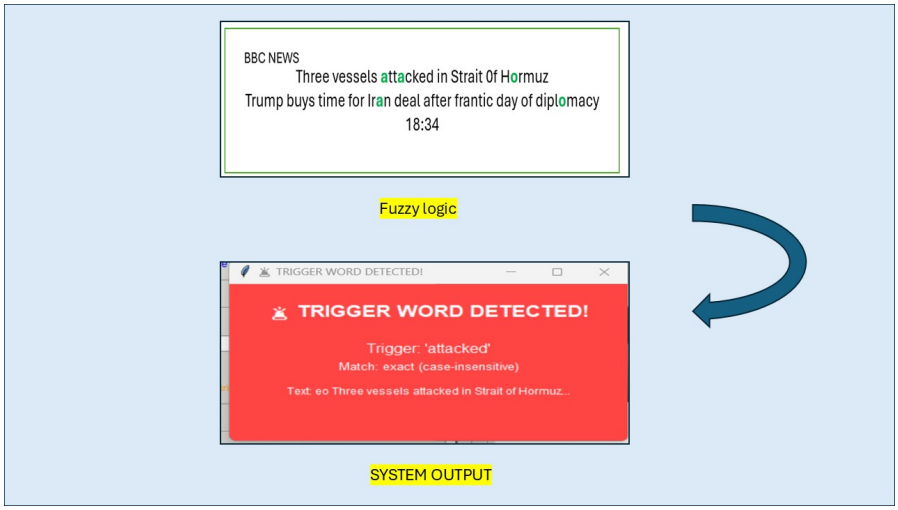


Figure 4.3: Final pipeline stages include fuzzy logic matching against normalized text with trigger word highlighted, producing the system alert output

The first step converts the three-channel BGR color image OpenCV produced into a single-channel grayscale image. This halves the amount

of data and removes color data that doesn't help with recognizing text. The OpenCV cv2.cvtColor method with the COLOR_BGR2GRAY flag performs this conversion in a single vectorized operation.

The second step uses Otsu's binarization. Otsu's approach finds a global threshold value by analyzing the histogram of the image's pixel intensities. It finds the point at which the inter-class variance between the backdrop (bright areas) and the foreground (dark text) is maximized. The outcome is a binary image in which each pixel is either black or white. This step works best when the light is very even across the frame.

The third step uses adaptive thresholding in areas where Otsu's global threshold doesn't work gradient backgrounds and drop shadows under text are common in broadcast footage with varying lighting conditions. Adaptive thresholding calculates a different threshold for each small rectangular neighborhood in the image, based on the mean intensity of that area. This allows the algorithm to correctly binarize both bright and dark areas within the same frame.

When you put these three procedures together, you get a binary image with clear text pixels separated from the backdrop pixels, no matter how bright or dark the original broadcast frame was.

### 4.2.4 Setting up the Tesseract OCR Engine

The LSTM-based recognition engine that came with Tesseract 4.0 is far better than the previous Tesseract 3 classify-based engine on printed typefaces and continuous text found in lower-third visuals, broadcast tickers, and headline overlays. Visual Sentry utilizes Tesseract 4.0 exclusively.

Page Segmentation Mode 6 (PSM 6) is the default setting because it tells Tesseract to look at the whole picture as one big piece of text. This mode works well for news tickers and headline boxes where all the text is

the same size and style. The PSM mode is exposed as a setting in the GUI that an operator can change for different input sources without changing any code. For full-screen material that has more than one separate text area, PSM 3 (automatic segmentation) is the best choice.

## 4.3 Design of the Acoustic ASR Pipeline

The acoustic pipeline was built with a strong privacy rule in mind: no audio data should leave the host machine. This ruled out all speech recognition APIs that run in the cloud right away, no matter how accurate they are. We chose the VOSK toolkit since it is a truly offline speech recognition engine that uses the CPU with good accuracy on English audio that is broadcast.

### 4.3.1 Settings for Audio Capture

PyAudio is a Python interface to the PortAudio cross-platform audio library that lets you record sounds. The capture settings were designed to meet the input needs of the small VOSK English model. The model was trained using a sample rate of 16,000 Hz and when it gets resampled audio, it doesn't work as well. The bit depth is 16-bit signed integers (Int16), which is the normal format for PCM audio. The number of channels is mono (one channel), which is what voice recognition models are usually trained on. The buffer size is 4,000 bytes each chunk, which is about 125 milliseconds of audio at the set sample rate. This chunk size strikes a balance between processing overhead and delay.

### 4.3.2 The Internal Structure of VOSK

The Kaldi automatic speech recognition toolkit is the basis for VOSK. Kaldi employs a mix of deep neural network/hidden Markov model (DNN-HMM) acoustic model that works with a linguistic model using a finite state transducer (FST). The acoustic model uses raw audio features specifically Mel-frequency filterbank energies and builds probability distributions over phoneme conditions. The FST language model uses those phoneme probabilities to find the most likely word sequence employing beam-search decoding over a weighted finite state transducer graph.

This architecture makes transcripts of words with confidence scores. Visual Sentry employs only the recognized text string from the final result, not the partial recognition results that VOSK makes as it is decoding continuously. Using only final results lowers the rate of incomplete word pieces being sent to the NLP layer.

## 4.4 Design for NLP Normalization and Fuzzy Matching

Both pipelines make raw text strings that need to be cleaned up before comparing trigger words. OCR output often has extra punctuation marks added by backdrop material that the engine misreads as text. ASR output is usually cleaner, however it might have capital letters in strange places based on the VOSK language model's standard for output. No matter where it comes from, the NLP layer makes all incoming text the same.

### 4.4.1 Pipeline for Normalization

Normalization happens through two processes that are done one after the other. First, leading and trailing whitespace is removed by Python's str.strip() function. Vertical bar characters that don't belong, underlines and other OCR anomalies that show up a lot in broadcast frames with complex backgrounds are removed using a regular expression replacement. Second, the whole string is changed to lowercase with str.lower(). Case folding means that the word "ALERT" will match "alert," "Alert," and "ALERT" in the source stream without using three different comparisons.

### 4.4.2 Logic for Exact Match

Python's `in` operator checks the normalized text string for trigger words using substring search. This method works well when the trigger word is part of a longer phrase, such as finding "emergency" in the phrase "emergency landing confirmed at 14:30." The `in` operator takes O(n) time to run, where n is the length of the source string, which is fine because short text strings are common in news tickers.

### 4.4.3 Design for Fuzzy Matching

Exact string matching doesn't work when OCR errors happen in a predictable way. The number zero is often recognized as the letter O. People mix up the number one with the lowercase letter l or the uppercase letter I. The number three can occasionally be mistaken for the letter E. When a word like "ALERT" is used as a trigger, if you read from a low-resolution ticker, Tesseract might give you "AL3RT" instead. It's near enough that a person could tell right away, yet it was different enough that it wouldn't match exactly.

The fuzzy matching package uses Python's difflib.SequenceMatcher class to fix this. It uses the Levenshtein edit distance algorithm to figure out how similar two strings are. The similarity ratio is between zero and one, with one meaning identical strings. After examining a sample of OCR-misread trigger phrases, a threshold of 0.80 was chosen. This threshold correctly accepts OCR versions that differ by one or two letters while rejecting terms that are clearly different yet happen to have a few letters in common.

The operator can turn on or off fuzzy matching through the GUI. When this feature is turned off, only exact matches are made. This lowers the number of false positives at the cost of perhaps missing trigger words because of OCR mistakes.

