Procedurally generated Phishing sites: Phase 1: Data Wrangling of website design

Qian ZhongJun

301541827

Instructor: Prof. Andrew H.

Simon Fraser University

October 2025

Abstract:

This report documents the development of a reproducible method for collecting design-structure data from websites. The process captures and quantifies band layout, element position, and adjacency to generate a dataset for a later Wave Function Collapse (WFC) layout generator. Several segmentation methods were tested and refined, from early computer-vision experiments to a final deterministic colour-encoding system. The report outlines each iteration, the reasoning behind design decisions, and how the resulting data pipeline now produces consistent, analyzable spatial maps of technical websites. Phase 1 establishes a validated framework for extracting the visual grammar of web design and prepares the dataset for rule-based procedural generation in later phases.

Introduction:

Modern websites share a narrow visual grammar: repeated grid systems, alignment rules, and familiar colour hierarchies. This consistency, while useful for usability, also allows imitation to appear authentic. The project therefore studies how ordinary site layouts can be abstracted, encoded, and regenerated to model the visual logic that underlies credibility.

Fifty technical and documentation-style websites form the reference corpus. Their clean, modular layouts make them ideal for isolating spatial patterns. Each page was processed into a heat-map dataset showing the frequency and relative placement of common interface elements. These maps will later be transformed into adjacency rules for a Wave Function Collapse (WFC) generator, enabling new but stylistically coherent layouts to emerge from data rather than design intuition.

The following sections document the methodological evolution, the automated capture pipeline, and the data's planned application in procedural layout generation.

Detection Methodology Evolution

Python.org serves as the running example because of its simple, document-like structure (Figure 1). The approach evolved through five stages, each addressing the weaknesses of the last.

Figure 1 Python.org reference site. Author's capture (2025).

Stage 1 OpenCV Segmentation

Early trials used the OpenCV computer-vision library to detect edges and contours. The intention was to segment headers, text blocks, and footers automatically. In practice, gradients and shadows produced noise, leading to overlapping or missing regions (Figure 2). The method could not yield reliable data for comparison.

Figure 2 OpenCV segmentation errors (Author, 2025).

Stage 2 Hybrid Color Assistance

Bright solid colours were added before analysis to give OpenCV sharper edges to follow. The change improved detection of small elements but destroyed the overall hierarchy: hundreds of tiny boxes replaced four or five structural bands (*Figures 3–4*). The experiment showed that contrast alone cannot represent layout meaning.

