ChatGPT

Extending the LinkedIn Easy Apply Bot into a Generic Form-Filling Engine

Research summary

1. Reliable selectors and resilient automation

- **Accessible and user-facing locators are preferred** – Playwright documents and practitioner guides recommend using user-facing locator methods (e.g., `getByRole()`, `getByLabel()`, `getByPlaceholder()`, `getByText()`, `getByAltText()`, `getByTitle()`) instead of brittle XPath or deeply nested CSS. These methods rely on ARIA roles, labels, placeholders and visible text, mirroring how real users interact with pages [1]. User-facing locators automatically wait for elements and return fresh DOM nodes on each action, which improves reliability [2]. A widely read best-practices article emphasises prioritising `getByRole()` and `getByLabel()` and only falling back to CSS selectors when user-facing attributes are unavailable [3]. XPath selectors should be avoided due to brittleness and performance issues [4].
- **Composite selector fingerprints and fallback strategies** – articles on locators recommend chaining locators (e.g., parent + child) and using filters or second arguments to refine matches [5]. When a page lacks stable attributes, scripts should compute multiple selector fingerprints: a user-facing locator, a CSS selector based on id/class, and possibly a fuzzy text match. At runtime the engine should try the primary locator and fall back to secondary locators if the target disappears.
- **Attribute fingerprinting** – to stabilise selectors across DOM changes, store a fingerprint containing key attributes (id, name, aria-label, text content, data-test-id) and use the shortest unique CSS or ARIA role combination. Persist these fingerprints along with the canonical form schema to allow recovery when the DOM structure changes.

2. Handling dynamic multi-step forms and custom controls

- **Multi-step UX considerations** – research on multi-step form design emphasises providing save-and-resume options, clear validation messages and splitting steps into small groups (no more than five fields per step) [6]. Although targeted at form designers, these guidelines inform automation: the engine should detect progress indicators or stepper components and persist interim state so users can resume manually.
- **Detecting step transitions** – multi-step forms often expose progress bars or stepper indicators in the DOM. The engine should monitor for changes in the number of visible input elements or progress bar attributes. When the DOM mutates significantly (e.g., a page navigation event, hidden fields appear), the canonicaliser should update the field list and log the transition. A heuristics-based rule (e.g., "new fields appear and a progress indicator increments") can trigger a new step in the canonical schema.
- **Custom dropdowns and overlays** – typical `select` elements can be automated with `locator.selectOption` or `setInputFiles` for file uploads. However, many modern sites implement dropdowns as div-based popovers. To interact with them, the bot should click the

dropdown container, wait for the menu to appear, and select an option by visible text or index. The automation API should support searching within an overlay and include fallback click coordinates when options are off-screen. For file uploads, Playwright's `setInputFiles` (Python: `locator.set_input_files()`) can upload one or more files [7], but the file path must be relative to the project for tests to run reliably [8].

- **Radio/checkbox groups** – treat radio buttons and checkboxes as grouped fields. When canonicalising the form, group options under a single field and choose the value with highest LLM confidence; the filler should click the corresponding label text.

3. Reproducibility, logging and debugging

- **Persistent contexts and storage state** – using a persistent context allows Playwright to reuse cookies and localStorage across sessions. A guide explains that a persistent context stores session data in a user-specified directory and lets tests run without re-logging each time [9]. This reduces flakiness and simulates a real-world browsing experience [9]. Another article shows how to implement a project-level setup step that logs in once and writes the storage state to disk using `storageState()`, then re-uses it for all tests [10]. The engine should adopt a similar mechanism: login manually in headful mode, save the session to `storageState.json` and reuse it in headless runs.
- **Traces, snapshots and HAR logs** – Playwright's tracing features capture screenshots, network requests and DOM snapshots for each action, viewable in the Trace Viewer. Although the official documentation wasn't accessible, visual-testing articles describe how Playwright stores before/after images and diff images in the `test-results` directory [11]. Visual tests compare the current page to a stored snapshot and report differences [12]. The engine should enable tracing when filling forms and save traces and HAR files per submission, allowing developers to replay failures and update snapshots with `--update-snapshots` [13].
- **Deterministic replay** – storing the HTML snapshot and relevant network responses enables deterministic unit tests. For each form, capture a DOM snapshot (via `page.content()`), the canonical schema, and a screenshot. Use these fixtures to run offline tests with mocked Playwright contexts; this ensures form canonicalisation and LLM answering logic are reproducible.

