How the LLM agent of Part 2 was actually built

Technical companion to Part 2 of the SHORA trilogy. Not part of the trilogy itself. For readers curious about the pji-agent pipeline that produced the 9% navigation rate on 10.5 million product pages.

> **Technical companion to Part 2 — not part of the trilogy.** Audience: engineers, researchers, the open-source community. Every code reference below is a deep link into the public MIT-licensed repository at github.com/crospector/pji-agent.

Why this post exists

The main part of Part 2 reports the outcome of an LLM-agent experiment: 10.5 million product pages, 1,056 French retailers, 480 locations, $32,341 of AWS spend, 9% journey-navigation success. That post is written for a business audience asking whether SHORA can be the deterministic measurement layer for their data-quality outcome. This post is written for a technical audience — engineers, researchers, the open-source community — asking *how the pipeline was actually built and how to reproduce it*.

The open-source codebase lives at github.com/crospector/pji-agent under MIT license. Every claim below references a specific file or line in that repository.

Scope and disclaimer

The published repository contains **everything a reader needs to reproduce the experiment this post describes**: the extraction agent, the journey agent, the prompt scaffolding, the evidence-capture layout, the DynamoDB schema references, and the Docker runtime. Configure the required AWS resources, provide your own credentials, and the pipeline runs.

> The repository **does not contain the production wiring** that ran SHORA's full experiment. To protect proprietary assets unrelated to the LLM agent itself, the following classes of code and documentation are deliberately omitted: the operations dashboard and its DynamoDB aggregation logic, the swarm orchestration layer that coordinated multiple agent fleets, the guardian preflight and auto-fix system, the validator API and its service mesh, the infrastructure-as-code that provisioned the AWS resources, the operational state files and audit logs that accumulated during the experiment, the internal architecture specs and roadmap documents, and the unrelated codebases (the deterministic engine, the dashboard frontend, internal tools) that share SHORA's monorepo but have nothing to do with the agent. What is published is the agent itself and the minimum infrastructure files needed to

> run it, nothing more. References to internal paths below describe the agent in production; the published repository contains the same files with secrets parameterised via environment variables.

The pipeline at a glance

The experiment ran on a two-machine architecture, both on AWS `c6i.8xlarge` instances (32 vCPU, 64 GB RAM, NVMe local SSD), located in `eu-west-3` (Paris):

- **Extraction machine.** Crawled the catalog of 1,056 French retailers. Discovered category pages, paginated through product listings, and extracted structured product records (name, price, URL, brand, category hierarchy, reviews). Implemented in `orchestrator/main.py` with the LLM-extraction step in `orchestrator/opus_extractor.py`.
- **Journey machine.** For each extracted product, executed a full purchase journey: navigate to the PDP, capture the PDP price, click add-to-cart, navigate to the cart, capture the cart price, classify the outcome. Implemented in `journey_orchestrator/main.py`, with the per-product test logic in `journey_orchestrator/journey_tester.py`.

Both machines ran 20–40 concurrent Camoufox browser instances (an anti-detect Firefox fork), driven by Playwright. The model in the data path was `eu.anthropic.claude-opus-4-6-v1` on Amazon Bedrock, called from `orchestrator/opus_extractor.py` (lines 23–24):

```
BEDROCK_MODEL_ID = "eu.anthropic.claude-opus-4-6-v1"
BEDROCK_REGION = "eu-west-3"
MAX_HTML_CHARS = 80_000
```

The model received page HTML (capped at 80,000 characters), an accessibility tree, and screenshots. Identity-grounding of every visible UI element on the screenshot was solved by an annotation layer that drew bounding boxes carrying GUIDs traceable back to the underlying DOM elements — so the model never had to "find" a button or a price by raw vision. Identity was given; semantic grounding (which box is the price, which is the add-to-cart, which is the cart total) remained the model's job. That distinction is the architectural core of the Part 2 argument.

The extraction agent

What it was asked to do

Per category page on each retailer, the model was asked to return a JSON array of product records with twelve fields each: `category`, `subcategories`, `name`, `reference`, `url`, `description`, `price`, `promo_price`, `brand`, `image_url`, `reviews_count`, `reviews_avg`,

and `trust_signals`. The prompt (from `orchestrator/opus_extractor.py` lines 29–62) enforced six "critical rules":

> 1. ONLY extract REAL PRODUCTS — items you can actually buy
> 2. NEVER extract: navigation links, category names, banners, ads, footer links, CSS, JS, HTML tags
> 3. If a product's name looks like code/HTML/CSS/JS, SKIP IT
> 4. Price must be a valid number > 0 (if no price visible, set to null)
> 5. URL must be a valid product page URL on this retailer's domain
> 6. If the page has no real products, return empty array

The HTML the model received

A lightweight in-page script (`CAPTURE_PAGE_JS` in the same file) tried to identify the product area of the rendered page and extract its inner HTML, falling back to a body clone with `script`, `style`, `nav`, `footer`, `header`, cookie banners, and modal popups stripped. The reasoning: feed the model the smallest, cleanest possible payload that still contained every product card. In practice this constraint was almost never the binding one — the binding constraint was the model's ability to apply semantic rules to a heterogeneous HTML fragment, not the size of the fragment.

