Vision-Based GUI Test Infrastructure Generation for Industrial HMIs

Gökhan Tontul
Siemens A.Ş.
Istanbul, Turkey
gokhan.tontul@siemens.com

Ömer Korçak
Marmara University
Istanbul, Turkey
omer.korcak@marmara.edu.tr

***Abstract*—Testing Graphical User Interfaces (GUIs) in embedded industrial Human-Machine Interface (HMI) systems is challenging because these closed-source platforms lack source-code access and platform-specific Application Programming Interfaces (APIs) required by conventional GUI test automation. Consequently, vision-based testing, which operates directly on rendered screen content, offers a practical alternative. While existing vision-based GUI testing approaches primarily focus on GUI element detection, they generally stop at detection and do not provide an end-to-end workflow that transforms detected GUI elements into reusable Page Object Models (POMs) and executable test artifacts. This paper presents an end-to-end framework that combines visual GUI component detection, automated JavaScript Object Notation (JSON)-based POM generation, type-specific component abstractions, and hardware-in-the-loop test execution. The proposed layered architecture transforms detection results into reusable page representations that decouple test logic from low-level operations such as Optical Character Recognition (OCR)-based text extraction, viewport scrolling, coordinate management, and input simulation. The visual detector was evaluated on a held-out test set of 33 previously unseen real-device screenshots containing 346 annotated component instances, achieving a precision of 0.921, recall of 0.901, mAP@50 of 0.970, and mAP@50–95 of 0.910. The framework was also evaluated on real industrial HMI device configurations, where it generated reusable POMs and supported repeated execution of industrial GUI test scenarios. Across 60 full-suite executions of 18 page-specific test scenarios, 1079 of 1080 individual test executions completed successfully. The results provide evidence that the proposed framework can support practical, reusable, and maintainable GUI test automation in the evaluated embedded industrial HMI setting.**

***Index Terms*—GUI Testing, Industrial HMI, Object Detection, YOLO, Vision-based Testing, Page Object Model, Test Automation**

I. Introduction

A. Background and Motivation

Graphical User Interfaces (GUIs) serve as the primary interaction point between users and software systems. Ensuring the quality and reliability of GUIs through automated testing is an important requirement; however, manual GUI testing remains time-consuming and error-prone, and it struggles to scale with increasing application complexity [1], [2], [3]. The rapid evolution of mobile and web applications, together with the reliability requirements of industrial HMI interfaces, has further increased the complexity of GUI testing and strengthened the need for more advanced testing methodologies [1], [3].

Traditional GUI testing approaches can be categorized into manual and script-based testing, model-based and event-driven testing, and image-based or vision-based testing [3]. Manual and script-based approaches have limitations in scalability, repeatability, and maintenance, while approaches that rely on internal or platform-specific representations may diverge from the interface as it is actually rendered on screen [1], [2], [3]. Model-based approaches reason about GUI behavior through abstract models, but their practical use depends on the availability or construction of suitable models [3], [4], [5]. Therefore, image-based and vision-based testing approaches, which operate directly on rendered screen content, provide a suitable alternative for systems where access to internal representations is limited [1], [2], [3].

Human-Machine Interface (HMI) panels are primary interfaces through which operators monitor, control, and configure industrial processes. These interfaces present real-time process states, alarm information, and configuration options to operators. Modern industrial HMIs may contain domain-specific components, configuration options, and interactive control flows [5]. In addition, GUI layout and appearance may vary with screen size, resolution, scaling, and device configuration, which makes validation across relevant deployment conditions an important concern for HMI test automation [1], [2], [6]. Because these factors are ultimately reflected in the rendered interface observed by the user, vision-based GUI testing provides a suitable basis for HMI test automation [1], [2], [3].

However, the challenge in industrial HMI test automation is not limited to detecting visual interface elements. The information extracted from HMI screens must also be connected to operator interactions, configuration behavior, and application-specific test logic in order to support executable test scenarios [5]. In addition, manual testing of industrial HMIs remains labor-intensive, error-prone, and difficult to scale with the growing complexity of modern industrial HMI interfaces [5]. This suggests a practical gap in how GUI information obtained from rendered screens can be transformed into structures that are directly usable, maintainable, and reusable in automated test scenarios. This work addresses this gap by transforming

This work is supported by Siemens.

visual information from industrial HMI screens into structured representations for test automation.

To make vision-based HMI testing executable, the visual information extracted from the screen must be represented in a form that test code can use. Page Object Models (POMs) provide such an abstraction by wrapping interface details behind application-specific methods. As Fowler [7] explains, a page object allows test code to manipulate page elements without directly handling the underlying widget details. Leotta et al. [8] demonstrated that the Page Object pattern can improve test-suite maintainability in an industrial case study. In the present context, preparing such page representations involves identifying visible components, preserving their locations and types, connecting them to interaction methods, and validating that the resulting representation supports correct test execution.

For industrial panels with diverse configuration modules, automating this transformation from visual observations into reusable POMs can help reduce repeated representation-preparation effort and provide a stable basis for component-level test automation. The present work follows this direction by combining vision-based detection, JSON-based POM generation, and type-specific component abstractions that expose detected GUI elements through reusable component classes for industrial HMI testing.

B. Research Questions

This study addresses the following research questions:

RQ1: Can visual detections from embedded industrial HMI screenshots be transformed into GUI component representations that are usable for POM generation and test automation?

RQ2: What practical benefits does framework-supported test execution provide compared with manual GUI test execution in an industrial HMI configuration scenario?

RQ3: Can automated POM generation and type-specific component abstractions create reusable page representations while decoupling test code from low-level GUI interaction logic?

