FieldBrief.

PROJECT CASE STUDY · SEPTEMBER 2026

How FieldBrief *was built.*

A look at the data, the AI, and the decisions behind a demo for investigating oil wells.

The project explained in 13 short chapters · 8 diagrams

By Jason · Updated September 8, 2026

FROM SOURCE TO BRIEF

01 **Start with well records.**

Real names and locations, with sample daily readings.

02 **Check the numbers.**

Compare recent readings with the week before.

03 **Put the findings into words.**

Ask the AI for help, check its sources, and save a brief.

RECORDS → READINGS → INVESTIGATION → BRIEF

58,673
Texas wells from public records

5,280,570
sample readings in the original dataset

90 days
per well, ending September 6, 2026

This is a practice workspace. The production data is made up, and team approvals are turned off in the hosted demo.

01 / THE IDEA

Help someone decide what to check next.

FieldBrief is a demo for people who review oil-well production. It helps them spot a change, look through the records, and write a short summary of what needs attention.

Imagine a well usually produces about 100 barrels of oil a day, but its latest reading is 80. That deserves a closer look. It could be a real drop, a missing report, or something that needs a field check. The number alone cannot tell you the cause.

FieldBrief puts the map, production history, source records, and an AI assistant in one place. You can check the records yourself, ask the assistant to explain them, and save a brief for later. Each chat stays attached to the data it started with.

01 / NOTICE

Find a change

See which wells have lower production or missing readings.

02 / CHECK

Look at the records

Compare recent readings and review the sources behind an answer.

03 / SUM UP

Prepare a brief

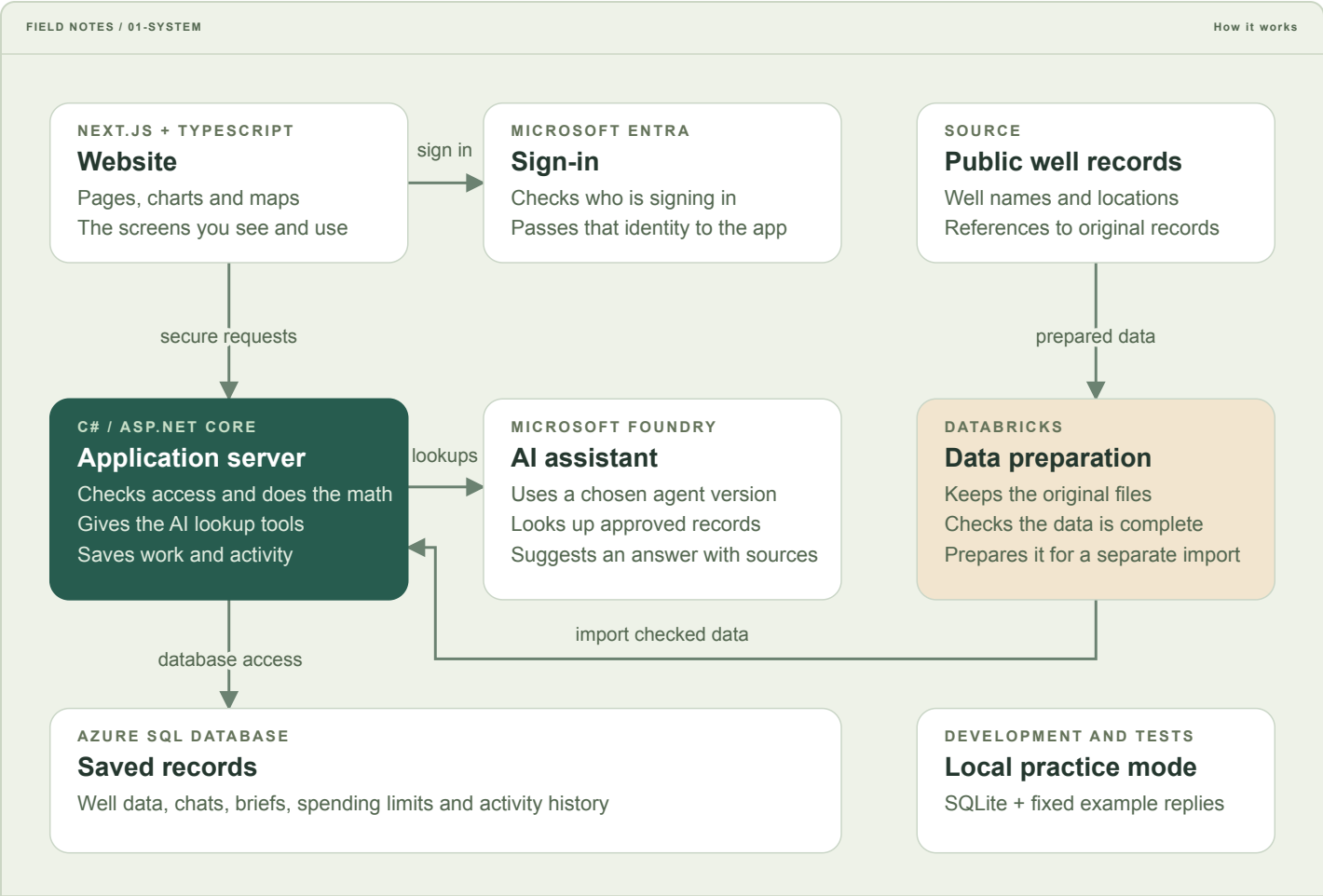
Save what you found and explain what someone should check next.

