CSC490 Assignment 2: OpenBrowser-AI
Data Processing Pipelines and Infrastructure as Code

Muhammad Enrizky Brillian (1009713712), Madhav Kanna Thenappan (1007841659),
Rohit Shetty (1007979020), Joseph Yu (1009196521), August Zheng (1009868635)
University of Toronto

February 13, 2026

1 Part One: Aspirational Datasets

We propose two complementary aspirational datasets: one for supervised fine-tuning (SFT) and one for reinforcement learning (RL). Together, these datasets are designed to address the core challenges of training a agentic language model for robust browser automation: grounded action selection, robustness to failure, and long-horizon decision-making.

The datasets would ideally be very large and cover a diverse and comprehensive set of browser tasks, spanning across a variety of domains (research, data entry, content creation, social media management, etc.).

1.1 Dataset 1: Task-Decomposed Supervised Fine-Tuning Dataset

Purpose. The objective of the SFT dataset is to train models to map browser states (consisting of DOM snapshots, screenshots, etc.) and task goals to well-reasoned actions (e.g. function calls along with reasoning tokens).

Structure. The dataset would be organized hierarchically:

- A *task* represents a real-world web objective (e.g., submitting a form, purchasing an item).
- Each task consists of a sequence of *atomic interaction steps*, each corresponding to a single browser action.
- Tasks may include branching paths that address correct alternatives and/or common error or failure states (e.g., validation errors, missing fields, unexpected page transitions).

Each atomic step is treated as an independent supervised training example, enabling flexible sampling strategies during training.

Atomic Step Schema.

```
{
  "task_id": "string",
  "task_goal": "string",
  "state": {
    "url": "string",
    "dom_snapshot": "serialized_dom",
    "accessibility_tree": "a11y_tree_json",
    "observation": {
      "error_message": "optional string",
      "dom_delta": "optional"
    },
    "history_summary": "concise textual summary"
  },
  "target_output": {
    "reasoning": "brief grounded justification",
    "action": {
      "tool": "browser",
      "type": "click|type|scroll|wait|abort",
      "target": {"semantic_role": "string"},
      "value": "optional"
    }
  }
}
```

1.2 Dataset 2: Reinforcement Learning Web Environment Suite

Purpose. The RL dataset addresses aspects of browser interaction that are difficult or impractical to specify through static supervision alone, including termination decisions, recovery strategies, and long-horizon credit assignment. The RL dataset also allows more freedom for the agent to explore potentially better strategies than those present in the SFT dataset.

Structure. Rather than a static dataset, this component consists of a suite of simulated web environments. Each environment defines:

- A task objective, success conditions, and episode termination conditions
- A browser state space consistent with the SFT dataset schema
- A shared action space identical to the supervised setting

All interactable elements in these web environments will be designed to allow easy implementation of reward signals (e.g., form field completion, navigation success, error resolution).

1.3 Aspirational Data Schema

Figure 1 (Appendix A) shows the unified relational schema that would back our aspirational data infrastructure. The schema is color-coded by domain: orange for the model registry, blue for core task data, green for evaluation, and purple for training. All relationships are one-to-many, reflecting the natural hierarchy from tasks to actions, results, training examples, and reward signals.

2 Part Two: Reality Check

We have identified five datasets that are actually available for evaluating and training OpenBrowser-AI. These range from established academic benchmarks to our own custom stress tests.

2.1 Dataset Comparison Table

Dataset	**Description**	**Commentary**
WebArena GitHub Paper	812 long-horizon tasks across 4 self-hosted web environments: e-commerce (90K products), social forums (Reddit-like), GitLab, and CMS. Functional correctness evaluation with exact_match, must_include, and fuzzy_match methods. Human trajectories for 170 tasks.	**Primary benchmark** for end-to-end agent evaluation. Docker-based reproducible environment eliminates website change issues. Best GPT-4 achieves 14.41% vs human 78.24%. Directly tests our Agent vs CodeAgent modes on realistic multi-step tasks. Deployed via Docker containers (Shopping, Shopping Admin, Reddit) on EC2.
Mind2Web GitHub HuggingFace	2,350 tasks across 137 real websites in 31 domains. Average 7.3 actions per task. Includes raw HTML, cleaned HTML, and element candidates (pos/neg). Three test splits: Cross-Task, Cross-Website, Cross-Domain.	Tests **generalization** to unseen websites—critical for OpenBrowser-AI's goal of working on any website. Provides action-level annotations (CLICK, TYPE, SELECT) matching our action space. HuggingFace hosting enables easy integration. Used for flow matching model training data.
FormFactory GitHub Paper	1,250 form-filling instances with 13,800 ground-truth field-value annotations across 20 web forms in 8 domains (Academic, Professional, Healthcare, Legal, etc.). Interactive Flask-based evaluation platform.	**Specialized for form-filling**—a core use case for browser automation. Tests fine-grained semantic alignment between source documents (resumes, papers) and form fields. Current MLLMs achieve ¡5% click accuracy, showing significant room for improvement. **Full 100-task evaluation completed on AWS** across 4 model/agent combinations (see Section 3.7): Gemini 2.5 Flash CodeAgent achieved 100/100 (100%) in 47.68s avg.
Our Stress Tests GitHub Pages Local: stress-tests/	40 custom tasks: 13 core interactions (radio, checkbox, slider, hover, file upload, canvas CAPTCHA) and 27 framework-specific forms (React, Vue, Angular, Svelte, Shadow DOM, iframes). Ground truth validation included.	**Custom benchmark** we control. Tests specific browser challenges: Shadow DOM penetration, iframe navigation, dynamic forms, non-Latin characters. Hosted on GitHub Pages for consistent availability. JSON schema matches our evaluation runner. Essential for regression testing.

Dataset	Description	Commentary
WebVoyager GitHub Paper	End-to-end web agent benchmark focusing on multimodal (vision + text) evaluation. Tests agents on real websites with screenshot-based observation.	Useful for comparing our **hybrid DOM+vision approach** against pure vision-based agents like SeeAct. Referenced in A1 landscape analysis. Less suitable for reproducible evaluation (real websites change), but valuable for multimodal capability assessment.

2.2 Detailed Data Schemas

2.2.1 WebArena Task Schema

WebArena tasks are stored in `config_files/test.raw.json`:

```
{
  "task_id": 108,
  "instruction": "Find the cheapest product in Electronics category",
  "sites": ["shopping_admin"],
  "task_type": "RETRIEVE",
  "reference_answer": "$29.99",
  "evaluation_method": "must_include",
  "require_login": true,
  "start_url": "http://localhost:7770/admin"
}
```

Action Space: click, type, hover, press, scroll, new_tab, tab_focus, close_tab, goto, go_back, go_forward, stop [answer]

Observation Space: URL, open tabs, page content (HTML DOM / screenshot / accessibility tree)

2.2.2 Mind2Web Task Schema

Mind2Web data available via HuggingFace `datasets` library:

```
{
  "annotation_id": "a1b2c3d4",
  "website": "amazon",
  "domain": "shopping",
  "subdomain": "e-commerce",
  "confirmed_task": "Add iPhone 15 Pro to cart",
  "action_reprs": ["Click on 'iPhone 15 Pro'", "Click 'Add to Cart'"],
  "actions": [
    {
      "action_uid": "act_001",
      "raw_html": "<!DOCTYPE html>...",
      "cleaned_html": "<div class='product'>...",
      "operation": {
        "op": "CLICK",
        "original_op": "CLICK",
        "value": ""
      },
      "pos_candidates": [
        {
          "tag": "a",
          "is_original_target": true,
          "backend_node_id": "node_123",
          "attributes": "{\"href\": \"/product/iphone\", \"class\": \"product-link\"}"
        }
      ],
      "neg_candidates": [...]
    }
  ]
}
```