C. Approach

Recent advances in GUI element detection provide a basis for transforming rendered interface screenshots into structured GUI component representations. Chen et al. [9] characterized GUI element detection as a domain-specific object detection problem and showed that learning-based detectors can improve over traditional image-processing approaches, while still requiring attention to GUI-specific characteristics. Building on this direction, we develop a vision-based framework that:

1) **Analyzes** embedded industrial HMI screenshots using a fine-tuned YOLOv8n detector as the visual detection component [10]
2) **Detects and classifies** nine component types (BUTTON, CHECKBOX, COMBOBOX, RADIOBUTTON, SLIDER, TEXT, TEXTBOX, TABLE, INFOBOX)
3) **Transforms** detection results into schema-validated JSON-based POMs
4) **Maps** generated POMs to reusable type-specific component classes for automated test execution

D. Related Work

1) From Internal Representations to Vision-Based GUI Testing: GUI test automation has traditionally relied on internal or platform-provided representations of the interface, including source code, behavioral models, DOM structures, accessibility APIs, or widget metadata [3], [4], [11]. These approaches remain effective when such representations are stable and available, but they become difficult to apply when access to the underlying interface representation is limited or unreliable. Early visual automation systems such as Sikuli [12] demonstrated that GUI interaction can be driven directly from screenshots, avoiding dependence on source code or platform-specific identifiers. Recent surveys describe this broader shift from code- and layout-based testing toward vision-based techniques that operate on the interface as seen by the user [1], [2], [3]. This transition provides the foundation for industrial HMI testing, where automation may need to rely on rendered screen content rather than software instrumentation.

2) Vision-Based GUI Understanding and Element Detection: Once testing relies on rendered screen content, a central technical requirement is to identify and classify the visible interface elements. Chen et al. [9] established GUI element detection as a domain-specific computer vision problem and showed that deep learning-based approaches can improve over traditional image processing methods, while still requiring attention to GUI-specific visual characteristics. White et al. [13] demonstrated that image-based widget detection can guide automated GUI exploration, while Khaliq et al. [14] examined deep learning-based UI detection in relation to functional testing workflows. More recent work has further explored CNN-based real-time mobile GUI detection [15] and YOLO-based GUI element detection, with Daneshvar and Wang [16] reporting strong performance across modern YOLO variants. These studies show that visual detection can provide useful mechanisms for locating and classifying GUI components, but they do not by themselves provide reusable page representations or component-level test interfaces. For industrial HMI testing, detection is only an intermediate step: the detected elements must be transformed into reusable page models, type-specific component abstractions, and executable test interfaces.

3) Cross-Device and Platform-Independent Visual Testing: Vision-based testing has also been investigated as a way to support non-intrusive execution across heterogeneous devices and platforms. Qian et al. [17] presented RoScript, a visual script-driven robotic testing system for touch-screen applications, and Xie et al. [6] proposed NiCro for cross-device and cross-platform GUI testing through visually matched widgets. Yu et al. [18] further explored platform-independent GUI testing by combining image embedding with reinforcement learning to abstract GUI states and widgets from rendered interfaces. These studies indicate that visual interaction can improve portability when instrumentation is unavailable or unreliable. However, their primary focus is on cross-device replay, state exploration, or platform-independent execution, rather than on converting vision-derived GUI information

into reusable Page Object Models and component-level test infrastructure for industrial HMI panels.

4) Test Abstraction, Maintainability, and Industrial Applicability: At the test design level, abstraction is central to maintainable GUI automation. Fowler [7] introduced the Page Object pattern to separate test intent from interface-specific details, and Leotta et al. [8] showed in an industrial case study that this pattern can substantially reduce maintenance effort. Industrial evidence further shows that GUI test automation remains affected by persistent challenges such as fragile execution, widget identification, script maintenance, and integration into real workflows [5], [11], [19]. These findings are especially relevant for industrial HMI panels, where tests must remain usable across evolving interfaces, device-specific execution conditions, and industrial workflows. The present work builds on the Page Object principle but differs from conventional usage in a key respect: rather than assuming manually written page objects, it derives structured page representations from screenshots and connects them to reusable, type-specific component abstractions.

Table I summarizes the positioning of the proposed framework against representative lines of related work.

The novelty of the proposed approach therefore does not lie in introducing a new object detector or a new POM concept individually. Rather, it lies in operationalizing the transformation from visually detected GUI elements into persistent page representations and type-specific component abstractions that can be used directly in closed-source industrial HMI test execution.

E. Research Gap and Contributions

The reviewed literature shows that existing work can detect or visually interact with GUI elements, but less often addresses how vision-derived GUI information can be converted into reusable POMs and component-level test infrastructure for industrial HMI panels. This leaves several limitations that hinder practical adoption in industrial contexts:

1) **Lack of End-to-End Integration:** Existing studies often address GUI element detection, visual interaction, replay, or test generation as separate concerns, rather than integrating them into a complete workflow from rendered screen analysis to executable test infrastructure [3], [5].
2) **Insufficient Abstraction:** Visual GUI testing approaches may still require test logic to remain close to visual anchors, coordinates, tool-specific actions, OCR calls, or other low-level interaction mechanisms, increasing implementation and maintenance effort [1], [2], [11].
3) **Limited Evidence on Detector-to-Test Integration:** Although YOLO-based models have demonstrated strong performance in GUI element detection, limited evidence is available on adapting lightweight detectors to specialized industrial HMI interfaces using small, domain-specific annotated datasets and integrating the resulting detections into practical test automation workflows [3], [16].

To address these limitations, this paper presents an end-to-end vision-based GUI testing framework evaluated on real industrial HMI device configurations. The main contributions of this paper are as follows:

- **End-to-End Test Automation Framework:** We present an end-to-end vision-based GUI testing framework that automates the workflow—from screenshot capture through object detection and POM generation to test execution—without requiring access to source code or platform APIs. The framework's layered architecture produces structured, machine-readable test artifacts for industrial GUI test automation.
- **Multi-Layer Abstraction Architecture:** We propose a layered abstraction architecture combining JSON-based Page Object Models (POMs) with type-specific component classes that encapsulate low-level interactions, enabling test developers to write concise, maintainable test code without knowledge of coordinates, OCR, viewport management, or input simulation.
- **Detection-to-Component Abstraction Mapping:** We define a mapping from detected GUI elements to reusable, type-specific component objects, allowing test code to operate on semantically named GUI components rather than raw bounding boxes, coordinates, or OCR outputs.

