ChatGPT

Feature: Live Metadata Display for Job Analysis (Step 2)

Feature Name	**Live Job Metadata Display (Post-Parse)**
Objective	Improve user trust & transparency to boost Verified Placement Rate (OKR VPR-120) by showing complete, accurate job details immediately after parsing.
Persona Focus	Alex (Job Seeker) – also extensible to Maya (Recruiter) & Jordan (Analyst) use cases
Impacted Systems	**Frontend:** Job Analysis UI (dashboard & results components) **Backend:** Parsing pipeline (WebLLM parsing, platform parsers), Analysis API **Database:** `job_listings` (parsed fields), `sources` (URL metadata), `analyses` (results), `key_factors` (signals) [1] **AI/Agents:** WebLLM `JobFieldValidator` & parser registry (field confidence), Ghost Analysis algorithm (consumes parse results)
Success Metrics	*UX:* ≥90% of analyses show title & company within 2s, all fields <3s [2] [3]; high user trust (e.g. NPS > 80). *Accuracy:* ≥85% field parsing accuracy across platforms [4] (minimize "Unknown" fields). *Business:* Increase Verified Placement Rate (VPR-120) by ensuring users quickly identify real jobs and avoid ghost listings.

1. Product Context & Goal

Alex is a job seeker using Ghost Job Detector to analyze a job posting and determine if it's a "ghost job." In the current flow, after Alex submits a job URL, the system parses the posting and proceeds to analysis. However, **the Step 2 UI (post-parse metadata display) is minimal – showing only Job Title, Company, and a few "Key Factors"** (positive signals) [5] [6]. This limited view hinders transparency and user confidence. To support the OKR of increasing Verified Placement Rate, we aim to enrich this step with *all* relevant job details the system has parsed, **giving Alex immediate, comprehensive insight into the posting before and during the ghost analysis.**

Objective: Provide an elegant, real-time display of the structured job metadata (title, company, location, posted date, source/platform, and a snippet of the description) as soon as they are parsed. By transparently showing these fields with live updates, Alex can verify that the system "got it right" (e.g. correct job title/company) and trust the ensuing ghost analysis. This will reduce uncertainty, encourage Alex to follow through with legitimate jobs, and thus improve the Verified Placement Rate (more real job applications and placements resulting from our analyses).

This feature aligns with the platform's emphasis on **AI transparency and user trust** – complementing the real-time AI reasoning terminal [7]. It ensures the *inputs* to the ghost detector are clearly visible, reinforcing trust in the outputs (risk scores and recommendations). Ultimately, a clearer Step 2 experience helps users

like Alex make informed decisions, increasing the likelihood they pursue genuine opportunities (boosting VPR-120).

2. User Experience & Functional Requirements (Frontend)

2.1 Step 2 UI – Comprehensive Metadata Card

After Alex clicks **"Analyze Job"**, the UI will present a **Job Details card** immediately once parsing begins. This card will display all key metadata extracted from the job post, updating field-by-field in real time as parsing progresses:

- **Job Title & Company** – Display prominently (e.g. as a header) in an "`<Title>` **at** `<Company>`" format. These usually parse first and should appear within ~2 seconds [2]. If available, use capitalization as in source.
- **Location** – Show the job's location (city/region or "Remote") with an appropriate icon (e.g.). This field might parse slightly after title/company; it should populate as soon as the parser confirms it [8]. If parsing yields multiple locations or "Multiple locations", display accordingly (comma-separated or "Multiple locations").
- **Posted Date** – Display when the job was posted (or last updated). Show as a relative age ("Posted X days ago") if possible, for quick insight (and consistency with analysis factors) [9] [10]. For example: "🛈 Posted 10 days ago". If only an exact date is available, format it clearly (e.g. "Posted on Aug 1, 2025"). This field is crucial for ghost detection (stale vs recent) [9] [10], so exposing it adds transparency.
- **Source / Platform** – Indicate the posting source with a label or icon. Derive a user-friendly source type from the URL (e.g. "Source: LinkedIn (Job Board)", "Source: Company Careers Page") [11] [12]. If available, use the new `platform` field computed in the backend (e.g. "LinkedIn", "Indeed", "Workday/Company Site") [13]. This lets Alex know *where* the job is posted (important since postings only on third-party boards can be a red flag [11] [12]). The source field should include a hyperlink to the original URL for reference, using the domain as link text (e.g. "jobs.company.com").

- **Job Description (Preview)** – Show an excerpt of the job description text, truncated to ~3–5 lines. Use a slightly smaller or muted font for this section to distinguish it. At the end of the preview, include a "… **Show more**" link or toggle that expands the card to reveal the full description text. The full description can appear in a scrollable container or modal for readability. This allows Alex to review the job's details without leaving our app. The preview should appear once we have parsed the description (likely after other fields, but still within the initial ~3s parse) – e.g., show a loading indicator in this section until available. **Important:** preserve basic formatting (line breaks, bullet points) in the description text for fidelity, but strip any HTML. Long descriptions (e.g. >1000 characters) should definitely be collapsed by default to keep the UI tidy.