The selectors used to find "the product area"

```
document.querySelector(
  'main, #main, [role="main"], .main-content, #content,
   .products, .product-list, .product-grid,
   [class*="product-list"], [class*="ProductList"]'
);
```

This worked for the majority of retailers. When it failed, the fallback (cleaned body) was used. The model could not in general tell the difference between "I'm reading a product listing page that has products on it" and "I'm reading a category-tile page that has subcategory navigation that *looks* like products to a language model." This is one of the failure modes the Part 2 main post attributes to **semantic grounding**: the boxes were addressable, but the model did not always know which boxes were products versus which were links to other category pages.

The journey agent

Once the extractor produced a catalog of products with valid URLs, the journey orchestrator (`journey_orchestrator/main.py`) round-robined through them. For each product,

`journey_orchestrator/journey_tester.py` executed the eight-step journey documented at the top of the file:

> 1. Navigate to product URL (fallback: search by name on retailer site)
> 2. Capture PDP price
> 3. Click "Add to cart"
> 4. Navigate to cart/checkout
> 5. Capture cart price
> 6. Compare PDP vs cart price
> 7. Clear cart
> 8. Determine outcome: JOURNEY_COMPLETE, JOURNEY_BLOCKED, PRICE_MISMATCH

The journey tester ran in two modes (defined in `journey_orchestrator/config.py`):

- **Heavy mode.** Used in Lille (the baseline location) and on every anomaly re-test. Full evidence capture per step: screenshot, annotated screenshot, original HTML, signed HTML (HTML annotated with the same GUID from screenshots), accessibility tree, vitals (Core Web Vitals), cookies, storage, mapping of UI signatures to application IDs, network HAR, console log, timing data, security log. All of it uploaded to S3 alongside a per-journey `video.webm` recording and a Playwright `trace.zip`.
- **Light mode.** Used for all 479 other French locations. Prices and outcome only, written to DynamoDB. If light mode detected an anomaly, the orchestrator immediately re-tested the same product in heavy mode to capture full evidence.

The two-mode design was a cost optimization: full evidence capture across 480 locations × millions of journeys was prohibitive. Heavy mode existed for the cases where evidence mattered (the Lille baseline as ground truth, and any anomaly worth documenting).

Anti-bot detection

Camoufox bypassed roughly 92% of the WAFs and bot defenses encountered. The remaining 8% were detected via the signature list in `journey_tester.py` lines 62–71:

```
ANTIBOT_SIGNATURES = [
    "captcha", "recaptcha", "hcaptcha", "challenge-platform",
    "cloudflare", "cf-browser-verification", "ray id",
    "attention required", "checking your browser",
    "access denied", "403 forbidden", "just a moment",
    "datadome", "incapsula", "imperva", "perimeterx",
    "bot detection", "security check", "verify you are human",
```

```
    "please enable cookies", "please turn javascript on",
]
```

When a signature was detected, the orchestrator retried with a fresh browser session, fresh IP egress (via residential proxy rotation when configured), and a backoff. Anti-bot challenges that survived retries were never counted in the journey outcome — they were logged as infrastructure failures and the journey was re-queued. The 9% navigation rate reported in Part 2 is computed *after* those infrastructure retries, on journeys where the agent reached the pages it was asked to read.

Outcome classification

The journey tester emitted one of three outcomes per product:

Outcome	Meaning
`JOURNEY_COMPLETE`	The agent navigated the site end-to-end and reached an add-to-cart that worked with the PDP price intact.
`JOURNEY_BLOCKED`	The agent navigated the site end-to-end and concluded that the site itself blocked the buyer: an explicit stockout indicator, a non-functional add-to-cart, a server error during checkout, a UI dead-end, a discontinued-product page, or any other site-side friction that would have stopped a real buyer too. `JOURNEY_BLOCKED` is a signal of customer-experience friction the agent caught, not a signal of agent failure.
`PRICE_MISMATCH`	The agent navigated the site end-to-end and observed a cart price strictly greater than the PDP price.

A login wall encountered *after* the cart was reached was classified `JOURNEY_COMPLETE` (the journey did its job; the retailer's checkout flow then required authentication, which is outside the agent's scope).

> The 9% Part 2 reports is the rate at which the agent *emitted any classification at all* — meaning it navigated the site end-to-end and reached one of the three terminal states above (`JOURNEY_COMPLETE` , `JOURNEY_BLOCKED` , or `PRICE_MISMATCH`). The 91% remainder is the rate at which the agent failed to navigate the site end-to-end — the multi-step chain broke down somewhere along the way and the agent never reached a conclusion. This 91% is what Part 2 §"Why the architecture hit a ceiling" attributes to semantic-grounding errors compounding across the chain, per-step error rates that collapse multiplicatively, and the model not learning from prior failures.