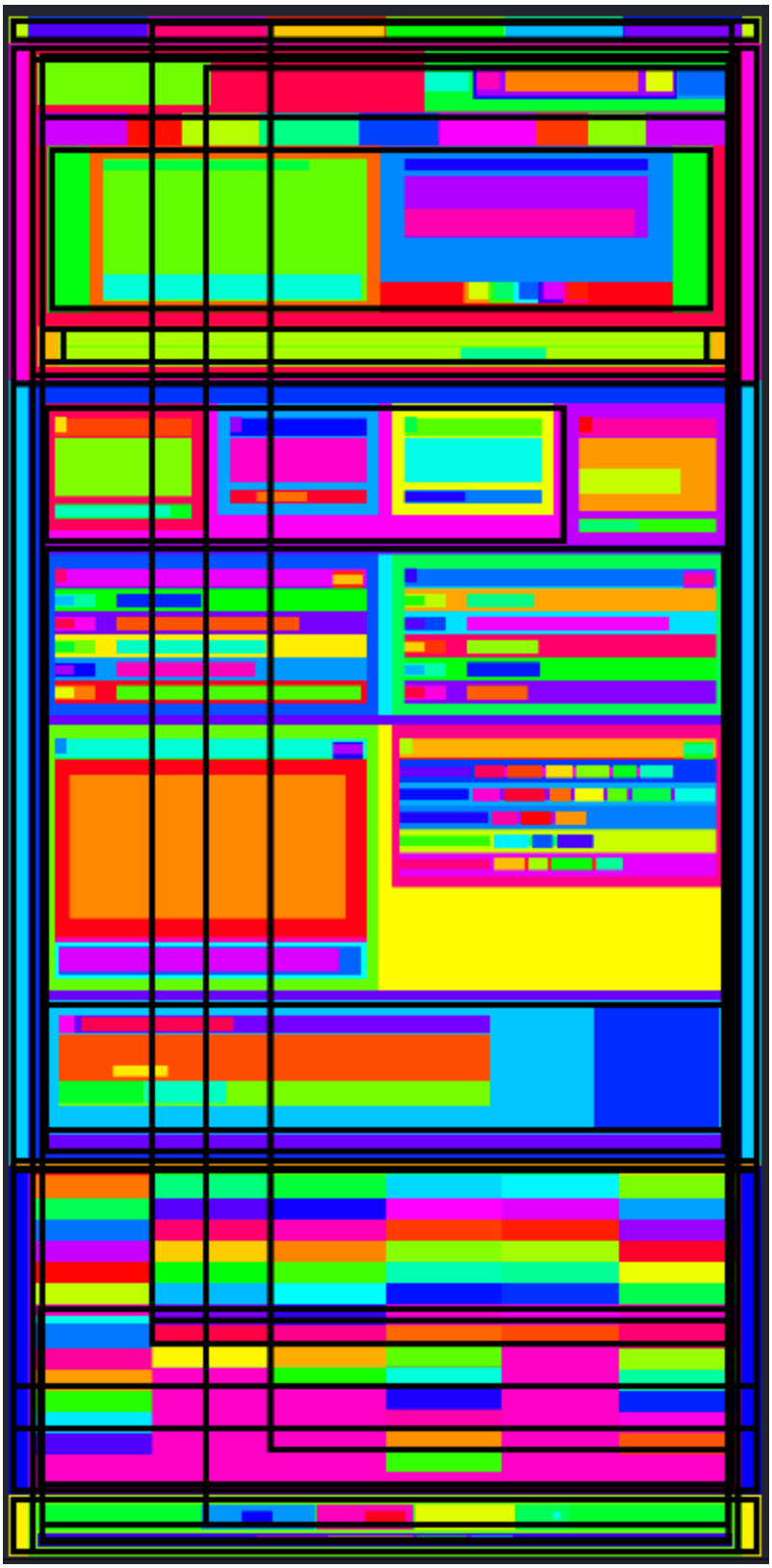

Figure 3 Hybrid colour-assisted detection results (Author, 2025).

Figure 4 Hybrid colour-assisted detection results per band(Author, 2025).

Stage 3 Random RGB Test

Computer vision was replaced by browser-side colour encoding. Each DOM element received a random RGB value, creating a perfect mosaic of unique regions (*Figure 5*). Yet because colours were random, no two sites could be compared—the same element type changed hue each time.