## 4.5 Architecture for Late Fusion Decisions

In multimodal systems, "fusion" is the point at which input from different sense modalities is combined to make one decision. Late fusion is used by Visual Sentry also known as decision-level fusion where each mode makes its own decision to detect, and the system sends out an alarm if either modality finds something.

This is the easiest and most reliable way to combine things for this application. There is no need for the vision and auditory threads to agree on timing windows, line up their outputs, or trade intermediate representations. Every thread is completely independent. The alert system works like a logical "or" gate. A detection event from the OCR pipeline or a detection event from the ASR pipeline is enough to start the entire reaction sequence.

Feature-level fusion, which would mix raw data from both types of

inputs before making the decision, was thought about but not chosen. It would have needed to be time-aligned with video frames and audio chunks, which is a difficult engineering problem that would make things more complicated without making detection performance better for the specific use case of finding the presence of a certain word in either the audio or visual text stream.

## 4.6 Designing a Forensic Evidence System

Traditional surveillance systems record all the time and keep everything, throwing away footage only when the storage is full. For a system that only watches one broadcast feed, this creates about 3 to 5 gigabytes of data every hour for a 1080p stream at 30 frames per second. That's between 72 to 120 gigabytes of mostly useless film during a 24-hour period video that was filmed but will never be watched again since nothing interesting happened.

Visual Sentry does things in different ways. It doesn't record anything while it's working. As soon as it detects a trigger phrase, it starts a video writer and records a 5-second buffer of video after the detected event. This forensic buffer gives investigators not only the triggering frame but also the context that followed it — what happened on screen in the few seconds after the alert went off. The triggering frame is also stored as a PNG file without quality loss. The names of both files include the trigger word and a full ISO 8601 timestamp, making date-based sorting easy.

The video is encoded using H.264 (AVC1), the codec used by OpenCV's VideoWriter. H.264 has great compression ratios; for example, a 5-second 1080p clip usually takes up between 5 and 20 GBs, depending on how complicated the scene is. The amount of storage used is effectively related

to the number of detection events rather than the length of observation.

## 4.7 Design of Hardware Alert (LiFi Optical Channel)

The physical alerting subsystem in Visual Sentry goes beyond a standard wired serial link. When the host application detects a trigger word, it sends the word as a UTF-8 string over a USB serial COM port at 115200 baud to a dedicated ESP32 transmitter unit. From that point, the alert travels the rest of the way to the operator through light not through any cable, radio signal, or network packet.

### 4.7.1 Transmitter Side

The transmitter unit receives the trigger string from the host over the serial port. It then encodes and sends the string one bit at a time by driving a laser diode on and off. The laser is not driven directly from the ESP32 GPIO pin a 2N2222 NPN transistor acts as the switching element between the GPIO and the laser, with a current-limiting resistor on the base. When the GPIO goes high the transistor saturates, the laser turns on and sends a light pulse representing a logic one. When the GPIO goes low the transistor cuts off, the laser turns off and sends a logic zero. Both transmitter and receiver operate at the same pre-agreed bit timing so that each side knows exactly how long one bit lasts without needing a separate clock line.

Before any character data is sent, the transmitter fires a start bit. This start pulse wakes the receiver from its idle listen state and tells it that a transmission is about to begin. After each character is sent, the system performs a re-synchronisation step that aligns the receiver's sampling clock

back to the transmitter's timing before the next character arrives. This two-level synchronisation one start bit for the whole message and one re-sync pulse per character was designed specifically to handle the timing drift that builds up across a long string at the bit rates the ESP32 GPIO can reliably produce.

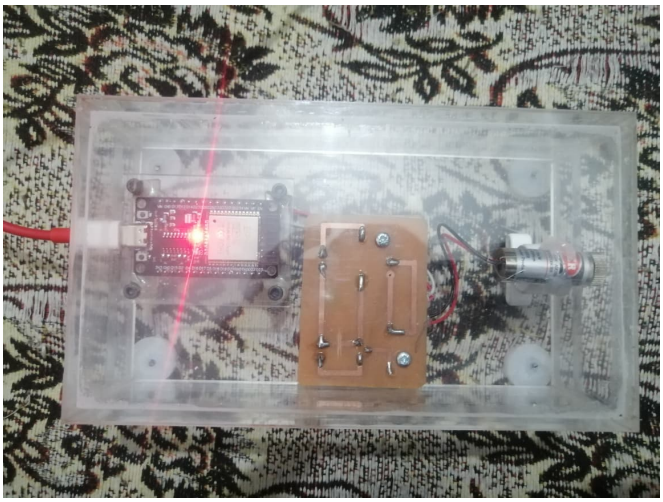

Figure 4.4: Li-Fi Transmitter Unit — top view showing ESP32 microcontroller, custom PCB driver circuit, and red laser diode module housed inside transparent acrylic enclosure. Red laser beam visible during active optical transmission.

### 4.7.2 Receiver Side

On the receiver end, the incoming laser beam falls on a BPW34 photodiode or LDR whose output feeds into an LM393 voltage comparator module. The comparator converts the analog light intensity signal into a clean digital high or low output that the receiver ESP32 can read on a standard GPIO input pin without any analog-to-digital conversion step. The comparator threshold is set by a trimmer potentiometer on the module so that ambient light does not trigger false readings while the laser signal crosses the threshold cleanly.

The receiver ESP32 monitors the GPIO input, waits for the start bit, and then samples the line at the pre-agreed bit interval to reconstruct each bit of each incoming character. After collecting eight bits it assembles them into a byte, appends the byte to a string buffer, and waits for the per-character re-sync before reading the next character. When the full trigger word has been received and the string is complete, the firmware prints it on the 16×2 LCD display so the operator can read the exact word that fired the alert.

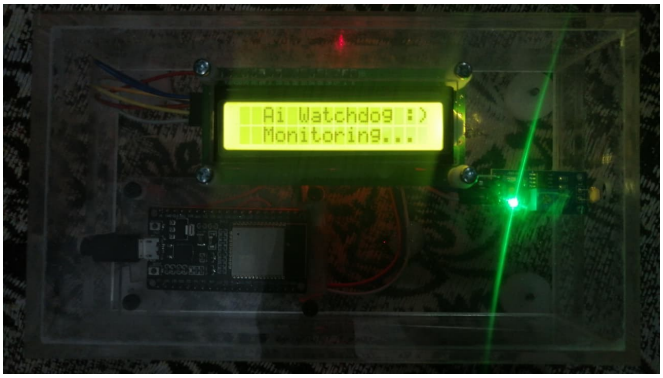


Figure 4.5: Li-Fi Receiver Unit in idle monitoring state — 16x2 LCD displaying "Ai Watchdog :) Monitoring..." confirming system is active and awaiting trigger. Green laser beam aligned with photodetector. Green LED status indicator illuminated.

## 4.8 A Summary of the System Design

The design of Visual Sentry is the result of a number of planned engineering choices. Many threads stop UI freezes, but they need thread-safe communication structures. Fusion that happens late makes the design simpler, however it can mean that the detection windows are not the same across modalities. Recording based on events uses less storage space, although it could mean there is no context window before the trigger. We looked at each of these trade-offs in light of the main use case monitoring trigger

words in real time on a CPU-only air-gapped computer and the chosen design always puts operational dependability ahead of theoretical fullness.