II. Methodology

A. Framework Overview

Our methodology is implemented in a modular framework designed to automate the framework-supported workflow from visual input to test execution. The framework is structured as a multi-stage pipeline that progressively transforms raw screen captures into high-level, executable test artifacts. This layered approach separates the core concerns of detection, modeling, and abstraction, supporting both maintainability and extensibility. Fig. 1 illustrates the end-to-end architecture of the framework.

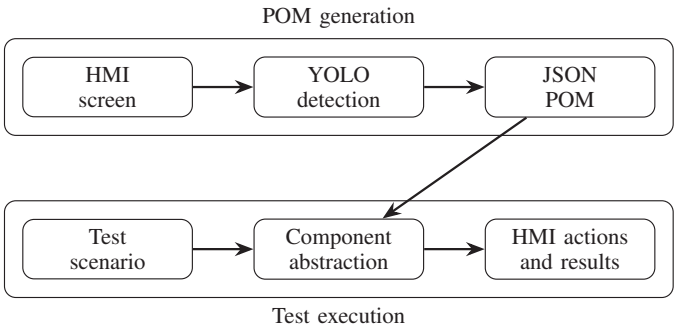


Fig. 1. Simplified two-phase workflow of the proposed framework, converting rendered HMI screens into reusable JSON POMs and using them through component abstractions to execute high-level test scenarios on target HMI configurations

The process begins with capturing a screenshot from the target industrial HMI configuration. This image serves as

TABLE I. CONCEPTUAL POSITIONING OF THE PROPOSED FRAMEWORK RELATIVE TO REPRESENTATIVE RELATED-WORK DIRECTIONS

Approach	Primary focus	Page abstraction	Test execution role
Visual scripting and replay [12], [17]	Visual search/action	Image anchors	Scripted actions
Cross-device visual testing [6], [18]	Platform portability	Matched visual states/widgets	Automated execution
GUI element detection [9], [13], [16]	Localization/classification	Detector outputs	Testing support
Conventional Page Objects [7], [8]	Maintainability	Manual page objects	Executable tests
Proposed framework	HMI test infrastructure	Generated page representations	Typed component tests

the input to our detection layer, which employs a fine-tuned YOLOv8n model to identify and classify GUI components. The detection results are then structured into a JSON-based Page Object Model (POM), creating a machine-readable representation of the GUI. This POM is subsequently transformed by the abstraction layer into type-specific component classes, which are finally used by pytest scripts to execute test scenarios. Each of these stages is detailed in the following sections.

B. Detection Layer: YOLOv8n with Transfer Learning

The core of our framework's detection capability is a YOLOv8n object detection model [10] fine-tuned for the specific domain of industrial HMI components. This section details the model selection, training methodology, and the process of transforming raw detections into structured data.

1) Model Selection: The nano variant of YOLOv8 was selected because it offers an appropriate balance between inference speed and detection accuracy, making it suitable for the present hardware-in-the-loop testing context. As a single-stage detector, YOLOv8 performs localization and classification simultaneously, following the broader YOLO design principle of unified real-time object detection [20]. With approximately 3.2 million parameters and a model size of about 6 MB, YOLOv8n provides a lightweight solution that remains suitable for the downstream POM generation task in the evaluated workflow.

2) Training with Transfer Learning: To train the model for this domain-specific task, a transfer learning strategy was adopted. Rather than training a detector from scratch, the training process was initialized from standard YOLOv8n weights pre-trained on the COCO dataset [21]. This starting point provides the model with general visual priors, including edges, shapes, and textures, before adaptation to the industrial GUI domain.

The model was then fine-tuned on a manually annotated dataset of 46 industrial HMI screenshots, split into 37 training and 9 validation images, with a total of 451 labeled component instances covering the nine component types. Training was performed for 200 epochs with an input size of 640×640 and a batch size of 16. In addition, 33 previously unseen real-device screenshots were independently annotated after model development and used exclusively as a held-out test set for quantifying detection performance across 346 component instances. Table II reports per-class metrics on the validation set, and Table III reports the corresponding metrics on the independent test set. Overall, the fine-tuned detector provides structured component detections that are then consumed by the POM generation layer.

TABLE II. VALIDATION-SET GUI COMPONENT DETECTION METRICS FOR THE FINE-TUNED YOLOV8N MODEL

Class	Precision	Recall	mAP@50	mAP@50–95
All	0.865	0.838	0.863	0.623
TEXT	0.906	0.857	0.860	0.552
BUTTON	0.731	0.250	0.257	0.170
COMBOBOX	0.922	1.000	0.995	0.895
TEXTBOX	1.000	0.949	0.995	0.859
CHECKBOX	0.955	0.857	0.978	0.795
RADIOBUTTON	1.000	0.788	0.995	0.467
SLIDER	0.476	1.000	0.828	0.352
INFOBOX	0.933	1.000	0.995	0.895

TABLE III. INDEPENDENT TEST-SET GUI COMPONENT DETECTION METRICS (33 PREVIOUSLY UNSEEN REAL-DEVICE SCREENSHOTS, 346 INSTANCES)

Class	Precision	Recall	mAP@50	mAP@50–95
All	0.921	0.901	0.970	0.910
TEXT	0.906	0.866	0.911	0.752
BUTTON	0.914	1.000	0.994	0.981
TABLE	0.782	1.000	0.995	0.995
COMBOBOX	0.978	0.778	0.925	0.866
TEXTBOX	0.951	0.981	0.991	0.906
CHECKBOX	1.000	0.790	0.974	0.896
RADIOBUTTON	0.835	0.841	0.972	0.914
INFOBOX	1.000	0.951	0.995	0.971

The reported precision, recall, and mAP values evaluate component classification and bounding-box localization jointly under standard object-detection criteria. On the 33-image independent test set, which contains 346 component instances, the detector attained an overall precision of 0.92, recall of 0.90, mAP@50 of 0.97, and mAP@50–95 of 0.91. These results suggest that the fine-tuned model retained high detection performance on unseen real-device screens within the evaluated HMI domain. Classes with no instances in an evaluation split are omitted from the corresponding per-class table; therefore, TABLE is not listed for validation and SLIDER is not listed for the independent test set. The validation and independent test metrics should not be directly compared as measures of relative dataset difficulty because the two splits differ in their component-class composition and sample distribution. Because both the validation and independent test sets are modest in size, these detection scores should be interpreted as evidence for the detector used in the framework pipeline rather than as a broad benchmark of industrial HMI GUI detection accuracy.