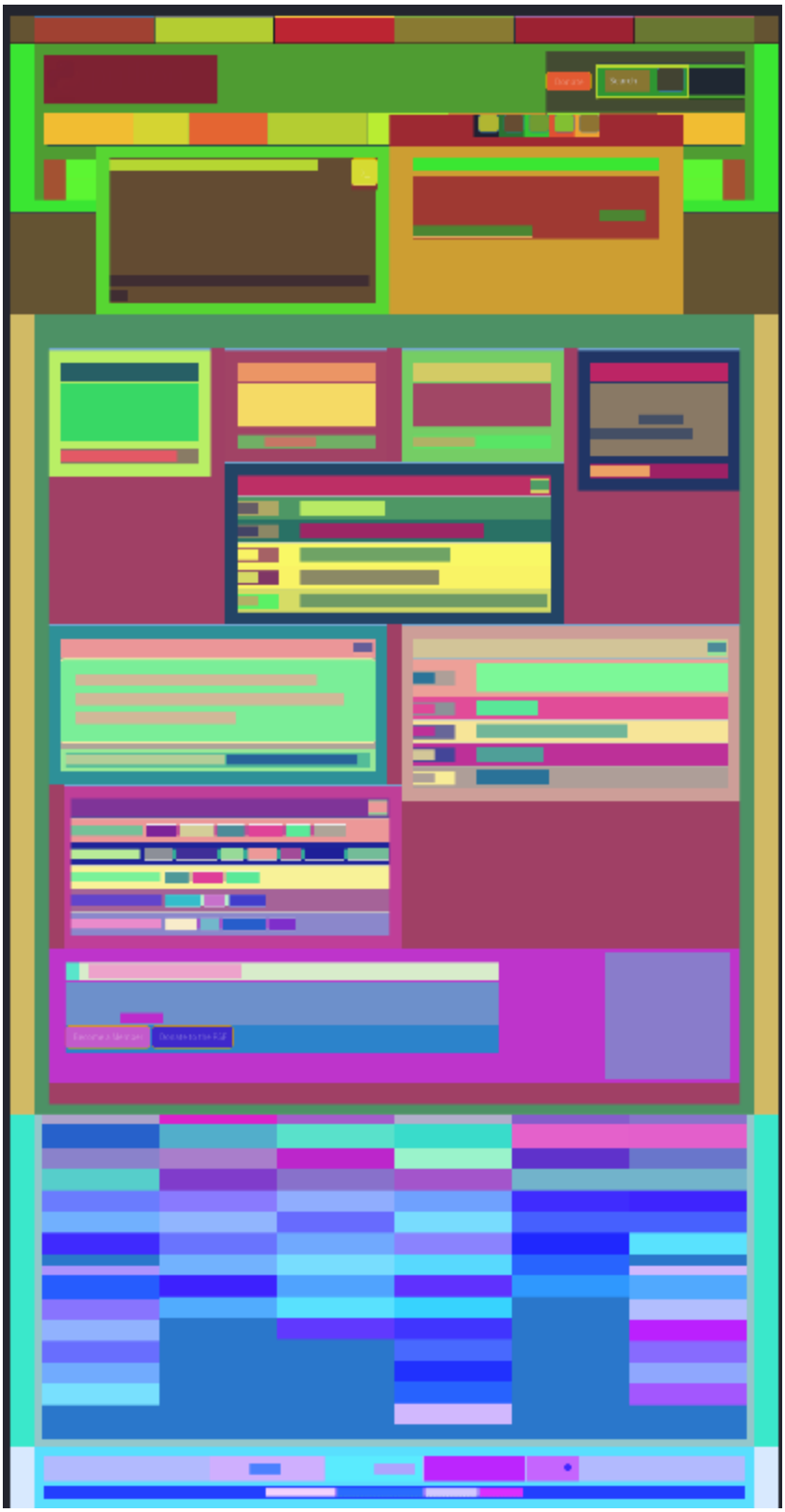

Figure 5 Random RGB encoding outputs (Author, 2025).

Stage 4 Deterministic RGB Encoding and Band Detection

A fixed mapping solved the consistency problem: red encodes element identity (DOM-path hash), green encodes semantic type (text, image, button, etc.), and blue later records vertical band. *Figures 6–8* illustrate element encoding, band map, and derived band mask. This deterministic system made results reproducible and comparable across all fifty sites.

Through these iterations, the project shifted from perception-based detection to structural encoding. Each failure clarified what was missing: stability, hierarchy, or meaning, and the final approach captured all three.