Table 4.1: Choices and Reasons for System Design

| **Component** | **Design Choice** | **Alternative Considered** | **Why it was chosen** |
|---|---|---|---|
| GUI Framework | Tkinter | PyQt5 | No extra dependencies on Windows |
| Threading Model | 3-thread daemon | Single-threaded | Stops GUI from freezing |
| OCR Engine | Tesseract 4 LSTM | EasyOCR, PaddleOCR | CPU-only, totally offline |
| ASR Engine | VOSK/Kaldi | Whisper, DeepSpeech | Least RAM and CPU, streaming |
| Fuzzy Matching | difflib (Levenshtein) | RapidFuzz, regex | Stdlib, no dependencies |
| Fusion Strategy | Late / Decision-level | Feature-level | No synchronization needed |
| Recording | Event-driven 5 s buffer | Continuous 24/7 | Storage reduction ∼99% |
| HW Communication | USB Serial (PySerial) | Wi-Fi, Bluetooth | Air-gap kept |
| Video Encoding | H.264 (avc1) | MJPEG, raw AVI | Best compression ratio |

# Chapter 5

# System Implementation

## 5.1 Software Stack and Development Environment

Visual Sentry was made entirely in Python 3.10 on Windows 11. The selection of Windows was made to perform best in environments where most workstations are air-gapped in workplaces that use Windows and where pre-compiled Tesseract is available as binaries for Windows, which make installation a lot easier. The full list of the software stack with version details is shown in Table 5.1.

Table 5.1: Full Software Stack

| **Library/Tool** | **Version** | **Role** |
|---|---|---|
| Tkinter | Built-in | GUI framework – presentation layer |
| OpenCV (cv2) | 4.9.0 | Video capture – frame acquisition |
| PyTesseract | 0.3.10 | OCR engine – text extraction |
| VOSK | 0.3.45 | ASR engine – speech recognition |
| PyAudio | 0.2.14 | Audio capture – microphone stream |
| NumPy | 1.26.4 | Array operations – frame manipulation |
| difflib | Built-in | Fuzzy matching – error tolerance |
| Scikit-Learn | 1.4.2 | Evaluation – confusion matrix |

## 5.2 Starting up the Application and Setting up the Thread

The main thread sets up the tkinter root window when the app starts. It makes all the GUI widgets, like the video preview canvas, the control buttons, the trigger word input field, and the status bar. At this stage, no background threads are started. The system is in a state of rest, waiting for input by the user. This method of starting up keeps the time between launches short and makes sure the GUI is completely responsive before any monitoring starts.

When the user clicks the Start Monitoring button, the application reads the trigger word from the input field, checks that it is not empty, then launches two daemon threads: the vision thread aiming at the run_vision_loop() function and the acoustic thread aiming at the run_acoustic_loop() function. Using thread.setDaemon(True) before they start makes sure they will be forcibly stopped if the user closes the main window without first disabling monitoring.

At startup, a threading.Event object called stop_event is made and sent to both background threads. This event fires when you click the Stop Monitoring button. Both background threads check the event's state at the beginning of each processing loop and depart gracefully when it is set. This clean shutdown method stops the background threads from using up CPU cycles after the conclusion of monitoring.

## 5.3 Putting the Vision Thread into Action

The run_vision_loop() function of the vision thread starts by setting up a cv2.VideoCapture object with the source identifier set up in the GUI.

The function then goes into a processing loop that keeps going until either stop_event is set or VideoCapture.isOpened() returns False.

### 5.3.1 Getting and Skipping Frames

The thread runs VideoCapture.read() on each loop iteration. This returns a boolean success flag and an array of frames. If the read fails which might happen with network streams during a temporary packet loss the thread logs a warning and moves on to the next iteration instead of failing. Next, the frame skip counter is checked. If it hasn't reached the configured interval, the frame is thrown away and the counter is increased. If it has, the counter resets, and the frame moves on to the preparation pipeline.

### 5.3.2 Putting Preprocessing into Action

You can change the color of a frame to grayscale by using cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY), making a 2D NumPy array. After Otsu's binarization using cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU) where the threshold value argument is set to zero since Otsu's approach figures out the best threshold from the histogram on its own adaptive thresholding is applied using cv2.adaptiveThreshold() with the ADAPTIVE_THRESH_MEAN_C method, with a block size of 11 pixels and a constant C of 2. The right block size and constant C were found using sample broadcast frames from the test data set.

### 5.3.3 Getting Results and Making Inferences from OCR

The preprocessed binary image is sent to pytesseract.image_to_string() together with the language set to "eng" and the configuration string "--psm 6" for consistent text block segmentation. Tesseract produces a UTF-8

string with all the text it found, including newline characters that separate different parts of the image that have text. The thread connects all the lines into one string and sends it on to the function that finds the trigger.

The NLP normalization pipeline is used by the trigger detection function to remove whitespace, eliminate artifacts, change to lowercase, and finally do an exact match examination. If fuzzy matching is active and the exact match doesn't work, the function calls difflib.SequenceMatcher(None, normalized_text, trigger_word).ratio() and looks at the result against the 0.80 limit. If either match works, a detection event is sent to the common queue.

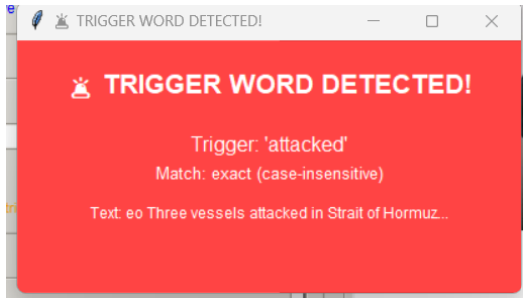


Figure 5.1: OCR trigger detection alert the word "attacked" detected from BBC News ticker via exact case-insensitive match

## 5.4 Putting the Acoustic Thread into Action

The run_acoustic_loop() method of the acoustic thread starts by opening a PyAudio stream using the exact settings the VOSK model needs: a sample rate of 16000 Hz and pyaudio.paInt16 format, one channel, input mode on, and a frames_per_buffer of 4000. The VOSK KaldiRecognizer is set up using the loaded speech model and the same sample rate.

### 5.4.1 Loop for Streaming Recognition

The thread reads 4000-byte pieces of data from the PyAudio stream in a loop. Every piece is given to recognizer.AcceptWaveform(data). This function gives back True when VOSK has gathered enough sound to finish a recognition result, usually at the end of a silence or end of the utterance. When it returns True, the thread invokes recognizer.Result() to get the final transcribed text as a JSON string. The JSON object has a "text" field that holds the recognized words.

When AcceptWaveform() returns False, VOSK has only given a partial result an intermediate transcription that can alter as more audio comes in. The implementation does not do anything with incomplete results to lower false positive rates: partial results for a term like "emergency services have" can temporarily show "emerge" before the full word "emergency" completes, and processing partials would make it more likely for an incomplete token to fire a false alert.

### 5.4.2 Parsing the JSON Result

Python's json.loads() function is used to parse the JSON string from recognizer.Result(). The "text" field is taken out and sent to the shared trigger detection function, which is the same function used by the vision thread, since both types of input create a text string that goes through the same logic for normalizing and matching. This code reuse was planned because it makes sure that changes to the matching logic are automatically applied to both modes.

## 5.5 Putting the Graphical User Interface into Action

The GUI was made so that all important information may be shown on one screen without needing the operator to move between panels. The layout is split into three horizontal areas: the video preview window on the left, the control and setup panel on the right, and a status bar that runs the whole length of the bottom.