2.2.3 FormFactory Instance Schema

FormFactory provides structured form-filling evaluation:

```
{
  "instance_id": "job_app_001",
  "form_id": "academic_job_application",
  "domain": "Academic & Research",
  "source_document": "data/data2/resume_001.md",
  "fields": [
    {
      "field_id": "first_name",
      "field_type": "String",
      "label": "First Name",
      "ground_truth": "John",
      "bbox": [120, 45, 280, 75]
    },
    {
      "field_id": "degree",
      "field_type": "Dropdown",
      "label": "Highest Degree",
      "options": ["Bachelor's", "Master's", "PhD"],
      "ground_truth": "PhD"
    }
  ],
  "evaluation": {
    "click_accuracy": 0.85,
    "value_accuracy": 0.92
  }
}
```

2.2.4 OpenBrowser Stress Tests Schema

Our custom benchmark in `stress-tests/InteractionTasks_v8.json`:

```
{
  "task_id": "14",
  "name": "Vanilla Form",
  "confirmed_task": "Go to the URL and complete the vanilla HTML form ...",
  "category": "FRAMEWORK_FORM",
  "ground_truth": "dumbledore"
}
```

Categories: CORE_INTERACTION (13 tasks), FRAMEWORK_FORM (27 tasks)

Frameworks Tested: Vanilla HTML, jQuery Bootstrap, AngularJS, Angular, React Hook Form, Formik, Svelte, Ember, Vue, Material-UI, Shadow DOM, Dynamic Forms, Progressive Forms, Iframe Inception (3 levels)

3 Part Three: Data Processing Pipelines

Our evaluation pipeline is a fully implemented Python system that handles data ingestion from four heterogeneous dataset sources, normalizes them into a unified task schema, executes browser-based agent evaluation, and produces structured results with per-task artifacts (video recordings, agent history traces).

3.1 Pipeline Architecture Overview

The pipeline is implemented across six Python modules totaling approximately 1,720 lines of code in `infra/eval/pipelines/`:

Module	Responsibility	Lines
`eval_benchmark.py`	Main orchestrator, agent execution	869
`data_loader.py`	Unified dataset loading and normalization	503
`results_schema.py`	Pydantic data models (TaskResult, RunSummary)	126
`eval_config.py`	Dataclass configuration with validation	94
`eval_report.py`	Cross-run markdown and CSV reporting	128
`results_aggregator.py`	Cross-project analysis and statistics	143

Figure 2 (Appendix A) illustrates the end-to-end evaluation pipeline architecture. The system spans local and AWS environments: datasets are ingested through the unified `data_loader.py` dispatcher, agent execution is orchestrated by `eval_benchmark.py` using Playwright browser sessions, and results are collected as structured JSON with per-task video recordings and agent history traces. The pipeline supports multiple LLM backends (Gemini, OpenAI, Anthropic, Ollama, vLLM) and two agent types (Agent and CodeAgent).

Figure 3 (Appendix A) details the data processing pipeline from raw sources through ingestion, cleaning, and transformation to the data lake. Four evaluation datasets (HuggingFace Hub, GitHub repositories, local JSON) flow through dataset-specific loaders into a unified task schema. Training sources are separately preprocessed into SFT JSONL and flow JSONL formats, with a reward function providing online signals during GRPO training.

Figure 4 (Appendix A) shows the S3 data lake architecture with prefix-based partitioning across two buckets: a data bucket for raw and processed datasets with Glacier lifecycle transitions, and a results bucket for evaluation runs with 365-day deletion policies. IAM access controls, SSM Parameter Store for secrets, and VPC security groups are all managed as Terraform resources in `main.tf`.

3.2 Data Ingestion

The `data_loader.py` module implements a unified `load_dataset()` dispatcher that routes to dataset-specific loaders. Each loader normalizes its source into a common task dictionary schema:

```
{
  "task_id": "string",
  "name": "string",
  "instruction": "string",
  "category": "string",
  "ground_truth": "string",
  "dataset": "stress_tests|mind2web|formfactory|webarena"
}
```

Dataset-specific ingestion details:

1. **Stress Tests**: Reads `stress-tests/InteractionTasks_v8.json` directly. No transformation needed beyond field renaming (`confirmed_task` → `instruction`).
2. **Mind2Web**: Calls the HuggingFace `datasets` library to fetch the `osunlp/Mind2Web` train split. Builds rich instructions that include the website URL, domain context, and reference action step descriptions.
3. **FormFactory**: Reads 25 ground truth JSON files from `data/formfactory/data/data1/`. Maps each form to its Flask route via a hardcoded `FORMFACTORY_ROUTE_MAP` (25 entries across 8 domains). Constructs instructions that include the form URL (`http://localhost:5050/form_name`) and all field values to fill.
4. **WebArena**: Downloads `test.raw.json` from the GitHub `web-arena-x/webarena` repository. Filters to supported sites only (shopping on port 7770, shopping_admin on port 7780, reddit on port 9999). Resolves URL placeholders (`__SHOPPING__` → `http://{hostname}:7770`) and injects login credentials.

3.3 Data Cleaning and Transformation

For training pipelines (distinct from evaluation), two preprocessors transform raw datasets into model-consumable formats:

- `formfactory_preprocessor.py`: Converts 25 FormFactory ground truth JSON files into two formats: SFT JSONL for Qwen3-8B QLoRA fine-tuning and flow JSONL for discrete flow matching training.
- `mind2web_preprocessor.py`: Converts Mind2Web data into flow training JSONL with action sequences for the flow vector field estimator.

All preprocessing outputs are written to `data/processed/` and are referenced as Makefile dependencies—running `make train-submit-sft` automatically triggers the preprocessor if the JSONL file does not exist.

3.4 Data Schemas

The pipeline uses JSON/JSONL/Pydantic document structures stored on S3 and local disk. Figure 5 (Appendix A) shows these document schemas with their relationships and join keys. The diagram maps out four primary document types: `TaskResult` (per-task evaluation output with video and history paths), `RunSummary` (aggregate statistics across all tasks in a run), SFT JSONL (instruction-response pairs for fine-tuning), and GRPO reward signals (per-rollout reward components and advantages). Join keys such as `run_id`, `task_id`, and `form_name` link documents across the evaluation and training workflows.

Results are structured using Pydantic models in `results_schema.py`, ensuring type safety and enabling JSON serialization/deserialization.