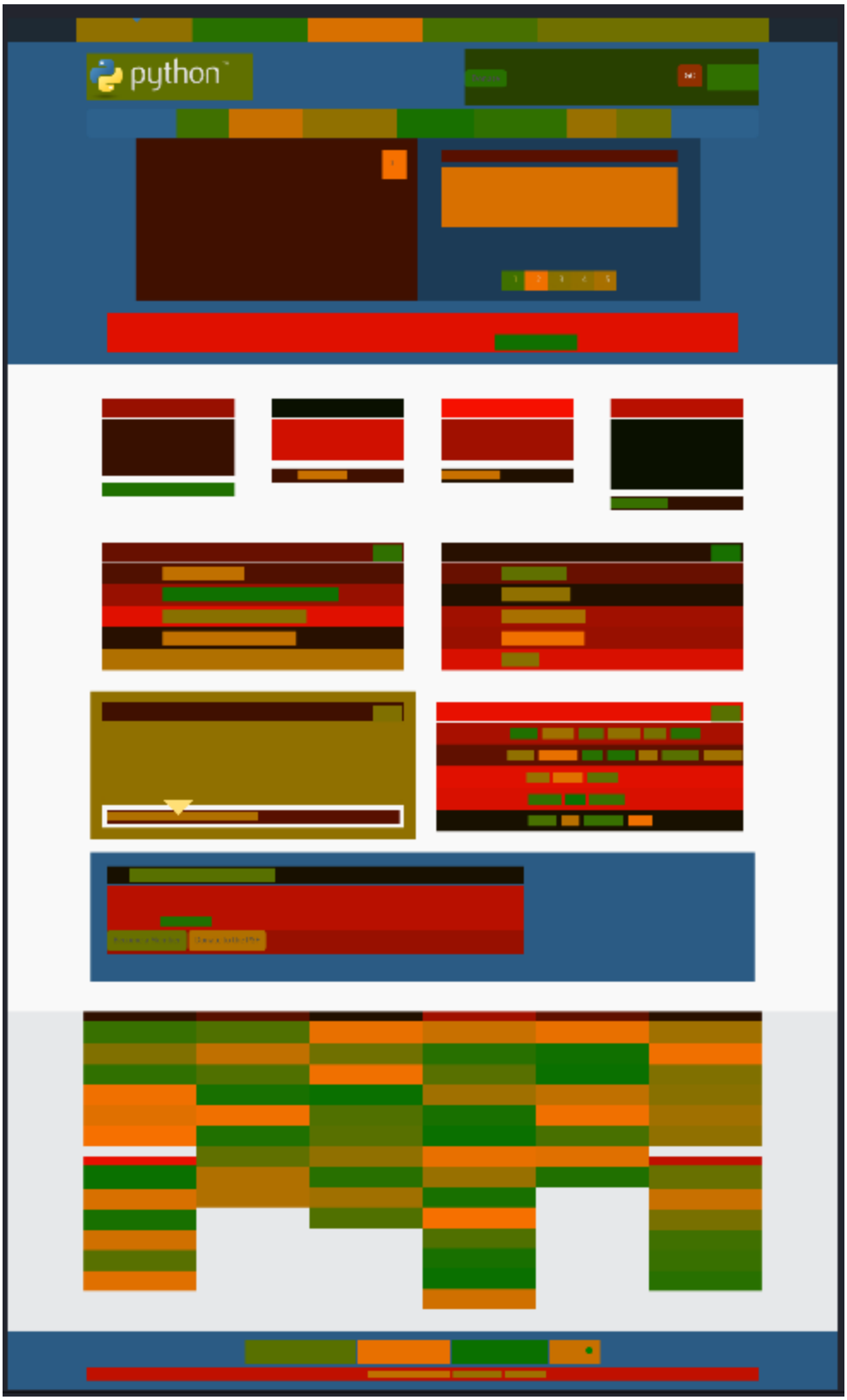

Figure 6 Deterministic RGB element encoding (Author, 2025).

Figure 7 Band encoding output (Author, 2025).