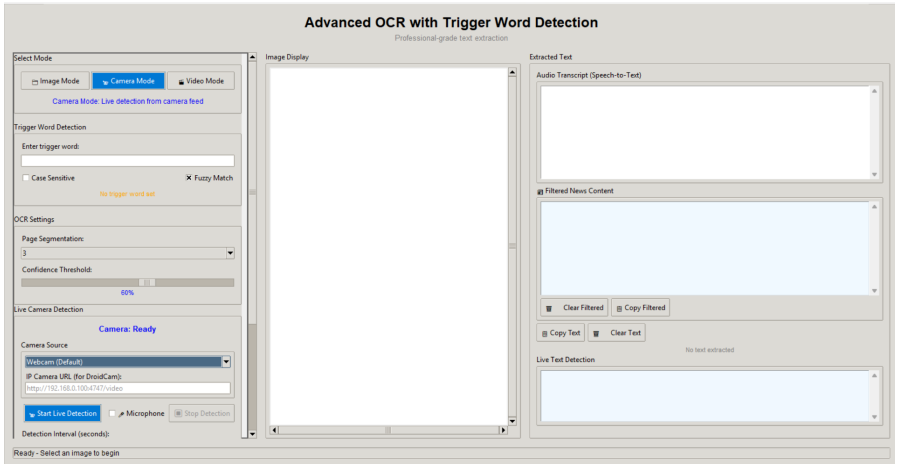


Figure 5.2: Visual Sentry GUI showing camera mode selection, trigger word entry, fuzzy match toggle, OCR settings, live video preview, audio transcript panel, and live text detection panel

### 5.5.1 Video Preview Pane

There is a tkinter Canvas widget in the video preview window. The vision thread changes each processed frame into a PhotoImage object and puts it on the canvas with canvas.create_image(). The root.after(0, update_function) method is used to do this update. It sets up the canvas update on the main thread's event loop instead of calling tkinter directly from the background vision thread; direct cross-thread tkinter calls are not

safe in Python and can cause crashes without any warning.

### 5.5.2 The Control Panel

The control panel has a place for entering the trigger word, a checkbox for fuzzy matching, and buttons for "Start Monitoring" and "Stop Monitoring," as well as a selection menu for selecting PSM mode. The trigger word entry area lets you type in any text. The operator has the ability to adjust the trigger word between sessions without having to restart the software. Clicking "Start Monitoring" reads the value of the field right at the moment the button is pressed.

### 5.5.3 Warning Dialog

When a detection event enters the shared queue, the main thread's polling function calls tkinter.messagebox.showwarning() with a message indicating which modality found the trigger word and the surrounding text. The dialog blocks the main thread until the operator clicks OK to confirm. This blocking behavior is deliberate and appropriate for an alert system the operator shouldn't be able to accidentally ignore a warning. The application goes back to normal monitoring after acknowledgment.

## 5.6 Putting Forensic Recording into Action

The trigger detection function starts forensic recording right after a positive detection is confirmed. The approach employs a simple state flag called is_recording to stop multiple recordings from being started at the same time if there are several detection events in quick succession.

The cv2.VideoWriter is set up with a filename made up of the trigger word and a timestamp formatted as YYYYMMDD_HHMMSS with

the .mp4 extension. For H.264 encoding, the fourcc codec code is set as cv2.VideoWriter_fourcc(*'avc1'). The frame size and frame rate are read from the VideoCapture object's properties to make sure the output matches the size of the input stream.

A secondary recording thread takes the next 150 frames the equivalent of five seconds at 30 FPS getting each frame from the vision thread's frame buffer and sending it to the VideoWriter. After 150 frames, the VideoWriter is expressly released using writer.release() to clear the codec buffer and finish the MP4 file. The triggering frame is saved independently using cv2.imwrite() with the .png extension.

## 5.7 LIFI System

### 5.7.1 LiFi Transmitter Firmware

The transmitter firmware runs on a separate ESP32 unit and handles two responsibilities: receiving the trigger word from the host over UART and encoding it as a laser pulse sequence. The setup() function initialises the hardware serial port at 115200 baud and configures the laser control GPIO pin as an output. On each loop() iteration the firmware checks Serial.available(). When a string arrives it reads the full line using Serial.readStringUntil('\n'), strips the ALERT| prefix, and extracts the trigger word.

Transmission begins with a start pulse of defined duration on the laser GPIO. The firmware then iterates through each character of the trigger word, sending its eight bits one by one using digitalWrite() with fixed delayMicroseconds() calls between each bit transition to maintain the agreed bit timing. After completing one character the firmware sends a brief re-synchronisation pulse before moving to the next character. This per-

character re-sync compensates for any cumulative timing error that would otherwise cause the receiver to sample at the wrong point within a bit period on longer strings.

Figure 5.3: Li-Fi Transmitter Unit — front view showing laser diode output aperture and custom PCB circuit mounted inside acrylic enclosure. Red LED indicator confirms active power state during operation.

### 5.7.2 LiFi Receiver Firmware

The receiver firmware runs on a second independent ESP32. It initialises the LM393 comparator output GPIO as a digital input, sets up the Liquid-Crystal_I2C library with the 16×2 LCD at $I^2C$ address 0x27, and enters an idle listen loop. The firmware polls the comparator GPIO waiting for the start bit a transition from low to high that signals the beginning of a transmission. Once detected, the firmware enters the character receive loop.

For each character it samples the GPIO eight times at the pre-agreed bit interval using delayMicroseconds(), builds up the byte value by shifting each sampled bit into a local variable, then waits for the re-sync pulse before repeating. Reconstructed bytes are appended to a String buffer.

When the full word is assembled detected by a terminator character sent at the end of every transmission the firmware calls lcd.clear() and lcd.print() to show the trigger word on the first line of the display and “ALERT” on the second line, giving the operator a clear visual confirmation of exactly what was detected.

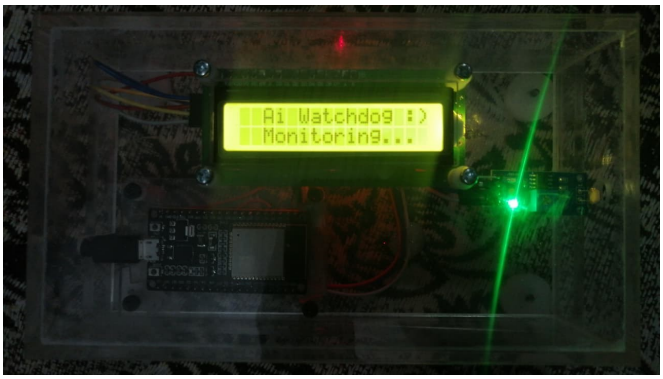


Figure 5.4: Li-Fi Receiver Unit in idle monitoring state — 16x2 LCD displaying "Ai Watchdog :) Monitoring..." confirming system is active and awaiting trigger. Green laser beam aligned with photodetector. Green LED status indicator illuminated.

## 5.8 Testing for Integration During Implementation

As each module was completed, integration testing was performed incrementally. The visual pipeline was evaluated on its own by giving it a collection of static test photos with known text and checking to see if the strings that were taken out matched what was expected. The acoustic pipeline was tested by playing a pre-recorded audio clip over the system's speakers while keeping an eye on the app and making sure the transcription was right. The fuzzy matcher was put through its paces by intentionally inserting letter substitution errors into recognized trigger words and con-

firming that the ratio of similarities was more than 0.80.