- **Key Factors (Positive Signals)** – Continue to display the key positive signals identified (if any) as was done previously, but now within this metadata card. These could be shown as badges or a bulleted list titled "Key Factors Identified". For example, if the posting is recent and on a company site, show badges like "Recently Posted" or "On Company Site" [10] [12]. This gives immediate feedback on legitimacy signals even before the full analysis results. (These are derived from the same fields above – e.g. a "Recently posted" badge if postedAt $\leq$30 days [10], or "Company Site" if sourceType indicates

ATS [12] – so they will populate as those fields are parsed.) Key factors should be visually distinct (e.g. green pill-shaped labels or an info icon list) to catch Alex's eye.

Layout & Styling: The metadata card should be clean and easy to scan: - Use a card or panel with a subtle border or background to distinguish it from the rest of the page. - Title/Company as a header (e.g. larger text or bold). - Other fields in a 2-column grid or list format with icons/labels for clarity (responsive design should collapse to one column on mobile). For example: a row with a location pin icon and location text, a row with a calendar icon and posted date, etc. - Ensure the "Show more" for description is obvious (underline or chevron icon) and that collapsing back is possible ("Show less"). - If any field is not available, omit its row entirely or show as "N/A" if critical – see error handling below.

Real-Time Field Updates: As the parser extracts each field, update the UI immediately: - Use a subtle loading placeholder for each field initially (e.g. grey skeleton text or spinner next to the field label) so the card layout is in place. For example, show a grey bar under "Location" label until the actual location text is parsed. - When a field's value arrives, populate it and remove the placeholder. Consider a short fade-in highlight to draw attention to newly filled data. - The Title & Company should appear almost instantly (they're usually parsed first) – if not, show "Parsing title..." briefly. Location and posted date might follow shortly; description last. The card should **not** wait to be fully complete – partial data is acceptable and expected in the first 1–3 seconds. - **Confidence gating:** Only render a field once we're reasonably confident in it to avoid flicker or showing incorrect info. The backend will provide confidence scores per field (via `JobFieldValidator`/parsers) [8]. If a field's confidence is below a threshold (e.g. 0.7), we could continue to treat it as "loading" or show a warning indicator until confirmed. (In practice, parsing accuracy after Phase 2 is high – 80–85%+ for key fields [4] – so low-confidence cases should be rare.)

2.2 Live Parse Progress UX

To reinforce that the system is actively working (and not stalled), implement a live progress indicator for the parsing step: - A small animated indicator (e.g. dots or spinner) next to the "Analyzing..." status text or on the metadata card until all fields are filled. For instance, the card header could show "Parsing job details... ⏳" which turns into "Job Details ✓" when done. - Alternatively or additionally, use the existing AI thinking terminal logs (if open) to reflect progress – e.g., logs already show messages like "💎 Analyzing: *Title at Company*" and " Source: ..." [14] [15]. Ensure these logs align with the UI field updates (the UI shows the same title/company at roughly the same time the log prints it). - Once all metadata fields are populated (parse complete), the UI can either automatically proceed to display the analysis results (ghost score, risk factors) below, or keep the metadata card in focus and show a "Analysis complete" message before listing detailed results. This depends on the existing flow: likely, the ghost analysis (scores) is fast (milliseconds) [16] and can be shown almost immediately after. We should ensure the user notices the metadata card is complete before drowning it in other info. A short delay (e.g. 500ms) or a subtle highlight on the completed card can help signify "all fields ready".

2.3 Error & Fallback UI States (Frontend)

Robust error handling is crucial for a good UX: - **Partial Parse Failure:** If certain fields cannot be parsed from the source (e.g. the posting has no clear location or the parser is uncertain), the UI should handle this gracefully. For any missing field after parse completes: - Show a placeholder like "Unknown" or "Not available" next to that label (in italics or a lighter color) rather than leaving it blank. For example: "Location: Unknown". Include a tooltip on the label explaining "This detail could not be parsed from the posting." - Do

not show fields that are entirely not applicable. E.g. if `postedAt` is not obtained, we can omit the "Posted" row to avoid confusion, *unless* we want to explicitly convey it's unknown. Since our system's accuracy is high post-improvement, such cases are expected to be rare [17] [18]. We prefer a clean UI, so likely omit missing fields unless the team decides to show "N/A" for transparency. - **Complete Parse Failure:** In the unlikely event the parser fails to extract even the basics (title/company) – which prior to Phase 2 was common but now should be exceptional [17] [19] – handle as follows: - Display a brief error message on the card like "⚠ Could not extract job details. Please check the URL or try again." - The analysis process should abort or not proceed to ghost scoring if core fields are missing. The UI should reflect that (no ghost score shown). Possibly provide a fallback action, such as: allow Alex to manually input the title/company if we choose to support that in future (though the prior manual correction feature was removed for simplicity [20]). For now, a message and suggestion to try another link is sufficient. - **Networking/Timeout issues:** If the parse API call fails (network error, etc.), show a notification (toast or inline message) like "Error: Unable to retrieve job posting details. Please check your connection." and allow re-submission. The metadata card can remain in a "loading" state with a spinner in such cases, or be hidden if parsing never started. - **Expansion edge cases:** If the description is extremely long or contains unusual formatting: - Ensure the "Show more" expansion can handle it (maybe open a modal with scroll if too large to comfortably expand inline). - If description text includes sensitive or unsupported content (rare for job posts), still display it fully – it's user-provided content – but if there's any filtering (like profanity or PII) needed by policy, that should be handled at parse time. Generally, we display what's on the job post.