4. LLM structured-output patterns for forms

- **Structured outputs ensure type-safety** – OpenAI's structured outputs feature enforces that model responses adhere to a provided JSON Schema. The documentation states that structured outputs guarantee reliable type-safety (no need to validate or reformat responses), explicit refusals (refusals are detectable) and simpler prompts [14]. The model returns JSON matching the schema, avoiding hallucinated keys or invalid enum values [14]. The OpenAI SDK supports JSON Schema and Pydantic/Zod definitions in Python and JavaScript [14]. By defining a canonical form schema and per-field answer schema, we can call the model in "json_schema" mode and obtain structured output without brittle prompt parsing.
- **Selecting output mechanisms** – an industry guide compares structured outputs via JSON Schema, function-calling and JSON mode. It notes that JSON Schema provides the strictest guarantees where supported; function-calling is portable across providers; and JSON mode requires robust prompts and client-side validation [15]. For our engine we should use OpenAI's `json_schema` (or function-calling fallback) to generate canonical field descriptors and answers. Prompt engineering should instruct the model to return numeric values only when the field is numeric, select one option for radio groups, and leave fields blank if unknown.

5. Anti-bot and verification handling

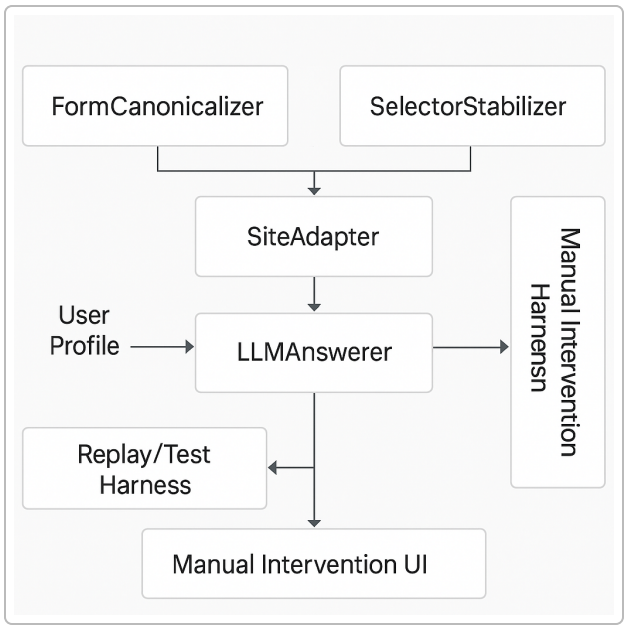
- **Respecting Terms of Service** – security blogs warn that bypassing CAPTCHAs or stealth automation may violate websites' terms of service. An article on bypassing CAPTCHAs explicitly states that bypassing CAPTCHAs "for illegal or malicious motives violates ethical and legal standards" and encourages readers to read the target site's ToS [16]. The engine must avoid evading anti-bot measures: when encountering a CAPTCHA or verification step, it should pause and allow manual user input. Provide a human-in-the-loop UI for solving CAPTCHAs and verifying information.
- **Headful/manual verification modes** – run the automation headfully during dry-run or verification modes. Present a summary of detected fields and proposed answers to the user before submission. This aligns with multi-step form best practices that emphasise clear error handling and save-and-resume options [6].

6. Test and CI strategy for generalised form filling

- **DOM snapshots and deterministic mocks** – to test the canonicaliser and filler deterministically, capture DOM snapshots and store them alongside test fixtures. Use Playwright's `expect(page).toHaveScreenshot()` to produce baseline images; failing tests store before, after and diff images [11]. Tests should run with offline snapshots; a mock Playwright context loads the saved HTML into a new page and runs canonicalisation and filling logic. This allows reproducible unit tests even when the real site changes.
- **Coverage metrics** – measure the percentage of required fields automatically answered. For LinkedIn, the MVP goal is $\geq$90% auto-filled fields; target $\geq$95% after iteration. For the non-LinkedIn PoC, aim for $\geq$80% coverage. Selector fallback success (primary vs. secondary selectors) should be $\geq$95% across snapshots.