C. POM Generation Layer

Following component detection, the framework converts the extracted GUI elements into a JSON-based Page Object Model (POM). Rather than storing only a transient list of detections, this layer creates a persistent, machine-readable page representation that can be reused during automated test execution. The generated POM preserves page-level metadata together with component identities, types, and device-specific screen coordinates.

1) POM Construction Workflow: POM generation is performed through an iterative acquisition process. After the target page is opened, the framework first captures the visible region of the interface, then applies component detection, and appends the resulting elements to the page representation. For pages extending beyond a single screen, the framework scrolls through the interface and repeats the same procedure until no new components are discovered. In this way, a single POM can represent configuration pages that extend across multiple vertically scrollable views.

Before serialization, the detection results undergo limited post-processing. In this stage, low-confidence results are filtered out, OCR support is used where needed to improve component naming, and proximity relationships on the screen are used to make some elements more meaningful. The resulting elements are then organized into a regular structure and saved in JSON format.

2) JSON-Based POM Structure: The resulting POM stores the page and its components in a structured format. This design not only preserves the detection results, but also provides a reusable interface description during test execution. As a result, the visually perceived interface is transformed into an intermediate layer that can be directly consumed by the test infrastructure.

Listing 1 shows an abbreviated excerpt from the JSON POM generated for the Network Settings page. Each page records page-level metadata together with a list of widgets, where every widget stores its index, name, component type, and resolution-specific bounding box. The `pageSize` height (2116) exceeding the screen height (800) reflects that the page spans multiple scrolled viewports.

Unlike temporary detector outputs, the POM provides a more stable interface description that can be validated against a predefined schema before use. It therefore serves as a bridge between the detection and execution layers. Although the current structure is intentionally kept lightweight, it provides an organized context for reusable component-level test execution.

```
{
  "pageIndex": 512, "device": "ARM",
  "pageName": "fw_NetworkSettings",
  "pageSize": [
    { "resolution": "1280,800", "orientation": "landscape",
      "w": 1280, "h": 2116 }],
  "widgets": [
    { "index": 1, "name": "deviceName",
      "type": "TEXTBOX",
      "panel": [{ "resolution": "1280,800",
        "x": 617, "y": 251, "w": 364, "h": 45 }]
    },
    { "index": 4, "name": "obtainAnIpAddressViaDhcp",
      "type": "RADIOBUTTON",
      "panel": [{ "resolution": "1280,800",
        "x": 615, "y": 537, "w": 30, "h": 32 }]
    },
    /* ... 25 more widgets ... */
  ]
}
```

Listing 1. Abbreviated JSON POM excerpt for the Network Settings page

D. Abstraction Layer

After POM generation, the framework transforms the structured page description into component objects that can be used directly in test code. This layer serves as the principal link between visual analysis and test execution. While the detection layer identifies interface elements and the POM layer preserves them as widget records, the abstraction layer maps these records to reusable component classes. As a result, the framework output is elevated from a set of detected screen regions to an executable interface that can be used by the test automation workflow. Fig. 2 shows this simplified mapping from POM entries to representative component abstractions.

1) Component-Based Access: The framework exposes interface elements through component-based objects. This design allows test scenarios to interact with the GUI not as a collection of raw visual regions, but as a set of meaningful software entities. Test logic can therefore be expressed in terms of intended operations rather than low-level interaction details, improving readability and supporting reuse across different pages and scenarios.

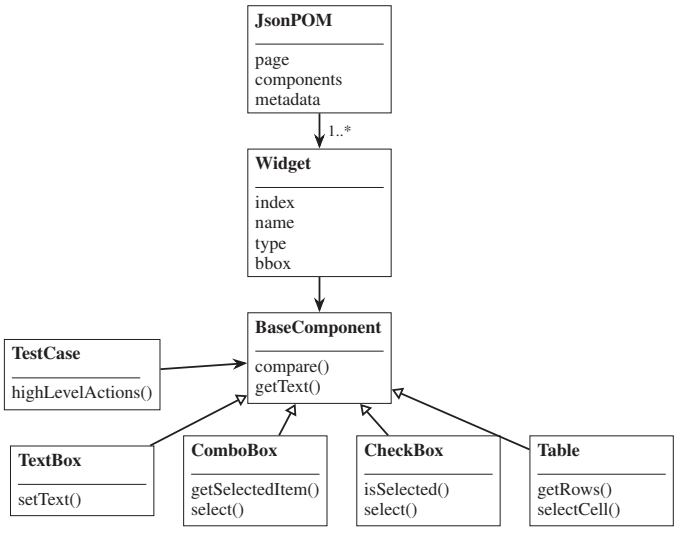


Fig. 2. Simplified component abstraction model, mapping widget records from the generated JSON POM to reusable component classes that allow test cases to use high-level operations instead of low-level GUI interaction details

2) Type Validation and Consistency: The abstraction layer also provides an early validation mechanism against incorrect component usage. When a component object is instantiated, the framework verifies that the referenced interface element matches the expected type. In this way, consistency is preserved between the visual representation stored in the POM

and the operations applied by the test layer, helping detect invalid assumptions before execution proceeds.

3) Encapsulation of Low-Level Interactions: The primary contribution of this layer is the separation of low-level GUI interaction logic from test code. In practice, operations such as calculating screen coordinates, scrolling the page upward or downward to bring an element into view, selecting the correct target element, sending keyboard input, and extracting text via OCR can easily make test scripts fragile and difficult to maintain. The abstraction layer encapsulates these concerns within component behavior and allows the test developer to work with higher-level operations instead.

This benefit becomes particularly evident for components requiring multi-step interaction. Entering data into a text field, retrieving the available options of a drop-down list, or selecting a specific entry within a table may appear as a single logical operation from the perspective of the test case, while internally involving multiple auxiliary actions. By centralizing this recurring interaction logic, the framework reduces duplication and improves maintainability.