Throughout error states, **do not crash the entire analysis view**. If parse fails, ideally the system should still allow the ghost job algorithm to run on whatever data is available (though with missing fields the result may be unreliable). The UI should handle a scenario of "analysis completed but with low confidence due to missing data" by messaging accordingly. (For example, if title/company unknown, the ghost score might be meaningless – better to prompt the user to try another posting or contact support.)

2.4 Accessibility & Responsiveness

- **Semantic Markup:** Structure the metadata information in a semantically clear way for screen readers. A recommended pattern is a definition list (`<dl>`) with each field label as a `<dt>` and value as `<dd>`. This way, a screen reader will read "Title: Software Engineer at Acme Corp. Location: San Francisco, CA. Posted: 5 days ago." etc. Alternatively, use appropriate headings and paragraphs with ARIA labels. Ensure the "Show more" link is keyboard-accessible and announces itself (e.g. `aria-expanded` on the description container).
- **Live Region:** Consider marking the metadata container or individual field values as an ARIA live region (polite) so that screen reader users get notified as fields update in real time. For example, the location field can have `aria-live="polite"` so that when the text changes from "Loading location..." to "Location: New York, NY", it's announced.
- **Color & Contrast:** Use sufficient color contrast for text (follow WCAG AA at minimum). If using colored badges for key factors (e.g. green background), ensure the text is readable (white on green with enough contrast). Avoid conveying status purely by color; include an icon or text ("Recently Posted" label is self-descriptive, for instance).
- **Keyboard Interaction:** All interactive elements (like the description expand link, or perhaps a copy-to-clipboard for the URL if provided) must be focusable via keyboard. The expansion should be toggleable with Enter/Space. If a modal is used for full description, trap focus inside it and provide a clear close button.

- **Responsive Design:** On smaller screens (mobile, tablets), the metadata card should resize gracefully:
- Likely the layout will stack vertically: Title/Company (still prominent), then each field on a new line. We should avoid a wide table layout; instead use flex or block layout that wraps.
- Ensure long text (like an extended description or a very long title) doesn't overflow the screen – use CSS to wrap words or truncate with ellipsis if needed (for titles, maybe truncate after a certain length on small screens with the full title in a tooltip).
- Test on common mobile viewport widths to ensure readability without horizontal scrolling.
- **Testing:** We will verify the accessibility by using screen reader testing (NVDA/VoiceOver) and ensure that dynamic updates are announced. We will also test on various screen sizes and devices for responsive behavior.

3. Backend Requirements & API Design

3.1 Parsing Pipeline Enhancements

The backend parsing system (WebLLM client-side model + platform-specific scrapers) must reliably extract the full set of metadata fields for each job posting URL: - **Ensure field availability:** The parser should populate `url`, `title`, `company`, `location`, `postedAt`, and `description` for every analysis whenever the source provides them. Recent Phase 2 improvements have already boosted extraction accuracy (80–85% for title/company across LinkedIn, Workday, Lever, Greenhouse, etc. [4]). We must verify that: - *Location:* is captured by each platform parser. For example, the LinkedIn parser should pull the location text; Workday/Greenhouse might have it in the HTML; the LLM can assist if needed. If any existing parser lacks location extraction, update it accordingly. - *Posted Date:* is parsed. We likely store a timestamp or date string as `postedAt`. The algorithm currently uses this to compute recency [9], so it is being set. Confirm each parser (or the LLM) finds a posted date or "X days ago" string and converts it to a date. Implement any needed date parsing logic (e.g. relative times to actual date). - *Source Type:* Determine how to classify `sourceType` or `platform`. We have logic in the algorithm to identify if a URL is a job board vs company site [11] [12]. Rather than duplicating string checks on the frontend, the backend should provide a field for this. We have the new `platform` computed property in v0.1.8 [13] – likely a string like "LinkedIn" or "CompanySite/Greenhouse". If not already, implement a mapping in the parse result: e.g. set `sourceType` = "Job Board" or "Company ATS" or "Unknown" based on URL. This can be done during platform detection. - *Description:* Extract the full text content of the job description. Likely our `raw_documents` storage already holds the HTML or text. The parser (or WebLLM) should return a cleaned text version of the description for display. Ensure we strip HTML tags and scripts for safety, but preserve line breaks and lists as plaintext. If the description is extremely long, consider sending a truncated preview in the initial response to lighten payload, with an option to fetch the full text on demand (though for simplicity, sending full text is fine since typically few KB). - **Data confidence & validation:** The backend's `JobFieldValidator` (AI agent) and `CrossValidationService` should continue to verify these fields. For instance, cross-check that the company name matches known patterns (we have company normalization via Levenshtein etc. [21]), or that the title wasn't a generic fallback. If any discrepancies or low confidence, the backend could flag it. For now, expose a `parsingConfidence` score (already stored per job listing) [22]. We will use that for logging/metrics rather than UI, except possibly gating as noted. - **Performance:** The parser must deliver the metadata quickly. We target <3 seconds for full parse completion and field population (most in ~2s) [2] [3]. This is achievable given the extraction speed benchmark (<2s per URL) [2]. To meet this: - Ensure asynchronous operations in parsing (network fetch, LLM inference) are optimized. For instance, start fetching the HTML immediately, and if using the WebLLM