Proposed architecture

Below is a high-level architecture for a generic form-filling engine. The diagram shows the components and data flow.



1. **FormCanonicalizer**: Given a live form or a saved DOM snapshot, it extracts a canonical representation of each field. It identifies field id, label, type, options, validation rules, numeric flag and required status, and generates multiple selectors (user-facing and CSS). It persists the canonical form and DOM snapshot for reproducibility.
2. **SelectorStabilizer**: Generates and ranks selector fingerprints for each field (ARIA locator, CSS selector, fallback text). It records attribute hashes and updates selectors when DOM changes. A storage layer stores snapshots and selector metadata under `devdata/forms/<site>/<form_id>.json`.
3. **SiteAdapter**: Implements site-specific logic for canonicalisation and submission. For LinkedIn it adapts the existing `_build_form_config_from_dialog` output into the canonical schema. Additional adapters support other platforms (Greenhouse, Lever, Workday). Each adapter handles login flows, navigation to forms, and detection of multi-step progress bars.
4. **LLMAnswerer**: Uses the canonical form schema and user profile to generate answers. It calls an LLM with a structured output schema and returns per-field answers along with confidence and rationale. It respects numeric fields and selects appropriate options. For unknown fields it leaves the answer blank.
5. **GenericFormFiller**: Consumes the canonical form and LLM answers, matches fields using selector stabiliser, and fills the form using Playwright. It detects step transitions, updates progress, validates numeric input, skips pre-filled values, and supports dry-run or headful modes for manual confirmation.
6. **Replay/Test Harness**: Loads stored DOM snapshots and runs canonicalisation, LLM answering (mocked) and filling logic in a deterministic environment. It supports snapshot tests and diff images for visual regressions. CI runs these tests to ensure changes don't break existing behaviour.
7. **Manual Intervention UI**: Presents a summary of fields and proposed answers to the user in dry-run or CAPTCHA scenarios. Allows manual edits and resumes automation after confirmation.

Canonical form schema

The engine represents any web form as a list of field descriptors. Below is a JSON Schema describing this canonical format:

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "CanonicalWebForm",
  "type": "object",
  "properties": {
    "form_id": { "type": "string", "description": "Unique identifier for the
form instance (e.g., URL hash or DOM snapshot hash)." },
    "site": { "type": "string", "description": "Name of the site or adapter
(e.g., linkedin, greenhouse)." },
    "fields": {
      "type": "array",
      "items": {
        "type": "object",
        "required": ["field_id", "label", "type", "required", "numeric",
"selectors"],
        "properties": {
          "field_id": { "type": "string", "description":
"Stable DOM fingerprint or GUID for the field." },
          "label": { "type": "string", "description": "Human-readable label
associated with the field." },
          "type": { "type": "string", "enum": ["text", "textarea", "dropdown",
"radio", "checkbox", "file", "number", "date", "email", "tel"], "description":
"Normalized field type." },
          "required": { "type": "boolean", "description": "Whether the field is
required." },
          "numeric": { "type": "boolean", "description": "If true, the answer
must be numeric only." },
          "options": {
            "type": ["array", "null"],
            "items": { "type": "object", "properties": { "label": {"type":
"string"}, "value": {"type": "string"} }, "required": ["label", "value"] },
            "description": "List of options for dropdown/radio/checkbox fields;
null for text inputs."
          },
          "validation": {
            "type": ["object", "null"],
            "properties": {
              "minLength": { "type": "integer" },
              "maxLength": { "type": "integer" },
              "pattern": { "type": "string", "description": "Regex pattern for
validation." }
```

```
            },
            "description": "Optional validation rules derived from the DOM
attributes."
          },
          "selectors": {
            "type": "array",
            "items": { "type": "string" },
            "description":
"Ordered list of selector fingerprints (ARIA locator, CSS, text) from primary to
fallback."
          },
          "notes": { "type": ["string", "null"], "description": "Additional
heuristic notes (e.g., hidden until previous step)" }
        }
      }
    }
  },
  "required": ["form_id", "site", "fields"]
}
```

Example canonical form (simplified):

