Technical Addendum — Tools, Pipeline Stages, Inputs & Outputs

*Companion to *The Municipal BD Intelligence System: A Step-by-Step Replication Playbook* (TUJ Edition)*

> **TUJ edition v1.5 — October 24, 2026.** Mirrors the case-study page and the Build Guide; the three are published together.

Audience note. This addendum is written for students who have completed a basic Python course and are new to agent building. It documents, at the technical level, what the original system was actually built with, what flows into and out of every stage, and how each piece maps to programming concepts you already know.

1. The Toolchain — What Was Actually Used

The original system was **not** built with a code editor or a coding copilot. It was built by *instructing an AI agent* — the builder wrote specifications, schemas, templates, and rules; the agent produced the artifacts. The complete stack:

Layer	Tool	What it does in this system
Agent platform	**Google Gemini — a "Spark" agent**, running a current Gemini model (the model under Spark changes over time — see the case-study page, Step 1.1, Platform facts)	The runtime that executes the weekly pipeline: browses portals, reads documents, updates strategy artifacts, renders the dashboard
Web access	The agent's built-in **real-browser browsing**	Essential: many procurement portals are JavaScript single-page apps; only a real browser render exposes their content (a plain HTTP fetch returns an empty shell)
Reasoning & extraction	The LLM itself (a current Gemini model)	Reads posting pages, CIP documents, agendas, and bios; extracts structured records that conform to the schemas in §2
Interface	**A single HTML file**: HTML + **Tailwind CSS** (CDN) + **Font Awesome** (CDN) + **vanilla JavaScript**	The dashboard. No framework, no build step, no server — it opens from `file://` or any static host
State	**Browser `localStorage`** inside the dashboard + **JSON export/import**	`localStorage` persists the dashboard's state between sessions; the JSON file is the backup *and* the interchange format handed from one weekly run to the next
Scheduling	A **scheduled agent run** (weekly, on a fixed schedule)	The trigger that starts each cycle — the equivalent of a cron job
Distribution	**Google Drive** archive + **email** push	Each cycle's dashboard file is

		archived to a dated Drive folder and emailed to the operator
Methodology content	Miller Heiman templates (Gold Sheet, dossier, fit rules)	Embedded in the agent's instructions as schemas and checklists — content, not code

Explicitly not used in this build (students often ask): **GitHub Copilot** — no. **Cursor** — no; no code editor was involved, because no application code was hand-written. **Python** — not in the production build. **A database server, a backend API, user accounts** — none; the design deliberately trades those away for a zero-infrastructure artifact (see the playbook's R8).

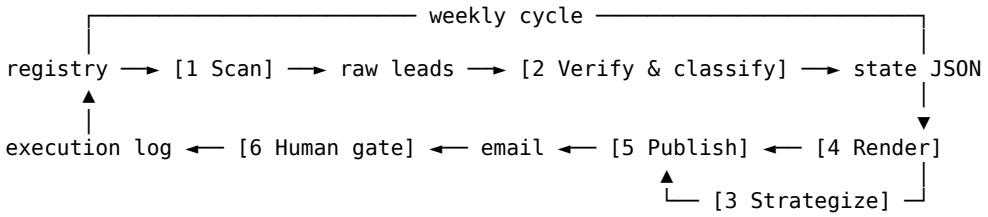
> **The one-sentence mental model.** The "program" in this system is not source code. It is five things: **(1) data schemas, (2) prompt modules, (3) a state file, (4) a schedule, (5) a rendered artifact** — plus a human verification gate. Everything below is an expansion of that sentence.

2. The Pipeline, Stage by Stage

One weekly cycle runs six stages. For each: the actor, the exact input, the transformation, and the exact output.

