FH University of Applied Sciences TECHNIKUM WIEN

BACHELOR PAPER

Thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science in Engineering at the University of Applied Sciences Technikum Wien - Degree Program Information and Communication Systems and Services

Precision at Pixel-Level: You only live once doing UI test automation

By: Nikolaus Rieder

Student Number: 2010258028

Supervisor: Dr. Dietmar Millinger

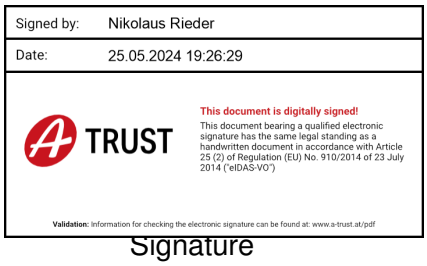
Vienna, May 25, 2024

Declaration

“As author and creator of this work to hand, I confirm with my signature knowledge of the relevant copyright regulations governed by higher education acts (see Urheberrechtsgesetz /Austrian copyright law as amended as well as the Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I hereby declare that I completed the present work independently and that any ideas, whether written by others or by myself, have been fully sourced and referenced. I am aware of any consequences I may face on the part of the degree program director if there should be evidence of missing autonomy and independence or evidence of any intent to fraudulently achieve a pass mark for this work (see Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I further declare that up to this date I have not published the work to hand nor have I presented it to another examination board in the same or similar form. I affirm that the version submitted matches the version in the upload tool.“



Vienna, May 25, 2024

Signature

Abstract

Developing and executing test suites on embedded devices with a graphical user interface (GUI) can be a time-consuming and exhaustive process. Nevertheless, it is a necessary task in interfaces of critical infrastructure systems, like fire alarm system (FAS) or health care system (HCS) developed at Schrack Seconet. This research analyzes the viability of using machine learning for identifying and detecting user interface (UI) widgets based purely on screenshot data. The analysis is written from the perspective of test automation (TA) performed on critical systems like the aforementioned. The focus is on high performing results due to the qualitative requirements in UI test automation of such systems, where upcoming problems during development are considered potentially critical if not detected.
During the development of this paper, a You Only Look Once (YOLO) model was trained on randomly synthesized datasets of UI screenshots. The datasets were generated with the Light and Versatile Graphics Library (LVGL), a common and popular library in embedded systems. During development of the datasets and model training processes, key problem factors are identified and potential mitigations addressed. The findings showcase these points in the context of a nurse call system, where reliability of the user interface plays a critical role and margin for error is non-existent.
The project follows a Design Science research approach, in which a UI generator for LVGL was created, that is capable of generating datasets of randomly chosen and placed widgets on a screen, as well as the generation of realistic looking designs by using a system based on a JavaScript Object Notation (JSON) schema. In order to create larger datasets, a large language model (LLM) was used to design multiple UIs from a list of pre-defined contexts and themes.

This research aims to reduce time spent in test development and improve test coverage of varied UIs in devices of critical systems, while aiding human developers in manual and automated testing.

Keywords: YOLOv8, UI Testing, embedded devices, widget detection, widget classification, automated testing

Acknowledgements

I extend my gratitude to the faculty and staff at UAS Technikum Vienna for their invaluable guidance and the wealth of knowledge they have shared, which has culminated in the completion of this bachelor thesis. Special appreciation is given to my academic advisor, Dietmar Millinger, for his expertise and dedication. Additional thanks are due to Dietmar Millinger, Karl M. Göschka, Lorenz Froihofer, and Susanne Teschl for their inspiration and exceptional contributions to my understanding of machine learning, computer science, and mathematics, respectively. Thanks should also go to my academic peers, for the collaborative environment fostered on our student community server and the valuable networks we have built throughout this educational journey. I am particularly grateful to my partner, whose support and patience has been unwavering during my academic endeavors at the UAS Technikum Vienna. I would also like to express my sincere thanks to Schrack Seconet AG for providing me with the opportunity to develop this paper in a professional context. The support from colleagues has been instrumental in aligning my academic pursuits with practical applications in the workplace. Lastly, I thank my own body and mind for not breaking apart throughout the long days and mostly nights that were involved up until this point.

Contents

1 Introduction

In embedded systems providing critical infrastructure, it is often mandatory to have well-established and high-quality testing procedures. One such system is nurse call used in hospitals and health care facilities, which consists of devices that allow humans to call for help in situations ranging from mundane *(e.g. requesting patient service)* to life-threatening *(e.g. cardiac arrest alarm)* use-cases. These systems require interfaces for human interaction and such interfaces need to be rigorously tested, to provide feedback on system quality in early stages of development, where the potential for harm is very low. Additionally, automated testing becomes necessary, to reduce possible variation in test execution when compared to manual testing, where human-error is more prevalent than with machines repeating test cases.
However, when such interfaces only consist of a touch display, it can become very challenging to automate, as it requires tools for the machine to analyze the visual image and to interact with the UI using stimuli *(e.g. simulated touch)*. The automation gets further complicated, the more inputs and outputs the machine needs to deal with in order to perform a representative human test routine. Once such test routines are established, they need to be continously maintained in order to adapt to changes in the UI, otherwise such changes can break the test system and halt the overall development process.
This research project will demonstrate a machine learning tool to aid in analyzing images to obtain metadata about the displayed UI. The metadata consists of position and size of individual widgets, which then can be used in testing routines for precise functional interactions. By training a model that can detect and classify widgets based on screenshots, the testing routines can adapt to many UI arrangements dynamically, which can reduce labor involved for maintenance of test repositories.
Since covering all possible UI libraries and frameworks would be tremendous in scope, the research will limit focus on LVGL as the basis for the UI. It begins by creating an image and annotation generator for LVGL, since there is no available dataset for this library specifically and it also allows for targeted adjustments in dataset size, variation and balance of widget types. Furthermore, it allows for validation based on a real world example from the company where the author is employed at. The work will also be limited to a subset of available widget types, to provide a basic baseline for the viability of the tool, which may be easily extended in the future. From a testing perspective, this paper is only concerned with the prediction accuracy of the model on UI widgets.

2 Problem description

When developing automated test cases for a product which has very long life cycles, due to being incorporated in building infrastructure for 10 years or more, it can be expected that involved maintenance labor of existing test cases surpasses the creation of new test cases. But even in such long-lasting products, big system changes *(e.g. UI redesign, new controller architecture, new graphics libraries...)* eventually occur. Such changes require a new evaluation and redesign of existing test procedures, to accommodate for the new system or UI, even if the actual use-case and performed steps fundamentally remain the same. For the business it is therefore beneficial to have testing tools, which reduce the maintenance costs of such legacy test cases, by being dynamic and adaptable to the given test environment.

2.1 Problem context at Schrack Seconet

New developed products by Schrack Seconet will use the Light and Versatile Graphics Library (LVGL) [1], a popular choice in embedded hardware for developing user interfaces. This framework already comes with implementations that allow for simulation of the UI rendering on a development machine, which already improves development time since it doesn't require display hardware to develop and test UI design. But being a library targeted at embedded systems, there are no integrated testing tools that would allow a test automation developer to verify the functionality of those designs on the actual hardware. The nurse call system Visocall IP (VCIP) [2] is a distributed embedded system, which relies on multiple hardware devices to interact with each other and UI functionality is tied to the signals and messages of the system as a whole. When it comes to simulation, it marks unit testing of the user interface as an unusable solution, due to the inherent complexity of the system and the required simulation of multiple devices or peripherals for the UI to function properly.
Instead, the developers at Schrack Seconet integrated a low-level kernel Application Programming Interface (API) into the development firmware of UI devices, which allows for simulation of user interactions on the devices, mimicking human behavior on the display. The mimicked behavior includes interactions such as touch, press, release, drag and combined variations of the aforementioned.
However, this approach requires knowledge about the metadata of the UI by the test developer, so that the defined interactions are performed on the correct coordinates of the user interface element.

A common example of such a test case using coordinates is visible in Fig.1, showing a routine which unlocks the screen by swiping.

Figure 1: Example of a lock screen test case using absolute coordinates by Schrack Seconet

The expected functionality of the device is then assessed by aggregating device log data and verification of expected signal states from accessible hardware outputs. These outputs typically are Light Emitting Diodes (LEDs) on the test device itself or other interconnected devices in the distributed system, which are connected through proprietary IP-based protocols or other proprietary hardware bus technologies employed by the Schrack Seconet system. In addition to these expectations, the raw frame buffer of the controller can be exported to the test machine, which will also depict any visual errors, as long as the error occurred in firmware and not during transmission to the display driver. An example of such a visual error is depicted in Fig.2, which shows display errors for system time in the upper right and also the font symbols, for an informative text on how to unlock the screen below the icon, are not displayed properly.

Figure 2: Example of a visual error on the lock screen of a VCIP product in development

Demand for new testing methods

The current approach works well for a given user interface design which is not affected by frequent changes, but we currently lack in verification on the visual aspect of the user interface itself. Furthermore, it comes with an inherent problem in maintenance and development time, since the test automation engineer is required to determine the element coordinates through reverse engineering or by inspecting the firmware code-base.

The latter approach is generally frowned upon in the company, as we prefer a tendency towards black-box-testing, which mitigates the chance of overlooking software bugs due to code blindness. It is preferred for the test automation engineer to have as little knowledge about the implementation of the firmware as possible, so that the tester can focus on unbiased verification of the target device. It also would not be a viable approach to look through the codebase to determine the element coordinates, since it typically is not blatantly written inside the code and would require the tester to work through the programmatic abstractions.

Coordinates of UI elements are typically reverse engineered by inspecting screen dumps from the test device itself. The generated image file has the exact dimensions as the device display, and it is easy to determine coordinates through generally available image viewers, such as IrfanView [3]. The positions of UI elements is then embedded into automated test case routines, which will perform the defined UI interactions. Functional expectations are then verified through existing test framework capabilities.

This approach comes with a caveat, as the hard-coded metadata is bound to become invalid when UI changes, like when new products are developed as previously mentioned. This results in a rework of all affected test cases. This can be mitigated through abstractions (e.g. global variables, re-usable routines), but ultimately remains a very labor-intensive maintenance process due to the sheer amount of individual coordinates for specific elements and menus.

Another perspective on the current approach is the existing dependency on manual testing, since the described test method does not involve visual verification. The existing test automation can reliably assess the correct execution of functions in the system, but is incapable of finding errors that exist in the displayed image and animation on the device itself. The visual aspect of the UI plays an important in nurse call systems, since it involves multiple languages for text, standards for visual and acoustic signalization described in the VDE-0834 [4], and also the visual prioritization of emergency calls.

Making use of the aforementioned screen dumps to assess these visual aspects would be a valid approach, but this has not been implemented so far. The development of a solution began through this paper. This visual verification however would still not account for errors that happen between transmission from the microcontroller unit (MCU) frame buffer to the display driver chip, but that is a negligible error source.

A solution, which is capable of dealing with the dynamic nature of user interfaces, visual verification and future UI development changes is highly sought after in the company. Computer vision and machine learning is assumed to be a promising approach for the purpose of reducing the cost and time involved in manual and automatic testing of these user interfaces.

3 Research questions

- **Is YOLOv8 a viable model for detecting LVGL widgets based on screenshots?**
  In order for a model to be viable in widget detection for the previously stated problem context of test automation, it would need to have a high prediction confidence with accurate results. If the selected model fails to perform well, it will only bring rise to many false-positives when performing test routines, ultimately failing adoption and qualitative trust in the tool. The exact target accuracy to overcome was not possible to determine, as humans can make errors in writing coordinates and given how minor these incidents are, they are generally not documented. But given the critical context of the system under test (SUT) a general approximation of minimum 80% confidence and accuracy seems like a fair target. This number was chosen out of experience, as the remaining 20% error margin would be manageable with review processes of the model output, as it is already the case for test case development.

- **Can machine learning (ML) be a viable tool in aiding test developers through automatic widget localization?**
  In order for the ML tool to properly aid the developer, it would need to make the general test development process faster and provide automation. Since the purpose of the tool is to provide coordinate metadata and the current process involves acquiring this information manually per single image, it would already be viable enough if the tool could automate this behaviour over a given set of images and return the desired coordinates.

It was considered to answer these questions for the ongoing development of a new UI design at the company, however the release planning and starting time for that development did not correlate well with the thesis deadline. The answers to these questions will thereforee be written based on the results of the synthesized data.

4 Existing literature and theoretical basis

The topic of the thesis is specific to applying a ML based object detection model for acquiring coordinate metadata of UI elements, to be used in visual GUI testing (VGT) on devices of a distributed embedded systems. A direct comparison in that regard has not been found at the time of writing. But the comprehensive survey [5] on embedded systems testing serves as a good start for systematic approaches.

In an article [6], the method of using record-and-play is described, with the goal of improving test accuracy. It uses the display data sent to a host PC and analyzes events performed by a manual tester to record test cases. In another step, the recorded events are played back to the device. However, this approach is not very easy to perform on distributed embedded systems, with various displays involved. It also comes with a performance challenge and the cost of proprietary technologies used.

In the field of mobile applications, a few studies were made using ML models to perform testing or create metadata of the graphical model of the UI. (i.e. element types, hierarchical structure, detection and recognition, state) [7]–[13] A common dataset used in these studies is the RICO dataset [14]. These studies share similarities of the upcoming research in regard to their usage of a ML model for gathering metadata about UI. Notably their software applications and user interfaces typically are written in languages and frameworks with better tooling support for performing visual automated testing.