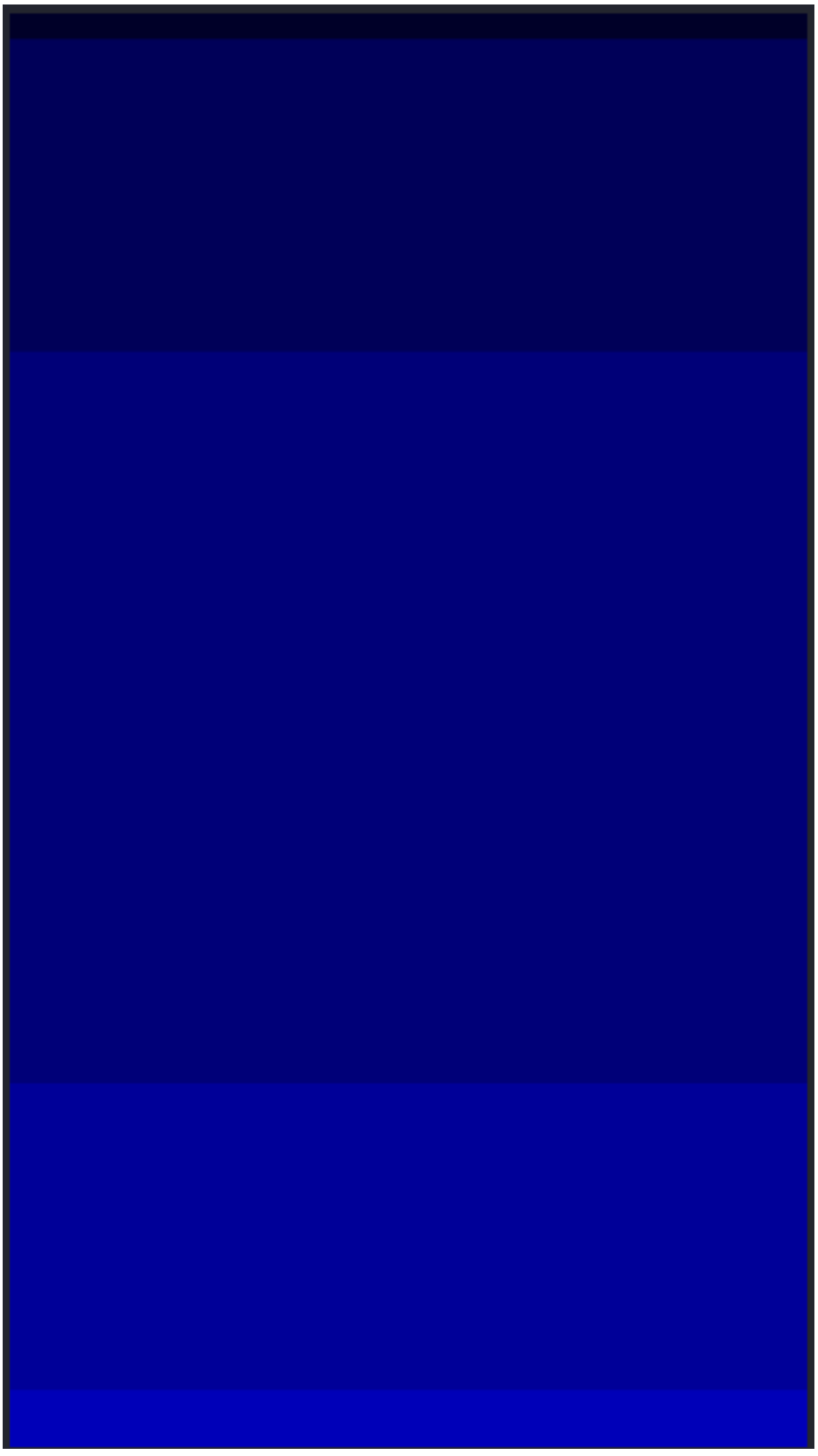

Figure 8 Derived band mask (Author, 2025).

Pipeline Design

Once colour encoding was finalised, the capture and aggregation process was automated into a single pipeline. A batch script runs every step in order, ensuring identical parameters across sites.

Overall Workflow

A batch script coordinates all steps in sequence for every website in the study. It reads a list of fifty URLs and runs a series of Python programs that record, segment, and classify the page layout. The automation ensures that each site is processed under identical conditions

Stage 1 Element Encoding

A headless browser paints each visible element with its deterministic RGB value and saves both natural and encoded screenshots (*Figure 6*).