Stage	Actor / tool	Input	Transformation	Output
0. Configure (once, then as needed)	Human + agent, conversationally	Portal registry worksheet; methodology templates (Gold Sheet, dossier); fit rules; the firm's capability inventory	The human states the methodology as instructions; the agent restates it as schemas and checklists for confirmation	The agent's **standing instruction pack**: registry + schemas + guardrails (playbook §7, R1–R8)
1. Scan (weekly)	Agent with real-browser browsing	The portal registry; last cycle's state JSON (to diff against)	Visits every registered portal — including JavaScript SPAs — plus CIP/agenda/appointment sources; collects candidate changes	A **raw candidate list**: unverified leads (postings, plan items, expirations, appointments)
2. Verify & classify	Agent applying the fit rules; official posting pages as ground truth	Raw candidate list; the two-tier schema; fit/prioritization rules	Each candidate is checked against its official source, then classified: Tier 1 (verified live) or Tier 2 (early signal) or rejected	Updated **state JSON**: Tier 1 records (ID, deadline, link — all verified) and Tier 2 records (catalyst, timing — *no deadline field exists*)
3. Strategize	Agent, using the methodology templates	Updated state JSON; existing Gold Sheets; public sources for dossiers	Rewrites affected Gold Sheet sections; logs gaps as Red Flags; refreshes six-dimension dossiers for officials on active	Updated **Gold Sheets + dossiers** inside the state; a Red Flag list that becomes next

			pursuits	week's first actions
4. Render	Agent generates the file; vanilla JS renders at view time	State JSON; the dashboard's module layout (playbook §6, O1–O9)	The agent emits the complete single-file HTML with the new state embedded; in the browser, JS functions render each module from state	**This cycle's dashboard HTML file** — the single artifact everyone reads
5. Publish	Scheduled run + Drive + email	The rendered HTML file	Archive a dated copy; email the file to the operator	A permanent, dated **archive** + the operator's inbox copy
6. Review gate	**Human operator** (15–30 min)	The dashboard; spot-checks at official sources	Accept / correct / reject each change; write the week's actions and observations into the execution log	The **approved state** and the **execution log** — which become Stage 1's diff-baseline next week. *The loop closes here.*



Two design points deserve emphasis for new builders:

- **The schema is a contract.** A Tier 2 record *has no deadline field* — not "the field is usually empty," but the data structure does not contain it. In the prompt-based build this rule is enforced by the schema text, the Step 2.2 checkpoint, and the weekly audit — discipline, not physics. A code build can make it structural: a Python `dataclass` without a `deadline` attribute cannot accidentally carry one (§9 of the case-study page describes that optional extension).
- **JSON is the system's bloodstream.** Every stage consumes and produces the same state JSON. The dashboard is a *view* of that JSON; the archive is a *snapshot* of it; next week's scan *diffs* against it. If you understand JSON, you understand this system's architecture.

3. Anatomy of a Prompt Module — What the Builder Actually Writes

If the schemas are the data types, the **prompt modules** are the functions. Each stage's instruction to the agent follows the same six-part anatomy:

Part	Purpose	Python analogue
ROLE	Who the agent is for this task ("You are the BD analyst for an engineering practice...")	The function's name and intent

CONTEXT	The inputs, pasted or referenced (registry excerpt, prior state, the account's Gold Sheet)	Arguments
TASK	The transformation, as numbered steps	The function body
RULES	Hard constraints — the guardrails (e.g., "never state a deadline you have not seen on the official posting")	Assertions and input validation
OUTPUT SCHEMA	The exact fields of the result, in order	The return type
EXAMPLE	One correct input→output pair	A unit test / doctest

Worked example — the Stage 2 verification module (template, following the system's documented rules):

> **ROLE:** You are verifying municipal solicitations for an engineering BD dashboard.
>
> **CONTEXT:** Candidate list (attached); portal registry; fit rules (core / adjacent / out-of-scope service lines).
>
> **TASK:** For each candidate: (1) open the official posting page — not an aggregator; (2) confirm the solicitation ID, the deadline (date *and* time), and copy the direct link; (3) classify fit; (4) if any of ID/deadline/link cannot be confirmed on the official page, do **not** guess — move the item to the Red Flag list with the reason.
>
> **RULES:** Tier 1 requires all three of ID + verified deadline + direct link. Early signals go to Tier 2 with a catalyst and timing window; Tier 2 output has no deadline field. Public sources only.
>
> **OUTPUT SCHEMA (per item):** agency · solicitation_id · scope_one_line · application_deadline · question_deadline · submission_deadline · opening_date · portal · direct_link · fit · status.
>
> **EXAMPLE:** *(one verified item shown in full, plus one Red-Flagged item shown with its reason).*