in-browser, make sure the model is loaded in advance (perhaps on app load or in a warmup step) to avoid the 10s load penalty [2] during first analysis. - If model loading is needed, consider showing a specific loading state ("Loading AI model...") separate from field placeholders, so user knows it's working. But ideally, keep that within the initial delay and still show partial info ASAP. - If the user's browser does not support WebGPU or the model (meaning it falls back to server), ensure the server function is likewise speedy. The Node backend should perhaps run the same parsing logic (maybe via smaller model or regex scrapers) and return fields swiftly. - **Streaming/Pipelining:** The backend should stream results if possible. Instead of waiting to parse everything then sending one response, it could send data as it's obtained. Two approaches: 1. **Incremental API responses (Streaming):** Implement the `/api/analyze` endpoint to flush partial data (e.g. first the title/company, then later fields) over a single connection. This could be done via Server-Sent Events (SSE) or web socket messages. For example, the API sends an event "field_parsed" with `{field: "title", value: "Software Engineer"}` as soon as title is found. The frontend would subscribe and update accordingly. This provides the best real-time feel. 2. **Short-Polling:** If streaming is complex to add on our current stack (serverless on Vercel might complicate persistent connections), implement a quick polling mechanism. For instance, after initiating analysis, the frontend polls an endpoint like `GET /api/analysis-status?analysisId=XYZ` every 0.5s, which returns the currently parsed fields and their status. The analysis process can save fields to the DB or an in-memory cache as they are parsed, and the status API reads from there. Given the short timeframes, even a single poll after 1s and 2s might suffice. The goal is to avoid a noticeable gap of nothing happening. 3. **Parse-Preview Endpoint:** Note that we already have a `POST /api/parse-preview` endpoint [23]. We can leverage this by making it return the parsed fields quickly (without waiting for ghost score). The front-end flow could be: call `parse-preview` on URL submission to get metadata JSON (as soon as ready), populate UI, *then* automatically call the full `/api/analyze` (or perhaps `analyze` can be split to take the already parsed data to skip re-parsing). For simplicity, if `analyze` currently triggers its own parse, we might accept the double parse cost in favor of simpler integration, but ideally we reuse results: - **Backend Integration:** We could modify `analyze` to check if an analysis for that URL is already in progress or recently parsed. For example, `parse-preview` could create a `job_listing` record and return an `id`. Then `analyze` can use that record (passing `jobListingId`) to avoid re-fetching content. This requires some refactoring of `analyze` API (allow passing existing parsed data or an ID reference). If not trivial, an alternative is to call `analyze` first and have it internally yield partial results to the client (back to streaming idea). - We will coordinate with backend dev to choose between streaming vs polling vs parse-preview approach based on what's feasible with our architecture. The **primary requirement** is that the system *supports delivering field-level updates in real-time** to the client.

3.2 API Response Contracts

We need to define the data structure that the frontend will receive for this metadata, whether via the existing analyze API or a new mechanism: - The **Analysis API (`/api/analyze`)** should include all parsed fields in its response. Currently, it likely returns an analysis result object that includes title, company, and maybe the algorithm outputs. We must extend or ensure it contains `location`, `postedAt`, `sourceType/platform`, and a `descriptionSnippet` or full description text. For example, a response JSON might look like:

```
{
  "analysisId": "12345",
  "url": "...",
```

```
  "platform": "LinkedIn",
  "title": "Software Engineer",
  "company": "Acme Corp",
  "location": "San Francisco, CA",
  "postedAt": "2025-08-15T00:00:00Z",
  "description": "This is the full job description ...",
  "keyFactors": [ "Recently posted (5 days ago)", "Posted on company career
site/ATS" ],
  "riskFactors": [ ... ],
  "ghostScore": 0.32,
  "verdict": "Likely Legitimate",
  ...
}
```

Here `platform` (or `sourceType`) is a human-readable string or enum. `postedAt` could be ISO timestamp or an integer (days ago) – clarify and stick to one (timestamp is more precise; frontend can compute days). If these fields are already part of the analysis record in the DB, just expose them. According to our schema audit, many details are stored, sometimes redundantly in JSON [24]. With v0.1.8 improvements, data is more normalized (e.g. KeyFactor table instead of JSON) [25], but the API can still assemble the needed output. - If using **parse-preview**, define its output contract. Likely it returns a subset like:

```
{
  "jobListingId": "abc123",
  "url": "...", "platform": "...", "title": "...", "company": "...",
  "location": "...", "postedAt": "...", "description": "...",
"parsingConfidence": 0.95
}
```