4) Contribution to Test Execution: The abstraction layer is the key structure that converts the outputs of detection and POM generation into an executable testing interface. Accordingly, the contribution of the framework is not limited to identifying GUI elements, but extends to enabling reusable automated interactions over them. While the POM serves as an intermediate representation, the abstraction layer transforms that representation into a directly usable execution interface for automated tests. This conversion constitutes a central part of the engineering contribution of the paper.

E. Framework-Supported Test Execution

The practical value of the framework becomes apparent during test execution, where the generated POMs and component abstractions are used to drive interactions on the evaluated HMI configurations. At this stage, the framework not only provides a static representation of the GUI, but also assists the tester by preparing the execution context, opening the relevant page, coordinating the required interactions, and supporting runtime verification and failure analysis. As a result, test scenarios can focus on the intended behavioral checks rather than device-dependent operational details.

1) Execution Preparation: Before a test scenario is executed, the framework establishes the appropriate execution context by resolving device-specific properties such as panel type, screen resolution, and orientation, and by loading the corresponding page representation. The required page is then opened through the framework's built-in navigation logic. This preparation reduces repetitive setup effort from the test scenario and helps provide a more consistent execution environment across supported panel configurations.

2) Assisted Runtime Interaction: During execution, interactions are expressed through the abstraction layer, while the underlying operational steps are handled by the framework in the background. These steps include locating the relevant element, bringing it into view when necessary, handling coordinate-dependent interaction details, and executing actions such as clicking or text input. In particular, when a target element is not immediately visible, the framework manages upward or downward page scrolling as part of the interaction process. This design allows the tester to express actions at a higher level without dealing directly with the mechanics of device interaction.

3) OCR-Supported Runtime Verification: The framework also supports runtime verification through OCR-based reading of rendered interface content, using Tesseract OCR where the configured backend requires it [22]. This allows visible values, selected items, and other textual information displayed on the screen to be checked directly during test execution. In addition, execution can proceed with any OCR option supported by the framework, allowing the tester to continue with the configured OCR backend according to scenario needs. In this way, OCR serves not only as a supporting technique within the framework, but also as a practical mechanism for assertion and validation during runtime.

4) Debugging and Failure Analysis: An important benefit of the execution workflow is the support it provides for traceability and debugging. The framework can capture page-level and component-level images during execution and preserve visual evidence for later inspection. When a failure occurs, these captured images, together with descriptive execution errors, support analysis of whether the problem originates from navigation, visual detection, interaction handling, or the GUI state itself. This improves the inspectability of the test flow and facilitates failure diagnosis in industrial HMI testing scenarios.

III. Experimental Evaluation

This section evaluates the proposed framework through a Network Settings case study conducted on real industrial HMI devices. The evaluation focuses on the same stateful Network Settings workflow on ARM and AMD hardware configurations. The evaluated configurations were an ARM 1280×800 landscape configuration and AMD 1366×768 configurations covering both landscape and portrait orientations. The Network Settings workflow provides a suitable evaluation target for POM generation and component-based test execution because it contains multiple GUI component types, including text fields, radio buttons, checkboxes, comboboxes, buttons, and information boxes. At the same time, it includes settings such as IP, DNS, SNMP, and DCP, which can directly affect device accessibility. Therefore, the evaluation examines both the executable use of the framework and its applicability to a stateful industrial configuration interface.

The experimental scope is intentionally limited to a single functional Network Settings page family and a small number of industrial HMI device configurations. On the AMD configurations, the same Network Settings workflow is represented through split page sections for the evaluated layouts; therefore, this scope should be understood as one functional configuration workflow rather than one identical physical page representation. Accordingly, the results should

not be interpreted as a general benchmark for industrial HMI systems; instead, they provide bounded empirical evidence from the evaluated Network Settings workflow. This scope enables a focused observation of how the framework operates on closed-source, visually observed, and stateful industrial configuration interfaces.

A. POM Generation and Reuse Benefit

This subsection reports the observed output of automated POM generation for the Network Settings workflow and examines the reuse benefit of this output for the test infrastructure. The framework automatically applied YOLO-based component detection to screenshots of the page and transformed the detected GUI elements into schema-compliant JSON POMs. As illustrated in Fig. 3, the fine-tuned model detects multiple component types directly on the rendered HMI screen. The raw detections produced by the model were not exposed to the tester as a manual clean-up task. Instead, the framework performed the required post-processing internally: component types were normalized to the POM schema, OCR-derived component names were adjusted to the POM naming format, long names that could exceed schema constraints were shortened, and duplicate or partially visible detections caused by viewport transitions were filtered during scrolling.

For an ARM Network Settings representation, the entire POM generation is triggered by a concise two-line framework invocation, shown in Listing 2. The first line applies component detection across the scrolled page and serializes the schema-compliant JSON POM, while the second line prepares the corresponding reference images from the generated representation.

```
Fw.helper_autoFillJsonFile(
  PageTree.NetworkAndInternet.NetworkSettings,
    "fw_NetworkSettings")
Fw("fw_NetworkSettings").helper_saveReferenceImages
    ()
```

Listing 2. Two-line framework invocation for generating the JSON POM and reference images for the ARM Network Settings representation

As a result, a reusable JSON POM containing 27 widgets was generated for the ARM Network Settings layout. The same POM was then reused to prepare reference images and support page-specific test authoring, without requiring the page representation to be regenerated for subsequent preparation or execution steps. In the observed ARM run, the combined workflow of POM generation and reference-image preparation completed in 674 seconds on the target test setup. When reference-image preparation was executed using the already generated POM, without regenerating the page representation, the run completed in approximately 117 seconds. These POM-generation timing measurements are reported only for the ARM 1280×800 landscape configuration, where the page is represented as a single vertically scrollable POM. The AMD representations were generated as split POM sections and then manually checked for widget-name consistency and page-section coverage before the repeated execution study. The repeated-execution results in Table IV then report reuse and stability evidence across the evaluated ARM and AMD configurations.