```
{
  "form_id": "linkedin_easy_apply_12345",
  "site": "linkedin",
  "fields": [
    {
      "field_id": "input_first_name",
      "label": "First name",
      "type": "text",
      "required": true,
      "numeric": false,
      "options": null,
      "validation": { "maxLength": 50 },
      "selectors": ["getByLabel('First name')", "input#first-name"],
      "notes": null
    },
    {
      "field_id": "radio_us_work_authorization",
      "label": "Are you legally authorized to work in the United States?",
      "type": "radio",
      "required": true,
      "numeric": false,
      "options": [
        {"label": "Yes", "value": "yes"},
        {"label": "No", "value": "no"}
```

```
      ],
      "validation": null,
      "selectors": ["getByRole('radiogroup', { name: /authorized/i })"],
      "notes": "radio group with two options"
    }
  ]
}
```

Structured output schemas and example prompts

A. Form-config inference schema (LLM output)

This schema instructs the LLM to extract field metadata from a DOM snapshot and produce the canonical field descriptors. Each `FieldDescriptor` corresponds to an entry in the canonical form.

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "FormConfigExtraction",
  "type": "object",
  "properties": {
    "fields": {
      "type": "array",
      "items": {
        "type": "object",
        "required": ["field_id", "label", "type", "required", "numeric",
"options", "selectors"],
        "properties": {
          "field_id": { "type": "string", "description":
"Unique DOM fingerprint for this field." },
          "label": { "type": "string" },
          "type": { "type": "string", "enum": ["text", "textarea", "dropdown",
"radio", "checkbox", "file", "number", "date", "email", "tel"] },
          "required": { "type": "boolean" },
          "numeric": { "type": "boolean" },
          "options": { "type": ["null", "array"], "items": { "type":
"string" } },
          "selectors": { "type": "array", "items": { "type": "string" } },
          "notes": { "type": ["null", "string"] }
        }
      }
    }
  },
  "required": ["fields"]
}
```

Prompt example:

> "You are a form canonicalisation assistant. The following HTML snippet represents a form step. Extract all input fields and return a JSON object matching the provided `FormConfigExtraction` schema. Use the `field_id` as a unique key derived from stable attributes (`id`, `name`, or combination of `aria-label` and index). Recognise dropdowns, radio groups and checkboxes, list their option labels, and set `numeric` to true only for numeric fields. Include multiple selectors (e.g., `getByLabel()`, `input#id`) for each field. Do not include explanatory text—only return valid JSON conforming to the schema."

B. Per-field answers schema (LLM output)

This schema instructs the LLM to produce answers for each field given the canonical schema and user profile.

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "FormAnswers",
  "type": "object",
  "properties": {
    "answers": {
      "type": "object",
      "additionalProperties": {
        "type": "object",
        "required": ["answer", "confidence", "type"],
        "properties": {
          "answer": { "type": ["string", "boolean", "number", "null"],
"description": "The value to fill in." },
          "confidence": { "type": "number", "minimum": 0, "maximum": 1 },
          "type": { "type": "string", "enum": ["text", "textarea", "dropdown",
"radio", "checkbox", "file", "number", "date", "email", "tel"] },
          "rationale": { "type": ["null", "string"], "description": "Brief
explanation of why this answer was chosen." },
          "validation_suggestion": { "type": ["null", "string"], "description":
"Constraints such as 'numeric only' or 'uppercase required'." }
        }
      }
    },
    "unanswered_fields": { "type": "array", "items": { "type": "string" } },
    "meta": {
      "type": "object",
      "properties": {
        "prompt_tokens": { "type": "integer" },
        "completion_tokens": { "type": "integer" }
      },
      "required": ["prompt_tokens", "completion_tokens"]
```