It might not include ghost analysis fields (score, riskFactors) since it's just parse. The frontend would call `analyze` after (or the backend could even trigger analysis in background). We should ensure consistency: the fields names/types are the same across both responses so the UI can handle seamlessly. - **Field formats:** - `postedAt` – use a format that is easy for frontend to parse into "X days ago". ISO 8601 string is fine. Or supply an integer `daysSincePosted` directly from backend (since backend already calculates it for algorithm) [26]. Possibly include both: exact date and a human diff. - `platform` vs `sourceType` – We need clarity on naming. Perhaps we define `platform` as the specific site (LinkedIn, Indeed, Greenhouse, Workday, etc.), and `sourceType` as category ("Job Board", "ATS"). We can include both if useful, but one might suffice for UI. Let's say the API returns `platform` (specific) and maybe a boolean like `isCompanySite` as well. However, to avoid confusion, a single field like `sourceType: "LinkedIn (Job Board)"` can be constructed. We'll coordinate with design how they want it phrased. - `description` – return as plain text (UTF-8). If extremely long, we might limit length (e.g. first 10000 chars) for payload. The remainder could be fetched on expansion via another call or just included if not too big. - `keyFactors` – list of strings (already the algorithm generates this [10] [27]). Ensure they are included in either parse result (some key factors can be determined immediately from parse data, like the two examples above) or certainly in the final analysis result. We will display them as part of Step 2 UI, but they are also part of the final result summary. - **No breaking changes**: Maintain backward

compatibility. If the analyze API is already consumed elsewhere (history, etc.), ensure new fields don't break anything. We might simply be extending the response. The `analysisHistory` API should also include these fields so that when Alex or Maya views past analyses, the metadata is visible there too, not just live.

3.3 Database Considerations

Leverage the existing schema to store and retrieve the needed metadata: - The `job_listings` **table** (19 fields) is the primary store for parsed job data [28] . It should already have columns for `title` , `company` , likely `location` , `posted_at` , and possibly `description` (or at least an ID linking to full text in `raw_documents`). We need to confirm: - If `description` is not stored fully in `job_listings` due to size, we might need to fetch it from `raw_documents` . The system currently stores an HTML snapshot in `raw_documents` and some normalized fields in `job_listings` . It may have a `descriptionSnippet` or similar. We should add a convenient way to get the description text. Option 1: store a text excerpt in `job_listings.description` (maybe done already via parsing). Option 2: query `raw_documents` when needed (by sourceId) to get text content. Given performance concerns, better to extract text once and store it for quick access. - `posted_at` field: ensure it exists and is filled. If not, add it in schema (DateTime). The parser should convert relative times to an absolute timestamp for storage. - `location` : ensure a field exists (likely does, as location is a basic attribute). - We also saw fields like `parsingConfidence` , `extractionMethod` in job_listings [22] . These are more for internal use/metrics, but we should store the confidence for each field as well if needed for analysis. We might not need separate columns per field confidence; the parse algorithms themselves handle it. - The `sources` **table** (11 fields) stores source metadata, possibly including the URL and type (e.g. `type: "URL"` vs `"PDF"`). If the URL is stored there, the job_listing might reference `source_id` . Ensure that linking and retrieval are working so we can show the original URL (for the hyperlink). We might not need anything else from sources for this feature. - The `key_factors` **table** (6 fields) is used now to store positive/negative factors in normalized form [29] . Since we are showing Key Factors, ensure that when analysis completes, the relevant positive factors (with type "positive") are inserted here. The UI can simply use the `keyFactors` array from the API (the API likely still composes it for convenience), but under the hood it's coming from this table. No schema change needed; just confirm all factors have human-readable text (they do, like "Recently posted (5 days ago)" etc.). - **No new tables** are anticipated. We are mostly reading from what parse and analysis already store. We should audit if any field is missing: - If `location` or `posted_at` were not originally in `job_listings` (perhaps because older versions didn't use them in the algorithm), we need to add them. Given location was being validated in logs [30] and postedAt used in v0.1.7 algorithm, likely they exist now. If not, a migration to add them is required. - Ensure indices if needed: e.g. if we allow filtering by company or location in history, those indices might exist (the audit suggests an index on `[company, createdAt]` for company analysis [31]). No immediate performance concern since we just fetch by primary key or join by foreign keys which should be indexed. - **Data flows:** When an analysis is initiated, a `job_listings` entry is created (or re-used if duplicate URL hashed – duplication logic exists to avoid re-analysing the same posting [32]). Then an `analysis` record (with ghost score, etc.) is created linking to it [1] . The front end might currently wait only for the analysis result. We will adjust so that the front end effectively taps into the `job_listings` data as soon as it's available, rather than waiting for the full analysis. - If using polling, we might call an endpoint that reads `job_listings` by analysis ID or by URL to get fields. - If using SSE, the events can be triggered server-side as soon as `job_listings` is populated (perhaps our backend can send an event when the DB write occurs – though in serverless, easier might be to send just before writing). - **Extensibility:** Designing this with future journeys in mind: - Maya (a recruiter persona) might upload a list of jobs or have a feed; our data model already supports multiple sources and analysis history. The metadata display component could be reused to show each job's details in a list or a comparison view. The structured

fields in the DB make it possible to present consistent info for each job. - Jordan (perhaps an analyst or admin) might be interested in aggregated data; ensuring each analysis entry has these fields stored means they can be used for analytics (e.g., how many jobs had location “Remote”, how often postedAt was older than 90 days, etc.). By fully capturing these fields now, we support such future BI queries (some might already be happening in `stats.js`, which currently uses jobListing.company grouping [33]). - Also consider non-URL inputs: if in future we allow PDF job descriptions or screenshots (maybe Maya’s use case), the same fields should be populated (perhaps via OCR/LLM). The UI should be able to display them just the same. Our schema is general enough for that, as `sources` can hold PDFs. Just ensure naming is not URL-specific (we use “Source” generically).