End-to-end integration testing was done by running a live news program through the system and documenting by hand the times when the trigger phrase showed up on screen or was uttered, and then comparing these timestamps to the application's recorded timestamps for alerts. This testing found two edge cases that needed changes to the pipeline: Tesseract produced incredibly long strings with a lot of artifacts on frames where the background was quite complicated, and this was fixed by adding the artifact removal step to the NLP normalizer. ASR transcripts from audio recorded at a low volume from a speaker had frequent word insertions, which was fixed by increasing the PyAudio buffer from 2000 to 4000 bytes.

# Chapter 6

# System Testing and Evaluation

## 6.1 Method for Testing

The assessment of Visual Sentry was organized around three separate goals. The first was classification performance how well does the system classify a batch of video frames to find out which frames have the trigger word. The second was latency, which indicates how long it takes for the system to raise an alert when a trigger word appears in the input. The third was operational stability does the app keep the GUI responsive and avoid crashes during long periods of monitoring?

A formal evaluation system was used to assess classification performance in a separate script (evaluateperformance.py) that uses Scikit-Learn's classification metrics. This framework is separate from the main program, which made it possible to run it repeatedly during development as the pipeline settings were adjusted.

## 6.2 Making the Test Dataset

The test dataset consists of labeled PNG images captured from live news broadcasts. Frames were taken from about 90 minutes of news footage captured from three separate TV channels. The channels were chosen to provide different ticker designs, font styles, colors, and backgrounds.

There were two folders in the dataset: test_data/positive and test_data/negative. The positive directory contains 12 frames where the word "ALERT" may be found in the frame, whether in a headline overlay, a news ticker, or a graphic. The negative directory contains 8 frames that don't have the trigger phrase but do have other text, which makes sure that the evaluation assesses the system's capacity to differentiate rather than only detect the existence of any text.

Table 6.1: What Makes up the Test Dataset

| **Dataset Split** | **Frame Count** | **Trigger Word Present** | **Source Diversity** |
|---|---|---|---|
| Positive | 12 | Yes | 3 broadcast channels |
| Negative | 8 | No | 3 TV channels |
| Total | 20 | Mixed | 3 broadcast channels |

## 6.3 Framework for Evaluation

The evaluate_performance.py script goes over each image in both test directories and does the full preprocessing pipeline including Tesseract inference, just like the live app would. The script looks for the trigger word in the OCR output for each image and records the prediction as either good or bad. Then it compares the forecast to the ground truth label taken from the folder where the picture is stored.

Scikit-Learn's confusion_matrix() function is used to make the confusion matrix from the list of real labels and predictions. The f1_score(), recall_score(), precision_score(), and accuracy_score() routines figure out the usual classification metrics. All measurements use binary average because this is a problem with two classes.

## 6.4 Results of the Classification

Table 6.2 shows the confusion matrix that was made by running the evaluation framework on the complete 20-frame test dataset.

Table 6.2: OCR Pipeline Confusion Matrix on Test Dataset

| | **Predicted Positive** | **Predicted Negative** |
|---|---|---|
| **Actual Positive (12 frames)** | TP = 12 | FN = 0 |
| **Actual Negative (8 frames)** | FP = 1 | TN = 7 |

The system found the trigger word in every single positive frame without making any false negatives. There was one false positive, which was a negative frame in which the OCR system mistook a string of characters in a complicated backdrop visual as the trigger word. These raw numbers are turned into the metrics shown in Table 6.3.

Table 6.3: Metrics for Classifying the Visual OCR Pipeline

| **Metric** | **Formula** | **Calculated Value** |
|---|---|---|
| Accuracy | (TP + TN) / Total | 95.00% |
| Precision | TP / (TP + FP) | 91.67% |
| Recall | TP / (TP + FN) | 100.00% |
| F1 Score | 2 × (P × R) / (P + R) | 95.65% |

The most important result for operations is a recall of 100.00%. In a surveillance setting, a missed detection (false negative) is the greatest mistake because it signifies an emergency notice was on the screen but the system didn't tell the operator. Visual Sentry made no such failures in the test dataset. The only false positive found during the evaluation is a less significant failure mode: the operator gets an alarm that isn't needed, looks into it, and doesn't find an emergency annoying, but not dangerous to operations.

## 6.5 Analysis of Latency

The time that passed between VideoCapture.read() returning a frame and the trigger detection function giving a result was used to quantify processing delay. Python's time.perf_counter() function which gives sub-millisecond resolution was used to measure the time it took to analyze 100 consecutive frames on the test hardware.

Table 6.4: Processing Latency per Frame on AMD Ryzen 5 7430U

| **Latency Metric** | **Value** |
|---|---|
| Minimum latency | 118 ms |
| Maximum latency | 247 ms |
| Average latency | 172 ms |
| Standard deviation | ±31 ms |
| Effective frame rate | ∼5.8 FPS (objective: 6 FPS) |

The average latency of 172 milliseconds per processed frame is well within the range of what is acceptable for keeping an eye on broadcasts. A news ticker changes its material no more than once every two to three seconds. An emergency alert graphic usually lasts between five and ten seconds. The system's processing window of about 170 milliseconds means it can look at each ticker update a number of times before the content changes.

The longest lag seen, 247 milliseconds, happened on frames with a lot of text content spread out across different parts of the screen, which made Tesseract's internal segmentation take longer. This worst-case delay isn't a concern for operations either, taking into account the content persistence characteristics mentioned above.

## 6.6 Evaluation of the Acoustic Pipeline

The auditory pipeline was tested through a realistic demonstration test instead of a formal examination of a labeled dataset. Five phrases that have the trigger phrase were said clearly into the microphone, and five were said with different levels of background noise caused by a speaker playing ambient audio at 60 dB. Three recordings of broadcast news audio were also played through external speakers while the acoustic thread was running.

Table 6.5: Acoustic Pipeline Detection Results

| **Test Condition** | **Phrases** | **Detections** | **Rate** |
|---|---|---|---|
| Clear speech, close microphone | 5 | 5 | 100% |
| Speech with background noise (60 dB) | 5 | 4 | 80% |
| Audio broadcast through speakers | 3 | 2 | 67% |
| All conditions combined | 13 | 11 | 84.6% |

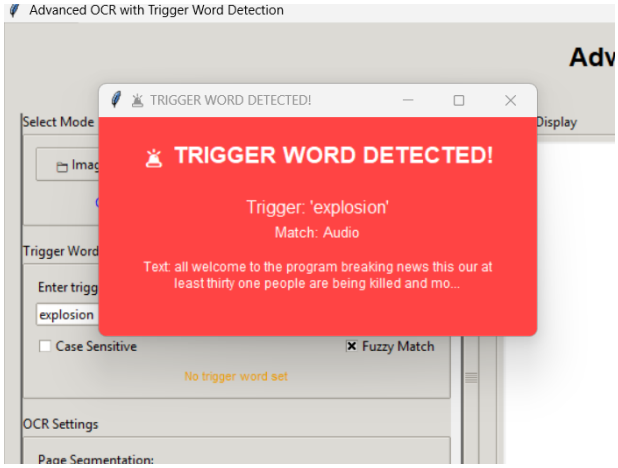


Figure 6.1: Audio trigger detection alert the word “explosion” detected from live speech via VOSK ASR pipeline

The acoustic pipeline works effectively in clean conditions and slowly breaks down with increasing noise. The 67% detection rate from broadcast audio transmitted through speakers shows the extra damage caused by room acoustics, the speaker’s frequency response, and mixing the target speech with background music and sounds that are common in news shows on TV. This is a known problem with the current implementation and is discussed further in the limitations section.