3.4.1 TaskResult – Per-Task Output

```
{
  "task_id": "ff-1",
  "task_name": "Art Exhibition Submission (record 1)",
  "dataset": "formfactory",
  "category": "arts-creative",
  "instruction": "Fill out the Art Exhibition Submission form...",
  "ground_truth": "{...}",
  "agent_type": "Agent",
  "model": "gemini-2.5-flash",
  "project": "benchmarking",
  "run_id": "20260208_064830_583b722c",
  "success": true,
  "execution_time": 111.88,
  "steps_taken": 14,
  "final_output": "The Art Exhibition form has been submitted...",
  "error_message": null,
  "video_path": "tasks/ff-1/video/0698831e...mp4",
  "history_path": "tasks/ff-1/history.json",
  "agent_messages": [
    {
      "step": 1,
      "evaluation": "Starting task, need to navigate to form URL",
      "memory": "Task: fill Art Exhibition Submission form",
      "next_goal": "Navigate to form URL",
      "actions": [{"action": "navigate", "url": "..."}],
      "results": [{"extracted_content": "...", "is_done": false}]
    }
  ],
  "started_at": "2026-02-08T06:48:30.123Z",
  "completed_at": "2026-02-08T06:50:22.003Z"
}
```

3.4.2 RunSummary – Aggregate Statistics

```
{
  "run_id": "20260208_192554_b4834ad4",
  "project": "benchmarking",
  "started_at": "2026-02-08T19:25:54Z",
  "completed_at": "2026-02-08T20:37:12Z",
  "datasets": ["formfactory"],
  "models": ["gemini-2.5-flash"],
  "agent_types": ["Agent"],
  "total_tasks": 100,
  "total_successes": 98,
  "total_failures": 2,
  "success_rate": 0.98,
  "avg_execution_time": 67.49,
  "agent_summaries": {"Agent": {"total": 100, "success_rate": 0.98}},
  "dataset_summaries": {"formfactory": {"total": 100, "success_rate": 0.98}},
  "model_summaries": {
    "gemini-2.5-flash": {"total": 100, "success_rate": 0.98, "avg_time": 67.49}
  },
  "results": ["<array of 100 TaskResult objects>"]
}
```

The `compute_summaries()` method calculates per-agent, per-dataset, and per-model breakdowns automatically from the results list.

3.4.3 SFT Training Data – Per-Example Input

The training preprocessor (`formfactory_preprocessor.py`) converts raw FormFactory ground truth into JSONL for QLoRA fine-tuning:

```
{
  "instruction": "Go to http://127.0.0.1:5050/art_exhibition and fill out
```

```
    the Art Exhibition Submission form with the following information,
    then submit it.\n\nField values to enter:\n  - artist_name: Jane Doe
    \n  - medium: Oil on Canvas\n  - dimensions: 24x36 inches",
  "response": "Step 1: Navigate to http://127.0.0.1:5050/art_exhibition
    \nStep 2: Click on the 'artist_name' input field\nStep 3: Type
    'Jane Doe'\nStep 4: Click on the 'medium' input field\n...",
  "form_name": "Art Exhibition Submission",
  "domain": "arts-creative",
  "num_fields": 9,
  "url": "http://127.0.0.1:5050/art_exhibition",
  "ground_truth_fields": {"artist_name": "Jane Doe", "medium": "Oil on Canvas"}
}
```

Dataset size: 1,240 examples across 25 forms in 8 domains. The `instruction` field provides the form URL and all field values; the `response` field provides the step-by-step action plan. During SFT training, instruction tokens are masked (`labels=-100`) so the loss is computed only on response tokens.

3.4.4 GRPO Reward Signal – Per-Rollout Output

During online GRPO training, the model generates G=4 candidate action plans per prompt. Each rollout is executed in a headless browser against the live FormFactory form, and a composite reward is computed:

```
{
  "prompt": "Go to http://127.0.0.1:5050/art_exhibition and fill...",
  "rollout_text": "Step 1: Navigate to ...\nStep 2: Click on ...",
  "reward": 0.72,
  "reward_components": {
    "task_completion": 1.0,
    "field_accuracy": 0.65,
    "execution_completeness": 0.80
  },
  "advantage": 0.34,
  "group_mean": 0.58,
  "group_std": 0.21
}
```

Reward weights: task_completion (0.4) + field_accuracy (0.4) + execution_completeness (0.2). Field accuracy uses continuous string similarity (`difflib.SequenceMatcher`) for dense feedback even on partial submissions. Advantages are normalized within each group: $A_i = (r_i - \mu_G)/\sigma_G$.

3.5 Pipeline Execution Flow

The main orchestrator `eval_benchmark.py` executes the following stages:

1. **Configuration**: Parse CLI arguments into `EvalConfig` dataclass, validate parameters.
2. **Server Initialization**: Start required servers:
    - `FormFactoryServer`: Launches Flask subprocess on port 5050, waits 30s for health check.
    - `WebArenaServer`: Pulls Docker images, starts containers on ports 7770/7780/9999, runs post-start Magento URL configuration.
3. **Dataset Loading**: Call `load_dataset()` for each dataset with `max_tasks` filtering.
4. **Agent Execution**: Triple nested loop over datasets × models × agent types × tasks:
    - Create Playwright browser session with optional video recording (`record_video_dir`).
    - Instantiate LLM via factory pattern (`_get_llm()`) supporting Gemini, OpenAI, Anthropic, Ollama, and vLLM backends.

- Run agent with `max_steps` limit, capturing per-step messages (evaluation, memory, next goal, actions, results).
- Save per-task artifacts: `.mp4` video, `history.json` agent trace.

5. **Results Aggregation**: Build `RunSummary`, call `compute_summaries()`.
6. **Output**: Write `results.csv` (flat, 15 columns) and `summary.json` (nested). Upload to S3 if `--results-bucket` is set.

3.6 When Pipelines Run and Use Cases

Pipeline	Trigger	Use Case
`eval_benchmark.py`	On-demand (Makefile or `launch_eval.sh`)	Benchmark agent performance against datasets. Runs locally or on AWS EC2 spot instances.
`formfactory_preprocessor.py`	Make dependency (auto before training submission)	Transform FormFactory ground truth into SFT/flow JSONL for Qwen3-8B fine-tuning.
`mind2web_preprocessor.py`	Make dependency (auto before flow training)	Transform Mind2Web HuggingFace data into flow training format.
`eval_report.py`	On-demand (`make report`)	Aggregate multiple eval runs into cross-project/model/dataset markdown and CSV reports.
`download_datasets.py`	One-time setup (`make download-datasets`)	Download datasets: HuggingFace API (Mind2Web), git clone (FormFactory), Docker pull (WebArena).
`sft_trainer.py` `online_grpo_trainer.py`	Anyscale job (`make train-submit-sft`)	QLoRA fine-tune Qwen3-8B on FormFactory JSONL (1,240 examples). Online GRPO with browser execution on g5.xlarge (A10G).
`submit_job.py` `*.yaml` configs	On-demand (`make train-submit-*`)	Submit jobs to Anyscale Ray cluster. Custom Containerfiles with Playwright + Chromium for browser-in-the-loop training.

3.7 Verified Results

We verified the pipeline end-to-end with an initial 2-task dev run (Run ID: `20260208_064830_583b722c`, 2/2 success, 109.57s avg), then scaled to a **full 100-task FormFactory evaluation** across two models (Gemini 2.5 Flash, GPT-4o) and two agent types (Agent, CodeAgent) on four separate EC2 spot instances:

Model	Agent Type	Success	Rate	Avg Time	Run ID
Gemini 2.5 Flash	Agent	98/100	98.0%	67.49s	`b4834ad4`
Gemini 2.5 Flash	CodeAgent	100/100	100.0%	47.68s	`0c986d7e`
GPT-4o	Agent	99/100	99.0%	103.49s	`97fb407d`
GPT-4o	CodeAgent	100/100	100.0%	190.53s	`8ef9b0b7`

Key findings:

- **CodeAgent achieves 100% on both models**, while Agent achieves 98–99%. CodeAgent's ability to generate Python code for form interaction makes it more robust for complex forms.
- **Gemini 2.5 Flash is 1.5–4x faster** than GPT-4o across both agent types (67.49s vs 103.49s for Agent, 47.68s vs 190.53s for CodeAgent).
- **GPT-4o CodeAgent was severely impacted by OpenAI 429 rate limits**, with retry delays reaching 45 seconds. This inflated its average execution time to 190.53s (vs 47.68s for Gemini CodeAgent).
- All 3 failures occurred on the **Art Exhibition Submission form**: Gemini+Agent had 2 failures due to `TypeTextEvent` timeouts (DOMWatchdog action handler hung on a specific input element, exhausting the step limit), and GPT-4o+Agent had 1 failure due to a `BrowserStartEvent` timeout on the first task (browser session failed to initialize—a cold-start race condition).
- Per-task artifacts (MP4 video recordings, agent history JSON traces) were uploaded to S3 for all 400 tasks across the 4 runs.