Stage 2 Band Detection

The encoded image is scanned column-wise to identify horizontal bands based on consistent colour regions (*Figure 7*).

Stage 3 Band Extraction

Each band is sliced into separate images with coordinate metadata, isolating structural layers (*Figure 9*).

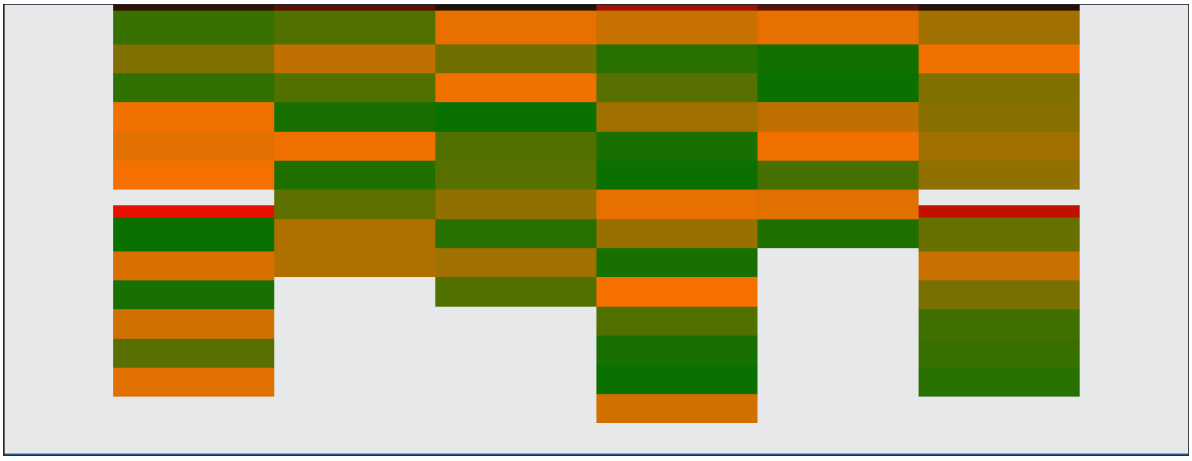

Figure 9 Deterministic RGB element encoding band (Author, 2025)

It's worth noting that the encoding structure doesn't cover the background colour, however because the R G and B channels are hard coded to mean something if all of them are an exact match, any collision in a 50 sized sampling would be a statistical anomaly and not impact the quality of the data as much.

Stage 4 Type Mapping

For each band, grayscale masks are created for key element types (text, media, button, input, etc.). Legends record bounding boxes and centroids (*Figure 10*).