What this project demonstrates

The demo shows how the software works from sign-in to a saved brief. It has not been tested in day-to-day field operations or shown to improve production.

Each part has a clear job.

The website shows the information. The server checks access, does the calculations, and saves the work. Other services prepare the data and provide the AI.



The main parts of FieldBrief. The website asks the server for information. The server checks access, reads saved records, and manages the AI. Databricks prepares the data before anyone opens the workspace.

The website uses Next.js, React, and TypeScript to build the screens, maps, charts, and chat. Bun runs the website on Azure. These are the tools behind what you see and click.

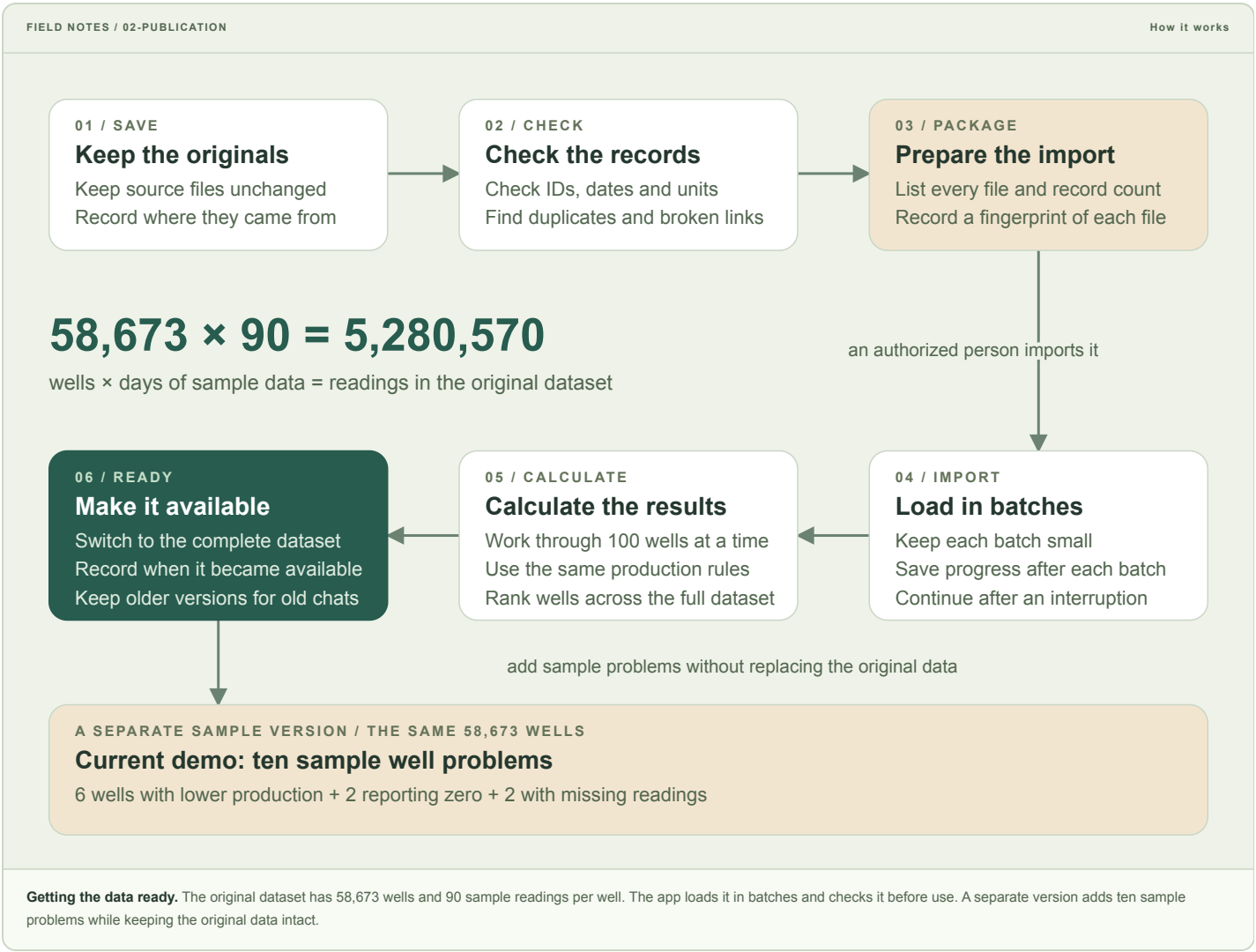
The server is written in C# using ASP.NET Core. The website talks to it through an API—a way for two pieces of software to exchange requests and results. The server decides which records you can see and applies the production rules. Azure SQL is the database where it saves well records, chats, and briefs.