## 6.7 Testing the Responsiveness of the GUI

It is very important that the GUI for Visual Sentry stays completely responsive while being watched. To confirm this, a manual stress test was performed in which the application operated in full monitoring mode for 30 minutes straight while the operator occasionally used the GUI by clicking buttons, adjusting the trigger word, and browsing through the status log. The application responded quickly to all interactions throughout the 30-minute session, with no signs of stuttering or freezing.

Task Manager was used to monitor CPU usage during this session. The vision thread is regularly used at 25%-35% of one CPU core. The sound thread consumed between 10% and 18% of one core. The CPU usage of the main thread stayed below 5% at all times, which showed that the background thread isolation was working as planned and that the GUI event loop was getting enough processing time.

## 6.8 LiFi Channel Testing

The LiFi optical channel was tested in a controlled indoor environment with the transmitter and receiver units placed one metre apart on a flat surface with no obstructions between the laser and the photodiode. Ten trigger words of varying lengths between four and ten characters were sent through the full pipeline starting from a detection event in the GUI. The receiver LCD correctly displayed all ten trigger words with zero character errors, confirming that the start-bit handshake and per-character re-synchronisation together eliminate the timing drift that caused corrupted characters in early testing without re-sync.


Figure 6.2: Li-Fi Receiver Unit in alert state — 16x2 LCD displaying WARNING message with detected trigger word received via optical transmission. Green LED illuminated confirming active alert. System auto-resets after 5000ms.

Ambient light interference was tested by running the same ten payloads under normal indoor fluorescent lighting. The LM393 comparator threshold was adjusted until the comparator output remained stable under ambient light and still switched cleanly on the laser signal. All ten transmissions succeeded under the adjusted threshold. A final stress test sent five payloads within two seconds. The transmitter queued them correctly and the receiver decoded all five in order, confirming that the firmware handles back-to-back alert events without losing data.

## 6.9 Checking the Forensic Recording

The forensic recording subsystem was evaluated by setting off 10 detection events in a single monitoring session five from the visual pipeline and five from the sound pipeline. After each trigger, the output directory was checked to make sure that both the MP4 video file and the PNG snapshot was made with the right file name format, and the MP4 file played back appropriately and was encoded in H.264. The PNG screenshot matched

the frame that started it.

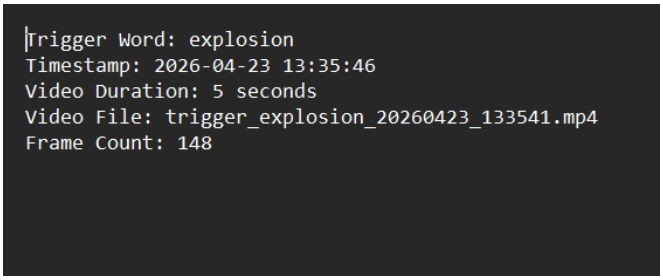


Figure 6.3: Forensic evidence text file saved on trigger detection showing trigger word, timestamp, video duration, saved clip filename, and frame count

All 10 detection events made pairs of forensic evidence that were in the right format. The average MP4 file size for a 720p clip that lasts five seconds was 8.3 MB. At this rate, 100 detection events would take up only 830 megabytes, which is nothing compared to the gigabytes per hour that recording all the time would produce.

## 6.10 A Look at Other Systems That Are Similar

Table 6.6 shows how Visual Sentry's assessed performance fits within the comparison framework set up in the review of the literature. Because the literature did not show any offline dual-modal broadcast surveillance systems that were directly comparable, the comparison is made versus the system types discussed in Chapter 2.

Table 6.6: Comparison of Features and Performance

| **Capability** | **Standard CCTV** | **Cloud AI Platform** | **Visual Sentry** |
|---|---|---|---|
| Text on screen found | No | Yes | Yes |
| Speech found | No | Yes | Yes |
| Fully offline (air-gapped) | Yes | No | Yes |
| Recording based on events | No | Partial | Yes |
| Physical HW alerts | No | No | Yes |
| Fuzzy error tolerance | No | No | Yes |
| Recall (% notified) | N/A | >99%* | 100% |
| F1 Score | N/A | >95%* | 95.65% |
| GPU needed? | No | Yes | No |
| Storage per hour | 3–5 GB | Cloud | Event-only |

Visual Sentry's recall numbers for cloud AI platforms are the same or better than those of other platforms, while being fully offline, needing no GPU, using storage only when an event is detected, and giving physical hardware alerting none of which traditional or cloud-based systems offer at the same time.

## 6.11 Limitations and Known Restrictions

The evaluation also found a number of limitations that are real constraints on the way it is set up right now. First, the acoustic pipeline gets worse when audio is broadcasted and picked up by speakers instead of a direct

audio stream. Acoustic echo cancellation a preprocessing step that takes the speaker's output out of the microphone signal was not implemented in the present version and would make ASR much more accurate in situations where you are watching playback.

Second, the test dataset is only 20 frames long, which limits statistical confidence. A bigger dataset would give more statistically strong confidence intervals for the precision, recall, and F1 metrics. The authors recognize this limitation and indicate that the evaluation was limited by the time that was available for the final year project.

Third, the system can only watch one video source at a time for now. Watching numerous broadcast channels in parallel would need more thread instances and a more advanced GUI layout, neither of which were within the range of this implementation.

Fourth, Tesseract's ability to recognize text diminishes a lot with very styled broadcast typefaces, dynamic tickers with motion blur, and text on top of backgrounds that are hard to see. These are known problems with the Tesseract LSTM engine that would need either fine-tuning the model on training data specific to broadcasting or switching Tesseract for a stronger deep learning OCR solution in a production environment.

## 6.12 Summary of the Results of the Evaluation

The assessment verifies that Visual Sentry fulfills all seven project objectives described in Chapter 1. The OCR pipeline got 95% accuracy, 91.67% precision, and 100% recall with an F1 score of 95.65% on the test dataset with labels. The average processing latency was 172 milliseconds per frame well within what is needed for real-time broadcast monitoring. The GUI remained receptive during long periods of monitoring. The system for

recording forensic evidence accurately recorded proof for all events that set off detection.

The one false positive found during testing, along with flawless recall, shows that a safety-critical alert system should be imbalanced to make it more likely to alert the operator excessively rather than quietly miss a real emergency.

# Chapter 7

# Conclusion

When the project started, the question was, can a standard laptop, no internet, no graphics card, no cloud subscription, watch a live broadcast and raise an alarm the moment a specific word appears on screen or comes through the audio? Not in theory. In practice, on real hardware, in real time. That was the bar. The answer, after months of design, implementation, and testing, is yes. Visual Sentry does exactly that. It watches video. It listens to audio. It reads text from scrolling tickers and transcribes spoken words simultaneously, compares everything against a user-defined trigger word, and responds in under 200 milliseconds when something matches. No data leaves the machine. No server is contacted. No GPU is needed. The system works entirely on a mid-range processor inside an air-gapped environment where most modern AI tools simply cannot operate. That is not a minor achievement. The entire artificial intelligence industry has spent the last decade moving computation into the cloud, making systems bigger and more powerful at the cost of making them completely dependent on network access. Visual Sentry goes in the opposite direction and proves that the trade-off was never as necessary as it seemed.