Students should notice: nothing in this module is "AI magic." It is a specification precise enough that a careful human could execute it — the agent simply executes it at machine speed and scale. ***Writing the specification** is *the programming.**

4. Screenshots — The Artifact, Module by Module

All figures are taken from the **fictional-data demo twin** of the dashboard — the same file shared with the class (`BD_Dashboard_Example.html`). Open it and compare as you read: every module below is live in that file.

Figure 1 — Header, controls, and the KPI band. The DEMO banner (the R5 guardrail, made visible); the cycle selector; the **Backup JSON** and **Save Changes** buttons — the JSON interchange that Stages 5–6 run on; and the KPI counts: portals monitored, active verified solicitations, pre-RFP strategic targets, Gold Sheets maintained, and the golden outreach window.

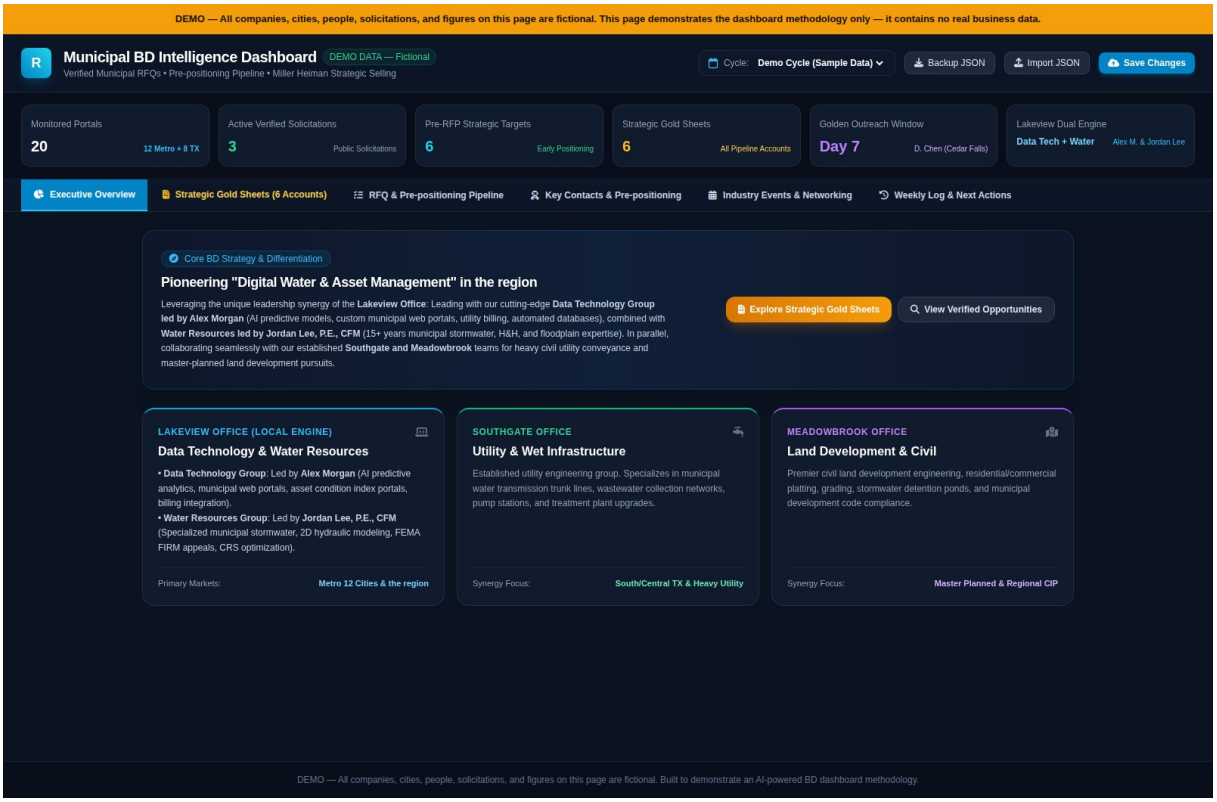


Figure 1 — Dashboard header, JSON controls, and KPI band (fictional demo data).