Infrastructure: Each run used a t3.small spot instance ($0.007/hr) in ca-central-1. Total EC2 compute cost across all 4 instances was approximately $0.50. Results uploaded to the S3 results bucket.

3.8 Next Steps

1. ~~Full FormFactory run~~: **DONE**. Scaled to 100 tasks across 4 model/agent combinations (400 total task evaluations). Results: 98–100% success rates.
2. ~~Multi-model comparison~~: **DONE**. Gemini 2.5 Flash vs GPT-4o, Agent vs CodeAgent. Gemini is 1.5–4x faster; CodeAgent achieves 100% on both models.
3. **WebArena integration**: Deploy Shopping + Shopping Admin + Reddit Docker containers on a larger EC2 instance (t3.medium) and run the 182 filtered tasks.
4. **Automated accuracy scoring**: Implement field-level accuracy comparison against FormFactory ground truth (currently only task-level success/fail).
5. **Fine-tuned model evaluation**: Qwen3-8B QLoRA fine-tuning is underway on Anyscale (g5.xlarge, A10G 24 GB GPU). Preliminary results on 25 training prompts with greedy decoding:

Model Configuration	**Prompts**	**Decoding**	**Nonzero Rate**	**Avg Reward**
Qwen3-8B zero-shot	2	greedy	0.0%	0.000
Qwen3-8B SFT-only (QLoRA)	25	greedy	100.0%	0.408
Qwen3-8B SFT+GRPO (QLoRA)	25	greedy	100.0%	0.424

SFT teaches the model to produce executable action plans (without SFT, GRPO achieves 0% reward). GRPO improves avg_reward by +3.9% over SFT-only on training data. Next: evaluate on held-out FormFactory forms via `ollama:` or `vllm://` prefix and compare against foundation model baselines (Gemini 2.5 Flash, GPT-4o).

4 Part Four: Infrastructure as Code Implementation

All infrastructure is provisioned via Terraform (HCL) in `infra/eval/terraform/`. The design separates *durable infrastructure* (Terraform-managed: VPC, S3, IAM, launch templates, SSM) from *ephemeral compute* (script-launched: EC2 spot instances), avoiding state drift when spot instances are interrupted or auto-stopped.

4.1 Repository Structure

```
infra/
  eval/
    terraform/          # IaC definitions (Terraform HCL)
      main.tf           # 341 lines -- VPC, S3, IAM, EC2 template, SSM
      variables.tf       # 67 lines  -- configurable parameters
      outputs.tf         # 35 lines  -- resource IDs for scripts
      user_data.sh       # 124 lines -- EC2 bootstrap script
    pipelines/           # Evaluation Python code
      eval_benchmark.py  # 869 lines -- main orchestrator
      data_loader.py     # 503 lines -- unified dataset loading
      results_schema.py  # 126 lines -- Pydantic models
      eval_config.py     # 94 lines  -- configuration
      eval_report.py     # 128 lines -- reporting
      results_aggregator.py  # 143 lines -- cross-run analysis
    scripts/             # Operations scripts
      launch_eval.sh     # 137 lines -- spot instance launcher
      download_datasets.py   # 499 lines
      download_results.py    # 52 lines
      upload_datasets.py     # 68 lines
      auto_shutdown.py       # 90 lines
  Makefile               # 203 lines -- 30+ operational targets
```

Total infrastructure code: 3,236 lines (567 Terraform/shell + 1,720 pipeline Python + 746 script Python + 203 Makefile).

4.2 Terraform Resources (`main.tf`)

The Terraform configuration creates 23 AWS resources (plus 4 data sources) in a single flat module:

Resource Group	Count	Purpose
VPC + Subnet + IGW + Route Table + Route Table Association	5	Isolated 10.0.0.0/16 network with public subnet in ca-central-1a
Security Group	1	SSH ingress (port 22), all egress
Datasets S3 Bucket + versioning + encryption + public access block + lifecycle	5	Versioned, AES256, IA transition at 30 days
Results S3 Bucket + versioning + encryption + public access block + lifecycle	5	Versioned, AES256, 90-day expiration
IAM Role + Instance Profile + Inline Policy + Managed Policy Attachment	4	S3 R/W, CloudWatch, SSM read, EC2 self-stop, SSM Session Manager
Launch Template	1	t3.small spot ($0.02 cap), 30GB gp3, Ubuntu 22.04
SSM Parameters (2)	2	SecureString placeholders for API keys
Total	**23**	

Key design decisions:

- **Spot instances for cost**: EC2 spot pricing (~$0.007/hr for t3.small) reduces compute costs by 60–70% versus on-demand.
- **SSM lifecycle ignore**: Terraform creates placeholder parameters; real API key values are pushed by `launch_eval.sh` from the local `.env` file. This avoids storing secrets in Terraform state.
- **Hardcoded AMI**: The organization SCP blocks `ec2:DescribeImages`, so we hardcode the Ubuntu 22.04 AMI ID for ca-central-1.

- **Default tags**: All resources tagged with `Project`, `Environment`, and `ManagedBy` for cost allocation and governance.

4.3 Variables and Parameterization (`variables.tf`)

```
variable "project_name"     { default = "openbrowser" }
variable "aws_region"       { default = "ca-central-1" }
variable "instance_type"    { default = "t3.small" }
variable "eval_datasets"    { default = "formfactory" }
variable "eval_max_tasks"   { default = 2 }
variable "eval_models"      { default = "gemini-2.5-flash" }
variable "eval_agent_types" { default = "Agent" }
variable "auto_run_eval"    { default = true }
```

All eval parameters are passed to `user_data.sh` via Terraform's `templatefile()` function, enabling different evaluation configurations without modifying code—only variable overrides are needed (e.g., `terraform apply -var="eval_max_tasks=0"` for a full run).

4.4 EC2 Bootstrap Script (`user_data.sh`)

The 124-line bootstrap script runs on first boot of each spot instance:

1. **System packages**: Python 3.12, uv, git, jq, AWS CLI, Playwright dependencies (fonts, X11 libraries), Xvfb.
2. **Repository clone**: `git clone` of the OpenBrowser-AI repository.
3. **Python environment**: `uv sync --all-extras` for all dependencies, `uv run playwright install chromium` for the browser.
4. **FormFactory dataset**: `git clone --depth 1` of formfactory into `data/formfactory/`.
5. **API keys**: Reads API keys from SSM Parameter Store, writes to `.env`.
6. **Auto-stop cron**: Checks every 10 min if eval is running; if idle 30 min, self-stops via EC2 API (IMDSv2).
7. **Auto-run eval**: If `auto_run_eval=true`, starts Xvfb (display :99) and runs the benchmark, logging to `/var/log/eval_run.log`.