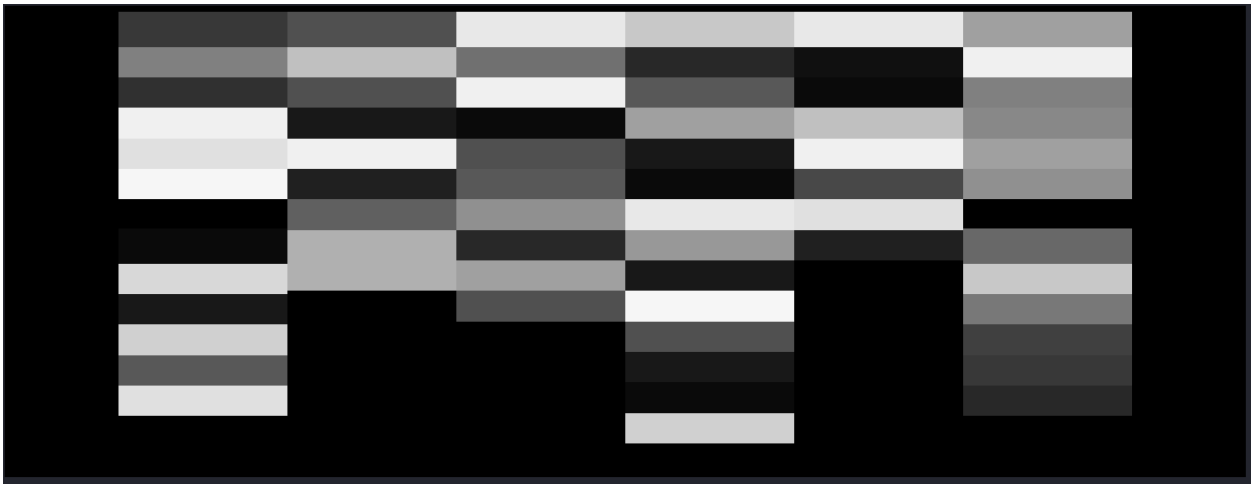

Figure 10 Example grayscale layer set (Author capture, 2025).

Stage 5 Aggregation and Heat-Map Generation

Grayscale layers from all sites are normalised, averaged, and rendered into heat-maps and block-maps showing spatial frequency (*Figures 11–12*).

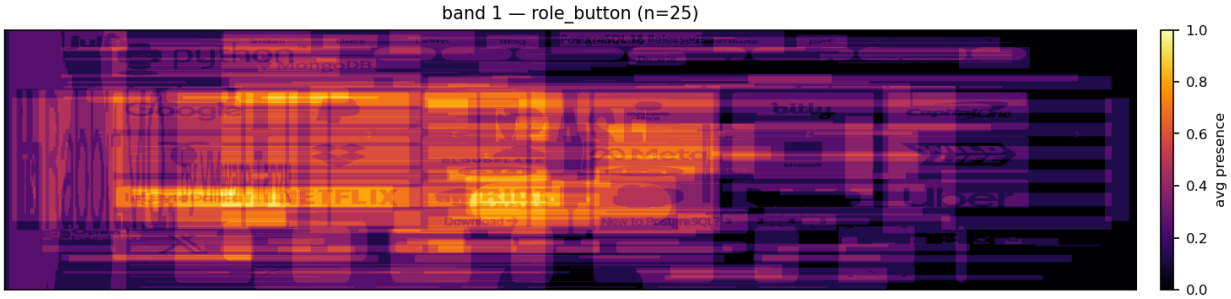

Figure 11 Aggregated heat-maps for role_button band 1 (Author render, 2025).

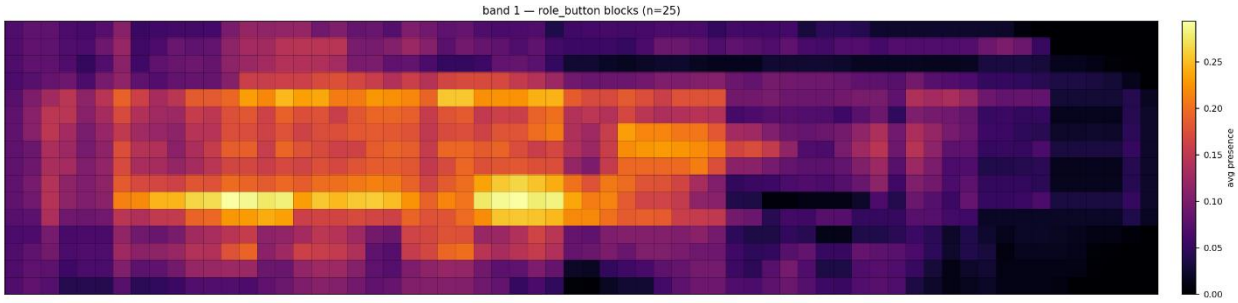

Figure 12 Aggregated block heat-maps for role_button band 1 (Author render, 2025).