Figure 2 — A Gold Sheet. One priority account on one scrollable page: account selector and **Print Sheet** button at top; then Account Overview, Response Mode, the Buying Influences matrix (each player with their Personal Win and Business Result), Competitive Landscape, Sales Strategy & Win-Results, and the Action Plan ordered by Red Flag elimination. This one-page artifact is what the whole system exists to maintain.

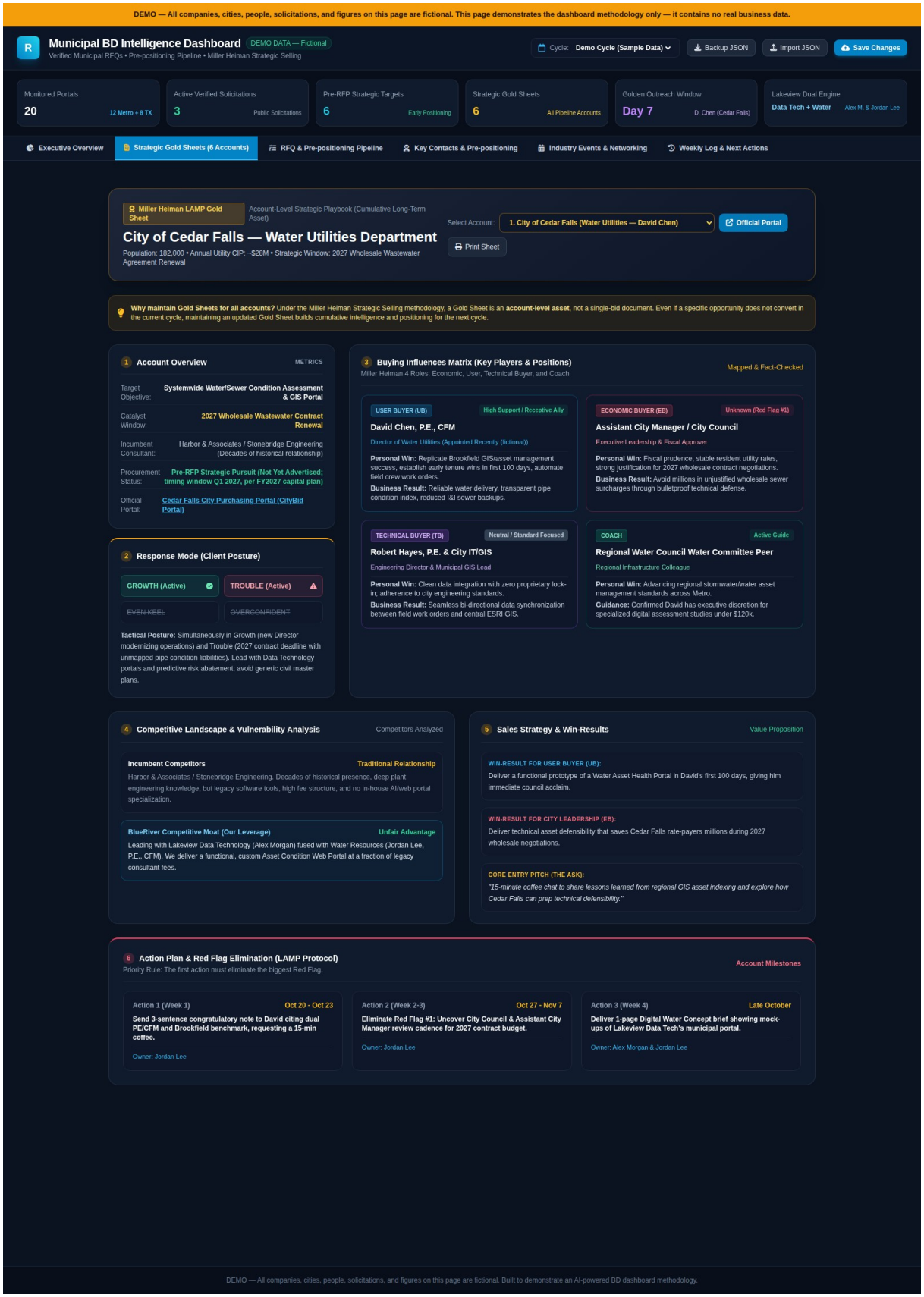


Figure 2 — A complete Gold Sheet for one priority account (fictional demo data).

Figure 3 — Tier 1 and Tier 2, side by side. Compare the columns. Tier 1 (verified live solicitations) carries **Verified Deadline** and **Direct Posting Link**. Tier 2 (pre-positioning pipeline) carries **Pre-positioning Catalyst & Timing**, **Strategic Action** and a **Source** (link, date, quoted line) — and *no deadline column at all*. The schema contract of Stage 2, visible in the UI.

