Engine — AI-Powered Data Intelligence Pipeline

The `engine/` folder houses a proprietary multi-stage AI pipeline that combines autonomous web intelligence gathering, computer vision, neural network inference, and structured data assembly to produce production-ready datasets from unstructured web sources.

Architecture Overview

```
Stage 1: Autonomous Web Intelligence Agent
  Cloudflare bypass → Stealth browser automation → DOM extraction →
Structured data capture
    ↓
Stage 2: Computer Vision & Neural Network Processing
  EasyOCR text detection → Dynamic mask generation → LaMa deep learning
inpainting
    ↓
Stage 3: Data Assembly & Identity Resolution
  UUID generation → Image–profile linking → Database–ready output
```

Stage 1 — Autonomous Web Intelligence Agent

Script: `engine/scrape_full_image.py`

The intelligence agent operates a headless Chrome browser with multiple layers of anti-detection to autonomously navigate, extract, and structure profile data from protected web sources.

Anti-Detection & Stealth Layer

The agent employs a three-tier evasion system:

1. **Undetected ChromeDriver** — A modified Chromium driver that patches known automation fingerprints (`AutomationControlled` flag, `navigator.webdriver` property) to appear as a legitimate human-operated browser session.

2. **Selenium Stealth Injection** — Injects a full browser fingerprint at runtime:

   - Language stack: `["en–US", "en"]`
   - Vendor signature: `Google Inc.`
   - Platform spoof: `Win32`
   - WebGL vendor: `Intel Inc.`
   - GPU renderer: `Intel Iris OpenGL Engine`
   - Hairline rendering fix for sub-pixel detection

3. **Cloudflare Turnstile Solver** — An adaptive bypass that detects Cloudflare challenge pages by monitoring page title signals (`"just a moment"`, `"attention required"`), then locates and interacts with the Turnstile iframe challenge:

- Identifies challenge iframes by `challenges.cloudflare.com` or `turnstile` in the source
- Switches browser context into the iframe
- Attempts checkbox click via CSS selectors (`.cb-i`, `input[type="checkbox"]`, `#challenge-stage`)
- Falls back to JavaScript-executed clicks on the iframe body
- Implements a 30-second adaptive retry loop with automatic page refresh after 15 seconds of failed attempts

Autonomous Data Extraction

The agent performs intelligent DOM traversal to extract structured data:

- **Profile Metadata** — Parses URL segments with regex (`/([^/]+)/\d+/`) to extract identity and profile IDs
- **Protected Contact Data** — Detects and programmatically clicks "SHOW PHONE" buttons via JavaScript execution, then captures dynamically rendered phone numbers using pattern matching (`(?:\+\d{1,3}\s?)?\(?\d{3}\)?[\s.-]?\d{3}[\s.-]?\d{4,}`)
- **Structured Field Extraction** — Isolates visible text between DOM markers, normalizes line breaks, filters noise (all-caps headers), and splits key-value pairs on delimiter boundaries
- **Full-Resolution Image Capture** — Goes beyond thumbnail collection by simulating user interaction:
  1. Identifies profile images via URL pattern matching (`/preview/(?:400x592|100x100|800x|1200x)/`)
  2. Locates and clicks parent elements to trigger lightbox overlays
  3. Captures full-resolution sources (`/preview/(?:1000x700|800x|1200x)/`) from the modal
  4. Dismisses the lightbox via ESC key injection
  5. Deduplicates across the session with a URL hash set

Output Artifacts

File	Contents
`page_output.json`	Complete raw extraction per profile
`info_output.json`	Normalized profile metadata
`image_output.json`	Full-resolution image URLs with `IMW_` identifiers

Stage 2 — Computer Vision & Neural Network Processing

Script: `engine/remove_watermark.py`

This stage applies a machine learning pipeline that combines Optical Character Recognition with deep learning-based generative inpainting to produce clean, watermark-free images.

EasyOCR Text Detection

The pipeline initializes an OCR reader trained on English text (`easyocr.Reader(['en'])`) and performs full-image text detection. Each detection returns bounding box coordinates, recognized text, and a confidence score.

Watermark Classification — Detected text regions are classified as watermark content using a dual-signal approach:

1. **Keyword Matching** — Normalized text (lowercased, whitespace-stripped) is matched against a vocabulary of known watermark fragments: `['euro', 'girl', 'escort', '.com', 'eurogirlsescort', 'eurogirls', 'girlsescort', 'ort.com', 'uro', 'irl', 'scort', 'ort', 'com']`
2. **Low-Confidence Detection** — Text regions with confidence below 0.4 are flagged as likely semi-transparent watermark overlays, capturing text that OCR can partially read but not fully resolve — a strong indicator of embedded watermarks

Dynamic Mask Generation

Once watermark regions are identified, the pipeline constructs a precise inpainting mask:

1. **Bounding Box Aggregation** — All detected watermark bounding boxes are collected and their Y-coordinates (vertical extent) are computed with padding
2. **Horizontal Band Expansion** — The mask is expanded to a full-width band (4% to 96% of image width) to capture watermark content that may extend beyond detected text boundaries
3. **Morphological Dilation** — An elliptical structuring element (`5×5`) is applied via `cv2.dilate()` to expand the mask by one iteration, ensuring complete coverage of watermark edges and anti-aliased boundaries

Fallback Mask — When OCR fails to detect any text (heavily stylized or very transparent watermarks), the system falls back to a fixed-position mask at 40-48% image height and 8-92% width, based on the known watermark placement pattern.