An interesting case was made with the AI-driven tool "Owl Eye" [15], which can detect visual defects or inconsistencies in graphical user interface, based purely on provided screenshots. Two models, Faster-RCNN and YOLO version 8 by ultralytics (YOLOv8), were trained on the RICO dataset, with a concluding choice on YOLOv8 due to higher accuracy and faster training time. The choice of model for this thesis was inspired by the results showcased in that paper.

These aforementioned studies hint towards a possible improvement through the usage of ML for visual testing and also the general improvement of quality, due to performing better than manual testing. As such, the thesis will explore if ML object detection is a viable and also reliable tool to aid in test development requiring coordinate metadata of UI elements.

4.1 Convolutional Neural Network (CNN)

CNNs are a type of deep neural networks, which are predominantly used for analyzing visual data. Resulting models have showcased remarkable success in various computer vision tasks, with the possibly fastest currently being the YOLO model. It was originally developed and researched by Joseph Redmon [16], but has since seen multiple version updates, with one of the latest being YOLOv8 [17].
The core idea behind CNNs is to learn spatial hierarchies of features from input images through the usage of convolution layers, meaning the underlying mathematical foundation is to perform convolution operations.
A CNN typically consists of several layer types:

- Convolution layers, which apply a set of convolution filters to the input image to capture local patterns (i.e. edges, textures, shapes)
- Pooling layers, which reduce the spatial dimensions of the input. This helps in reducing computational load and controlling over fitting. Common techniques include max pooling and average pooling.
- Fully connected layers, which are traditional neural networks typically placed at the end to perform the high-level reasoning based on the extracted features.

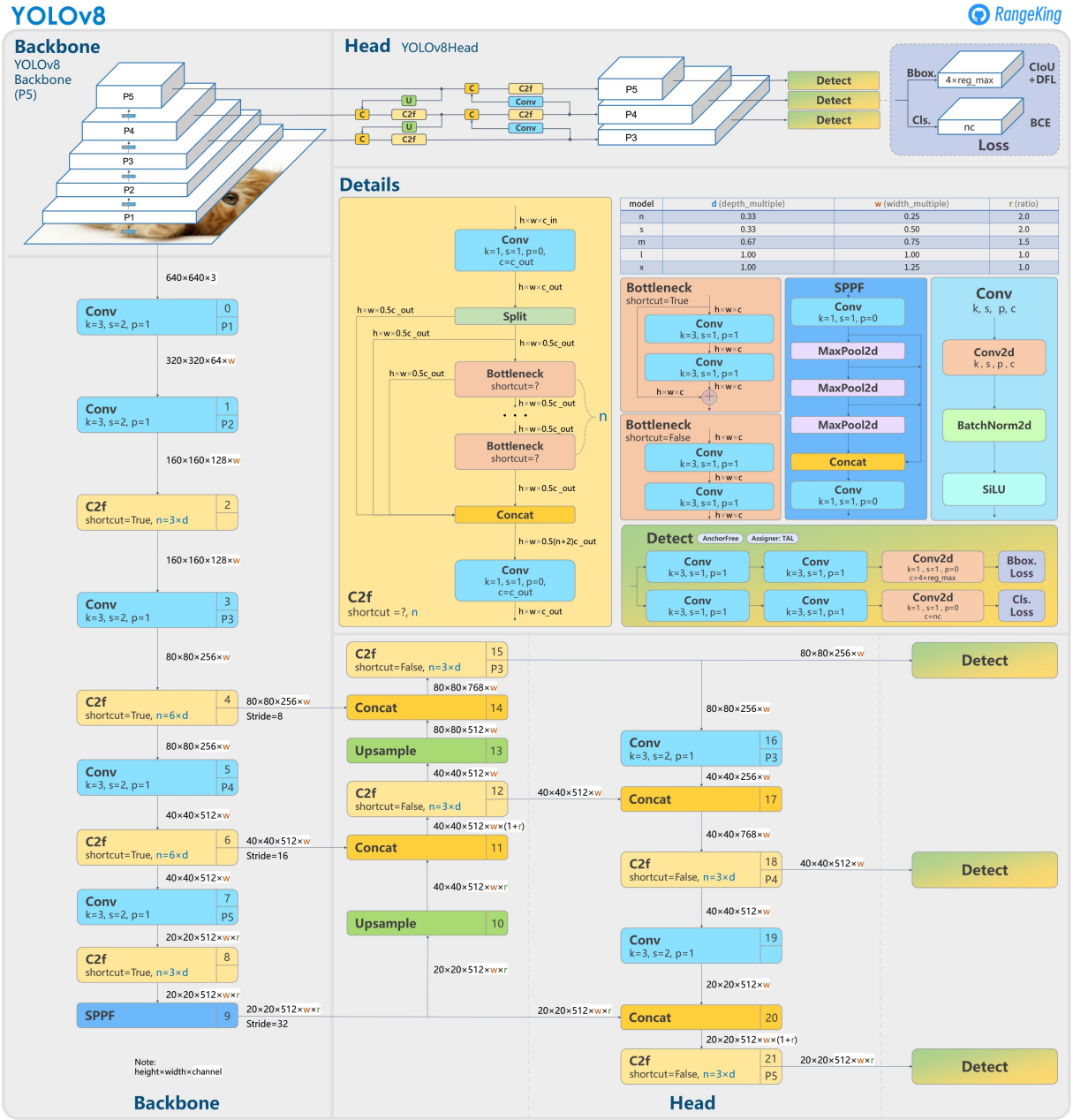
4.1.1 You Only Look Once (YOLO) architecture

The YOLO model architecture is a family of single-shot object detectors. They are known for their efficiency, due to their bounding box and class probability predictions performed directly on full images in a single evaluation. The single evaluation is also what gave the well-known model name "You only look once".
The architecture of the YOLOv8 model, further visualized in Fig.3, consists of two main parts:

- **Backbone**
  The backbone is responsible for extracting features from a given input image. It contains series of convolution layers, as well as further operations to down sample the image and capture the spatial hierarchies of features.
- **Head**
  The head processes features extracted by the backbone and is responsible for predicting the bounding boxes and class probabilities. A detection layer refines the feature maps and will predict the object locations and classes at multiple image scales.

Further details about the architecture can be determined through the open-source repository of the model [17].

Figure 3: Model structure of YOLOv8 detection models by RangeKing [18]



model	d (depth_multiple)	w (width_multiple)	r (ratio)
n	0.33	0.25	2.0
s	0.33	0.50	2.0
m	0.67	0.75	1.5
l	1.00	1.00	1.0
x	1.00	1.25	1.0

4.2 Light and Versatile Graphics Library (LVGL)

This graphics library [1] is a popular choice amongst embedded system engineers and has seen much usage, due to its portable support for a wide range of target architectures and devices. Its performance works well in resource constrained systems.
The library is generally very flexible, providing a rich set of widget types [19] and layout systems (flex, grid), which allow developers to create complex and custom UI. Fig.4 and Fig.5 showcase some of the UI capabilities of the library, with more available on their web page [20].

Figure 4: LVGL music player demo [20]

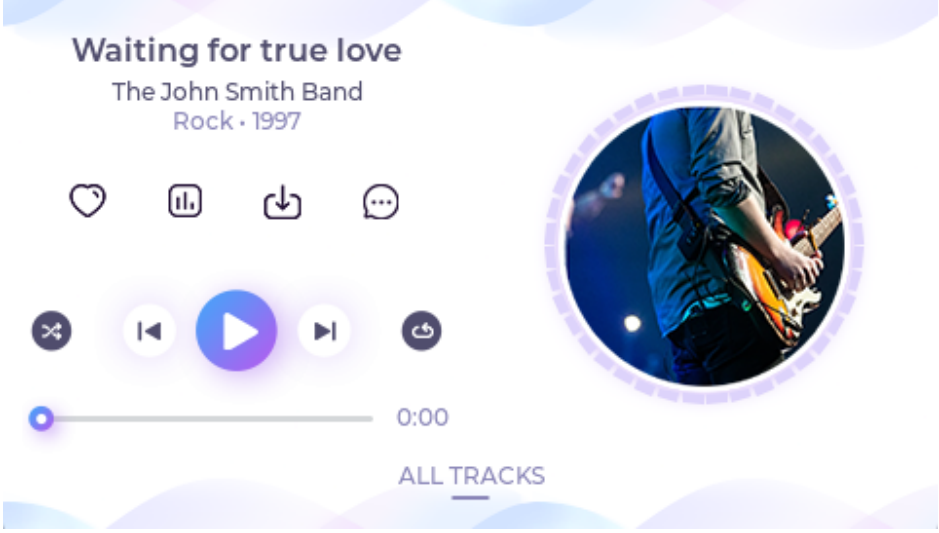
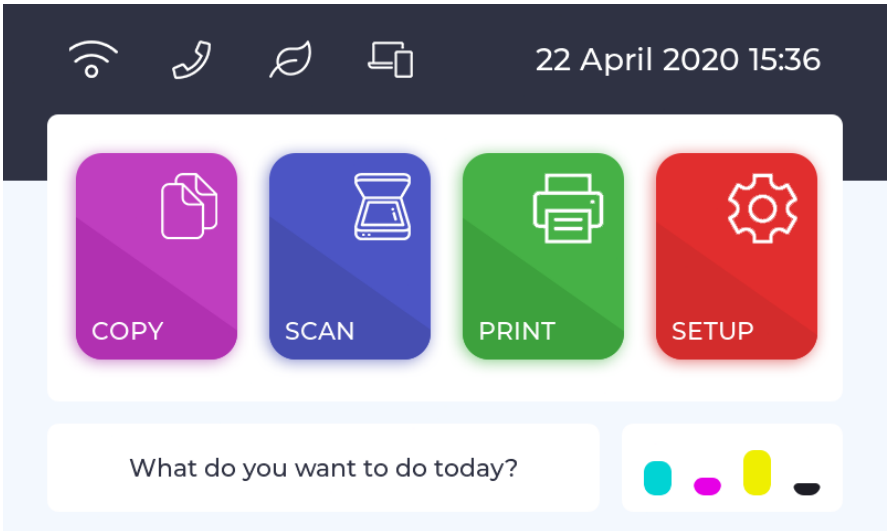


Figure 5: LVGL printer demo [20]



5 Development of synthesized datasets

There generally is no available dataset for the target graphics library (LVGL) of this research, so it was mandatory to create one during project development. Creating such a dataset artificially, meaning the images are not based on real world UIs, has the benefit of being able to adjust the size, variation and balance of the dataset very precisely. It would however be a complicated and time-consuming endeavour, if all of these user interfaces would be created manually using regular LVGL code, given that typical datasets for a ML model of this type require hundreds of pictures or even more.
This is why a generator must be built, which is capable of creating and placing widgets in a structured manner. A screenshot of the displayed window needs to be captured and any metadata (size and position) of placed widgets must be gathered for creating corresponding annotations. The generator will also need to handle variation in visual representation of widgets, otherwise resulting datasets would be biased and detection accuracy of the model would be bad when used on screenshots of real-world examples, which typically incorporate varying visual representations of the same widget type.
In order to create variety in individual widgets, a randomized approach is used, where the outputted widgets will have varying styles applied to them. To gain more fine-control on the final visual representation of an created UI, an additional design mode is necessary, where the creation of user interfaces can be statically described. The design mode will allow for creation of realistically looking UIs, while the random mode will allow for the creation of varying visual representation of the individual widgets.
Since the creation of a window in the generation process is singular, it will be necessary to have a mechanism to repeatedly call the generator to obtain a new image with each iteration. Using a modular approach, this paper will separate the generator, capable of producing a single image with annotations, from the so-called randomizer, which will loop the generator repeatedly for an arbitrary amount of iterations. This approach allows for a focused implementation that is capable of producing a good singular image, while having a secondary implementation that solely focuses on the organisation and creation of the dataset. Additionally, when written in the modular approach, it also allows for separation of concerns, where the generator will only need to deal with the intricacies of the LVGL API, while the randomizer can focus on the structural requirements of the dataset for the chosen model.

5.1 First generator version in C

The initial generator (v1) is forked from the existing VSCode simulator project of LVGL.[21] The simulator was stripped from any demo code and reused as a binary with Command-line interface (CLI) arguments. For proper operation, the `lv_conf.h` file was also modified, to increase memory allocation and deactivate unnecessary features. In the configuration, the default color depth was also modified to be 24 bit, so that output images would appear in higher quality. To be able to store screenshots, an additional library named `lv_100ask_screenshot` was used,[22] which deals with the proper JPEG encoding using a `tiny_jpeg` library internally.
Since the generator uses the allocation and storage mechanisms provided by LVGL, it is necessary to use an identifier for the target storage system as prefix to the output filename. For this the `\` character was chosen, so when supplying the CLI with a desired output filename, an example argument would look like this: `\screenshot.jpg`
The following usage information will explain the available console arguments for this generator version:

```
Usage: ./lvgl_ui_generator/build/bin/demo [OPTIONS]
Options:
-m <mode>                   Set the operation mode ("randomizer" or "design")
Randomizer Mode Options:
-w <width>                  Set the width of the UI
-h <height>                 Set the height of the UI
-c <widget_count>           Set the number of widgets to be randomized
-t <widget_types>           Comma-separated list of widget types to be used
-o <output_file>            Set the output file path
-d <delay_count>            Set the screenshot delay count
-l <layout>                 Set the layout type
```

Listing 1: Usage instructions of LVGL generator v1

When run, the generator will produce an image in the provided width and height, as shown in the example of Fig. 6
It will place randomly chosen widgets up to the given count on the window and use the selected layout (none, grid, flex) to structure them.
The layout structuring of widgets helped in outputting more placement variation in the final image. When using `flex`, the created widgets are aligned and will wrap onto the next line if they run out of width. In the `grid` layout, widgets would be placed in a grid-like pattern over the whole image. Finally in the `none` layout, widgets would be absolutely positioned randomly over the screen, sometimes also causing overlap of widget types.