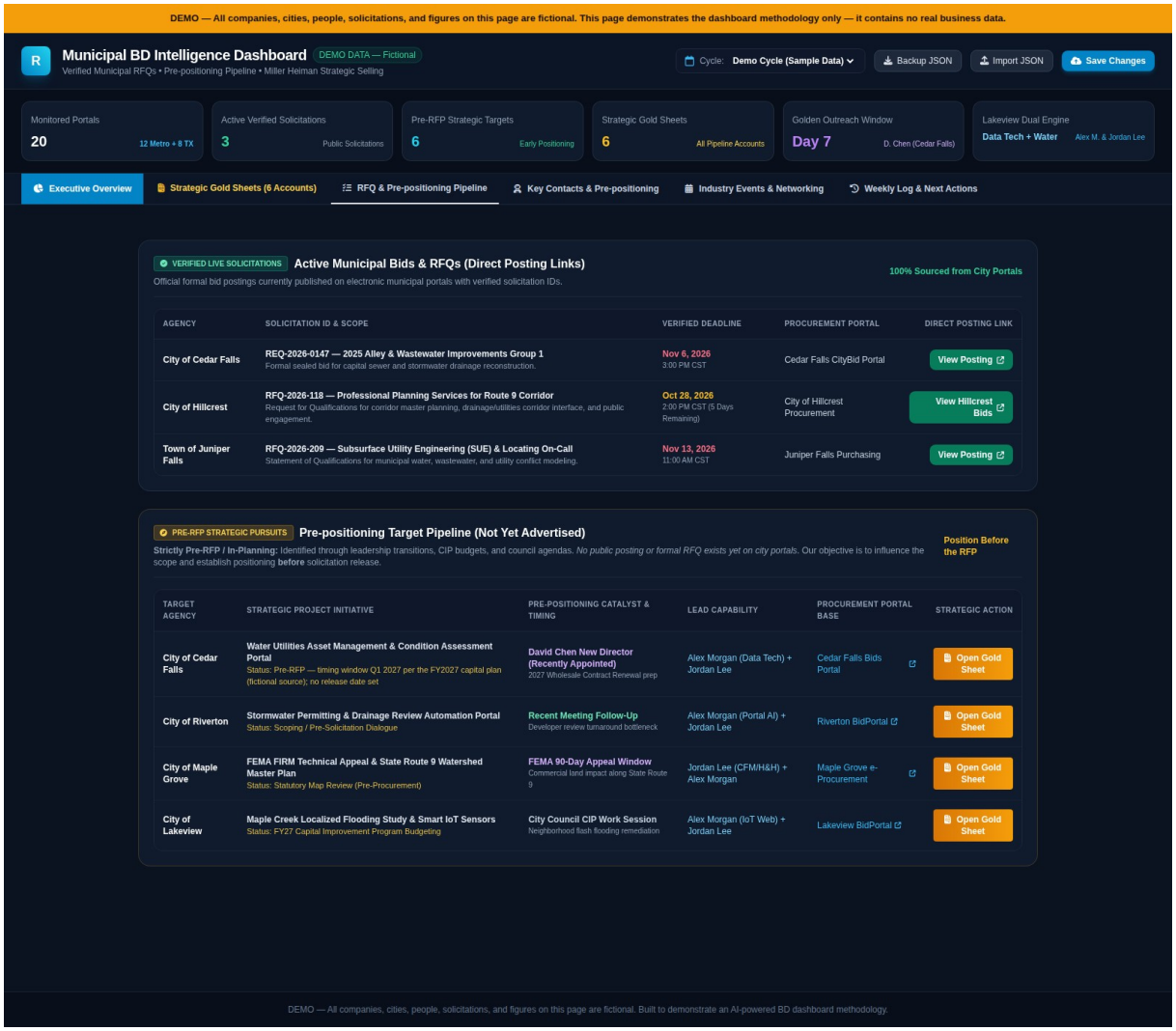


Figure 3 — Tier 1 verified solicitations above; Tier 2 pre-positioning pipeline below (fictional demo data).

Figure 4 — A decision-maker dossier. The six dimensions of the playbook's Step 2.2, ending in a ready ice-breaker pitch and a copyable outreach draft. Note the division of labor: the system *drafts*; only the human sends (guardrail R3).