4.5 Launch Automation (`launch_eval.sh`)

The 137-line shell script provides one-command eval deployment:

```
bash infra/eval/scripts/launch_eval.sh
```

It performs five steps:

1. Reads the local `.env` file and pushes API keys to SSM Parameter Store via `aws ssm put-parameter --type SecureString --overwrite`.
2. Reads Terraform outputs (`launch_template_id`, `results_bucket`).
3. Launches a spot instance via `aws ec2 run-instances --launch-template`.
4. Waits for the instance to reach "running" state and retrieves the public IP.
5. Prints monitoring commands: SSH tail of bootstrap and eval logs, SSM Session Manager connect, S3 results download command, and manual termination command.

4.6 Multiple Environments

The pipeline supports three execution environments with identical evaluation logic:

Environment	Trigger	Details
Local	`make eval-local` `make eval-formfactory` `make eval-stress`	Runs on developer machine. Direct Playwright browser (visible). Results saved to `results/`. No S3 upload.
Docker	`make docker-run-eval`	Containerized execution with `--shm-size=2g`. Dockerfile installs Playwright + Chromium. Volume-mounts `results/`. Useful for CI/CD integration.
AWS EC2 Spot	`make eval-aws-launch`	Terraform-provisioned infrastructure. Bootstrap installs all dependencies. Xvfb virtual display. Results uploaded to S3. Auto-stop after 30 min idle.
Anyscale Ray	`make train-submit-online-grpo` `make train-submit-online-flow-grpo`	GPU cluster (g5.xlarge, A10G). Custom Containerfile with Playwright + Chromium. Headless browser against FormFactory.

4.7 Makefile Targets

The `infra/Makefile` provides 30+ targets organized into six categories:

```
# Evaluation
make eval-local           # stress_tests, 5 tasks
make eval-stress          # All 40 stress tests
make eval-mind2web        # Mind2Web, max 20 tasks
make eval-formfactory     # FormFactory, max 10 tasks
make eval-aws-launch      # Launch spot instance on AWS
make eval-aws-results     # Download results from S3
make report               # Generate cross-run markdown report

# Data Management
make download-datasets    # Download all datasets
make upload-datasets      # Upload datasets to S3
make preprocess-formfactory  # Generate training JSONL

# Training (Anyscale Ray)
make train-submit-sft     # Submit Qwen3-8B QLoRA SFT job
make train-submit-grpo    # Submit offline GRPO fine-tuning job
make train-submit-online-grpo       # Submit online AR GRPO with browser execution
make train-submit-online-flow-grpo  # Submit online Flow GRPO with browser execution

# Terraform
make terraform-plan-eval  # Preview infrastructure changes
make terraform-apply-eval # Deploy infrastructure
make terraform-destroy-eval  # Tear down infrastructure
```

4.8 Cost Analysis

Resource	Specification	Cost
EC2 spot (t3.small)	2 vCPU, 2 GB RAM	˜$0.007/hr
EBS gp3	30 GB root volume	˜$2.88/mo (while instance exists)
S3 storage	Results ˜10 MB/run	Negligible
SSM parameters	2 SecureString	$0.15/mo
Dev run (2 tasks, ˜5 min)		¡$0.02
Full FormFactory (100 tasks × 4 configs)	4 spot instances, ˜5h total	˜$0.50

Actual cost for full evaluation (100 FormFactory tasks × 2 models × 2 agent types = 400 task evaluations): approximately $0.50 in EC2 spot compute across 4 instances. Gemini instances completed in ˜1.5 hours each; GPT-4o instances took 3–5 hours due to OpenAI API rate limiting. LLM API costs (external, not AWS): approximately $0.50–1.50 per 100-task run with Gemini 2.5 Flash, $2–5 per 100-task run with GPT-4o.

5 Part Five: Disaster Recovery Demonstration

This section describes the disaster recovery workflow for the OpenBrowser-AI evaluation infrastructure. The entire demonstration is performed exclusively through the command line (Terraform CLI, AWS CLI, and shell scripts) rather than the AWS Management Console. This is a deliberate choice, because all infrastructure is defined as code in Terraform and all operational workflows are scripted, the CLI is the authoritative interface for provisioning, destroying, and restoring resources. Using the web console would contradict the Infrastructure as Code philosophy and introduce manual steps that are not version-controlled or reproducible. The shell-based approach ensures that every action in the demo is repeatable, auditable, and consistent with how the infrastructure is managed in practice.

The screen recording demonstrating the full destroy-and-restore cycle is available at: `https://drive.google.com/file/d/1MY088jKVCVpOFyfTKb-BSm4LebZcMBBs/view?usp=sharing`

5.1 Recovery Architecture

Because the infrastructure is fully defined in Terraform and all application code lives in Git, disaster recovery is straightforward: `terraform destroy` followed by `terraform apply` recreates the entire environment from scratch. No manual console configuration is required.

What is recoverable:

- VPC, subnet, internet gateway, route tables, security group
- S3 buckets (datasets and results) with versioning, encryption, lifecycle policies
- IAM role, instance profile, and permissions policy
- EC2 launch template with spot configuration
- SSM parameters (placeholder values; real keys re-pushed by `launch_eval.sh`)

What is stored externally (not affected by infrastructure deletion):

- All source code and pipeline scripts in Git
- API keys in team members' local `.env` files
- Terraform state file (`terraform.tfstate`) stored locally
- Previously downloaded results (local `results/` directory)

5.2 Disaster Recovery Steps

5.2.1 Step 1: Destroy Production Environment

```
cd infra/eval/terraform
terraform destroy -auto-approve
```

This deletes all 23 AWS resources:

- VPC and all networking components
- Both S3 buckets (datasets and results)
- IAM role, policy, and instance profile
- EC2 launch template
- SSM parameters

5.2.2 Step 2: Verify Deletion

```
# Verify Terraform state is empty
terraform show
# Confirm S3 buckets no longer exist
aws s3 ls | grep openbrowser
# Confirm launch template is gone
aws ec2 describe-launch-templates \
    --filters "Name=tag:Project,Values=openbrowser" \
    --region ca-central-1
```

5.2.3 Step 3: Restore Infrastructure

```
terraform init
terraform apply -auto-approve
```

Terraform recreates all 23 resources with identical configuration. New resource IDs are generated but all functional properties (bucket policies, IAM permissions, launch template settings) are identical.

5.2.4 Step 4: Restore Secrets

```
# Re-push API keys from local .env to new SSM parameters
bash infra/eval/scripts/launch_eval.sh
# (launch_eval.sh Step 1 pushes keys before launching instance)
```

5.2.5 Step 5: Verification

```
# Verify all Terraform outputs are populated
terraform output

# Launch a test eval run
bash infra/eval/scripts/launch_eval.sh

# Monitor bootstrap
ssh ubuntu@<IP> 'tail -f /var/log/user-data.log'

# Monitor eval execution
ssh ubuntu@<IP> 'tail -f /var/log/eval_run.log'

# Download results from S3
```

```
uv run infra/eval/scripts/download_results.py \
    --bucket $(terraform output -raw results_bucket)
```

5.3 Recovery Verification Checklist

Component	Verification Method	Expected Result
VPC + Networking	`terraform output vpc_id`	New VPC ID returned
S3 Datasets Bucket	`aws s3 ls`	Bucket exists (empty)
S3 Results Bucket	`aws s3 ls`	Bucket exists (empty)
IAM Role	`terraform output`	Role ARN returned
Launch Template	`terraform output launch_template_id`	Template ID returned
SSM Parameters	`aws ssm get-parameter`	Parameters exist
EC2 Spot Launch	`launch_eval.sh` output	Instance ID + public IP
Bootstrap	`tail user-data.log`	"Bootstrap complete"
Eval Execution	`tail eval_run.log`	Tasks run, results saved
S3 Upload	`aws s3 ls` results bucket	CSV + JSON + videos

5.4 Data Recovery Considerations

S3 bucket data loss: Destroying Terraform deletes S3 buckets and their contents. Prior evaluation results should be downloaded locally before destruction (`make eval-aws-results`). For production use, cross-region replication or a separate long-term storage bucket outside the Terraform-managed lifecycle would be appropriate.