```
    }
  },
  "required": ["answers", "unanswered_fields", "meta"]
}
```

Prompt example:

> "You are a job-application assistant. Given a list of field descriptors (id, label, type, options, numeric flag, required), produce JSON answers matching the `FormAnswers` schema. Use the applicant's profile (e.g., name, location, work eligibility) to fill required fields. For numeric fields, return only digits. For dropdowns and radio groups, choose the option that best matches the profile. Provide a confidence score between 0 and 1 and a short rationale. Leave unknown answers null and include their field_ids in `unanswered_fields`. Output valid JSON only."

Prioritised implementation plan

MVP (2–3 weeks)

1. **Create canonical form modules**
   Add `src/forms/schema.py` defining the canonical JSON Schema and a Python dataclass equivalent. Include utility functions to validate and serialize forms.
   Add `src/forms/adapters.py` with an abstract `SiteAdapter` class and a `LinkedInAdapter` that wraps the existing `_build_form_config_from_dialog` function, converting its output into the canonical schema. Persist DOM snapshots and canonical forms under `devdata/forms/linkedin/<timestamp>.json`.
2. **Implement selector stabiliser**
   Add `src/forms/selector_stabilizer.py` to generate ordered lists of selectors using Playwright locators (`get_by_role`, `get_by_label`, fallback CSS). Save selector fingerprints into the canonical schema. Add a function to re-compute selectors for a field using a fresh page.
3. **Extend LLM client**
   Modify `openai_client.py` to accept a JSON Schema parameter and call the OpenAI API using `json_schema` structured-output mode [14]. Implement functions `infer_form_config(dom_html) -> canonical_form` and `generate_answers(form: CanonicalForm, profile) -> answers`. Add new prompt templates in `prompts.py` reflecting the examples above.
4. **Generic form filler**
   Refactor `apply_core.py` or equivalent to consume the canonical form and LLM answers. Use the selector stabiliser to locate elements, fill values, skip pre-filled fields, validate numeric inputs, handle radio groups, and detect multi-step transitions. Preserve existing behaviours (progress detection, numeric validation, skip prefilled fields, dry-run mode, and logging/screenshot capture). Provide a headful dry-run mode that pauses before final submission and displays a summary of fields and answers.
5. **Deterministic test harness**
   Add `tests/test_form_generalization.py` that loads saved DOM snapshots from `devdata/forms` and runs canonicalisation, LLM (mocked) and filling logic in a mocked Playwright context.

Use Playwright's snapshot testing to verify that the DOM is filled correctly and that selectors resolve. Provide fixtures for a LinkedIn Easy Apply dialog and at least one other site.

Version 1 (4–6 weeks)

1. **Site adapters** – Implement adapters for at least one common external platform (e.g., Greenhouse). Capture a sample job-apply form as a snapshot (`devdata/forms/greenhouse/<id>.html`) and implement `GreenhouseAdapter` to parse its fields. Write tests to assert ≥80 % required fields auto-answered.
2. **Selector learning** – Add heuristics to update selector fingerprints based on success or failure; if the primary selector fails but a fallback succeeds, promote the fallback. Periodically update stored selectors in the canonical form.
3. **User profile management** – Introduce a `user_profile.yaml` config with fields used by the answerer (name, contact info, work eligibility, etc.). Allow users to customise per-site values.
4. **Improved manual intervention UI** – Build a minimal web UI or terminal output summarising unresolved fields and allowing manual entry. When CAPTCHAs or anti-bot challenges are detected, pause automation and open the UI.
5. **Additional LLM providers** – Abstract the LLM interface to support Anthropic, Azure OpenAI or local models. Add fallback to function-calling when `json_schema` is unsupported [15].

Version 2 (future improvements)

1. **Machine-learned selector stabilisation** – Train a model (or use heuristic rules) to predict the most stable selectors across snapshots. Use page diffing to detect structural changes and update selectors automatically.
2. **Dynamic conditional logic** – Implement runtime evaluation of conditional logic; if the answer to one field determines which subsequent fields appear, re-invoke the canonicaliser after each step and merge new fields.
3. **Parallel form submissions** – Allow concurrent job applications using multiple browser contexts while respecting site rate limits and ToS.
4. **Analytics and metrics dashboard** – Track form coverage, success rate, time per application, and provide dashboards for monitoring progress and identifying failures.

Proof-of-concept plan for a non-LinkedIn site

- **Select a platform** – Greenhouse is widely used for job applications. Create a minimal fixture by applying to a dummy Greenhouse job, saving the DOM of the application form (`page.content()`), capturing a screenshot, and storing them under `devdata/forms/greenhouse/<sample>.html` and `.png`.