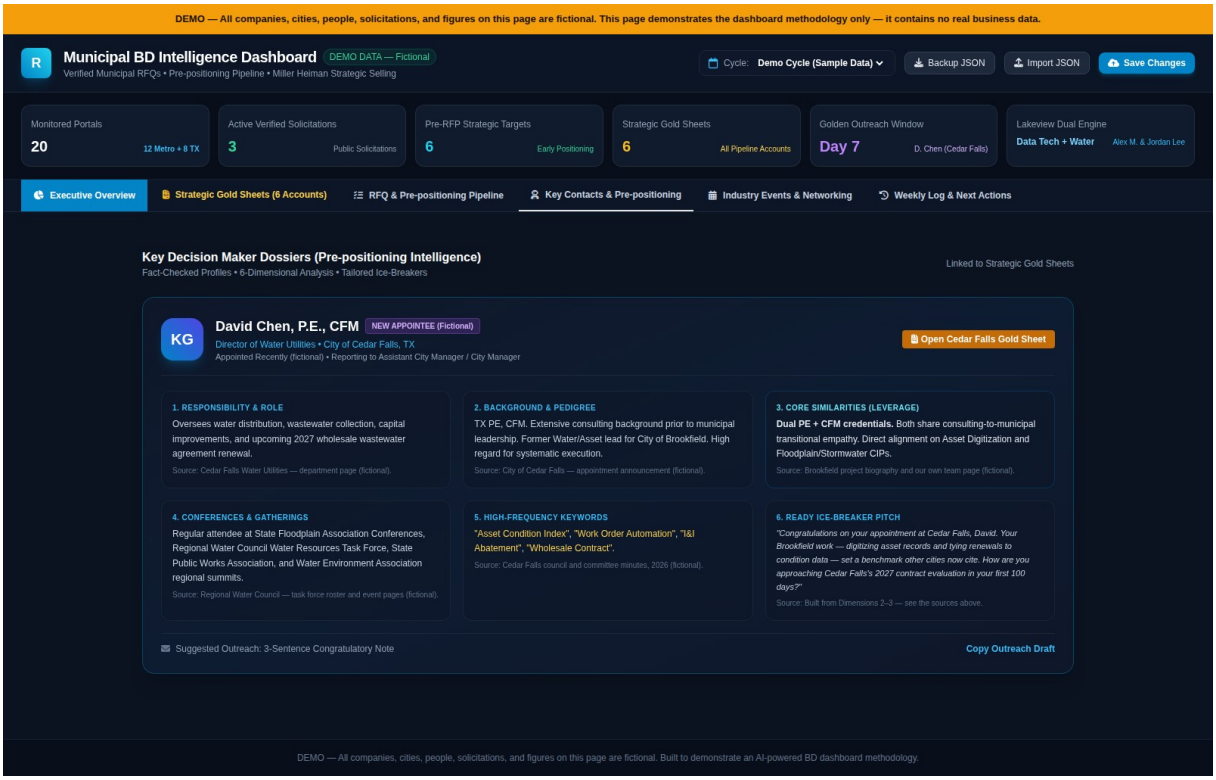


Figure 4 — A six-dimension decision-maker dossier (fictional demo data).

Figure 5 — The weekly execution log. The human's workspace inside the artifact: last week's accomplishments, next week's action items (with checkboxes and owners), and strategic observations — including, in this demo, the operator's own note of the operating discipline: "Zero fabrication. Pre-positioning before the posting." Saved to browser storage, this log becomes next cycle's Stage 1 baseline.