Microsoft Foundry supplies the AI. The server gives it approved ways to look up records, then checks the references in its answer. Databricks handles a separate job: preparing and checking large batches of data before they reach the app. Microsoft Entra handles sign-in.

> **There is also a local practice mode**
>
> On a developer's computer, SQLite stores the data and the assistant gives fixed example replies. This lets the workflows be tested without paying for AI calls. The hosted demo uses the real Microsoft services.

Real well records. Made-up production readings.

The names and locations belong to real wells in public Texas records. Their daily oil readings were generated for this demo.



Getting the data ready. The original dataset has 58,673 wells and 90 sample readings per well. The app loads it in batches and checks it before use. A separate version adds ten sample problems while keeping the original data intact.

The directory contains 58,673 wells across six counties: Culberson, Loving, Pecos, Reeves, Ward, and Winkler. Records without a usable name or a clear location were left out. The app keeps the public identifiers and source references so the well records can be checked.

Each well starts with 90 days of sample readings, ending September 6, 2026. That makes 5,280,570 readings in the original dataset. The dates stay fixed; this is not a live production feed.

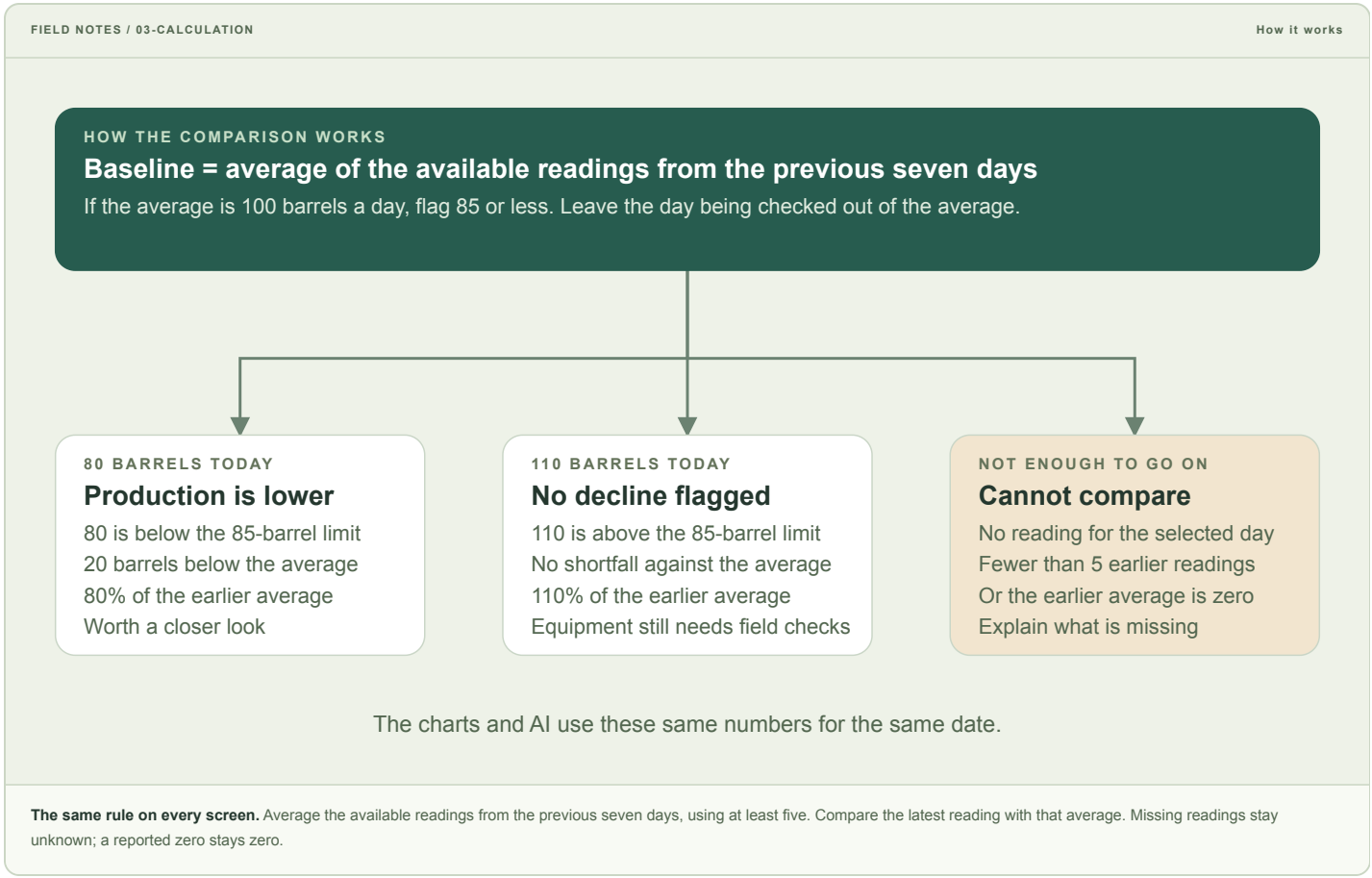
Databricks keeps the original files, checks them, and prepares them for import. The app loads the data in small batches and records its progress. It checks that nothing is missing or changed before making the new dataset available. If the upload stops halfway through, the incomplete data stays out of the workspace.

Ten wells give you something to investigate.

The hosted demo changes the last few readings for ten wells. Six have reduced production, two report zero oil, and two have missing readings. That gives eight measured declines and two reporting gaps, with 58,671 wells reporting a value on the selected day. These are sample problems, not reports of real outages.

Compare the latest reading with the week before.

FieldBrief uses the same calculation everywhere, so the chart, status card, and AI start with the same numbers.



First, it finds the average of the available daily oil readings from the previous seven days. This average is the baseline. At least five readings are needed, and the day being checked is left out of its own comparison.