The system that came out of this project is a multi-threaded Python desktop application with a clean three-layer separation between the user interface, the AI inference pipelines, and the response mechanism. The vision thread captures video frames, preprocesses them through a grayscale, binarization, and adaptive thresholding pipeline, and feeds the result into Tesseract's LSTM engine. The acoustic thread streams microphone audio in 125-millisecond chunks into VOSK's Kaldi-based decoder. Both threads run in the background while the interface stays completely fluid on the main thread no freezing, no blocking, no unresponsive windows even under sustained inference load. When either pipeline picks up the trigger

word, the system responds immediately on four fronts. A warning dialog appears on screen. A screenshot of the warning dialog frame gets saved to disk. A 5-second H.264 video clip gets written as forensic evidence. Simultaneously, the detected trigger word travels over the serial COM port to a LiFi transmitter ESP32, which encodes it as laser pulses through a 2N2222 switching circuit and fires it to a receiver unit where an LM393 comparator and BPW34 photodiode reconstruct the signal and display the word on a 16×2 LCD a fully wireless-free optical alert chain. The fuzzy matching layer running on top of both pipelines handles the OCR noise that is unavoidable in real broadcast conditions. A Levenshtein similarity threshold of 0.80 catches common character substitution errors, zeros misread as O's, ones confused with I's, threes that look like E's, without flooding the operator with false alarms from unrelated text. The result is a system that is simultaneously tolerant of imperfect recognition and precise enough for operational use. Average per-frame latency was 172 milliseconds, with a worst-case of 247 milliseconds, both well within the window needed for broadcast monitoring, where content persists on screen for seconds at a time. The GUI remained fully responsive across a continuous 30-minute stress test. The forensic recorder produced correctly formatted, playable evidence files for all ten triggered test events. These are not simulated results or theoretical benchmarks. They were produced by the actual deployed system running on an AMD Ryzen 5 7430U laptop with 8 GB of RAM, processing real broadcast footage, in the exact configuration that would be used in a deployment. The literature review in Chapter 2 identified four gaps that no existing system had filled together: offline broadcast monitoring, dual-modal late fusion for keyword detection, semantic event-driven forensic recording, and physical air-gap alerting without wireless interfaces. Visual Sentry fills all four in a single

integrated deployment. That combination is genuinely new. Individual components like Tesseract, VOSK, PySerial, and laser-based optical communication exist separately. Bringing them together into one coherent, air-gap-safe, CPU-only application with a working hardware alert chain and event-driven evidence capture has not appeared in the open literature, and the project has demonstrated that it works. The comparison table in Chapter 6 makes the point visually. Standard CCTV systems see pixels but understand nothing. Cloud AI platforms understand content but require internet access that air-gapped facilities cannot provide. Visual Sentry understands content and requires nothing outside the machine it runs on. It occupies a space the market has largely ignored because cloud dependency has been treated as inevitable, and this project shows that assumption is wrong.

No project this size ends without honest limitations, and pretending otherwise would undermine everything the testing chapter established. The acoustic pipeline degrades when broadcast audio reaches the microphone through room speakers rather than a direct line-in connection. Acoustic echo cancellation was not implemented and would make a meaningful difference in that configuration. The test dataset, at 20 frames, is enough to demonstrate the system's capability but not enough to produce narrow confidence intervals on the reported metrics. A production-quality evaluation would need hundreds of labeled frames across many more channels and trigger words. Single-source monitoring is a real constraint too. The current architecture handles one video feed and one microphone simultaneously. Scaling to multiple parallel feeds would require multiple thread pairs and a more sophisticated alert aggregation layer that was outside the scope of this final-year project. And Tesseract, despite performing well on the test data, remains vulnerable to highly stylized fonts and extreme

motion blur situations where a neural scene-text detector like CRAFT or DBNet would perform more reliably.

The architecture that Visual Sentry is built on is genuinely extensible. The most impactful near-term improvement would be replacing Tesseract with a lightweight scene-text detection model something like the MobileNet-based CRAFT detector which would dramatically improve accuracy on stylized and low-contrast broadcast text while still running on CPU. Adding acoustic echo cancellation through the WebRTC noise suppression library would fix the speaker-monitoring limitation without requiring a GPU. Extending the threading model to support multiple simultaneous video sources would open up multi-channel broadcast monitoring from a single workstation. Visual Sentry started as a question about whether intelligent surveillance could be done without the cloud, and it ended as a working proof that it can. The system monitors two modalities at once, preserves evidence automatically, alerts a human operator both on-screen and physically through hardware, and does all of this in real time on hardware that already exists in almost every secure facility on earth. The 172-millisecond average latency means the system responds faster than a human operator could notice the content change. The zero-network architecture means sensitive footage never leaves the room it was recorded in. The LiFi optical channel takes that physical alerting one step further no wire, no radio, just light carrying the exact trigger word to an operator display in a completely air-gap-safe manner. That combination intelligent, fast, private, and physically alerting is what the project set out to build. It is what was built.

# References


[1] Gary Bradski. The OpenCV library. *Dr. Dobb's Journal of Software Tools*, 25(11):120–125, 2000.

[2] Huu-Thanh Duong, Viet-Tuan Le, and Vinh Truong Hoang. Deep learning-based anomaly detection in video surveillance: A survey. *Sensors*, 23(11):5024, 2023.

[3] Ray Smith. An overview of the Tesseract OCR engine. In *Proc. 9th Int. Conf. Document Analysis and Recognition (ICDAR)*, volume 2, pages 629–633, 2007.

[4] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[5] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In *Proc. 23rd Int. Conf. Machine Learning (ICML)*, pages 369–376, 2006.

[6] Xiao-Feng Wang, Zhi-Huang He, Kai Wang, Yi-Fan Wang, Lu Zou, and Zhi-Ze Wu. A survey of text detection and recognition algorithms based on deep learning technology. *Neurocomputing*, 556:126702, 2023.

[7] Chenxia Li, Weiwei Liu, Ruoyu Guo, Xiaoyue Yin, Kaitao Jiang, Yongkun Du, Yuning Du, Lingfeng Zhu, Baohua Lai, Xiaoguang

Hu, Dianhai Yu, and Yanjun Ma. PP-OCRv3: More attempts for the improvement of ultra lightweight OCR system. *arXiv preprint arXiv:2206.03001*, 2022.

[8] Kabeh Mohsenzadegan, Vahid Tavakkoli, and Kyandoghere Kyamakya. A smart visual sensing concept involving deep learning for a robust optical character recognition under hard real-world conditions. *Sensors*, 22(16):6025, 2022.

[9] Nobuyuki Otsu. A threshold selection method from gray-level histograms. *IEEE Trans. Systems, Man, and Cybernetics*, SMC-9(1):62–66, 1979.

[10] Derek Bradley and Gerhard Roth. Adaptive thresholding using the integral image. *Journal of Graphics Tools*, 12(2):13–21, 2007.

[11] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukáš Burget, Ondřej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlíček, Yanmin Qian, Petr Schwarz, Jan Silovský, Georg Stemmer, and Karel Veselý. The Kaldi speech recognition toolkit. In *Proc. IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*, Hawaii, USA, 2011.