B. Page-Specific Test Authoring and Validation

This subsection evaluates how the generated POM representations were used for page-specific test authoring and how the resulting tests were validated on the evaluated hardware configurations. For the Network Settings page, 18 domain-specific test scenarios were defined. These scenarios covered page-specific operations such as reading and modifying the device name, validating the converted name, comparing the MAC address against the expected value, checking DHCP and static IP options, validating DNS fields through GUI-based set-and-verify steps, checking the mode and speed combobox value, inspecting SNMP-related checkbox behavior, verifying password masking, and reading infobox content through OCR.

The tests were not written directly against screen coordinates or low-level input commands. Instead, they were expressed through the generated POM representations and the framework's type-specific component abstraction layer. Consequently, the test code remained at the page-component level, while low-level interaction details were handled within the framework.

The final page-specific scenario suite and the generic POM smoke suite were evaluated through repeated hardware-in-the-loop executions on the ARM and AMD configurations summarized in Table IV. For these measurements, each table row corresponds to complete-suite executions under the specified hardware and layout condition, and the reported runtime is the average external wall-clock duration across those runs. The generic POM smoke suite completed 30 suite executions without failure across the ARM landscape, AMD landscape, and AMD portrait configurations, producing 270 passed test instances and 80 expected skips. The suite contains 7 generic test functions. On the ARM single-POM representation, each function was instantiated once, producing 7 passed pytest instances per run. On each AMD configuration, the same 7 functions were instantiated over two split POM sections, producing 14 collected pytest instances per run; 10 passed and 4 were skipped because the corresponding widget type was outside the scope of that split POM section. These skips were therefore treated as expected scope skips rather than failures.

The page-specific scenario suite was also executed repeatedly. For each evaluated hardware and orientation condition, 10 consecutive full-suite executions were run. In addition to these 10-run condition-level batches, an extra 30 consecutive robustness executions were run on the ARM landscape configuration. Thus, the ARM landscape condition produced 40 valid full-pass suite executions, corresponding to 720 passed individual test executions. The AMD landscape configuration produced 10/10 full-pass suite executions, while the AMD portrait configuration produced 9/10 successful full-suite executions. In the unsuccessful AMD portrait suite execution, one of the 18 tests, covering the default-gateway scenario, did not complete successfully. The single failure was a timeout associated with a delayed device response rather than an

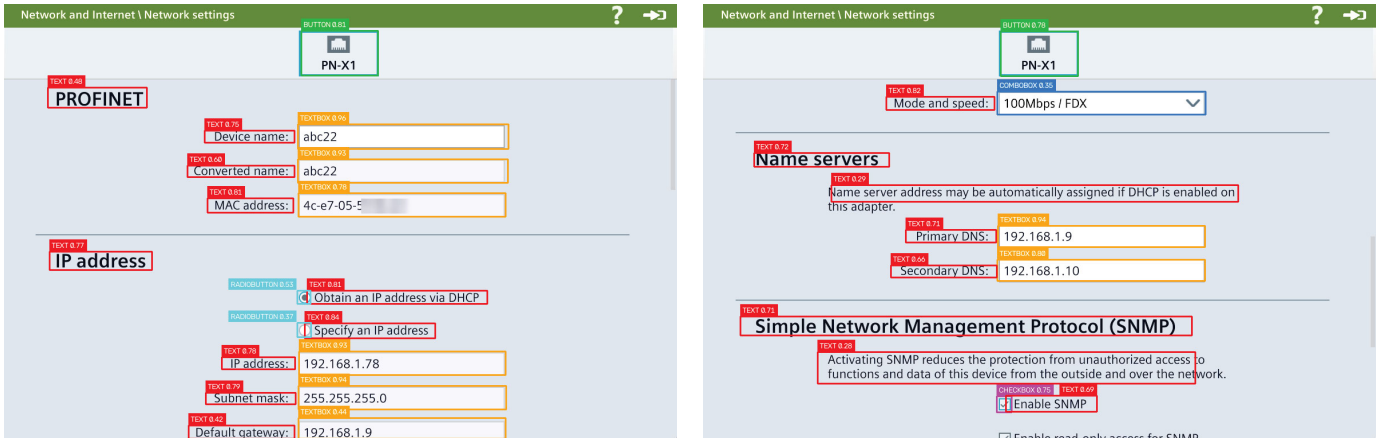


Fig. 3. Qualitative YOLOv8n detection results on the Network Settings page of the HMI panel, with detected regions annotated using the framework's component-type labels and confidence scores: (a) upper region of the page (left), showing `BUTTON`, `TEXT`, `TEXTBOX`, and `RADIOBUTTON` components; (b) lower region of the page (right), showing `BUTTON`, `COMBOBOX`, `TEXT`, `TEXTBOX`, and `CHECKBOX` components

incorrect GUI interaction. Across the page-specific scenario suite executions reported in Table IV, 1079 of 1080 individual test executions passed, corresponding to an observed test-level success rate of 99.91%. These results should not be interpreted as evidence for the absence of flakiness; they characterize the observed stability under the evaluated hardware, orientation, and test conditions.

C. Manual and Framework-Supported Execution Comparison

To evaluate RQ2, the framework-supported execution results were compared with a manual execution baseline for the same 18 Network Settings scenarios. In the observed UI-only manual execution, completing the scenarios directly through the HMI screen required 24 minutes, corresponding to 1.33 minutes per test. This baseline reflects direct GUI operation and visual observation only; it does not include backend state verification. When backend-supported checks were included in the manual workflow, the required execution effort increased to 50 minutes, corresponding to about 3 minutes per test in this evaluation context. For the framework-supported comparison, the ARM landscape configuration provided the directly comparable repeated-execution runtime: the page-specific scenario suite produced 40 full-pass runs on this configuration with an average external runtime of 1062.23 seconds, or approximately 17.7 minutes.

The comparison also shows that the benefit of the framework-supported approach is not limited to a single execution time. For a new screen size or device variant, the additional framework-side effort can primarily involve generating and validating a new or updated JSON POM for the relevant page, whereas manual execution requires the same scenario set to be repeated by a human tester for each new screen size, resolution, or device variant. Once created, the framework-supported test suite can also be reused in scheduled regression runs or pull-request validation workflows. This can enable repeated execution of the same scenario set without requiring the full manual execution effort for each run.