Evidence layout in S3

Per heavyweight journey (one per Lille product, plus every anomaly re-test in any of the 480 locations), the agent wrote an S3 prefix matching the layout below (the evidence layout is implemented in `journey_orchestrator/evidence_collector.py` and uploaded via `journey_orchestrator/s3_uploader.py`):

```
v2/journeys/{retailer}/{location}/{date}/{journey_id}/
├── manifest.json
├── journey.json
├── report.md
├── checksums.sha256
├── video.webm
├── trace.zip
├── logs/
│   ├── journey.log
│   ├── errors.json
│   ├── api_calls.json
│   └── agent_transcript.json
└── steps/{NN}/
    ├── screenshot.png
    ├── screenshot_annotated.png
    ├── original.html
    ├── signed.html
    ├── vitals.json
    ├── cookies.json
    ├── storage.json
    ├── accessibility.json
    ├── state.json
    ├── summary.json
    └── logs/
        ├── network.har
        ├── network.json
        ├── console.json
        ├── timing.json
        ├── coverage.json
        └── security.json
```

This layout was designed so that any single journey could be replayed and audited deterministically from its archived evidence alone, without needing access to the production system that produced it. The Lille baseline plus the anomaly archive together constituted the reproducible record of the experiment.

Storage and validation

Per-product price records were written to a DynamoDB single-table design, implemented in `journey_orchestrator/dynamo_writer.py`, with the following entity model:

Entity	PK	SK	Purpose
Retailer	`RETAILER#{slug}`	`METADATA`	Retailer definitions (URL, name)
Location	`LOCATION#{slug}`	`METADATA`	480 French locations
Product	`PRODUCT#{retailer}#{id}`	`PRICE#{location}`	Extracted product + price per location
Journey	`JOURNEY#{retailer}#{location}`	`{timestamp}#{journey_id}`	Journey test results
Dashboard	`DASHBOARD#STATS`	`GLOBAL`	Aggregate counters

Each journey was independently validated against four API endpoints exposed by an internal validation service. The client calls live in `journey_orchestrator/validator.py`; the service itself is not in the open-source repository (see the scope disclaimer at the top of this post). Its role was to verify that the S3 evidence layout was internally consistent (manifest claims matched archive contents, checksums verified, step counts matched declared totals) before a journey was promoted from "executed" to "validated."

The third-party benchmark context

The 9% reported in Part 2 is consistent with the published literature on web agents. For readers who want to verify the architectural ceiling claim against authoritative sources:

- **WebArena** — Zhou et al., ICLR 2024. The canonical academic benchmark, 812 long-horizon web tasks. At publication, best-GPT-4-based agent reached 14.41% vs human baseline 78.24%. Live leaderboard currently shows SOTA at 74.3% (Deepseek v3.2, Feb 2026), still below the human baseline.
- **VisualWebArena / Online-Mind2Web** — *An Illusion of Progress? Assessing the Current State of Web Agents*, Tao et al., COLM 2025. Introduces a 300-task realistic benchmark across 136 live websites; 2025 frontier drops to 61%, vs human 88.7% on VisualWebArena.
- **WAREX** — arxiv:2510.03285, 2025. Stress-tests released agents on WebArena, REAL, and WebVoyager under production-like conditions; documents "severe degradation under realistic conditions, exposing fundamental robustness gaps."
- **OSWorld** — Xie et al., NeurIPS 2024. Computer-use benchmark; original paper reports best model 12.24% vs human 72.36%.

- **Record-replay test fragility** — *Similarity-based Web Element Localization for Robust Test Automation*, Stocco et al., ACM TOSEM 2023. Empirical study of 1,065 test breakages across 453 web application versions; fragile locators caused 73.6% of failures. *Why do Record/Replay Tests of Web Applications Break?*, Hammoudi et al., 2016 is the canonical earlier study.

The above five families of evidence are the third-party basis for the Part 2 §"Why the architecture hit a ceiling" argument. Readers who reproduce the pji-agent pipeline on a comparable workload should expect numbers consistent with this literature, not with the SOTA on curated benchmarks.

A note on scale

The full experiment described in this post ran across 10.5 million product pages, 1,056 French retailers, and 480 locations, accumulating $32,341 of AWS spend over seven months. A curious reader interested in the architecture, the prompt scaffolding, or the journey-orchestrator loop does not need to run the experiment at that scale — the code, the prompts, and the evidence layout are all readable in the repository. See the repository README for setup details.

Part 2 of the SHORA trilogy: $32,341 of AWS spend, 10.5 million product pages, 9% reliability. Financial detail companion: part-2-financial-companion. Open-source repository: github.com/crospector/pji-agent.