[12] Mehryar Mohri, Fernando Pereira, and Michael Riley. Weighted finite-state transducers in speech recognition. *Computer Speech & Language*, 16(1):69–88, 2002.

[13] Rohit Prabhavalkar, Takaaki Hori, Tara N. Sainath, Ralf Schlüter, and Shinji Watanabe. End-to-end speech recognition: A survey. *IEEE/ACM Trans. Audio, Speech, and Language Processing*, 32:325–351, 2024.

[14] Alexei Baevski, Henry Zhou, Abdelrahman Mohamed, and Michael Auli. wav2vec 2.0: A framework for self-supervised learning of speech representations. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 12449–12460, 2020.

[15] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. In *Proc. 40th Int. Conf. Machine Learning (ICML)*, pages 28492–28518, 2023.

[16] Pradeep K. Atrey, M. Anwar Hossain, Abdulmotaleb El Saddik, and Mohan S. Kankanhalli. Multimodal fusion for multimedia analysis: A survey. *Multimedia Systems*, 16(6):345–379, 2010.

[17] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[18] Mordechai Guri, Gabi Kedma, Assaf Kachlon, and Yuval Elovici. AirHopper: Bridging the air-gap between isolated networks and mobile phones using radio frequencies. In *Proc. 9th Int. Conf. Malicious and Unwanted Software (MALWARE)*, pages 58–67, 2014.

[19] Mordechai Guri. Air-gap electromagnetic covert channel. *IEEE Trans. Dependable and Secure Computing*, 21(4):2127–2144, 2024.

[20] Mordechai Guri. POWER-SUPPLaY: Leaking sensitive data from air-gapped, audio-gapped systems by turning the power supplies into speakers. *IEEE Trans. Dependable and Secure Computing*, 20(1):313–330, 2023.

[21] Mordechai Guri. AIR-FI: Leaking data from air-gapped computers using Wi-Fi frequencies. *IEEE Trans. Dependable and Secure Computing*, 20(3):2547–2564, 2023.

[22] Mordechai Guri. RAMBO: Leaking secrets from air-gap computers by spelling covert radio signals from computer RAM. *arXiv preprint arXiv:2409.02292*, 2024.

[23] M. G. Sarwar Murshed, Christopher Murphy, Daqing Hou, Nazar Khan, Ganesh Ananthanarayanan, and Faraz Hussain. Machine learning at the network edge: A survey. *ACM Computing Surveys*, 54(8):1–37, 2021.

[24] Vittorio Mazzia, Aleem Khaliq, Francesco Salvetti, and Marcello Chiaberge. Real-time apple detection system using embedded systems with hardware accelerators: An edge AI application. *IEEE Access*, 8:9102–9114, 2020.

# Appendix A

# User Manual

This user manual provides step-by-step instructions for installing, configuring, and operating the Visual Sentry system. It is intended for users with basic familiarity with Python and Windows operating systems.

## A.1 System Requirements

Before running Visual Sentry, ensure the following minimum hardware and software requirements are met.

**Hardware Requirements:**

- Processor: AMD Ryzen 5 / Intel Core i5 or higher
- RAM: Minimum 8 GB
- Storage: Minimum 2 GB free disk space
- Camera: Built-in or USB webcam
- Microphone: Built-in or external microphone
- Operating System: Windows 10/11 (64-bit)
- Optional: ESP32 DevKit V1, 16x2 I2C LCD,

## A.2 Software Dependencies

The following software must be installed prior to running the application:

1. Python 3.9 or higher — `python.org`
2. Tesseract OCR Engine 4.0+ — `github.com/UB-Mannheim/tesseract`
3. VOSK Model (`vosk-model-small-en-us-0.15`) — `alphacephei.com/vosk/models`
4. Python libraries via terminal:

```
pip install opencv-python
pip install pytesseract
pip install vosk
pip install pyaudio
pip install pyserial
pip install scikit-learn
```

## A.3 Installation Steps

1. Download the project folder to your local machine.
2. Install Python — check *Add Python to PATH* during setup.
3. Install Tesseract OCR and note the installation path.
4. Open terminal in the project folder and run: `pip install -r requirements.txt`
5. Download and extract the VOSK model into the project root folder.
6. Verify the Tesseract path in `main.py` matches your installation.

## A.4 Running the Application

1. Open terminal in the project folder.
2. Run: `python main.py`
3. The Visual Sentry GUI will open.
4. Enter your trigger word in the configuration panel.
5. Select video source, PSM mode, and confidence threshold.
6. Click **Start Live Detection** to begin monitoring.

## A.5 Connecting ESP32 Hardware (Optional)

## A.5 Connecting Li-Fi Hardware (Optional)

Visual Sentry supports physical alerting through a Li-Fi (Light Fidelity) based optical communication link. This replaces traditional wired serial communication with a point-to-point optical transmission system, preserving the air-gap integrity of the environment.

**Required Hardware Components:**

- Li-Fi Transmitter Module (connected to the host laptop)
- Li-Fi Receiver Module (Photodetector unit)
- ESP32 DevKit V1 Microcontroller
- 16x2 I2C LCD Display

**Hardware Setup Steps:**

1. Connect the Li-Fi Transmitter Module to the host laptop via USB port.
2. Position the Li-Fi Receiver Module within direct line-of-sight of the transmitter. Ensure no physical obstruction between the two modules.
3. Connect the Li-Fi Receiver output to the ESP32 DevKit V1 input pin.
4. Connect the 16x2 I2C LCD Display to the ESP32 via the I2C bus (SDA and SCL pins).
5. Upload the provided firmware file `esp32_lifi_firmware.ino` to the ESP32 using Arduino IDE.

**Operation:**

1. Upon trigger word detection, Visual Sentry encodes the alert payload and transmits it via the Li-Fi Transmitter as optical light pulses.
2. The Li-Fi Receiver captures the light signal and decodes the payload, passing it to the ESP32.
3. The ESP32 parses the received trigger word and simultaneously:
    - Displays the trigger word on the 16x2 LCD via I2C bus
4. After 5000ms, the ESP32 automatically resets to idle state — LCD displays *"System Safe"*, buzzer and LED deactivate.

**Important Note:** The Li-Fi communication link operates exclusively through optical light transmission and does not use any radio frequency, Wi-Fi, Bluetooth, or network interface at any stage. The air-gap integrity of the system is fully preserved. Light signals are physically contained within the room and cannot propagate beyond the immediate physical environment.

## A.6 Forensic Evidence Files

All forensic evidence is saved automatically to the `/evidence` folder upon trigger detection:

- **Video clip:** 5-second MP4 file with timestamp filename.
- **Screenshot:** PNG image of the exact triggering frame.
- Example filename: `evidence_20250506_143022.mp4`

## A.7 Running the Evaluation Script

1. Place test images in `/test_data/positive/` and `/test_data/negative/` folders.
2. Run: `python evaluate_performance.py`
3. Confusion matrix results are displayed in the terminal and saved as `confusion_matrix.png`.

## A.8 Troubleshooting

| **Problem** | **Solution** |
|---|---|
| Tesseract not found | Verify path in main.py matches installation location |
| VOSK model not loading | Confirm folder name is exactly vosk-model-small-en-us-0.15 |
| PyAudio install fails | Use: pipwin install pyaudio |
| ESP32 not detected | Install CP210x USB driver from silabs.com |
| No text in OCR log | Check camera in Device Manager. Try Video Source index 1 |

Table A.1: Troubleshooting Guide