Application and Future Data Refinement

Phase 1 provides the spatial evidence for the next stage: a procedural layout generator guided by WFC. In WFC, layouts are built tile-by-tile according to adjacency probabilities. The aggregated heat-maps act as these probabilities. Bright zones correspond to frequent element positions; darker zones represent rare ones. Translated into adjacency tables, they define constraints such as *text commonly follows navigation* or *buttons cluster near lower content areas*. *Figure 10* conceptually illustrates this relationship.

Improving Data Wrangling

Although the current deterministic RGB method produces clean and repeatable masks, several refinements are planned:

1. **Edge Consistency Detection** – Introduce lightweight computer-vision filters (such as Sobel or Canny edges) not to define structure but to check alignment between adjacent bands and detect missing margins.
2. **Automatic Anomaly Flagging** – Use contour analysis to highlight irregular shapes or truncated elements before aggregation. This prevents distorted masks from influencing probability maps.
3. **Adaptive Resolution Resampling** – Apply image-based scaling so that tall pages and compact pages contribute equally to the dataset.
4. **Semantic Verification** – Cross-check DOM-based type labels with visual features (for example, confirming that a “button” mask has a consistent aspect ratio and central label).

Summary of the Transition

Phase 1 delivered a reproducible process for extracting structured spatial data from real websites. Phase 2 will use that data to derive adjacency probabilities and to test how far procedural generation can imitate the spatial grammar of functional online documents. Continued refinement of the encoding and validation steps, particularly through selective use of computer-vision checks, will increase confidence that any resulting layouts reflect design principles rather than artefacts of the capture process.

References

OpenCV Team. (2024). *OpenCV documentation (Version 4.9)*. https://docs.opencv.org/

Microsoft Corporation. (2024). *Playwright documentation (Version 1.44)*. https://playwright.dev/

Python Software Foundation. (2024). *Pillow (Python Imaging Library)*. https://python-pillow.org/

Hunter, J. D., and Matplotlib Development Team. (2024). *Matplotlib: Visualization with Python (Version 3.9)*. https://matplotlib.org/

Appendix A – Scripts Overview

run.sh / crawl.sh Batch controller; calls all capture scripts.

rg.py Browser automation; generates RGB element encoding.

band.py Scans encoded image to define horizontal bands.

cut.py Slices bands and stores metadata.

function.py Creates per-type grayscale masks and legend.

aggregate_heatmaps.py / aggregate_heatmaps_blocks.py Aggregates results into heat-maps.

Appendix B – Dataset Sources

1. https://www.python.org
2. https://www.rust-lang.org
3. https://go.dev
4. https://kotlinlang.org
5. https://swift.org
6. https://www.ruby-lang.org
7. https://www.php.net
8. https://www.perl.org
9. https://www.lua.org
10. https://julialang.org
11. https://www.r-project.org
12. https://www.haskell.org
13. https://ocaml.org
14. https://elixir-lang.org
15. https://www.erlang.org
16. https://clojure.org
17. https://scala-lang.org
18. https://www.typescriptlang.org
19. https://dart.dev
20. https://www.swi-prolog.org
21. https://www.common-lisp.net
22. https://racket-lang.org
23. https://fortran-lang.org
24. https://www.tcl.tk

25. https://nim-lang.org
26. https://ziglang.org
27. https://crystal-lang.org
28. https://www.gnu.org/software/gcc/
29. https://www.gnu.org/software/
30. https://www.gnu.org/software/octave/
31. https://www.lua.org
32. https://www.gnu.org/software/guile/
33. https://www.smlnj.org
34. https://www.adaic.org
35. https://www.gnu.org/software/patch/
36. https://www.eclipse.org/eclipse
37. https://www.djangoproject.com
38. https://flask.palletsprojects.com
39. https://www.fastify.io
40. https://expressjs.com
41. https://www.gatsbyjs.com
42. https://www.nextjs.org
43. https://www.rabbitmq.com
44. https://www.nginx.com/resources/wiki
45. https://www.postgresql.org
46. https://www.mysql.com
47. https://www.mongodb.com/docs/manual
48. https://www.sqlite.org
49. https://www.apache.org
50. https://www.gnu.org/licenses/