Figure 6: Example output of generator v1 (640x640, 40 random widgets, flex)

5.1.1 Development issues

During development and testing of the first generator, it became apparent, that implementing further widget types would create a lot of complexity due to parameter passing in code. Since the C language does not inherently have classes, it was considered to be much more difficult to implement multiple widget types, each of which require a different set of options to be displayed properly.
Additionally, issues like segmentation faults due to window deletion occurred, which were inherited from the forked project. The implementation of a design mode was attempted, but this only increased complexity, which resulted in more unreliability of the program, despite the authors experience in the language. Frequent errors in compilation as well as a high compilation time due to the included LVGL examples, which the author was not able to remove, were halting and slowing down development time. A lot of features *(widget styles and states, widget property randomization, ...)* were still left to be implemented at a low level and C being a systems programming language, it was becoming too big of a problem for the project.

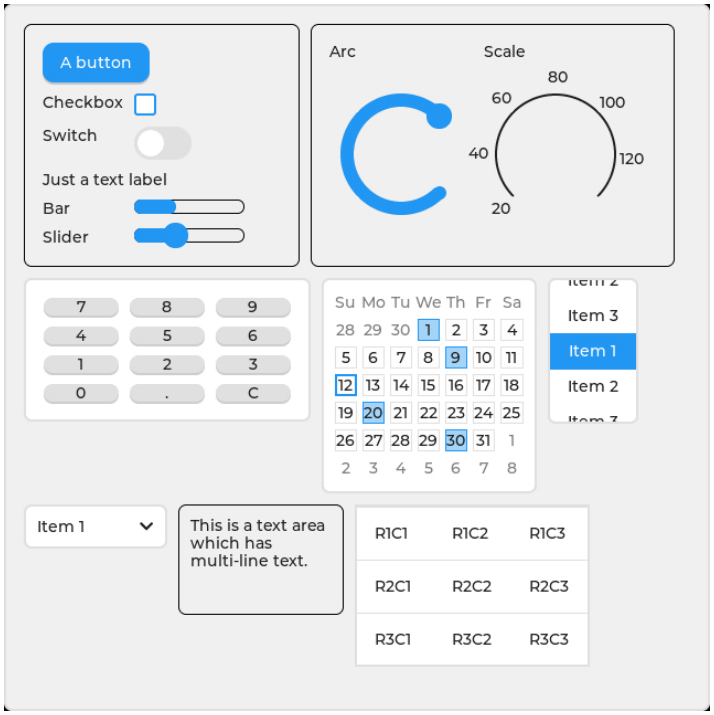
The general implementation based on C proved to be much more error-prone as a result of these issues and was therefore considered too complex for usage in a generation pipeline. A new solution was sought after, which would allow implementation of the generator in a more resilient and forgiving programming language.
LVGL provides C bindings [23] for usage and implementation in other programming languages. A language promoted by the creators themselves is micropython [24], which is a reduced implementation of python for running on hardware. While the issues faced with the initial generator certainly are solvable, there existed no premise to continue in the C language. A new version in that language was considered to be a simpler approach that would refocus the project on the dataset creation matter at hand.

5.2 Second generator version in Micropython

The second generator version is implemented in micropython, and was started as a fresh project using the `lv_micropython` project [23] as a necessary sub-module for compiling the micropython binary including the `lv_bindings` module [24].
Rewriting the functionality of the initial generator was much simpler, as the binary comes with `argparse` for easily creating a CLI interface. Having the functionality of classes, it was also easier to differentiate between the two modes with their own source files. Widget creation was divided into separate functions, one for each widget type, and then called via a mapping of the type name to the creation function. This proved to be much more maintainable throughout project development and issues were able to be faced on an individual basis per implemented type.
Both modes provide a `get_ui` function to retrieve the top-level container and all its children in a dictionary object format, containing all relevant metadata information gathered through usage of the `coords_t` object. This LVGL object is necessary, as it stores the coordinates of a widget in relation to the whole window, while other methods usually provide this information in relation to the parent container that the widget resides in.
The retrieved container is passed to a screenshot function, which uses a manually added JPEG encoding from the `flat` project [25] to create an image. Using the coordinate metadata, the annotation file of the resulting image is created in YOLO format (`class center_x center_y center_width center_height`) and also normalized if the user provides the corresponding flag. A resulting example image using design mode is illustrated in Fig.7 and its corresponding annotation file is listed below.
The following widget types were implemented and used in this generator version:

- Arc
- Bar
- Button
- Buttonmatrix
- Calendar
- Checkbox
- Dropdown
- Label
- Roller
- Scale
- Slider
- Spinbox
- Switch
- Table
- Textarea

Figure 7: Example output of generator v2 (640x640, design, widget_showcase.json)



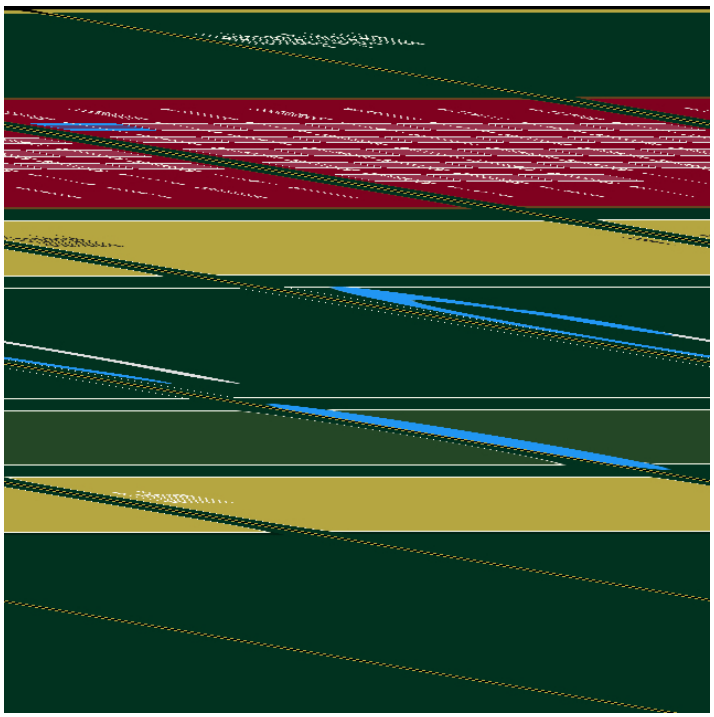

```
textarea 0.3625 0.784375 0.234375 0.15625
table 0.6625 0.821875 0.3328125 0.2296875
checkbox 0.2046875 0.140625 0.04375 0.03125
arc 0.5609375 0.2109375 0.203125 0.203125
label 0.0734375 0.2875 0.0390625 0.025
label 0.0859375 0.328125 0.065625 0.025
label 0.478125 0.065625 0.0375 0.025
label 0.7078125 0.065625 0.059375 0.025
label 0.1109375 0.1375 0.1140625 0.025
bar 0.2609375 0.2859375 0.15625 0.0203125
label 0.0921875 0.184375 0.078125 0.025
label 0.1421875 0.246875 0.1765625 0.025
switch 0.2234375 0.1953125 0.08125 0.046875
buttonmatrix 0.2296875 0.4875 0.40625 0.203125
button 0.1296875 0.08125 0.1515625 0.05625
calendar 0.6015625 0.5390625 0.3046875 0.3046875
scale 0.7796875 0.2109375 0.203125 0.203125
slider 0.2609375 0.3265625 0.15625 0.0203125
roller 0.8328125 0.4890625 0.128125 0.20625
dropdown 0.128125 0.7375 0.203125 0.0625
```

Listing 2: Generated annotation file for example output of generator v2

Distorted output images due to render race conditions

The used snapshot API sometimes causes output issues, since race-conditions can occur due to render updates while the referenced image data is still being encoded, resulting in distorted or sheared images as seen in the example Fig.8. This largely originates from adding the JPEG encoding in micropython, which inherently is much slower than its C counterpart. The issue was discovered very late in paper development and a problem solution using a compiled C module of the latest libjpeg-turbo [26] version was discussed in development forums of LVGL [27]. Adding this module improved encoding speed, but the faced race conditions due to underlying render updates by LVGL still occurred. Images which had this issue needed to be regenerated and datasets were fixed manually.

Figure 8: Example image with distortions from generator v2 (640x640)



Memory allocation errors

Since the image encoding takes place in micropython, it actually takes up a lot more memory in heap. When creating and encoding images with the generator, it may result in memory allocation errors, depending on available memory on the running machine. This issue is not solved in the final generator version of the paper and was circumvented by retry mechanisms when creating image files using the dataset pipeline.

5.3 Random mode

In random mode, the generator will produce a window of desired width and height. It randomly chooses widgets from the user provided type list and places them into a container. Since absolute positioning proved to be most suitable for random generation, the implementation of layouts was omitted. To avoid overlap, a spatial map was used, to find free space in the window during widget creation and placement.
The usage description in Listing 3 provides an overview of the parameters for this mode.

```
usage: src/main.py [-h] [-m, --mode mode] [-?, --usage] [-n, --normalize] [-o,
↪  --output_file output_file] [-W, --width width] [-H, --height height] [-c,
↪  --widget_count widget_count] [-t, --widget_types widget_types+] [-l, --layout
↪  layout] [--random-state]

Process CLI arguments for the UI generator.

optional args:
  -h, --help                        show this message and exit
  -m, --mode                        mode the mode to run the program in
  -?, --usage                       Print usage information for that mode.
  -n, --normalize                   normalize the bounding boxes
  -o, --output_file output_file     The output file (screenshot)
  -W, --width width                 the width of the UI
  -H, --height height               the height of the UI
  -c, --widget_count widget_count   the count of widgets
  -t, --widget_types widget_types+  A list of widget types
  -l, --layout layout               the layout option
  --random-state                    Use a random state for each created widget
  ↪  (experimental)
```

Listing 3: Usage instructions of LVGL generator v2 in random mode

5.3.1 Style and state variation

When using random mode, the generator will randomize the applied style of widgets to artificially create variation in created datasets. Additionally, there is an experimental feature, which will also randomize the state of widgets. The latter is not recommended for usage in training, as sometimes this might cause widgets to not be displayed if their state is invalid or disabled, which will confuse the model, as the created metadata will still include the annotation of said widget.
Listing 4 shows the properties and their range of possible values to be randomized. Listing 5 shows the available states.

```
# List of style properties to randomize
properties = [
    ('set_bg_color', lv.color_make),
    ('set_bg_opa', lambda: random.randint(0, 100)),
    ('set_border_color', lv.color_make),
    ('set_border_opa', lambda: random.randint(0, 100)),
    ('set_border_width', lambda: random.randint(0, 10)),
    ('set_outline_width', lambda: random.randint(0, 10)),
    ('set_outline_color', lv.color_make),
    ('set_outline_opa', lambda: random.randint(0, 100)),
    ('set_shadow_width', lambda: random.randint(0, 15)),
    ('set_shadow_offset_x', lambda: random.randint(0, 10)),
    ('set_shadow_offset_y', lambda: random.randint(0, 10)),
    ('set_shadow_color', lv.color_make),
    ('set_shadow_opa', lambda: random.randint(0, 100)),
    ('set_line_width', lambda: random.randint(0, 10)),
    ('set_line_dash_width', lambda: random.randint(0, 10)),
    ('set_line_dash_gap', lambda: random.randint(0, 10)),
    ('set_line_rounded', lambda: random.choice([True, False])),
    ('set_line_color', lv.color_make),
    ('set_line_opa', lambda: random.randint(0, 100)),
    ('set_text_color', lv.color_make),
    ('set_text_opa', lambda: random.randint(0, 100)),
    ('set_text_letter_space', lambda: random.randint(0, 10)),
    ('set_text_line_space', lambda: random.randint(0, 10)),
    ('set_opa', lambda: random.randint(0, 100)),
    ('set_align', lambda: random.choice([lv.ALIGN.CENTER, lv.ALIGN.TOP_LEFT,
    ↪  lv.ALIGN.TOP_RIGHT, lv.ALIGN.TOP_MID, lv.ALIGN.BOTTOM_LEFT,
    ↪  lv.ALIGN.BOTTOM_RIGHT, lv.ALIGN.BOTTOM_MID, lv.ALIGN.LEFT_MID,
    ↪  lv.ALIGN.RIGHT_MID, lv.ALIGN.DEFAULT])),
    ('set_pad_all', lambda: random.randint(0, 10)),
    ('set_pad_hor', lambda: random.randint(0, 10)),
    ('set_pad_ver', lambda: random.randint(0, 10)),
    ('set_pad_gap', lambda: random.randint(0, 10)),
    ('set_pad_top', lambda: random.randint(0, 10)),
    ('set_pad_bottom', lambda: random.randint(0, 10)),
    ('set_pad_left', lambda: random.randint(0, 10)),
    ('set_pad_right', lambda: random.randint(0, 10)),
    ('set_pad_row', lambda: random.randint(0, 10)),
    ('set_pad_column', lambda: random.randint(0, 10)),
    ('set_margin_top', lambda: random.randint(0, 10)),
    ('set_margin_bottom', lambda: random.randint(0, 10)),
    ('set_margin_left', lambda: random.randint(0, 10)),
    ('set_margin_right', lambda: random.randint(0, 10))
]
```

Listing 4: Available style properties which are randomized in given value range

```python
def randomize_state(widget: lv.obj):
    if hasattr(widget, "set_state"):
        state = random.choice([lv.STATE.CHECKED, lv.STATE.DISABLED,
        ↪  lv.STATE.FOCUSED, lv.STATE.PRESSED, lv.STATE.HOVERED,
        ↪  lv.STATE.EDITED])
        widget.set_state(state, True) # Add the state
    else:
        raise AttributeError(f"Widget {widget} does not have a 'state'
        ↪  property.")
```

Listing 5: Available state properties which are randomized in given options

5.4 Design mode

In design mode, the generator will parse a provided JSON file and create a UI according to the specified design. This mode allows for creation of a static UI design, which more resembles the expectation of a realistic interface in comparison to the random mode. To allow for variation in this mode, a special type `random` is available, which randomly chooses widgets from a given type list and randomizes their appearance. This option is similar to random mode and allows for randomization in specific portions of the UI, while keeping the rest statically defined. In the design file, various styles can be defined. Multiple styles can then be applied to individual widgets and containers by name reference. In general, all possible *setter* properties of LVGL styles are allowed, as the implementation checks for the availability of the corresponding setter attribute on a `lv.style_t` object.
Since the design mode only requires a single parameter *(i.e. the path to the design file)*, no further usage instructions are provided in this paper.