LaMa Neural Network Inpainting

The core of the image restoration uses **LaMa (Large Mask Inpainting)**, a deep learning model specifically architected for filling large masked regions in images.

How LaMa works:

- LaMa is a neural network trained on millions of image pairs (masked → original) that learns to generate contextually coherent content for missing regions

- Unlike traditional inpainting methods (Telea, Navier-Stokes) that propagate nearby pixels, LaMa uses **Fast Fourier Convolutions** to capture global image context — understanding scene structure, textures, and patterns across the entire image
- The model generates photorealistic fills that maintain consistency with surrounding content, effectively reconstructing the image as if the watermark was never present

Processing Flow:

1. Image converted from OpenCV BGR to RGB color space
2. Both image and mask converted to PIL format for model input
3. LaMa model initialized (`SimpleLama()`) and loaded into memory
4. Inference executed: `result = lama(pil_image, pil_mask)`
5. Clean image saved with `IMW_` prefix replaced by `IM_` (Image Watermarked → Image Clean)

Debug Output

Every processed image generates a corresponding mask visualization in `engine/watermark/debug/`, enabling visual verification of mask accuracy and OCR detection quality.

Stage 3 — Data Assembly & Identity Resolution

Script: `engine/build_data.py`

The final stage resolves identities across the pipeline outputs, assigning persistent UUIDs and linking all artifacts into a unified, database-ready dataset.

UUID Generation & Identity Mapping

Each scraped profile receives a globally unique identifier via `uuid.uuid4()`. A mapping dictionary (`id_to_uuid`) maintains the relationship between the source profile ID (e.g., `835389`) and the generated UUID, ensuring referential integrity across all downstream data.

Image-Profile Linking

Clean image files are scanned using the regex pattern `^IM_(.+)\.(jpg|jpeg|png|webp)$`. The captured group is split on underscore boundaries — the first segment yields the source profile ID, which is resolved to a UUID via the identity map. This creates a deterministic link chain:

```
Filename: IM_835389_01.jpg
  → profile_id: 835389
  → page_uuid: a1b2c3d4-...
  → sequence_number: 01
```

Output Artifacts

File	Contents
`app/data/page_data.json`	UUID-enhanced profile records, ready for `providers` table
`app/data/image_data.json`	Image records with profile UUID linkage, ready for `provider_images` table

Supporting Modules

Site Discovery Agent — `engine/site_scrapper/`

- **`scrape_deep.py`** — Recursive BFS (Breadth-First Search) crawler using a `deque` queue with depth tracking. Discovers up to 50 pages per crawl with same-domain filtering. Uses 40-second page load timeouts for JavaScript-heavy targets.
- **`scrape_surface.py`** — Single-depth link extractor for rapid site mapping. 10-second load timeout, no recursion. Used for initial reconnaissance before deep crawling.

Both modules use the same stealth stack (undetected ChromeDriver + selenium-stealth + Cloudflare bypass).

Thumbnail Scraper — `engine/page_scrapper/scrape_page_thumbnail.py`

Lightweight variant of the full image scraper that collects thumbnail-resolution images directly from `img src` attributes without lightbox interaction. Used for rapid previews and validation before committing to full-resolution extraction.

Dependency Stack

Machine Learning & Computer Vision

Package	Purpose
`easyocr`	OCR-based watermark text detection
`simple-lama-inpainting`	LaMa neural network for generative image restoration
`opencv-python` (`cv2`)	Image I/O, color space conversion, morphological operations
`numpy`	Array and mask tensor operations
`Pillow` (`PIL`)	Image format bridging between OpenCV and LaMa model

Web Intelligence & Automation

Package	Purpose
`selenium`	Browser automation, DOM traversal, element interaction
`undetected-chromedriver`	Anti-detection Chromium driver
`selenium-stealth`	Browser fingerprint spoofing

Standard Library

Module	Purpose
`json`, `os`, `re`, `uuid`	Data serialization, filesystem ops, pattern matching, identity generation
`urllib.request`, `urllib.parse`	Image download, URL resolution
`collections.deque`	BFS queue for deep crawling

Pipeline Execution

```
# Stage 1 — Web intelligence gathering
python3 engine/scrape_full_image.py

# Stage 2 — Computer vision & neural network processing
python3 engine/remove_watermark.py

# Stage 3 — Data assembly & identity resolution
python3 engine/build_data.py
```

Output is consumed by `database/seed.py` for Cloud SQL injection and `gcloud storage rsync` for GCS deployment. See data_injection.md for the full deployment flow.

SUCESS RATE

- Total Images 1500
- Total suceess 900
- Success Rate 60%
- Autotmation: 95% (5% is due to cloudflare security feature work around)
- Total Run Time For 1500 image processing: < 4 Hours (end to end)

URL

http://34.124.244.233/baligirls

Open Source

All solutions are gathered through libraries of open license not requiring any paid subscription.