Terraform state loss: If `terraform.tfstate` is lost, `terraform import` can re-associate existing AWS resources, or a clean `terraform apply` creates new resources. For team use, migrating to a remote backend (S3 + DynamoDB locking) is recommended.

Recovery time: Full infrastructure restoration (`terraform apply`) completes in approximately 30–60 seconds. EC2 bootstrap (package installation, Playwright setup, dataset download) requires an additional 5–8 minutes.

A Figures

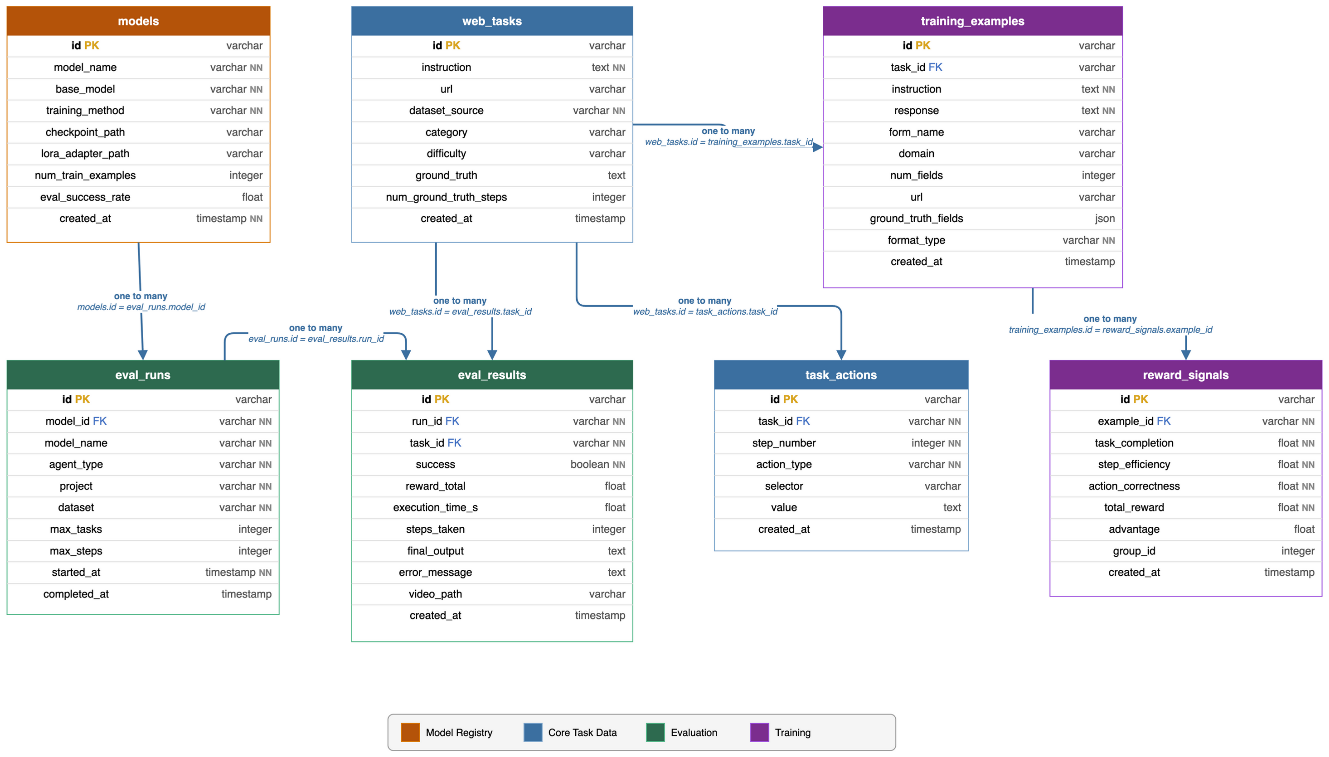


Figure 1: Aspirational Data Schema – unified relational ERD showing 7 tables with primary keys, foreign keys, data types, and NOT NULL constraints. Color-coded by domain.

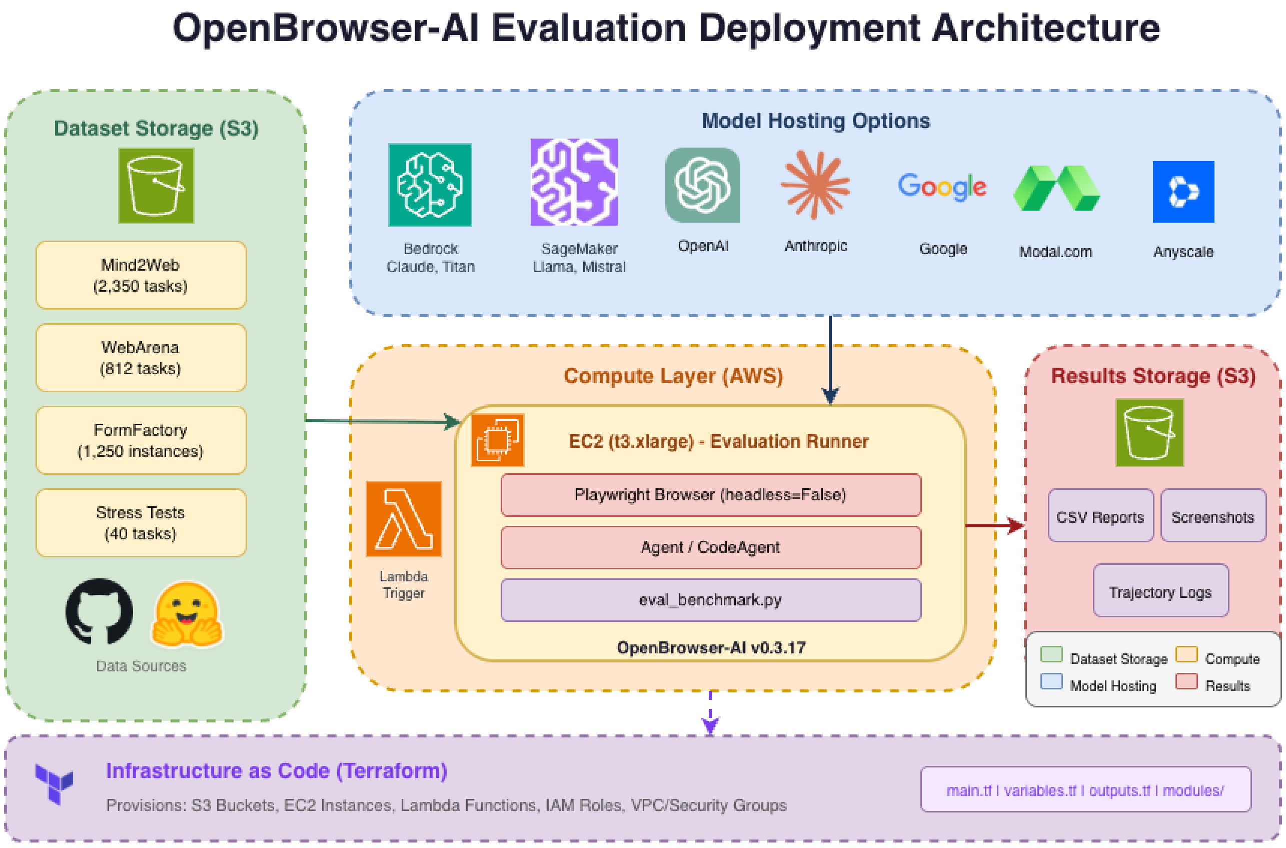


Figure 2: OpenBrowser-AI Evaluation Pipeline Architecture showing dataset ingestion, agent execution, and results collection across local and AWS environments.