5.4.1 The design file rules

The creation of design files require a certain set of rules that need to be followed, in order for the creation to work properly.

1. It is mandatory that the first widget object in `root` is a container, as the root widget is always a container.
2. The title of the window is not mandatory and not used by the generator. It is only there for reference to the user for describing and recognizing the contents of a design file.
3. The `styles` object is optional and can be omitted if no styles are defined.
4. Added styles are referenced by their name in the `style` array of each widget. If a style is not found, the generator will throw a `ValueError`.

5. A style defines a list of properties that are applied to widgets via the usage of a `lv.style_t` object. The possible properties are the same as documented in the LVGL API for styles [28]. Properties are verified by checking if the specified name has a corresponding `setter` attribute in the object. This is done by appending `set_` to the property name, thus the user is required to use the property setter function names without the prefix. For example, to set the background color of a widget, one would use the property `bg_color`. The generator will then look for the `set_bg_color` attribute in the object and apply the converted value to it.

6. If a provided property inside a style object does not actually correspond to an available attribute, the generator will ignore it and continue.

7. Values supplied to style properties are converted according to the required type of the property. Some properties taking in special objects, like colors, require a specific string to be supplied (e.g. `#RRGGBB` for any color property or `top-left` for the align property).

8. If value conversion fails, the property is ignored and the generator will ignore it and continue.

9. The id property is mandatory for widgets of type container, as it is required to reference the container inside the children array, when the special widget type random is used.

10. The special widget type random may be used to supply a list of widget types for the generator to randomly choose from and then create a random widget in similar fashion to the random mode. This is useful for randomizing widgets in certain areas of the UI, while keeping the rest of the UI static.

5.4.2 JSON schema

In order to validate design files before providing them to the generator, a regular python script is used. This was not included in the generator, as the corresponding package is not available in micropython. The schema was purposely made more strict than what the generator requires, so that validation errors could be provided to the later used LLM, to offer more detailed guidance during design generation. Listing 6 shows a simple example script to validate a design file with the available schema [29] in the source repository.

```
def load_json_file(filepath: str):
    import json
    with open(filepath, 'r') as f:
        return json.load(f)

def verify_design_from_file(design_file: str, schema_file: str) -> tuple[bool,
↪  Exception]:
    from jsonschema import validate
    from jsonschema.exceptions import ValidationError
    design = load_json_file(design_file)
    schema = load_json_file(schema_file)
    try:
        validate(instance=design, schema=schema)
        print(f"Provided design file {design_file} is valid.")
        return True, None
    except ValidationError as e:
        print(f"Provided design file {design_file} is invalid:\n{e}")
        return False, e

if __name__ == '__main__':
    verify_design_from_file('path/to/design_file.json',
    ↪  'path/to/design_file.schema.json')
```

Listing 6: Exemplary validation code for design file using JSON schema

5.5 Dataset creation

Both generator modes only ever produce a single image output with a corresponding text file in the YOLO annotation format. This choice was intended to allow for a separate implementation which will handle the organization of created files, by repeatedly calling the generator and moving the outputs into a dataset folder structure. It was also a necessary choice, as the micropython programming language would have been very limiting for implementing third-party integration.
Furthermore, it also allowed for balancing the representation of classes in random mode, by removing a type from the provided widget list, once a certain threshold of instances for that class is reached.
The dataset creation concept is further illustrated in Fig.9 and Fig.10.
The produced annotations in each iteration are post-processed before they are moved into the dataset structure. Widgets that are placed out-of-bounds of the window have annotation values which are not usable for model training, as such they will be removed from the annotation file during this step. The label modifications are generally minor, since they occur rarely, and their affect on model performance is negligible. In the annotations, the written class names also need to be processed, since the YOLO model cannot interpret classes from their string representation, so they are replaced by their index in the list of implemented types.

Figure 9: Dataset creation concept using random mode

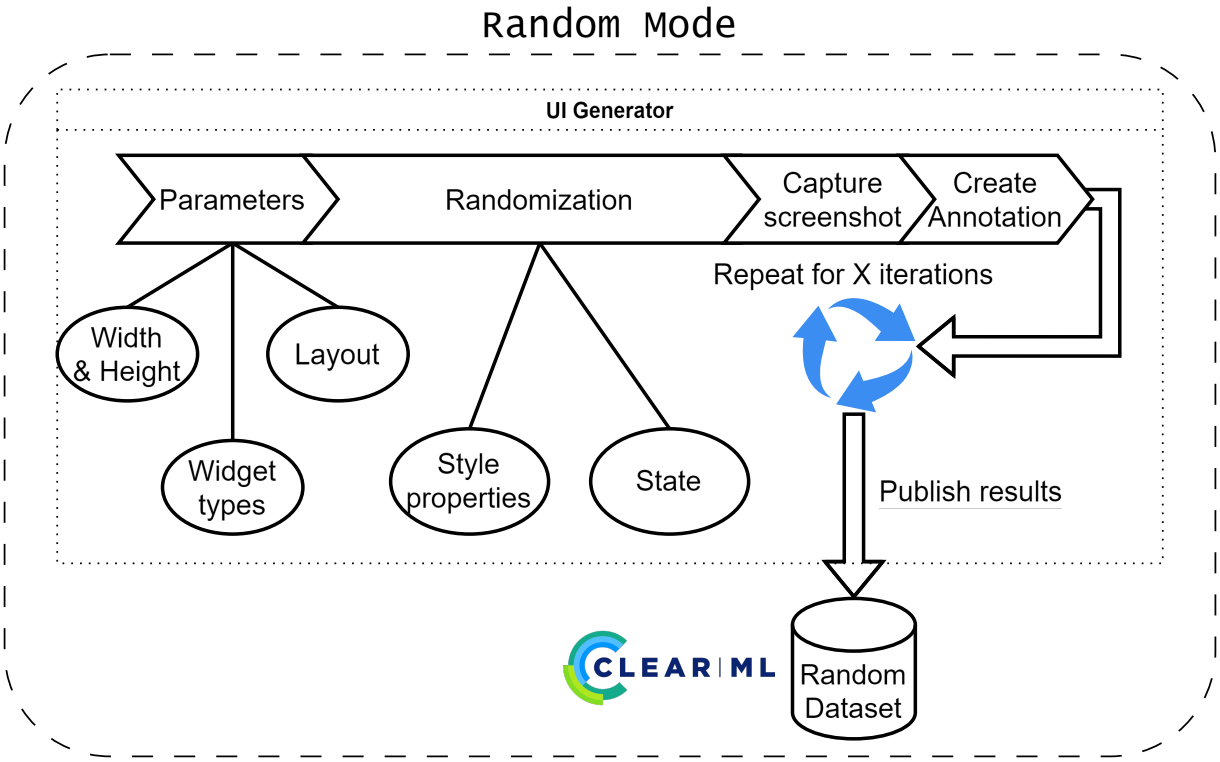
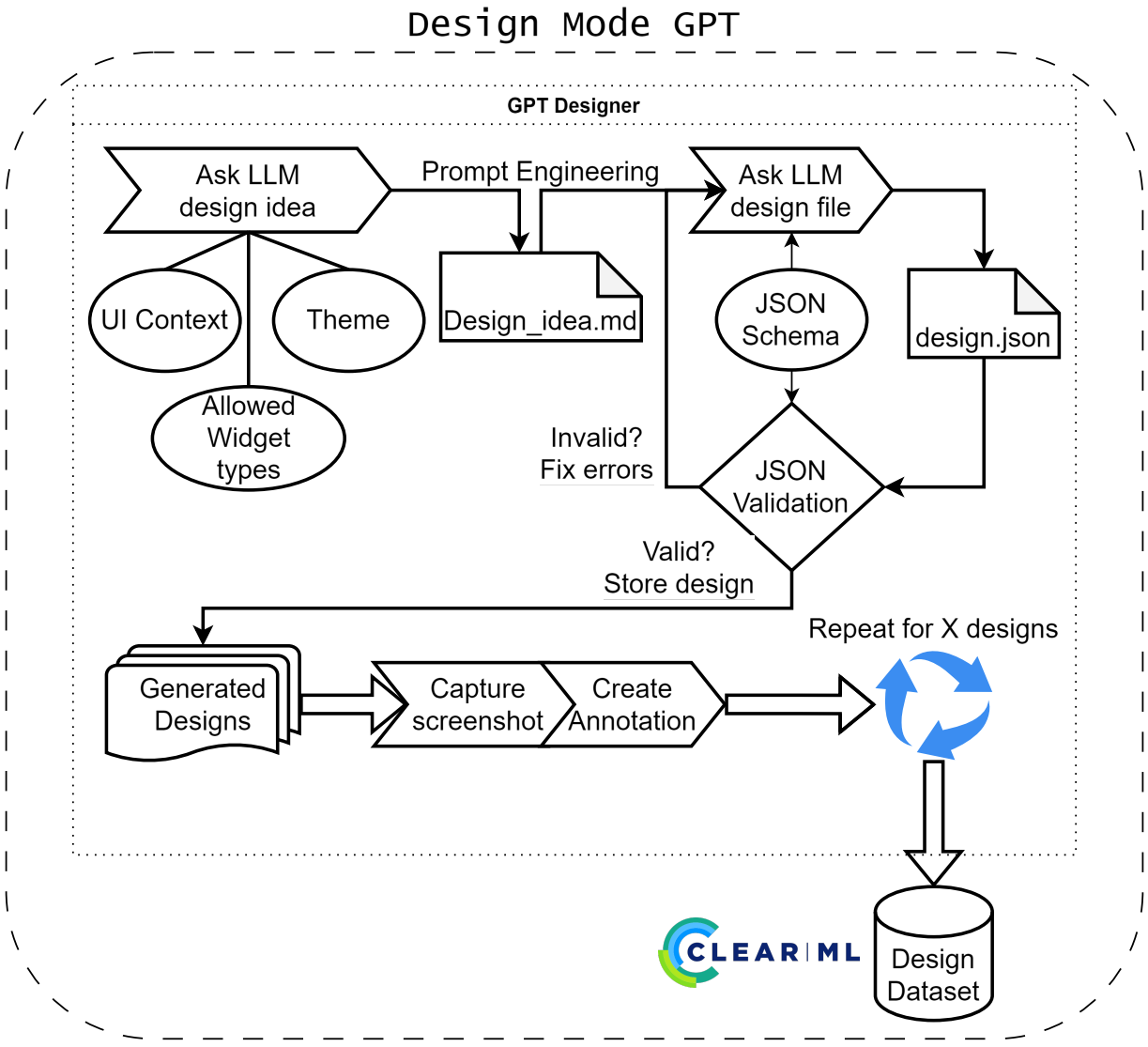


Figure 10: Dataset creation concept using design mode with gpt



5.5.1 Design contexts and themes

Since creation of a large design dataset would additionally require the labor-intensive manual definition of a large set of designs, it was decided to use a LLM which would handle the creation of those design files. For that purpose, a list of 100x contexts and 100x themes was created. By unique pair combination, this creates a possible list of 10000 design contexts and themes to be provided to the LLM. In the thesis, it was decided to use gpt-4-turbo as the LLM, as it provided the easiest API to use and is generally known for its good performance.
As illustrated in Fig.10, the LLM would first be instructed to write out a design idea in the provided UI context and style theme. Via a system prompt, the model is instructed to include the widgets to be used in the design, a high-level description of the UI purpose and a structural description of what the UI shall look like.
The idea output is then again fed into the LLM in a separate context and instructed to output a valid JSON file. It additionally receives the JSON schema as guidance on how to structure and construct the output JSON. The resulting output of the model is then validated against the schema and any validation errors are fed back to the LLM for correction. Once an output design is valid, it will be stored and added to a design file list, to be generated in a following step. This loop can be repeated for a desired amount of design files, for up to 10000 generations, given our combination list of contexts and themes.

5.5.2 Third-party integration

The dataset creation process relies heavily on third-party integration. To track individual versions of datasets, a popular experiment tracker named ClearML was used. In order to automate the design file creation process, the proprietary LLM named ChatGPT was used via OpenAIs available python package and API [30].

Integration with ClearML

Whenever a dataset creation starts, a new ClearML task is created and used for generating statistics about the process. The task stores all of the following information:

- Console output of the script
- Scalar plots displaying generation statistics over iterations
- Metadata about dataset (file split of train,val,train)
- Artifacts (ZIP folder of dataset contents)
- Parameters of the script/task
- Machine environment information
- Version control information (diff, commit hash, ...)