D. Reduction of Test-Side Implementation Burden

This subsection reports the amount of page-specific test code required when low-level GUI interaction mechanisms are handled by the framework. The final page-specific test file contained 319 physical lines and 238 source lines of code. This suggests that the test logic could be expressed through the POM-based component abstraction layer without embedding low-level GUI interaction details in the test code. Listing 3 illustrates this style with a representative page-specific test fragment.

```
def test_set_device_name():
  fw = Fw("fw_NetworkSettings")
  fw.TextBox(
    fw.getIndex("deviceName")).setText(
      EXPECTED_DEVICE_NAME)

  assert (fw.TextBox(
    fw.getIndex("deviceName")).getText()
    == EXPECTED_DEVICE_NAME)
```

Listing 3. Page-specific test code expressed through component abstractions

In this example, the test code only writes a value to a page component, reads the value back, and checks the result. Locating the element, bringing it into view, resolving coordinates, performing input actions, and reading the rendered value through OCR-supported mechanisms are handled by the framework.

Implementing the same test scope without such framework support would require the test developer to build additional support mechanisms for screenshot acquisition, widget-region cropping, OCR preprocessing, coordinate mapping, scrolling, input simulation, component-state extraction, reference-image comparison, and network-state restoration. A repository-local SLOC measurement of the framework and support functions exercised by this 18-test scope identified 1,440 executable source lines of supporting code outside the page-specific test file. This SLOC value reflects reused framework infrastructure rather than a direct measurement of developer productivity.

TABLE IV. REPEATED HARDWARE-IN-THE-LOOP EXECUTION RESULTS FOR THE NETWORK SETTINGS CASE STUDY

Hardware	Layout	Suite	Runs	Outcome	Avg. runtime	Notes
AMD	Portrait, 1366×768	Generic POM smoke	10	100 passed, 40 skipped, 0 failed	144.04 s	7 functions over 2 POMs
AMD	Landscape, 1366×768	Generic POM smoke	10	100 passed, 40 skipped, 0 failed	169.73 s	7 functions over 2 POMs
ARM	Landscape, 1280×800	Generic POM smoke	10	70 passed, 0 failed	281.77 s	7 functions over 1 POM
ARM	Landscape, 1280×800	Page-specific scenario	10+30	720 passed, 0 failed	1062.23 s	Initial batch plus robustness batch
AMD	Landscape, 1366×768	Page-specific scenario	10	180 passed, 0 failed	794.25 s	10/10 full-pass
AMD	Portrait, 1366×768	Page-specific scenario	10	179 passed, 1 failed (timeout)	604.10 s	Device response delay

E. Discussion of Research Questions

With respect to RQ1, the results indicate that visual detections can be transformed by the framework into structured GUI component representations suitable for POM generation. In the evaluated embedded industrial HMI workflow, visually detected elements were represented within a structured and schema-compliant page representation. Therefore, RQ1 is answered positively within the scope of this study; however, this result concerns the applicability of detection outputs to the POM generation pipeline in an embedded industrial HMI context, rather than general YOLO detection performance.

With respect to RQ2, the results indicate that the main benefit of framework-supported execution is not simply accelerating a single manual run, but transforming GUI checks into repeatable regression test assets. Although the single-run timing comparison provides useful context, the manual approach does not provide the same level of support for consistent re-execution, scheduled execution, and reuse across device or screen configurations. The extended repeated execution results strengthen this observation: the generic POM smoke suite completed 30 failure-free suite executions across multiple device/layout conditions, and the page-specific scenario suite produced 59 full-pass runs out of 60 reported suite executions, with one non-passing observation associated with a delayed device response rather than an incorrect GUI action. Therefore, RQ2 is answered by showing that the practical benefit of the framework emerges primarily in repeated validation, regression execution, and reuse scenarios.

With respect to RQ3, the results indicate that the generated POMs and type-specific abstractions can decouple page-specific test code from low-level GUI interaction details. Test scenarios could be expressed through page components and component-level operations instead of directly handling screen coordinates, scrolling, input simulation, OCR, or state-management logic. Therefore, RQ3 is answered positively in the evaluated scenario: the POM provides a reusable page context, while the type-specific abstractions turn this context into a higher-level and more maintainable execution interface for test code.

These findings should therefore be interpreted as bounded evidence from the evaluated Network Settings workflow and device configurations. The workflow provides a suitable hardware-in-the-loop GUI test automation case because it includes stateful configuration fields that can affect device accessibility, but it is not a general benchmark for industrial HMI systems. Broader claims about generalization require additional workflows, larger device families, and more diverse interface configurations.

IV. CONCLUSION

This study shows that visual information obtained from embedded industrial HMI screens does not have to remain as detection output, but can be transformed into POM-based, executable test infrastructure. The proposed framework converts detected GUI components into JSON-based page representations and places a reusable abstraction layer between page-specific test code and low-level screen operations.

The experimental results also indicate that POM generation does not need to be repeated for every test execution. Instead, it is an initial page-modeling step that can be reused during subsequent preparation and execution steps. For the Network Settings page, the reuse of the generated JSON POM containing 27 widgets showed that the same page representation could be used in later preparation steps without regenerating the POM.

Through this structure, manually authored test scenarios can be written in terms of page components and component-level operations instead of coordinate calculation, scrolling, OCR calls, image cropping, or input simulation. The type-specific component abstractions provide a consistent programming model for GUI elements such as text boxes, combo boxes, checkboxes, and tables, allowing test developers to focus on page-specific behavior rather than implementing separate low-level interaction flows for each component type. In the evaluated Network Settings scenario, the 18 tests were expressed in 238 SLOC of page-specific test code, while 1,440 SLOC of lower-level supporting infrastructure were separated from the test writer. These measurements illustrate the separation of page-specific test logic from reusable infrastructure rather than a direct quantification of development-effort reduction, highlighting the framework's contribution to consistent and maintainable test authoring.