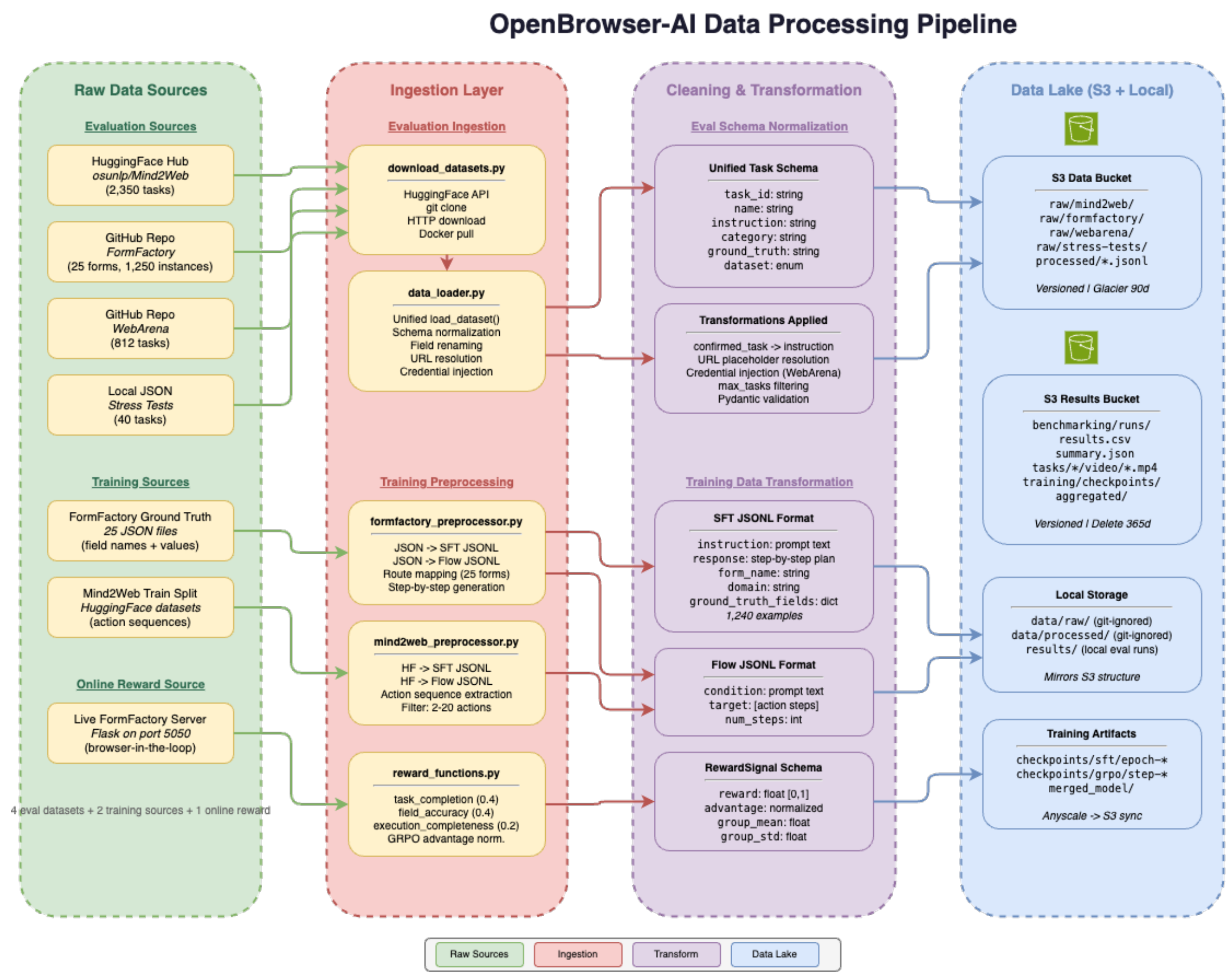


Figure 3: Data Processing Pipeline showing raw data sources, ingestion scripts, cleaning and transformation stages, and data lake storage targets for both evaluation and training workflows.