4. Real-Time Update Mechanism

As noted, real-time updates are essential for this feature. We outline how it will work in practice: - **Initiation:** When Alex submits a URL, the frontend will immediately call the analysis pipeline (via either `parse-preview` then `analyze`, or directly a modified `analyze` that streams). The UI renders the metadata card with placeholders right away. - **Streaming option:** The backend sends back chunks of data as they become ready. For example: 1. Initial response chunk: `{ title: "Software Engineer", company: "Acme Corp" }` (and perhaps `analysisId`). 2. Next chunk: `{ location: "New York, NY" }`. 3. Next: `{ postedAt: "2025-08-20T...Z" }`. 4. Next: `{ descriptionSnippet: "Join our team to build..." }` (or full description). 5. Finally: the analysis results (ghostScore, factors, etc.) or a flag that parsing is complete.

The frontend code (likely using Fetch API with `ReadableStream` or an EventSource for SSE) will update the UI on each message. We’ll use an identifier to know which field the chunk represents and place it in the right spot. - **Polling option:** The frontend might do: - Call `POST /api/analyze` which immediately returns an `{analysisId}` and perhaps whatever is instantly available (maybe title/company if quick). - Then enter a loop to `GET /api/analysis-status?analysisId=XYZ` every 500ms. This status endpoint will check the DB or in-memory cache: * It could look up the `job_listings` record by analysisId and return any fields that are non-null along with a flag for done. * Or if easier, the backend can keep an in-memory map of analysis progress (since analysis is short-lived, a simple object in the lambda or a shared Redis cache could store fields as keys when parsed). But storing in DB is fine as the parse writes to job_listings. - The polling stops when all expected fields are present or when analysis is done. - Note: the analysis likely completes very shortly after parse, so we might incorporate analysis completion in the same status. But the UI specifically wants the parsing fields ASAP, so we treat that as the critical path; the ghost score can come a moment after without issue. - **WebLLM (Client-side) nuance:** If the user’s browser is doing the parse via WebLLM (thanks to our WebGPU integration [34] [8]), we have an interesting scenario: the parsing is happening on the frontend itself. In this case, we should tap into that process: - The WebLLM parsing code can be instrumented to stream partial results to the UI directly (since it's running in the browser, perhaps in a web worker). We can expose callbacks when a field is extracted. For example, when the LLM identifies the job title in the HTML, call a JS callback to update UI. - This would bypass needing an API round-trip for those fields, making it extremely fast (sub-second possibly). The client could then send the final parsed data to the server for ghost analysis. - However, implementing that might be complex; if easier, we can still go through the API for consistency. But note, the system was designed with client-side AI to offload the server [35]. So for performance, leveraging that on the client is ideal. We should consult with the ML engineer to see if the WebLLM parsing function can return incremental results or if it only returns after full parse. - If we cannot easily get incremental results from WebLLM, we might just display nothing until it finishes (which is typically <2s anyway). But the spec goal is a “live update feel”, so even a two-second

silent period is not ideal. Perhaps we can break the LLM prompt into steps (like first ask it for title/company, then for location, etc.). This might be overkill; an easier approach: run lightweight regex scrapers concurrently to get title/company quickly while the LLM does the rest. In fact, platform-specific heuristics likely already catch title/company instantly [36] (like Workday URL giving company, etc.). - **Synchronization:** Ensure that our real-time updates do not conflict with the final rendering of analysis results: - The metadata card should remain visible even after the ghost score and verdict are shown. Likely it will stay at the top of the results section as a summary of the job analyzed. The ghost score/risk factors can be shown below it or in a side panel. This way, Alex always can refer back to the job info while reading the analysis. - When analysis is complete, if any final adjustments to metadata occur (e.g. the model revised a field), update it. (Unlikely – parse is done by then. But consider if an external verification step updated something, e.g., the system found a more official company name via LinkedIn – if so, update the company field accordingly.) - **Logging & Debug:** We will enhance logging to support this feature. The backend should log each field parsing event (e.g., "Parsed title: X" at time T) both to the server log and possibly to the AI thinking terminal (as it already does for some fields [14] [15]). We'll ensure the logs align with UI updates for easier debugging. If a field is parsed but not showing up, logs will help identify if it's an API issue or frontend issue.