All of that surrounding information regarding the creation process helps in getting a better understanding about errors and quality of the created dataset.

Integration with LLM (gpt-4-turbo)

In order to generate design files at a larger scope, OpenAIs gpt-4-turbo model was used in the dataset creation process when using the design mode of generator v2. In order for the model to properly behave and generate usable output, the generative pre-trained transformer (GPT) was instructed via system prompts. The prompt was designed, so that the format of the idea is structurally similar through each iteration and proper output from the JSON generation step can be expected. The used system prompt for the design JSON can be seen in Listing 7. Due to their length, the system prompt for the design idea and corresponding examples are omitted from the paper, but can be viewed in the source code repository [31].

```
You are a UI generator.
Your goal is to create a new single window UI using a specialized JSON format.
The format specification is available in the design.schema.json file below.
Follow the provided design guideline of the user when replicating the design idea
↪  using the structure of the JSON format.
Always output a valid JSON object that represents the UI design.

You ALSO MUST adhere to the following SPECIAL RULES:
- Never use 'grid' container. Use either 'none' or 'flex' container.
- Containers must be used to set style for the whole window or the specific group
↪  of widgets (such as background color and other general styles).
- Each widget placed inside a 'none' container must have an associated style
↪  defining its X and Y coordinates.
- Widgets may have multiple styles applied to them
- Window size MUST be 640 x 640 pixels
- You MUST make sure that coordinates of widgets do not overlap due to width and
↪  height of the widgets
- You MUST make sure that the widgets are within the window bounds (0, 0, 640,
↪  640)
```

Listing 7: System prompt for gpt-4-turbo generating design file (JSON)

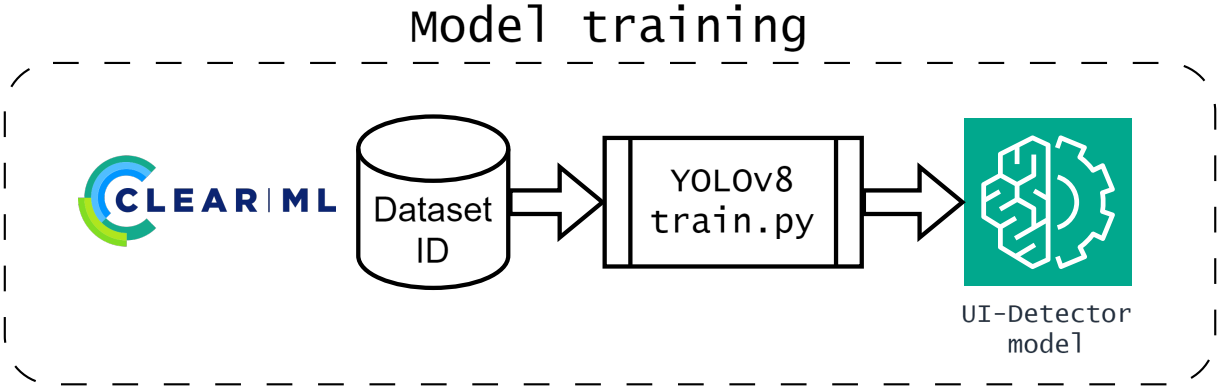
5.5.3 Randomizer script

In order to provide a randomizer script which does not rely on third-party integration, an extra module was created. The repository [32] contains scripts for calling both generator versions. These scripts also repeatedly call the generator and allow for manual dataset creation, without necessity of additional third-party services. An optional mechanism to upload the created dataset to ClearML afterwards, if desired by the user, is implemented in v2.

6 Training of YOLO model

The training process illustrated in Fig.11 uses the already built training mechanisms of ultralytics [17] YOLO model version 8.1 [33]. All experiments of the paper were performed using this model version and the specific python package version used was 8.1.47.
The model was trained on the detection and localization of widgets, to determine their bounding boxes. Through the bounding boxes, the model provides the necessary metadata for coordinates of widgets on a given screenshot.

Figure 11: YOLOv8 model training process



6.1 Training tasks in ClearML

Every training process of a YOLO model is automatically tracked via the existing integration of ClearML in the ultralytics YOLO engine [34]. This was a crucial step in the creation of reproducible results, since all hyper-parameters of the training process are stored and the used dataset is referenced by a unique ID. This also allows for later modification of a training process, by cloning of the training task and manual editing of parameters in the cloned task.

6.1.1 Training agents on local and remote machines

In order to train models remotely on a more powerful machine than the authors laptop, a ClearML agent was setup using Google Colab. The creators of ClearML provide a simple mechanism and already defined notebook [35] for starting such an agent. By having this agent readily available, any local or previously started training tasks could be remotely executed. The

built integration uses docker containers to execute individual tasks and execution was therefore immutable once finalized. Such an agent was also started on the local machine of the author, to compare results between the two machines and for debugging purposes of the pipeline itself.

6.2 Tuning for best hyperparameters

In order to optimize the training process for a given dataset to determine the best set of hyper-parameters, the existing tuning process from the ultralytics engine [36] is used. This tuning process has automatic optimization features, choosing different optimizer algorithms for different epochs.

6.3 Hyperparameter optimization

The tune process by ultralytics was not always best for producing results. In some cases, the process failed to record the output model weights, despite reporting good results, which led to the creation of a custom hyper-parameter optimization (HPO) pipeline.
ClearML offers mechanisms for performing HPO [37] on previously executed training tasks. For the optimization of parameters, the Optuna optimizer [38] is used, which offers an algorithm to optimize for a given metric using either discrete value ranges or a defined set of scalar values. The open-source optimizer implementation would then determine the best hyper-parameters for the training task by running it multiple times with a different set of parameters determined by the algorithm, while tracking the target performance metrics. In all HPO processes performed, the configured target metric was the mean average precision (mAP), which shows the model performance across different levels of detection difficulty. The result of this can be seen in Table 1. The author chose the mAP50-95 metric, as it provided a more comprehensive view of the models performance, which was representative of an overall better accuracy and recall.

6.3.1 Explanation of mAP metric

The YOLO training process provides two mAP metrics:

- mAP50 - Mean Average Precision at 50% Intersection over Union (IoU) threshold
- mAP50-95 - Mean Average Precision averaged over IoU thresholds from 50% to 95% (at 5% steps)

This value measures the accuracy of a model in detecting and localizing objects within an image. The value is calculated by evaluating the precision and recall of the model across different confidence thresholds and IoU thresholds.
Precision measures the accuracy of the positive predictions made by the model. It is the ratio of true positive predictions to the total number of positive prediction (total of both true positives

and false positives). A high value in precision indicates that the model makes very few false positive errors, meaning the predicted instances are actually positive.
Recall (otherwise known as sensitivity) measures the model's ability to correctly identify all positive instances. It is the ratio of true positive predictions to the total number of actual positive instances (total of both true positives and false positives). A high recall value indicates that the model makes very few false negative errors, meaning it correctly identifies most of the actual positive instances.
There exists a trade-off between those metrics, as an increase in precision can lead to a decrease in recall and vice versa. An increased threshold for classifying a positive prediction (i.e. to reduce false positives) might exclude some true positives, which reduces recall. This is illustrated in the precision-recall curve and the area under this curve (average precision) can be used to summarize the model's performance across different thresholds.
The mAP@50 refers to the mean average precision calculated at the IoU threshold of 50%. The mean is the average precision across all classes. The mAP@50-95 provides a more comprehensive evaluation, as it calculates the mean average precision across multiple thresholds in 5% increments, starting from 50% up to 95%.

6.4 Note on ClearML pipelines

The author calls the implementations in the UI detector repository pipelines, even though they are simply python scripts with a CLI. They are however implemented in the recommended pattern [39] that would later allow these individual scripts to be adapted to pipelines. This is also the reason that all imports that a function requires are located in the corresponding function body and not the top of the script. Pipelines were however not used during paper development, as they require a bit more implementation to work properly and it was considered out-of-scope for this paper to do so.

7 Research results

The final project resulted in the creation of a custom dataset generator capable of producing UI screenshot images with corresponding widget annotations. It provides two modes of operation, which can be used to create datasets of varying image sizes, styling properties and used widget types. In addition to that, a JSON design parser was written, which converts a design specification written in accordance to a JSON schema into an image representing a realistically looking UI. Multiple pipelines with integration to ClearML were written, to aid in dataset generation, YOLO model training and hyper-parameter optimization (HPO).

7.1 Model performance

The created generators were used in the research to train a baseline model based on a random dataset. The performance of this baseline model showed such good results, that the author doubted the viability in real-world scenarios, since the resulting metric scores showcased a form of over-optimization for the case of detecting isolated widget types on a white background. In response to that, another dataset using the implemented design mode was created with the help of an LLM to produce the necessary design files. The main purpose of the design mode is to create more realistic looking UIs, with an overall theme applied in the form of styles, which got rid of the white background and allowed the widgets to blend in more within the window.
The improved model was trained on the generated design dataset and shows a decrease in performance when compared to the baseline model. The drop in performance is attributed to the increased difficulty for the model when detecting widgets on colored and styled backgrounds, which slightly blends the visible features of individual widgets with their surroundings, making them harder to detect.

Dataset	mAP50	mAP50-95	model_variant
Train (random)	**0.95312**	**0.86023**	yolov8n
Train (design)	0.77673	0.74507	yolov8n

Table 1: Training results with random & design dataset

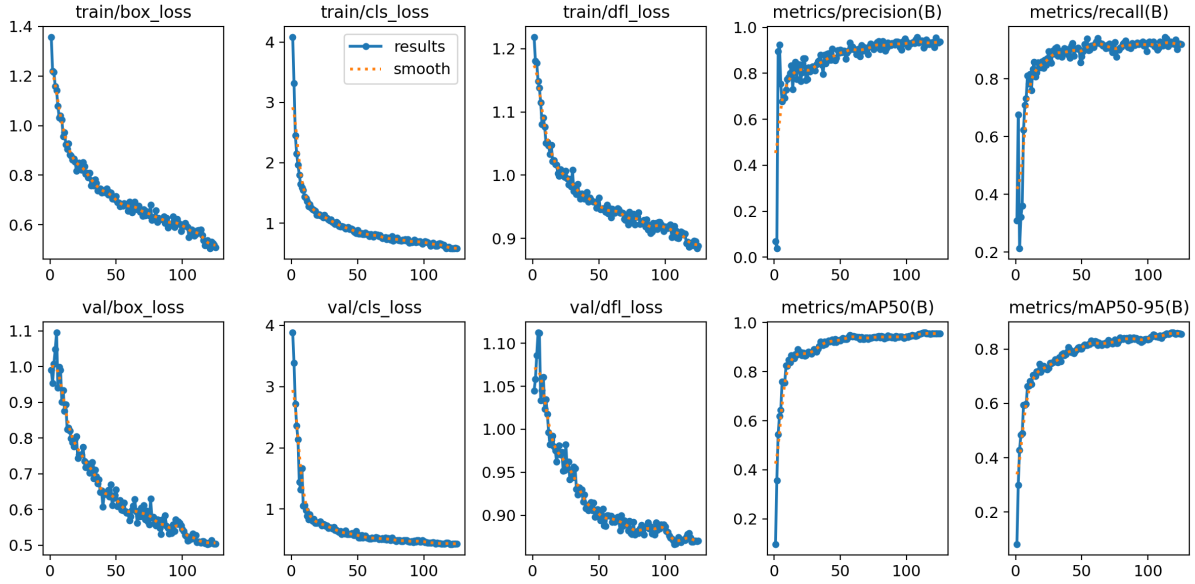
The results show the mean average precision of the model over all widget classes. The calculation of this value is described in section 6.3. Visualizations of training results for the random dataset are depicted in Fig.12,14,16 and 18. Further training visualizations for the design dataset

are depicted in Fig.13,15,17 and 19. They are interpreted and described in the following section 7.1.1

7.1.1 Interpretation of performance results

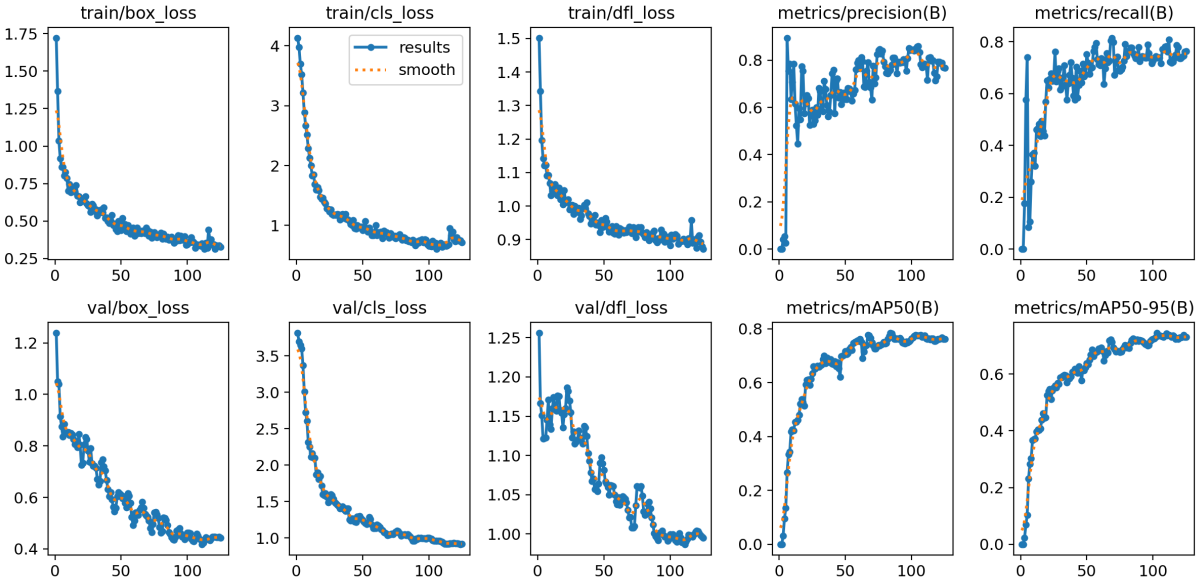
In both trained models the training results show an increase in the mAP value over the whole process. The random model results (Fig.12) generally show good performance with little variation, meaning the model gets more confident over time. This behaviour is to be expected, since the random dataset contains images of best-case scenarios where individual widgets are isolated in position and their characteristics are more easily differentiated.