- **Implement** `GreenhouseAdapter` – Derive field descriptors by querying labels (`label[for]`), input types, select options and radio groups. Map them into the canonical schema. Many Greenhouse forms use standard HTML inputs, so user-facing locators should suffice. Add logic to detect the "next" button and progress bar; treat each page as a separate form step.
- **Add tests** – In `tests/test_greenhouse.py`, load the saved snapshot, run `FormCanonicalizer` and ensure that at least 80 % of required fields are identified. Mock the LLM to return answers, run `GenericFormFiller` in a mocked Playwright context, and verify that inputs are populated. Assert that progress detection and step transitions work.

- **Metrics** – Measure coverage (fields auto-answered/required fields), fallback selector success, and end-to-end runtime. Compare results across multiple snapshots.

Acceptance criteria & metrics

- **Functional reproducibility** – Given a saved DOM snapshot and a fixed user profile, canonicalisation and answer generation must produce identical output on repeated runs. Filling the form in a mocked context should yield the same filled HTML across runs.
- **Coverage** – The MVP must automatically answer ≥90 % of required fields on LinkedIn Easy Apply (measured over multiple job postings). After implementing selector stabilisation and additional heuristics, the target is ≥95 %. The Greenhouse PoC should achieve ≥80 % coverage.
- **Robustness** – When the primary selector fails due to DOM changes but a fallback selector exists, the filler should still locate the field in ≥95 % of test snapshots. The engine should skip fields that are pre-filled and correctly handle numeric validation.
- **Test reliability** – CI tests using DOM snapshots and mocked contexts should pass consistently. Snapshots should only change when the underlying UI changes (updated via `--update-snapshots` flag). Trace and HAR files must be stored for each failed E2E test.
- **Manual/human-in-the-loop** – The engine must detect CAPTCHAs and verification steps and pause for manual input, aligning with ToS guidelines [16].
- **Security and privacy** – Profile data should be stored securely and only required fields should be used. Provide a mechanism to redact or delete stored answers. Do not recommend bypassing CAPTCHAs or anti-bot measures; always respect site policies [16].

Security, privacy and ToS checklist

1. **Respect website terms** – Do not bypass CAPTCHAs or anti-automation measures. Pause for manual intervention when such measures appear [16]. Provide clear user warnings about potential ToS violations.
2. **Data minimisation** – Only collect and store user data required for applications. Do not save unnecessary personal information.
3. **Storage of traces and snapshots** – Store DOM snapshots, traces and HAR files in a secure `devdata` directory. Provide a CLI command to purge old snapshots.
4. **Session management** – Use persistent contexts to avoid repeated logins and reduce risk of account locks [9]. Store `storageState.json` separately from code and exclude it from version control.
5. **Transparent AI behaviour** – Use structured outputs to ensure that model responses adhere to schemas, reducing unpredictable behaviour [14]. Log prompt tokens and completion tokens for cost visibility.
6. **Manual overrides** – Provide a UI for reviewing and editing answers before submission. Record manual edits for auditing.

---

[1] [2] 9 Playwright Best Practices and Pitfalls to Avoid | Better Stack Community
https://betterstack.com/community/guides/testing/playwright-best-practices/

[3] [4] [5] Playwright Locators: Best Practices for Robust Test Automation
https://www.bondaracademy.com/blog/playwright-locators-best-practices

6 3 Multi-Step Form Best Practices
https://www.formassembly.com/blog/multi-step-form-best-practices/

7 8 How to upload files with Playwright
https://timdeschryver.dev/blog/how-to-upload-files-with-playwright

9 Using Persistent Context in Playwright for Browser Sessions | by Anandkumar | Medium
https://medium.com/@anandpak108/using-persistent-context-in-playwright-for-browser-sessions-c639d9a5113d

10 Speed Up Playwright Tests with Shared StorageState
https://www.checklyhq.com/blog/speed-up-playwright-tests-with-storage-state/

11 12 13 Getting Started with Snapshot Tests in Playwright - DEV Community
https://dev.to/mikestopcontinues/getting-started-with-snapshot-tests-in-playwright-4gfj

14 Structured model outputs - OpenAI API
https://platform.openai.com/docs/guides/structured-outputs

15 Structured output from LLMs: more than just prompt engineering | by Paweł Twardziak | Sep, 2025 | Medium
https://medium.com/@twardziak.p/structured-output-from-llms-more-than-just-prompt-engineering-b47408a0f8d3

16 How to Bypass CAPTCHA With Playwright
https://oxylabs.io/blog/playwright-bypass-captcha