5. Ghost Analysis Integration

While the core ghost job detection algorithm isn't changing in this feature, we must ensure it works in tandem with the updated parsing flow: - The algorithm currently expects inputs: `{ url, title, company, description, postedAt }` [37]. Our parsing improvements ensure those are filled (location is not explicitly used in v0.1.7 logic). The algorithm uses postedAt for recency checks [9] and the URL (source) for job-board vs company-site check [11], and it scans `description` for language cues [38]. By surfacing these same elements to the UI, we make the analysis more explainable: Alex can see, for example, "Posted 60 days ago" on the card and understand why a risk factor "Stale posting" was flagged [39]. - **Live vs Final Data:** The ghost analysis will typically run after parse is done (immediately). So by the time the user is digesting the metadata, the ghost score and factors will appear. We need to present them in a coordinated way: - Possibly we highlight any field that directly led to a risk/key factor. For instance, if `postedAt` was 60 days ago, the system adds a risk factor "Posted 60 days ago (stale)" [40]. We could visually mark the Posted Date field in red or with a warning icon to tie it to that risk. Similarly, if `sourceType` came out as "Job board only", mark it with a warning since the algorithm adds a risk for that [41]. Conversely, a positive key factor "Posted on company site/ATS" [27] could be indicated with a green check by the Source field. This kind of cross-highlighting would enhance transparency. **Requirement:** Frontend should have the mapping of which factors relate to which fields: * Posted recency factor relates to postedAt. * Company site/board factor relates to url/sourceType. * Language factors relate to description content. * (Title and company themselves might not have direct factors unless extremely generic or long titles trigger something – algorithm does have minor checks like "generic title" +5% [42]. We could in future highlight title if it's flagged generic.) - Implementing this highlighting is a nice-to-have within this feature scope. At minimum, ensure the key factors list is shown (which already tells the user these things). - **Performance of analysis:** The ghost score computation is very fast (rule-based, <1ms) [43], so it won't be a bottleneck. It should not impede the field updates. We just need to ensure that if the analysis result is ready before some fields (unlikely, because analysis depends on fields), we don't show the score first. The sequence is naturally parse -> analysis, so it should be fine. - **API changes for analysis:** None significant besides including the parse fields in the response as discussed. The `analyzeJob` function might not currently accept location or sourceType as separate parameters (only uses URL, etc.), but in future if we enhance algorithm (v0.2+ might incorporate location, etc.), our pipeline is ready to supply them. We won't modify algorithm logic in this feature, but by capturing location now we pave the way for using it (e.g., perhaps flagging if location is

inconsistent with company HQ in future). - **Verified Placement Rate rationale:** By having these details clearly displayed and tied to analysis, Alex can make an informed decision ("This looks like a legitimate post – it's on the company site and posted 5 days ago"), which increases the chance he'll apply and potentially get placed, as opposed to ignoring a ghost posting. The transparency also fosters trust; if Alex trusts the analysis, he's more likely to continue using the tool for other applications, improving overall placements. We will monitor metrics like how often users proceed to click the original job link or mark a job as useful after seeing the metadata and analysis. A hypothesized improvement is a higher rate of applications to non-ghost jobs identified by our tool (thus a higher Verified Placement Rate).

6. Performance & Monitoring

To ensure this feature meets its targets: - **Speed:** We set a target that **initial fields (title, company)** display **within 2 seconds** of submission for >=90% of analyses, and **all metadata fields within 3 seconds** for >=95% (under normal load). This aligns with our system SLA of ~2s response time even under 100 concurrent users [3] . We will leverage the already optimized parsing (<2s) [2] and make use of client-side processing when available to meet these goals. - We will add performance telemetry: measure time from click "Analyze" to each field populated. This can be logged (perhaps as events like `parse.title.time=1200ms`, `parse.complete.time=2500ms`). If we detect consistently slower times, we'll investigate (maybe certain sites take longer – we might then optimize those parsers or degrade gracefully). - **Load testing:** Because this feature might introduce polling or streaming, we'll test how that behaves under load. If 100 users are polling every half-second, does it strain the backend? Likely not with a lightweight endpoint and short lifespan, but we can adjust the interval if needed. If using SSE, ensure the serverless function can handle many open connections or switch to a more suitable deployment (we might consider moving SSE to a separate Node process if needed, but that's beyond MVP scope). - **QA checks:** Automated QA can verify that after analysis, the UI shows all fields and none are null. We will write tests for a few sample URLs (maybe using a headless browser in CI with known job posts) to ensure the metadata appears correctly and quickly. We also include checks that for a known outdated post, the "Posted date" and corresponding risk factor are shown, etc., to validate end-to-end correctness. - **Regression risk:** This feature touches parsing and front-end. We must ensure it doesn't break existing history views or analysis flows. The history page should now show these metadata fields too for past analyses. We may need to backfill location/postedAt for older records if we want them visible (not strictly required, but nice if data available). Since the question scope is Alex's live journey, focusing on new analyses is fine, but we note the history UX alignment.

7. Cross-Functional Notes

- **Design/UX:** The design team should produce a high-fidelity mock of the new metadata card (web and mobile) with the field layout and loading states. They should also design the icons for location, posted date, source, etc., consistent with our style. The expansion interaction for description should be specified (animation, modal vs inline). Design should also consider how this card coexists with the analysis results section – perhaps as a sticky header when scrolling through analysis details.
- **Frontend Dev:** Will implement the React component for the metadata card. This likely involves creating a new component (e.g. `JobMetadataCard.tsx`) that accepts a job object (or subscribes to analysis context) and updates state field-by-field. Frontend dev will also handle the streaming or polling logic to feed data into this component. Using React state or context to store partial parse results will be needed. They should ensure that the component is easily reusable (for example, could be used in a "Analysis Details Modal" as well, or for Maya if she views multiple jobs).