Figure 12: Random dataset training - results



This is not the case for the design model results (Fig.13), as it shows trouble of properly identifying widgets over time, visible in the big bounces of the precision and recall values even at higher epochs.

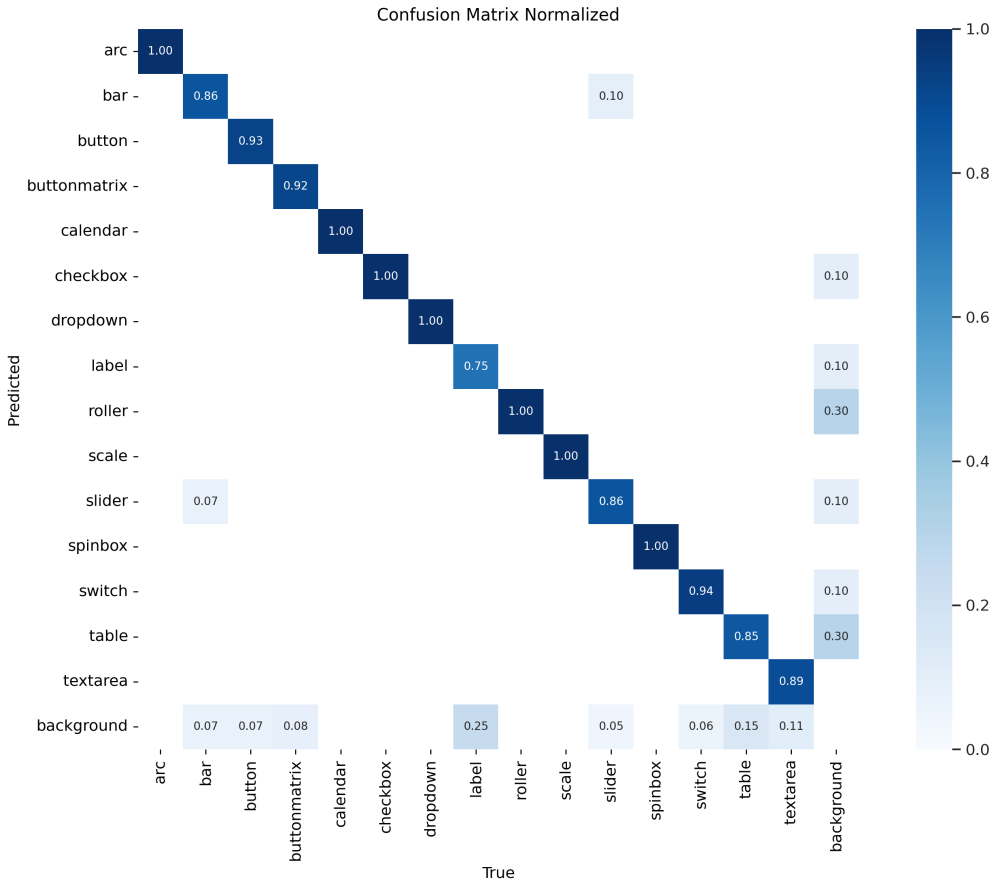
Figure 13: Design dataset training - results



The confusion matrix showcases the model's predictions compared to the actual labels. A good performing model will show a diagonal line across the table, meaning the model is capable of predicting the correct classes in the true positive cases. If the model makes mistakes, the table will show the confusion of a predicted label versus what the label actually should have been. The normalized version of the table shows this value in a range of 0 to 1, instead of the actual amount of predictions.

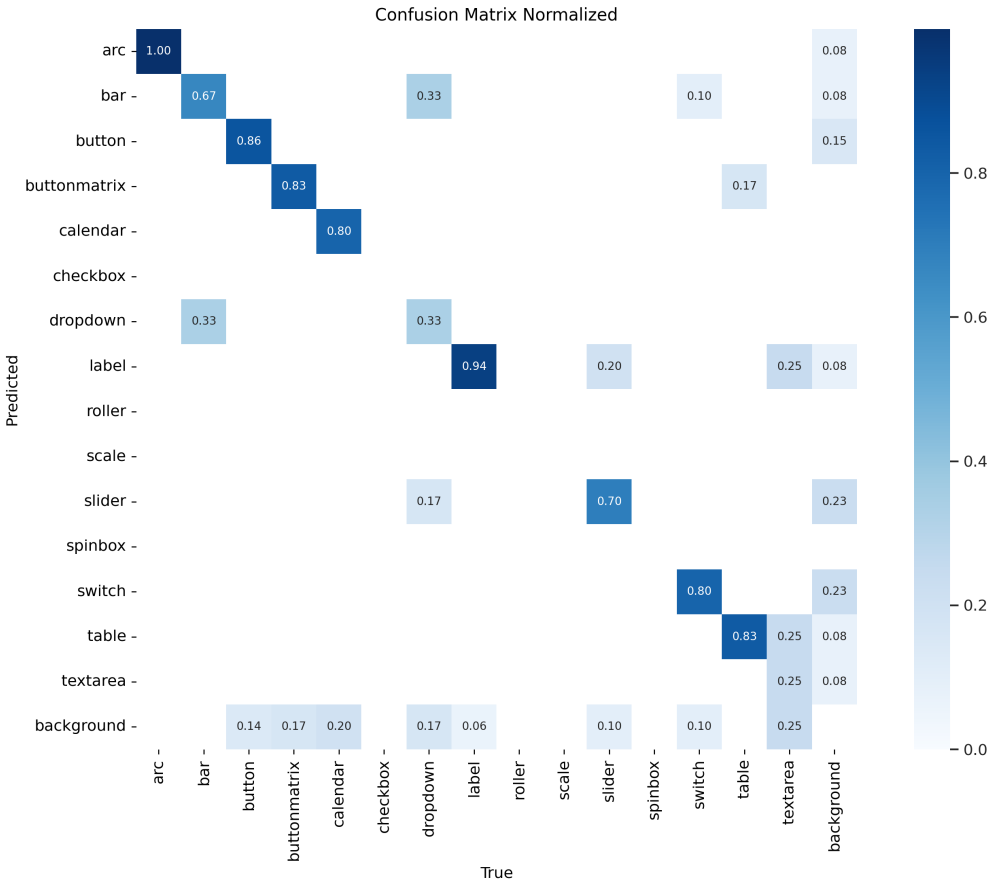
In our random dataset, we see that the model performs well and predicts the correct label in most cases (Fig.14). The confusion is mostly with the background of the image, meaning there was no class to predict in that part of the image.

Figure 14: Random dataset training - confusion matrix (normalized)



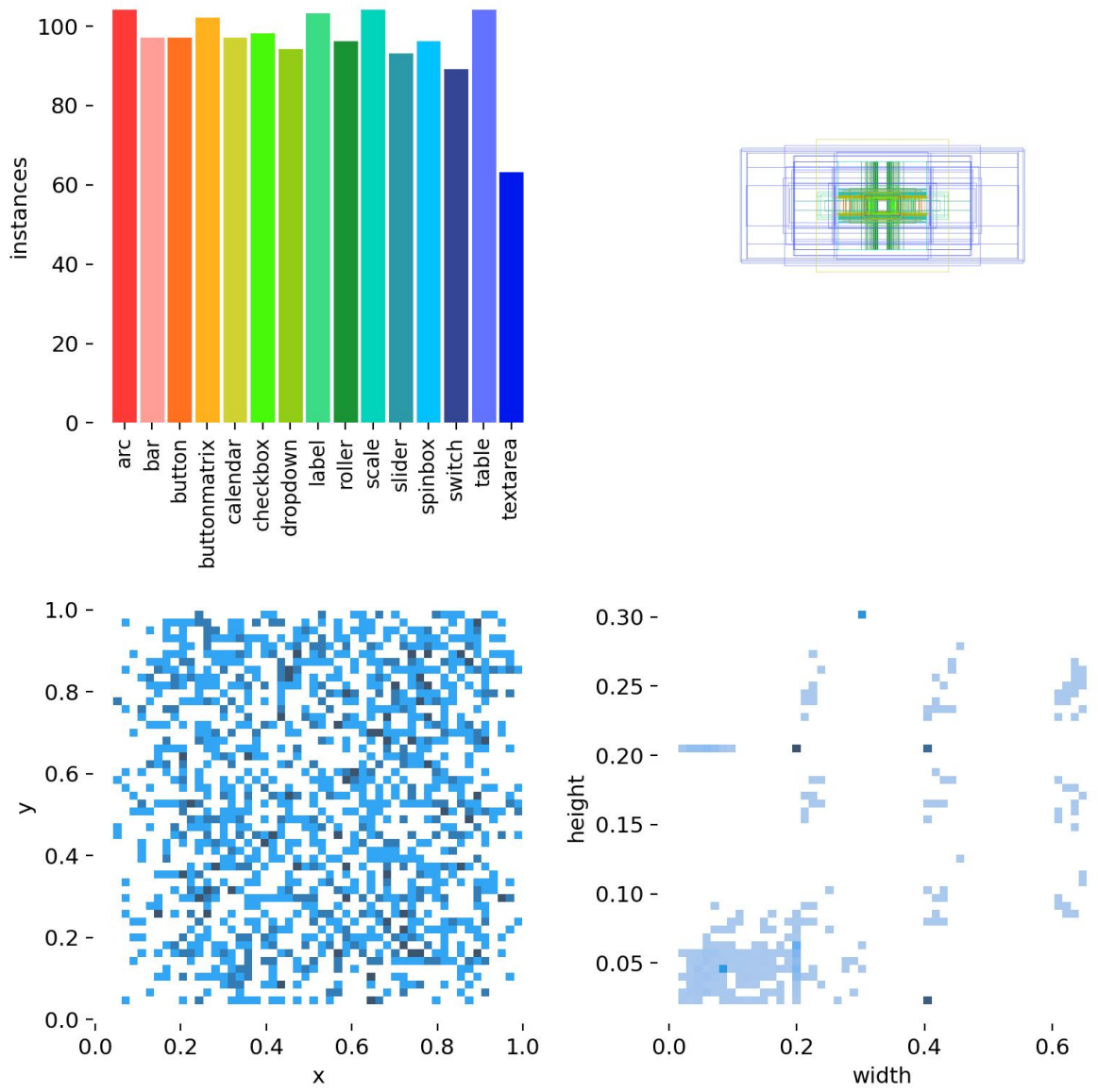
This is not the case in the model trained on the design dataset, as it predicts wrong labels for some classes. Due to the under-representation or missing of certain classes in the dataset, this behaviour can be expected. It means we require a generally bigger dataset with more balance.

Figure 15: Design dataset training - confusion matrix (normalized)