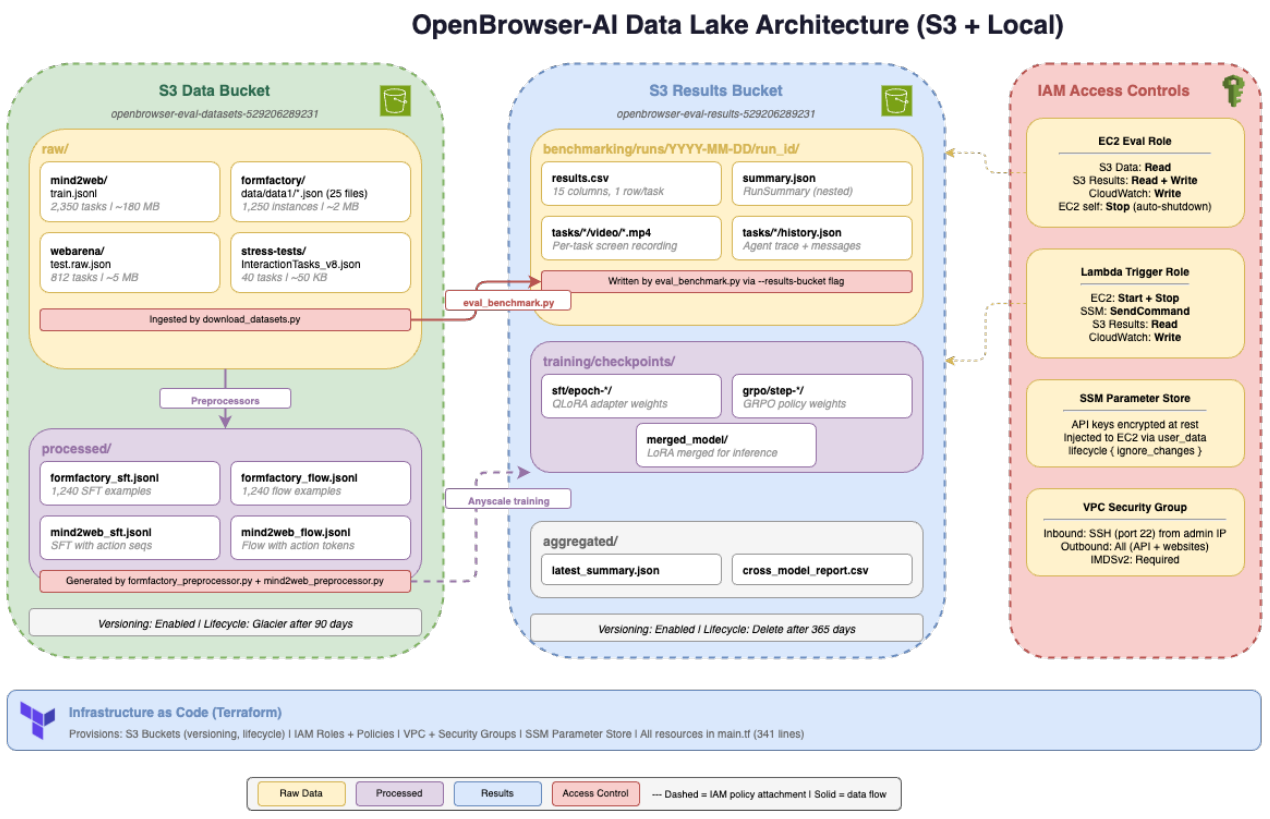
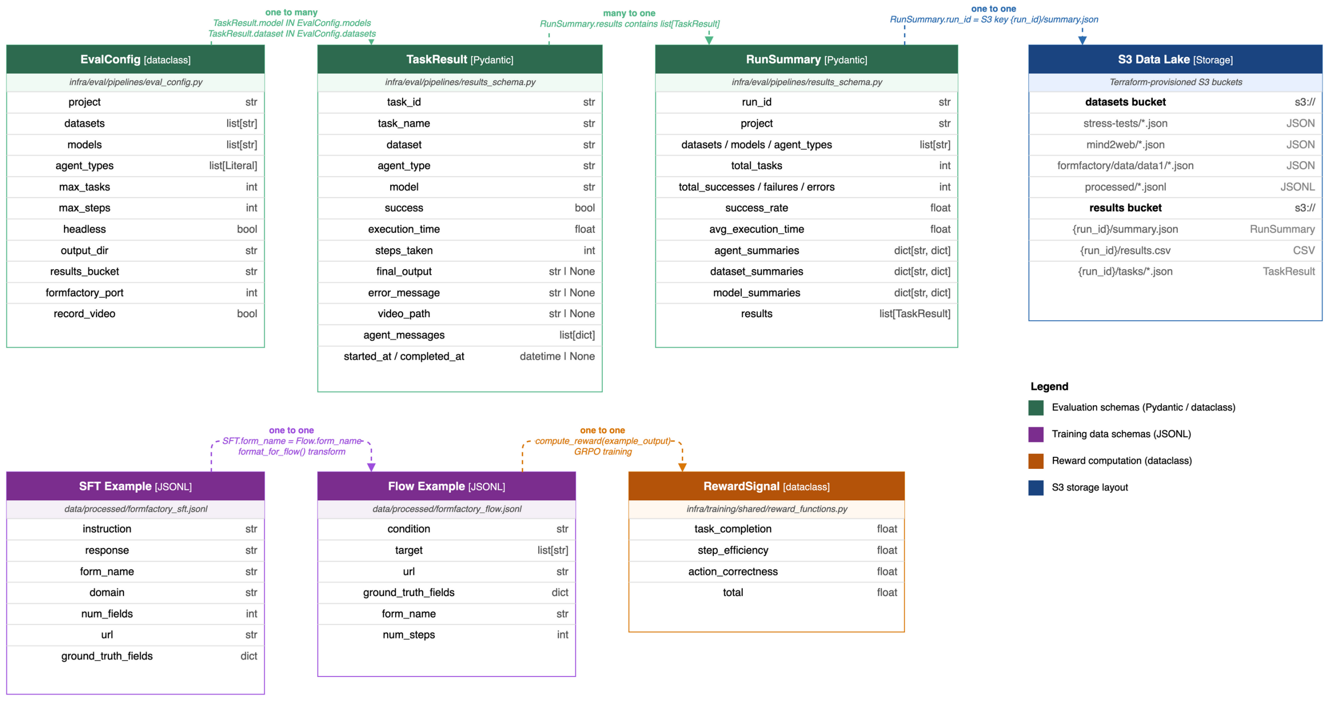


Figure 4: Data Lake Architecture showing S3 bucket structure with prefix-based partitioning, lifecycle policies, and IAM access control patterns.


one to many
TaskResult.model IN EvalConfig.models
TaskResult.dataset IN EvalConfig.datasets
many to one
RunSummary.results contains list[TaskResult]
one to one
RunSummary.run_id = S3 key {run_id}/summary.json
one to one
SFT.form_name = Flow.form_name
format_for_flow() transform
one to one
compute_reward(example_output)
GRPO training
Legend
🟩 Evaluation schemas (Pydantic / dataclass)
🟪 Training data schemas (JSONL)
🟧 Reward computation (dataclass)
🟦 S3 storage layout

EvalConfig [dataclass]	
infra/eval/pipelines/eval_config.py	
project	str
datasets	list[str]
models	list[str]
agent_types	list[Literal]
max_tasks	int
max_steps	int
headless	bool
output_dir	str
results_bucket	str
formfactory_port	int
record_video	bool

TaskResult [Pydantic]	
infra/eval/pipelines/results_schema.py	
task_id	str
task_name	str
dataset	str
agent_type	str
model	str
success	bool
execution_time	float
steps_taken	int
final_output	str I None
error_message	str I None
video_path	str I None
agent_messages	list[dict]
started_at / completed_at	datetime I None

RunSummary [Pydantic]	
infra/eval/pipelines/results_schema.py	
run_id	str
project	str
datasets / models / agent_types	list[str]
total_tasks	int
total_successes / failures / errors	int
success_rate	float
avg_execution_time	float
agent_summaries	dict[str, dict]
dataset_summaries	dict[str, dict]
model_summaries	dict[str, dict]
results	list[TaskResult]

S3 Data Lake [Storage]	
Terraform-provisioned S3 buckets	
datasets bucket	s3://
stress-tests/*.json	JSON
mind2web/*.json	JSON
formfactory/data/data1/*.json	JSON
processed/*.jsonl	JSONL
results bucket	s3://
{run_id}/summary.json	RunSummary
{run_id}/results.csv	CSV
{run_id}/tasks/*.json	TaskResult

SFT Example [JSONL]	
data/processed/formfactory_sft.jsonl	
instruction	str
response	str
form_name	str
domain	str
num_fields	int
url	str
ground_truth_fields	dict

Flow Example [JSONL]	
data/processed/formfactory_flow.jsonl	
condition	str
target	list[str]
url	str
ground_truth_fields	dict
form_name	str
num_steps	int

RewardSignal [dataclass]	
infra/training/shared/reward_functions.py	
task_completion	float
step_efficiency	float
action_correctness	float
total	float

Figure 5: Actual Document Schemas – JSON/JSONL/Pydantic structures used in the current pipeline, with relationships and join keys.

B Existing Benchmark Infrastructure

OpenBrowser-AI has foundational evaluation code predating the AWS infrastructure:

Benchmark Scripts (`examples/benchmarks/`):

- `agent_comparison.py`: Compares Agent vs CodeAgent on specific tasks
- `comprehensive_benchmark.py`: Full benchmark suite using stress tests

Stress Test Dataset (`stress-tests/`):

- `InteractionTasks_v8.json`: 40 tasks with ground truth
- HTML challenges hosted on GitHub Pages

Key Findings from A1 Benchmarks:

- Agent is 2.5x faster for simple UI interactions
- CodeAgent is 3x faster for data extraction tasks
- CodeAgent uses fewer steps by combining operations in Python

Key Findings from A2 Full FormFactory Evaluation (100 tasks $\times$ 4 configs):

- CodeAgent achieves 100% success on both Gemini 2.5 Flash and GPT-4o (vs 98–99% for Agent)
- Gemini 2.5 Flash is 1.5–4x faster than GPT-4o across both agent types
- Gemini 2.5 Flash + CodeAgent is the best configuration: 100% success, 47.68s average, fewest steps
- All failures occurred on a single form type (Art Exhibition Submission) due to DOMWatchdog timeouts
- GPT-4o CodeAgent is heavily impacted by OpenAI 429 rate limits, inflating avg time to 190.53s