- **Backend Dev:** Will adjust the API endpoints as discussed. Key tasks include enabling partial result delivery (which might involve refactoring how the analyze function works – potentially splitting parse and analysis steps or enabling an event stream). Also, updating platform parsers to fill any missing fields (e.g., ensure `postedAt` extraction for all). Database migrations if needed (to add missing columns or indexes) should be done carefully (the system uses Prisma – update schema and run `prisma migrate`). They must also update any relevant unit tests for parse correctness.
- **ML Engineering:** Should confirm that the WebLLM integration is robust for these fields. The ML team might want to adjust the LLM prompt or logic if it currently doesn't handle location or posted date reliably. For example, maybe add a prompt instruction like "Extract: title, company, location, posted date, description." If the LLM tends to hallucinate a posted date, rely more on regex. The ML validation team has emphasized trust – so ensure any AI-extracted field is cross-checked (e.g. if LLM says posted "10 days ago", verify the raw text had that substring). This could be part of the `CrossValidationService` improvements [44].
- **Analytics/Monitoring:** We should track usage of this feature. Add an event when metadata card is fully rendered, and perhaps if user clicks "Show more" (meaning they are engaging deeply – could be a good sign of trust/interest). If possible, track if Alex clicks the original job link from our UI (we can instrument that button). Those are proxy metrics for Verified Placement follow-through.
- **Extensibility for Maya & Jordan:**
- For Maya (likely a recruiter or HR professional using the system to audit their company's postings or competitors), this metadata display can be reused in her workflow. If Maya uploads a job description PDF or uses an internal ATS feed, the same fields (title, company, etc.) apply. We ensure the architecture supports parsing from multiple source types (URL or file) – which it does via `sources` table and a unified parse interface. The UI component should not assume the input was a URL; it could say "Source: PDF Document" or "Source: Internal Posting" in those cases (we can derive that from sourceType).
- For Jordan (maybe a product analyst or power user), the interest might be more in aggregated data or verifying the system's decisions. The metadata being fully exposed means Jordan can better analyze why the system made certain calls. E.g., they might export analysis history including these fields to do further analysis (the DB now stores them relationally, which is easier to query than JSON blob [24]). So, by implementing this feature, we are also effectively ensuring these fields are properly stored and accessible for any internal analysis Jordan might do.

In summary, this feature will deliver a **richer, more transparent Step 2** in Alex's journey. By seeing the job's title, company, location, posting recency, source, and description immediately – with dynamic updates – Alex gains confidence that the system understands the job posting correctly. This trust, combined with the subsequent ghost job risk assessment, will help Alex focus only on viable job opportunities. In turn, this drives the objective (VPR-120): more verified (real) job placements stemming from our platform. The implementation will involve close collaboration across frontend, backend, ML, and design teams to ensure the UI/UX is seamless and the technical delivery is robust and fast.

Sources:

- Ghost Job Detector parsing & analysis pipeline (WebLLM integration) [17] [19]
- Algorithm use of metadata (posted date, source site) [9] [11] and positive signals definitions [10] [12]
- WebLLM field extraction and validation process [8]
- Phase 2 parser improvements (accuracy and speed metrics) [4] [2]
- Database schema overview (JobListing fields, Source, KeyFactor) [28] [1]

- Removal of manual corrections (trust in auto-parse) [20]
- AI transparency and real-time update principles [45] [14]

---

[1] [22] [24] [28] [29] [31] [33] DATABASE_SCHEMA_AUDIT_REPORT.md
file://file-BKLbYD5b5zTDiBDqym8SJd

[2] [4] [17] [18] [19] [36] [44] PHASE2_EXTRACTION_LOGIC_FIX.md
file://file-WPrQ5hX1Y5sjSjwShRwK6Z

[3] ML_VALIDATION_RESEARCH_REPORT.md
file://file-PmLUAk4Z6bJRrFiBdS7jkd

[5] [6] [9] [10] [11] [12] [15] [16] [26] [27] [37] [38] [39] [40] [41] [42] [43] GHOST_JOB_DETECTION_ALGORITHM.md
file://file-Rg9DTF3NK5dprZYdzNK4fq

[7] [35] [45] GhostJobDetector_PRD_v0.1.7.md
file://file-9VCtRnRH2TENSaVgXGV2bc

[8] [13] [20] [25] [34] CLAUDE.md
file://file-UuSVcXHXcLnYYxbDe2vm4n

[14] [30] AI_THINKING_TERMINAL_DEMO.md
file://file-F6rgPBAqVpes8BU5oP5B5G

[21] [32] ARCHITECTURE.md
file://file-Uncukd1m8J5AUsTvCkfTNR

[23] CURRENT_IMPLEMENTATION_PARSING.md
file://file-QnuHcNFybbYGHTSj6gYw8j