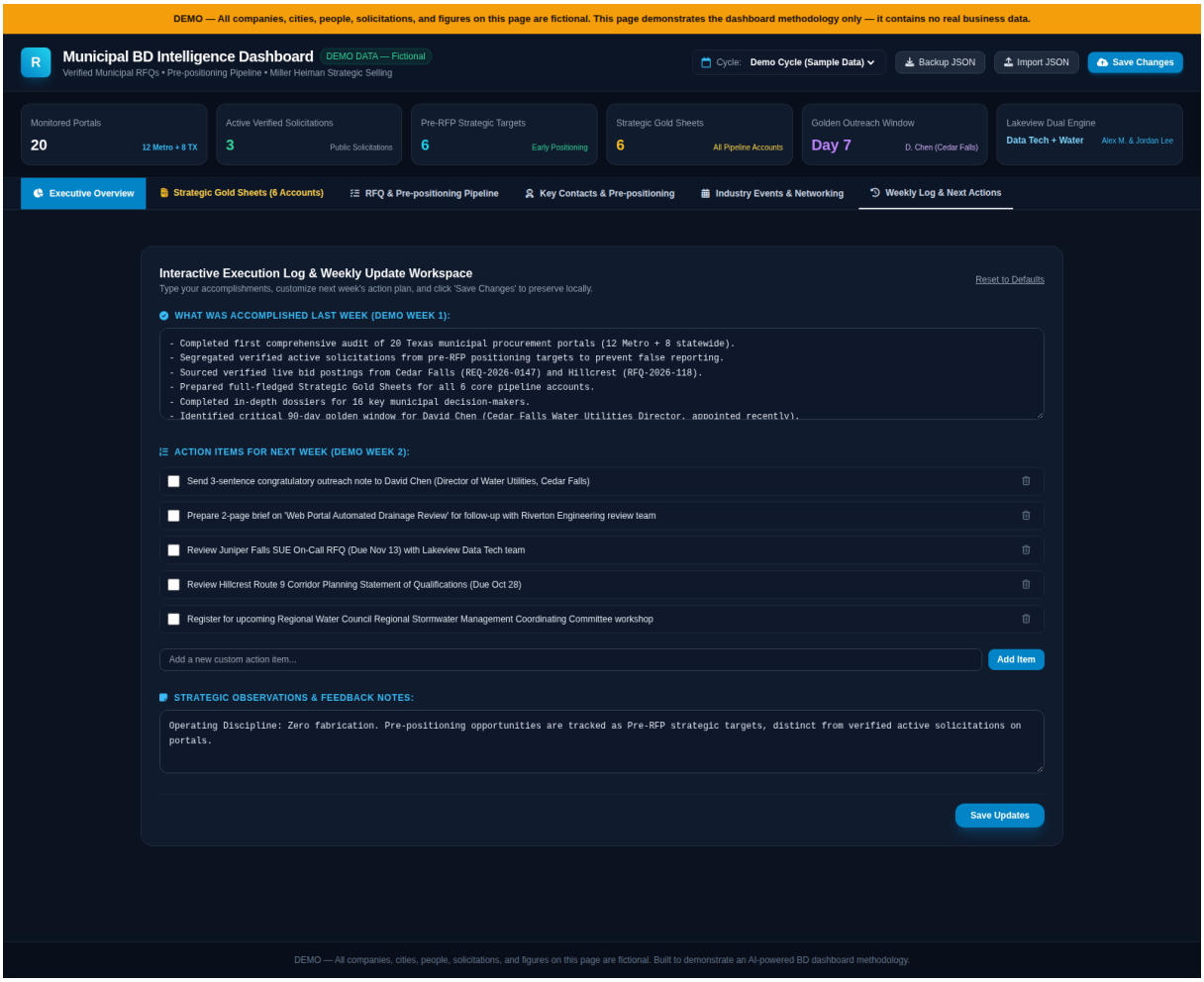


Figure 5 — The weekly execution log and update workspace (fictional demo data).

5. For Python Students — Concept Map and Starter Exercises

Nothing in this architecture is alien to you. Each piece is a concept from your Python course, wearing work clothes:

In this system	Python analogue	What you already know
State JSON	`dict` / list of dicts, saved with `json.dump`, loaded with `json.load`	Data structures, file I/O
Tier 1 / Tier 2 schemas	`@dataclass` definitions — Tier 2 simply has no `deadline` field	Classes, attributes
A prompt module	A function: arguments in (context), a spec (the prompt), a typed return value (the output schema)	Functions, docstrings
Verification rules (R1)	`assert` / validation `if`s that reject bad records into an error list	Conditionals, exceptions
The weekly schedule	cron / Task Scheduler running a script; `datetime` for the cycle	OS basics, loops
Dashboard rendering	Building an HTML string from data (f-strings or a template) and writing	String formatting

	it to a file	
`localStorage` state	A JSON file on disk that your program reads at start and writes at exit — same idea, browser-managed	Persistence

Exercise A — Rebuild the skeleton in Python (no AI account needed, ~1 hour). (1) Define `Tier1` and `Tier2` dataclasses; make it *impossible* for `Tier2` to hold a deadline. (2) Load the instructor's sample JSON of raw portal findings. (3) Write the validator: any Tier 1 record missing an ID, a deadline, or a link is rejected into a `red_flags` list with a reason. (4) Render a simple HTML page with the two tables from the validated records and open it in a browser. You have just rebuilt Stages 2 and 4 — everything except the agent.

Exercise B — Replace the validator with a prompt module. Take the Stage 2 module from §3 and run it (through any LLM you have access to) on the *same* raw findings. Compare its classification and Red Flag output against your Python validator on at least ten items, and log every disagreement. That log is your first verification-gate audit — and a concrete feel for why Stage 6 (the human) exists.

6. Honest Technical Limitations of This Stack

- **CDN dependencies.** Tailwind CSS and Font Awesome load from CDNs when the dashboard is opened; a fully offline copy must inline them. (Open the demo with the network off and watch the styling degrade — a good first debugging exercise.)
- **`localStorage` limits.** Per-browser, small (roughly 5 MB), and not synchronized. The design compensates with the JSON export ritual and the dated Drive archive — process compensating for platform, a recurring real-world pattern.
- **Probabilistic extraction.** The agent's browsing and extraction can err, and portals change without notice. The human verification gate (Stage 6) is therefore a *load-bearing* part of the architecture, not a courtesy review.
- **Single-operator by design.** No authentication, no multi-user state, no backend. The natural v2 — state moved to a small Python + SQLite service behind the same dashboard — is an excellent student extension project.

Questions on any stage are welcome — the builder is happy to join a class Q&A.