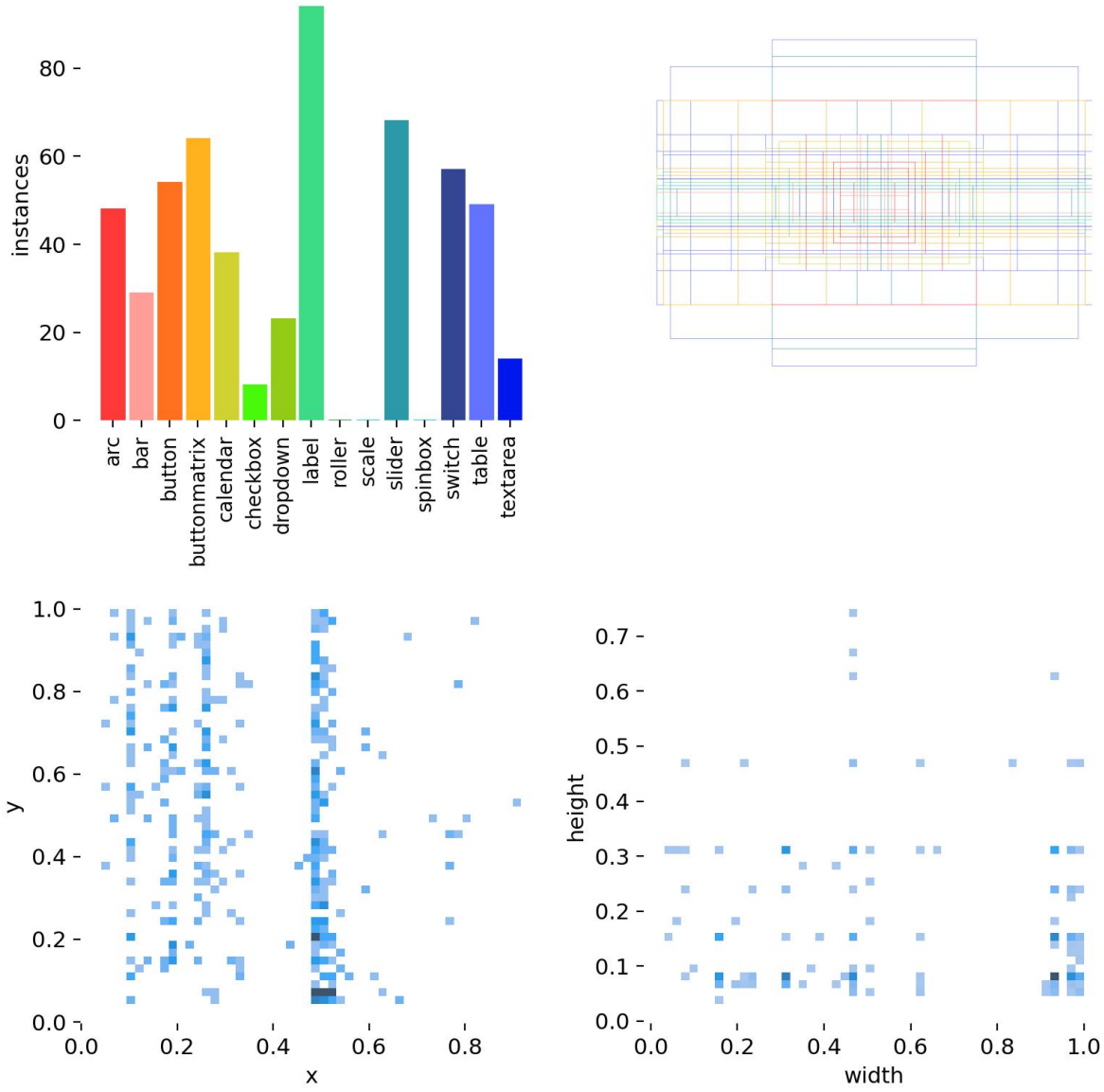
In Fig.16, the distribution of labels (upper-left) in the random dataset can be seen, which shows a general good balance across all classes with the exception of the textarea. This widget often got placed out-of-bounds, resulting in the removal of the label and thus is under-represented in comparison. The visualization further shows the distribution of bounding boxes (upper-right) and their size in comparison to different classes. The object location scatter plot (lower-left) represents the locations of widgets within the image dataset. The x and y correspond to the normalized coordinates of the object centers within the image. It showcases if widgets tend to cluster in certain regions or are evenly distributed in spatial position. For the random dataset, it illustrates that the random absolute positioning produces a mostly even distribution in placement. The scatter plot of object sizes (lower-right) shows the distribution of object sizes in terms of their width and height, which are also normalized. Each point represents the size of an object and showcases if the dataset contains a wide range of object sizes or if they have a tendency of similar dimensions. It showcases, that in the random dataset the size of widgets are clustering. This is to be expected, as size variation is not implemented and has need for improvement.

Figure 16: Random dataset training - label distribution



For the design dataset, the distribution (Fig.17) is not as good. An improvement of the design generation process with the LLM is required, to better enforce certain classes to be used. More balance is required, which means the LLM needs to be informed to not over-use certain types. The removal of types from the prompt once a threshold is met could improve the dataset creation output. It also showcases, that widgets are positioned in a more structural approach, as the position scatter plot forms a line in various columns. The sizes of widgets and their bounding boxes are better distributed in the design dataset than in the random dataset, since variation in size was possible for the design generation process.

Figure 17: Design dataset training - label distribution

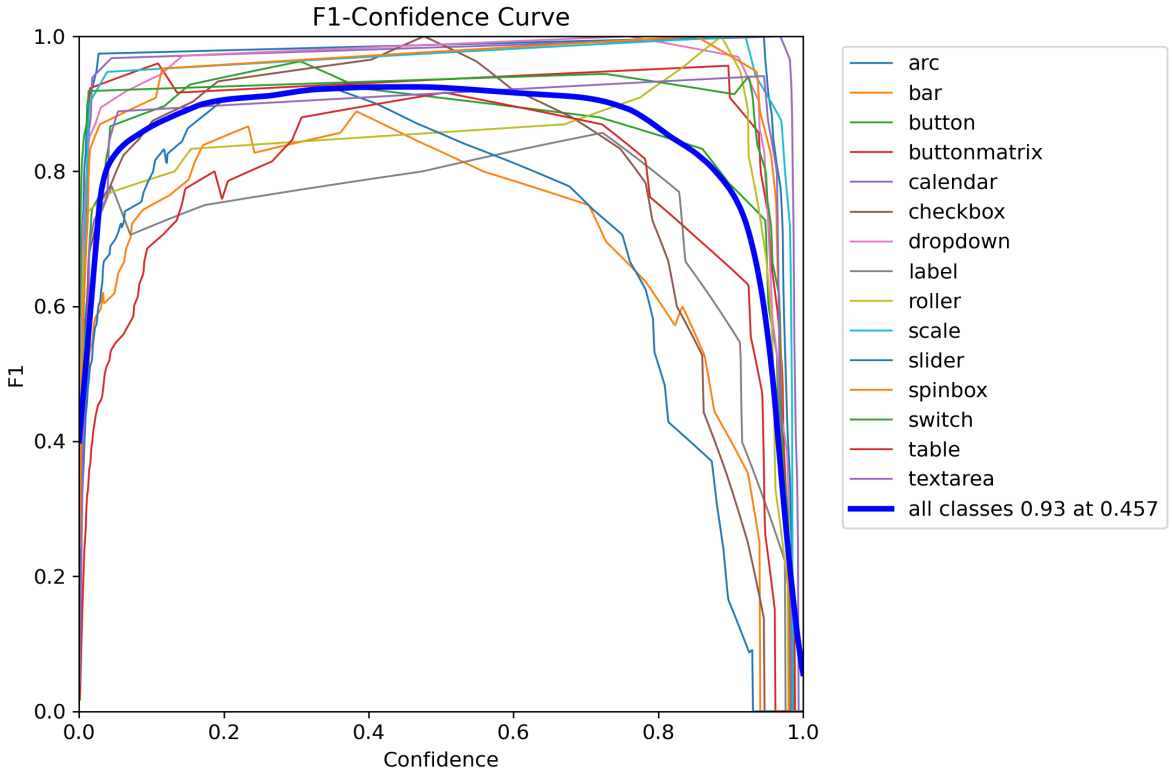


Another metric is the F1 score, which calculates the harmonic mean of precision and recall. It better represents model performance in imbalanced datasets. The metric can showcase the balance of class distribution and corresponding prediction confidence. Additionally, the F1 score can showcase which class types result in a decrease of the mAP value.

For the random dataset, the F1 curve (Fig.18) shows an overall good confidence in widget types. Bad results originate from the under-represented text area, as well as some other types (e.g. roller, spinbox, table). The confidence curve indicates that these widget types need to be improved in the generator, so that their representation is better identified and differentiated from other types. The curve also shows, that widgets like arc, button, buttonmatrix and more work

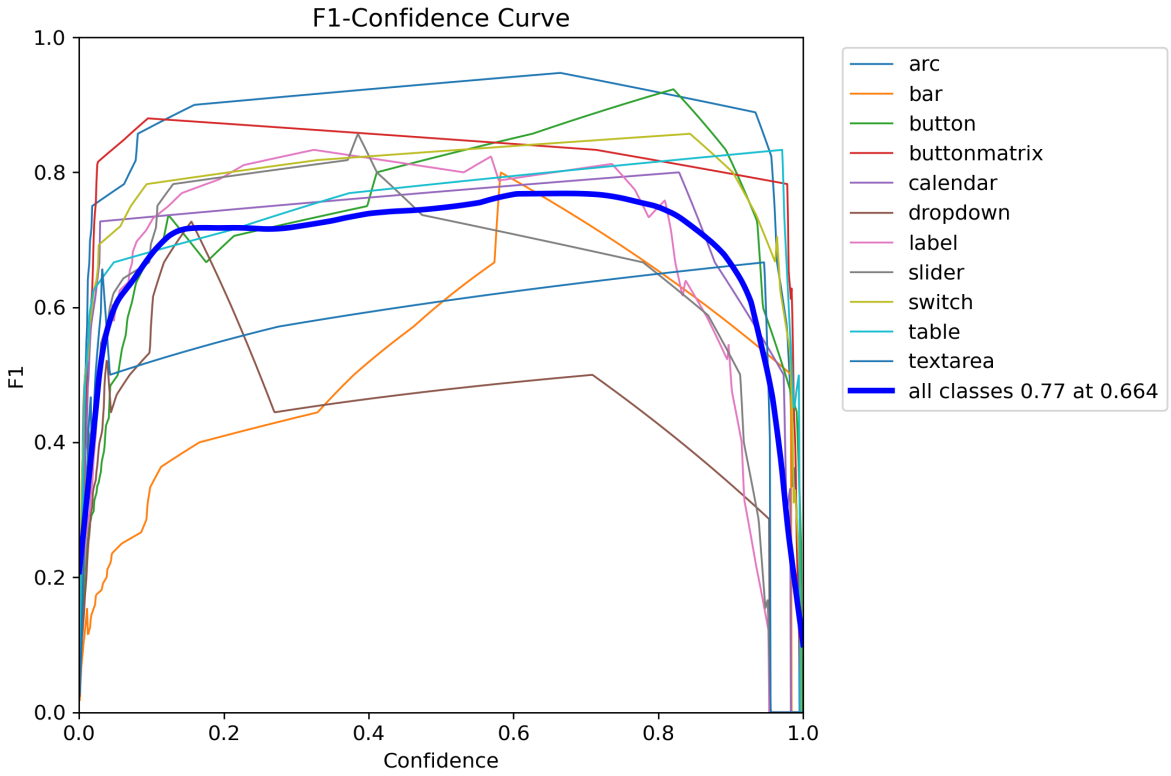
very well for the model, indicating that usage of those types more likely yields good results for the purpose of automating widget detection in a test automation process.

Figure 18: Random dataset training - F1 curve



The curve (Fig.19) of the model trained on the design dataset is much lower in the harmonic mean. This can be attributed to much worse detection in widget types, due to their under-representation in the dataset. However, the curve showcases better performance in the same widget types of the random dataset, further indicating that a subset of widget types is more suitable for detection in test automation processes.

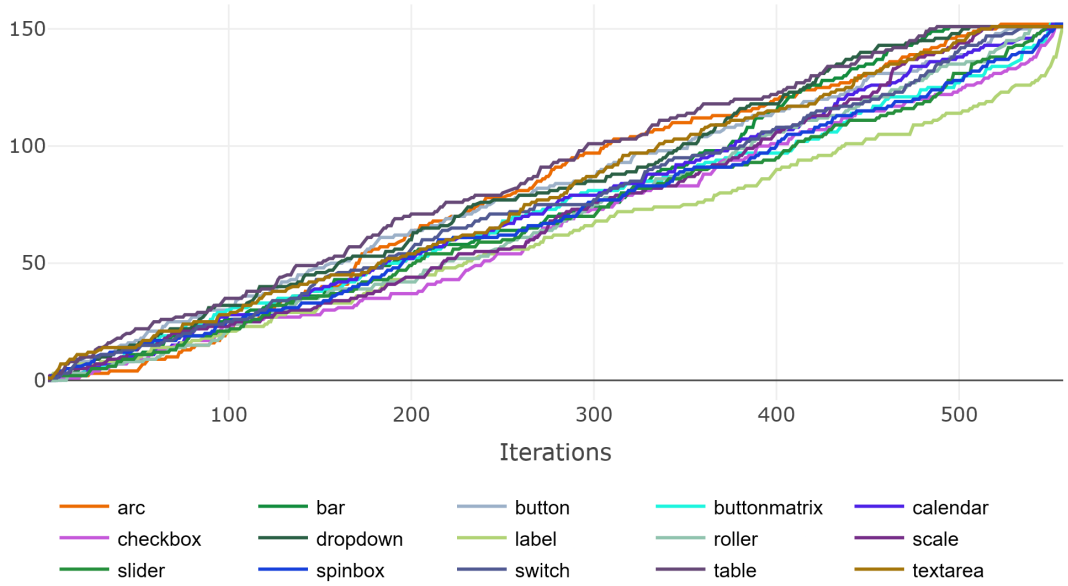
Figure 19: Design dataset training - F1 curve



7.1.2 Random based dataset

The used random dataset contains 15 different widget types of LVGL in varying positions and widget styles (excluding container style), where each widget is represented about 150 times, as showcased in Fig.20. The total amount of widgets in the dataset is 2250, spread over 406 image files. In the dataset, each image contained 6 widget instances. The dataset is uploaded to the repository of the paper [40] and can be referenced by ID `04da75baa7084aee83f3b31602c408c2`.

Figure 20: Generator v2 iteration metrics for random dataset



7.1.3 Design based dataset

The design dataset has 145 design files. Not all designs could be generated, so only an output of 132 generated images was achieved. Due to the described distortion issue in the generator, the dataset had to be manually corrected by removing broken images and adding in replacements of other design generations. The corrected design dataset contains 186 images.
Since the process of adding in images was performed manually, an exact widget count cannot be given, but is estimated to contain roughly 550 widgets. In Fig.17 the label distribution of the dataset can be seen.
Certain widget types were definitely more preferred by the LLM, while three (roller, scale, spinbox) are not represented at all. For the design mode it is inherently more difficult to create a balance in distribution, as certain widget types like buttons or labels will always be overrepresented when trying to create realistic looking UI. The original design dataset with some image distortions is referenced by ID `5c8e248bdcc24ee2bb57760f0961c690`. The manually corrected version is referenced by ID `2fe5e8b06a6249fcb9360c9645e0f7a6`. Both are uploaded to the repository of the paper [40].