From an execution-time and stability perspective, the repeated hardware-in-the-loop study showed that the framework-supported scenario suite could be reused across the evaluated ARM and AMD configurations. The ARM landscape configuration provided the strongest robustness evidence through an initial 10-run batch followed by an additional 30-run batch, while the AMD landscape and portrait configurations provided additional cross-layout execution evidence. The only non-passing observation was associated with delayed device response timing in the default-gateway scenario rather than an incorrect GUI action.

These results indicate that the practical value of the framework lies not only in the observed execution-time comparison, but also in reducing repeated manual interaction effort and producing repeatable and maintainable regression assets. In a manual approach, the same scenario set must be repeated by a human tester for each device, screen size, or variant, whereas the framework-supported approach allows the same test logic to be reused with new or updated POMs. The reported comparison concerns execution time and observed test stability under the evaluated conditions, and does not include the initial costs of dataset annotation, model training, framework development, POM validation, or test authoring.

Nevertheless, the evaluation remains limited to the Network Settings workflow and a small number of HMI device configurations. Additional studies on broader device families, different HMI pages, and larger independently labeled test sets are needed to further characterize the scope and limits of the approach. Future work may also investigate how the machine-readable GUI context provided by the generated POMs and type-specific abstractions can support future test-generation and maintenance workflows.

Taken together, the study demonstrates the feasibility of combining vision-based GUI detection, POM generation, and component-level abstraction to provide reusable and maintainable test infrastructure for the evaluated industrial HMI setting.

ACKNOWLEDGMENT

We extend our gratitude to Siemens Sanayi ve Tic. A.S. for their invaluable support.

REFERENCES

[1] S. Yu, C. Fang, Z. Tuo, Q. Zhang, C. Chen, Z. Chen, and Z. Su, "Vision-based mobile app GUI testing: A survey," *ACM Computing Surveys*, vol. 58, no. 6, pp. 1–46, 2025.

[2] M. A. Chaikovskyi and E. V. Malakhov, "Vision-based GUI testing: A systematic review of computer vision advances in software quality assurance," *Applied Aspects of Information Technology*, vol. 8, no. 4, pp. 397–423, 2025.

[3] A. A. Qassem, R. Bryce, and K. Alkhaldi, "A survey on visual GUI testing for automatic test case generation: Tools, AI techniques, and emerging trends," in *Proc. Int. Conf. Softw. Eng. and Knowledge Eng. (SEKE)*, 2025, pp. 108–113.

[4] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 2007.

[5] R. Ramler, G. Buchgeher, and C. Klammer, "Adapting automated test generation to GUI testing of industry applications," *Information and Software Technology*, vol. 93, pp. 248–263, 2018.

[6] M. Xie, J. Ye, Z. Xing, and L. Ma, "NiCro: Purely vision-based, non-intrusive cross-device and cross-platform GUI testing," *arXiv preprint arXiv:2305.14611*, 2023.

[7] M. Fowler, "Page object," https://martinfowler.com/bliki/PageObject.html, Sep. 2013, accessed: 2026-08-05.

[8] M. Leotta, D. Clerissi, F. Ricca, and C. Spadaro, "Improving test suites maintainability with the page object pattern: An industrial case study," in *Proc. IEEE Int. Conf. Softw. Testing, Verification and Validation Workshops (ICSTW)*, 2013, pp. 108–113.

[9] J. Chen, M. Xie, Z. Xing, C. Chen, X. Xu, L. Zhu, and G. Li, "Object detection for graphical user interface: Old fashioned or deep learning or a combination?" in *Proc. ACM Joint European Softw. Eng. Conf. and Symp. Foundations of Softw. Eng. (ESEC/FSE)*, 2020, pp. 1202–1214.

[10] G. Jocher *et al.*, "YOLOv8," https://github.com/ultralytics/ultralytics, 2023.

[11] E. Alégroth, R. Feldt, and P. Kolström, "Maintenance of automated test suites in industry: An empirical study on visual GUI testing," *Information and Software Technology*, vol. 73, pp. 66–80, 2016.

[12] T. Yeh, T.-H. Chang, and R. C. Miller, "Sikuli: Using GUI screenshots for search and automation," in *Proc. ACM Symp. User Interface Softw. and Technol. (UIST)*, 2009, pp. 183–192.

[13] T. D. White, G. Fraser, and G. J. Brown, "Improving random GUI testing with image-based widget detection," in *Proc. ACM SIGSOFT Int. Symp. Softw. Testing and Analysis (ISSTA)*, 2019, pp. 307–317.

[14] Z. Khaliq, S. U. Farooq, and D. A. Khan, "A deep learning-based automated framework for functional User Interface testing," *Information and Software Technology*, vol. 150, p. 106969, 2022.

[15] R. H. Degaki, J. G. Colonna, Y. Lopez, J. R. Carvalho, and E. Silva, "Real time detection of mobile graphical user interface elements using convolutional neural networks," in *Proc. Brazilian Symp. Multimedia and the Web (WebMedia)*, 2022, pp. 159–167.

[16] S. S. Daneshvar and S. Wang, "GUI element detection using SOTA YOLO deep learning models," *arXiv preprint arXiv:2408.03507*, 2024.

[17] J. Qian, Z. Shang, S. Yan, Y. Wang, and L. Chen, "Roscript: A visual script driven truly non-intrusive robotic testing system for touch screen applications," in *Proc. IEEE/ACM Int. Conf. Softw. Eng. (ICSE)*, 2020, pp. 297–308.

[18] S. Yu, C. Fang, X. Li, Y. Ling, Z. Chen, and Z. Su, "Effective, platform-independent GUI testing via image embedding and reinforcement learning," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, pp. 1–27, 2024.

[19] M. Nass, E. Alégroth, and R. Feldt, "Why many challenges with GUI test automation (will) remain," *Information and Software Technology*, vol. 138, p. 106625, 2021.

[20] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE Conf. Comput. Vision and Pattern Recognition (CVPR)*, 2016, pp. 779–788.

[21] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common objects in context," in *Proc. European Conf. Comput. Vision (ECCV)*, 2014, pp. 740–755.

[22] R. Smith, "An overview of the Tesseract OCR engine," in *Proc. Int. Conf. Document Analysis and Recognition (ICDAR)*, 2007, pp. 629–633.