7.2 Impact of variation

An important factor discovered during the research is the importance of variation when training a YOLO model on UI image data. Given that in the real-world UIs need to appeal to humans and are generally conforming to certain visual expectations, it is important to incorporate many variations (style, size, placement, nesting, ...) into training datasets. This avoids over-optimized datasets illustrating best-case detection scenarios where individual widget features would be more easily detected by the model. Incorporating themes and contexts into artificially generated UI brings the benefit of creating datasets, that are more aligned with worst-case detection scenarios, where individual widgets need to be differentiated with better precision. It is recommended to experiment with all kinds of visual widget properties when creating such datasets and training future models in the context of UI widget detection.

8 Discussion

While the results of the model on the synthesized datasets looks promising, they are far from being usable when used on real data. The output models of both final runs were used in a prediction test against design mock-ups of a new UI design. The test showed, that complex types (nested widgets) are confused with other simple types like switch that are looking similar. The additional usage of different fonts and also symbols made the model confuse simple buttons with types like arc. In the real data test, it was determined that the models perform much worse or not at all when faced with complexity in widget style, nested widgets, used symbols and fonts. This is largely attributed to the fact, that this complexity is not represented in the training datasets. They require additional implementation and improvement in the generation process for better model results on real data. In the test, both models were not able to properly predict most widgets. The most promising type in those test results was the switch and button, which was not only often correctly identified but also with a much higher confidence than the rest. One has to keep in mind that these proprietary designs are far more advanced than what the generator could provide, so bad results were expected. It was however an indicator as to what further improvements are most necessary and are aligned with the results determined with the artificial UIs datasets. Due to company secrets, it was not possible to showcase those test images in the thesis.
In regards to the stated research questions the viability of the ML object detection approach is reasonably possible, even though the definitive answer in the real test is negative. By further improving the dataset generation with more complex features and variation, it is anticipated that model performance will increase on real data. Further development in the created generators is necessary, which will be conducted at the discretion of Schrack Seconet.

Bibliography

[1] "LVGL - Light and Versatile Embedded Graphics Library." (), [Online]. Available: https://lvgl.io/ (visited on 01/30/2024).

[2] "Visocall IP | Moderne IP Kommunikationslösung," Schrack Seconet AG. (), [Online]. Available: https://www.schrack-seconet.com/de/healthcare/kommunikationssystem-visocall-ip/ (visited on 01/30/2024).

[3] "IrfanView - Official Homepage - One of the Most Popular Viewers Worldwide." (), [Online]. Available: https://www.irfanview.com/ (visited on 01/30/2024).

[4] "DIN VDE 0834-1 VDE 0834-1:2016-06 - Normen - VDE VERLAG." (), [Online]. Available: https://www.vde-verlag.de/normen/0800319/din-vde-0834-1-vde-0834-1-2016-06.html (visited on 01/30/2024).

[5] V. Garousi, M. Felderer, Ç. M. Karapıçak, and U. Yılmaz, "Testing embedded software: A survey of the literature," *Information and Software Technology*, vol. 104, pp. 14–45, Dec. 2018, ISSN: 09505849. DOI: 10.1016/j.infsof.2018.06.016. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584918301265 (visited on 01/24/2024).

[6] Y.-D. Lin, E. T.-H. Chu, S.-C. Yu, and Y.-C. Lai, "Improving the Accuracy of Automated GUI Testing for Embedded Systems," *IEEE Software*, vol. 31, no. 1, pp. 39–45, Jan. 2014, ISSN: 0740-7459. DOI: 10.1109/MS.2013.100. [Online]. Available: http://ieeexplore.ieee.org/document/6576113/ (visited on 01/25/2024).

[7] M. D. Altinbas and T. Serif, "GUI Element Detection from Mobile UI Images Using YOLOv5," in *Mobile Web and Intelligent Information Systems*, I. Awan, M. Younas, and A. Poniszewska-Marańda, Eds., vol. 13475, Cham: Springer International Publishing, 2022, pp. 32–45, ISBN: 978-3-031-14390-8 978-3-031-14391-5. DOI: 10.1007/978-3-031-14391-5_3. [Online]. Available: https://link.springer.com/10.1007/978-3-031-14391-5_3 (visited on 01/24/2024).

[8] J. Cheng and W. Wang, "Mobile Application GUI Similarity Comparison Based on Perceptual Hash for Automated Robot Testing," in *2021 International Conference on Intelligent Computing, Automation and Applications (ICAA)*, Nanjing, China: IEEE, Jun. 2021, pp. 245–251, ISBN: 978-1-66543-730-1. DOI: 10.1109/ICAA53760.2021.00052. [Online]. Available: https://ieeexplore.ieee.org/document/9653505/ (visited on 01/24/2024).

[9] Y. Li, G. Li, L. He, J. Zheng, H. Li, and Z. Guan. “Widget Captioning: Generating Natural Language Description for Mobile User Interface Elements.” arXiv: 2010.04295 [cs]. (Oct. 8, 2020), [Online]. Available: http://arxiv.org/abs/2010.04295 (visited on 01/24/2024), preprint.

[10] B. Selcuk and T. Serif, “A Comparison of YOLOv5 and YOLOv8 in the Context of Mobile UI Detection,” in *Mobile Web and Intelligent Information Systems*, M. Younas, I. Awan, and T.-M. Grønli, Eds., vol. 13977, Cham: Springer Nature Switzerland, 2023, pp. 161–174, ISBN: 978-3-031-39763-9 978-3-031-39764-6. DOI: 10.1007/978-3-031-39764-6_11. [Online]. Available: https://link.springer.com/10.1007/978-3-031-39764-6_11 (visited on 01/24/2024).

[11] T. Zhang, Y. Liu, J. Gao, L. P. Gao, and J. Cheng, “Deep Learning-Based Mobile Application Isomorphic GUI Identification for Automated Robotic Testing,” *IEEE Software*, vol. 37, no. 4, pp. 67–74, Jul. 2020, ISSN: 0740-7459, 1937-4194. DOI: 10.1109/MS.2020.2987044. [Online]. Available: https://ieeexplore.ieee.org/document/9064552/ (visited on 01/24/2024).

[12] T. Zhang, Z. Su, J. Cheng, F. Xue, and S. Liu, “Machine vision-based testing action recognition method for robotic testing of mobile application,” *International Journal of Distributed Sensor Networks*, vol. 18, no. 8, p. 155 013 292 211 153, Aug. 2022, ISSN: 1550-1329, 1550-1477. DOI: 10.1177/15501329221115375. [Online]. Available: http://journals.sagepub.com/doi/10.1177/15501329221115375 (visited on 01/24/2024).

[13] S. N. Cavsak, A. Deliahmetoglu, B. T. Ay, and S. Tanberk, “Cavsak et al. - GUI Component Detection Using YOLO and Faster-RCNN.pdf,”

[14] B. Deka, Z. Huang, C. Franzen, *et al.*, “Rico: A Mobile App Dataset for Building Data-Driven Design Applications,” in *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, Québec City QC Canada: ACM, Oct. 20, 2017, pp. 845–854, ISBN: 978-1-4503-4981-9. DOI: 10.1145/3126594.3126651. [Online]. Available: https://dl.acm.org/doi/10.1145/3126594.3126651 (visited on 01/25/2024).

[15] A. Gamal, R. Emad, T. Mohamed, O. Mohamed, A. Hamdy, and S. Ali, “Owl Eye: An AI-Driven Visual Testing Tool,” in *2023 5th Novel Intelligent and Leading Emerging Sciences Conference (NILES)*, Giza, Egypt: IEEE, Oct. 21, 2023, pp. 312–315, ISBN: 9798350381030. DOI: 10.1109/NILES59815.2023.10296575. [Online]. Available: https://ieeexplore.ieee.org/document/10296575/ (visited on 01/25/2024).

[16] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. “You Only Look Once: Unified, Real-Time Object Detection.” arXiv: 1506.02640 [cs]. (May 9, 2016), [Online]. Available: http://arxiv.org/abs/1506.02640 (visited on 01/24/2024), preprint.

[17] G. Jocher, A. Chaurasia, and J. Qiu, *Ultralytics YOLO*, version 8.0.0, Jan. 2023. [Online]. Available: https://github.com/ultralytics/ultralytics (visited on 01/25/2024).

[18] “Brief summary of YOLOv8 model structure · Issue #189 · ultralytics/ultralytics,” GitHub. (), [Online]. Available: https://github.com/ultralytics/ultralytics/issues/189 (visited on 05/23/2024).

[19] “Widgets — LVGL documentation.” (), [Online]. Available: https://docs.lvgl.io/master/widgets/index.html (visited on 05/24/2024).

[20] L. V. G. L. LLC. “Live demos - Test LVGL in your browser,” LVGL. (), [Online]. Available: https://lvgl.io/demos (visited on 05/24/2024).

[21] *Lvgl/lv_port_pc_vscode*, LVGL, May 20, 2024. [Online]. Available: https://github.com/lvgl/lv_port_pc_vscode (visited on 05/23/2024).

[22] *100askTeam/lv_lib_100ask*, 100askTeam, May 23, 2024. [Online]. Available: https://github.com/100askTeam/lv_lib_100ask (visited on 05/23/2024).

[23] “Lvgl/lv_binding_micropython: LVGL binding for MicroPython.” (), [Online]. Available: https://github.com/lvgl/lv_binding_micropython/tree/master (visited on 05/23/2024).

[24] “Lvgl/lv_micropython: Micropython bindings to LVGL for Embedded devices, Unix and JavaScript.” (), [Online]. Available: https://github.com/lvgl/lv_micropython (visited on 05/23/2024).

[25] “Flat/flat/jpeg.py at master · xxyxyz/flat,” GitHub. (), [Online]. Available: https://github.com/xxyxyz/flat/blob/master/flat/jpeg.py (visited on 05/23/2024).

[26] *Libjpeg-turbo/libjpeg-turbo*, libjpeg-turbo, May 23, 2024. [Online]. Available: https://github.com/libjpeg-turbo/libjpeg-turbo (visited on 05/23/2024).

[27] “How can I store a JPG using micropython and LVGL snapshot? - Micropython,” LVGL Forum. (Mar. 17, 2024), [Online]. Available: https://forum.lvgl.io/t/how-can-i-store-a-jpg-using-micropython-and-lvgl-snapshot/15135 (visited on 05/23/2024).

[28] “Lv_style_gen.h — LVGL documentation.” (), [Online]. Available: https://docs.lvgl.io/9.1/API/misc/lv_style_gen.html# (visited on 05/23/2024).

[29] “Ui-detector/schema/design_file.schema.json at 74795557012d5750fa35f1ff5a774d3172c6e953 · HackXIt/ui-detector.” (), [Online]. Available: https://github.com/HackXIt/ui-detector/blob/74795557012d5750fa35f1ff5a774d3172c6e953/schema/design_file.schema.json (visited on 05/23/2024).

[30] *Openai/openai-python*, OpenAI, May 23, 2024. [Online]. Available: https://github.com/openai/openai-python (visited on 05/23/2024).

[31] “Ui-detector/src/generate.py at 74795557012d5750fa35f1ff5a774d3172c6e953 · HackXIt/ui-detector (GPT design system prompt).” (), [Online]. Available: https://github.com/HackXIt/ui-detector/blob/74795557012d5750fa35f1ff5a774d3172c6e953/src/generate.py#L612 (visited on 05/23/2024).

[32] “HackXIt/ui_randomizer at 14d441c38a37850f4809471124bea3c36b9bc0b7,” GitHub. (), [Online]. Available: https://github.com/HackXIt/ui_randomizer (visited on 05/23/2024).

[33] “Ultralytics/ultralytics at v8.1.0.” (), [Online]. Available: https://github.com/ultralytics/ultralytics/tree/v8.1.0 (visited on 05/23/2024).

[34] Ultralytics. “ClearML.” (), [Online]. Available: https://docs.ultralytics.com/integrations/clearml (visited on 05/23/2024).

[35] “ClearML Agent on Google Colab | ClearML.” (), [Online]. Available: https://clear.ml/docs/latest/docs/guides/ide/google_colab/ (visited on 05/23/2024).

[36] Ultralytics. “Tuner.” (), [Online]. Available: https://docs.ultralytics.com/reference/engine/tuner (visited on 05/23/2024).

[37] “Hyperparameter Optimization | ClearML.” (), [Online]. Available: https://clear.ml/docs/latest/docs/fundamentals/hpo (visited on 05/23/2024).

[38] “Optuna - A hyperparameter optimization framework,” Optuna. (), [Online]. Available: https://optuna.org/ (visited on 05/23/2024).

[39] “PipelineDecorator | ClearML.” (), [Online]. Available: https://clear.ml/docs/latest/docs/pipelines/pipelines_sdk_function_decorators (visited on 05/23/2024).

[40] “Release Final paper results · HackXIt/ui-detector-paper,” GitHub. (), [Online]. Available: https://github.com/HackXIt/ui-detector-paper/releases/tag/final-paper-results (visited on 05/24/2024).

List of Figures

List of Tables

List of source codes

List of Abbreviations

API Application Programming Interface

CLI Command-line interface

CNN Convolutional Neural Network

FAS fire alarm system

GPT generative pre-trained transformer

GUI graphical user interface

HCS health care system

HPO hyper-parameter optimization

IoU Intersection over Union

JSON JavaScript Object Notation

LED Light Emitting Diode

LLM large language model

LVGL Light and Versatile Graphics Library

mAP mean average precision

MCU microcontroller unit

ML machine learning

SUT system under test

TA test automation

VCIP Visocall IP

VGT visual GUI testing

UI user interface

YOLO You only live once

YOLO You Only Look Once

YOLOv8 YOLO version 8 by ultralytics