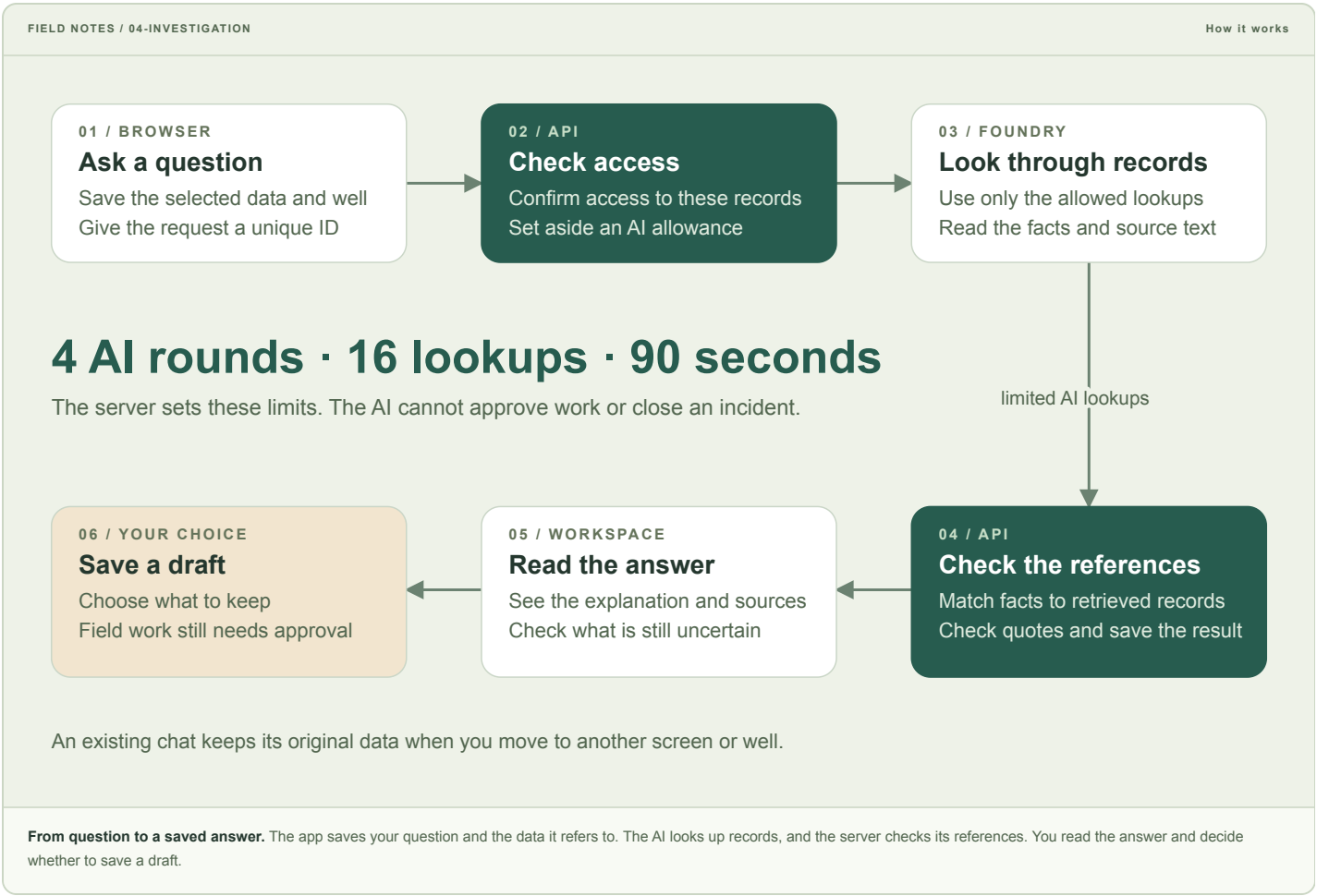
A reading at or below 85% of the baseline is flagged as a decline. For example, a well averaging 100 barrels a day would be flagged at 85 barrels or less. A reading of 80 has a shortfall of 20 barrels compared with that average. That difference is a reason to investigate; it does not confirm lost production or an equipment fault.

A missing reading stays unknown. A reported zero stays zero. If there are too few earlier readings, or their average is zero, the app explains why it cannot make the comparison. It also shows how many readings arrived and how long it has been since the last one.

The overview totals cover the full dataset, even though the screen shows only a selection of wells. This avoids a misleading total based on a single page of search results.

The assistant helps explain what you are looking at.

Ask about a well or the wider dataset. The assistant can look up records, explain the readings, and suggest a brief.



From question to a saved answer. The app saves your question and the data it refers to. The AI looks up records, and the server checks its references. You read the answer and decide whether to save a draft.

When you send a question, the app saves the selected data, date, screen, and any well or incident you are discussing. The assistant keeps that context if you move to another screen. A new selection does not quietly change an existing chat.

The assistant gets a small set of lookup tools, such as searching for wells or reading production history. It cannot browse the database freely. The server checks that the facts and quotes in its answer came from the records it was given.

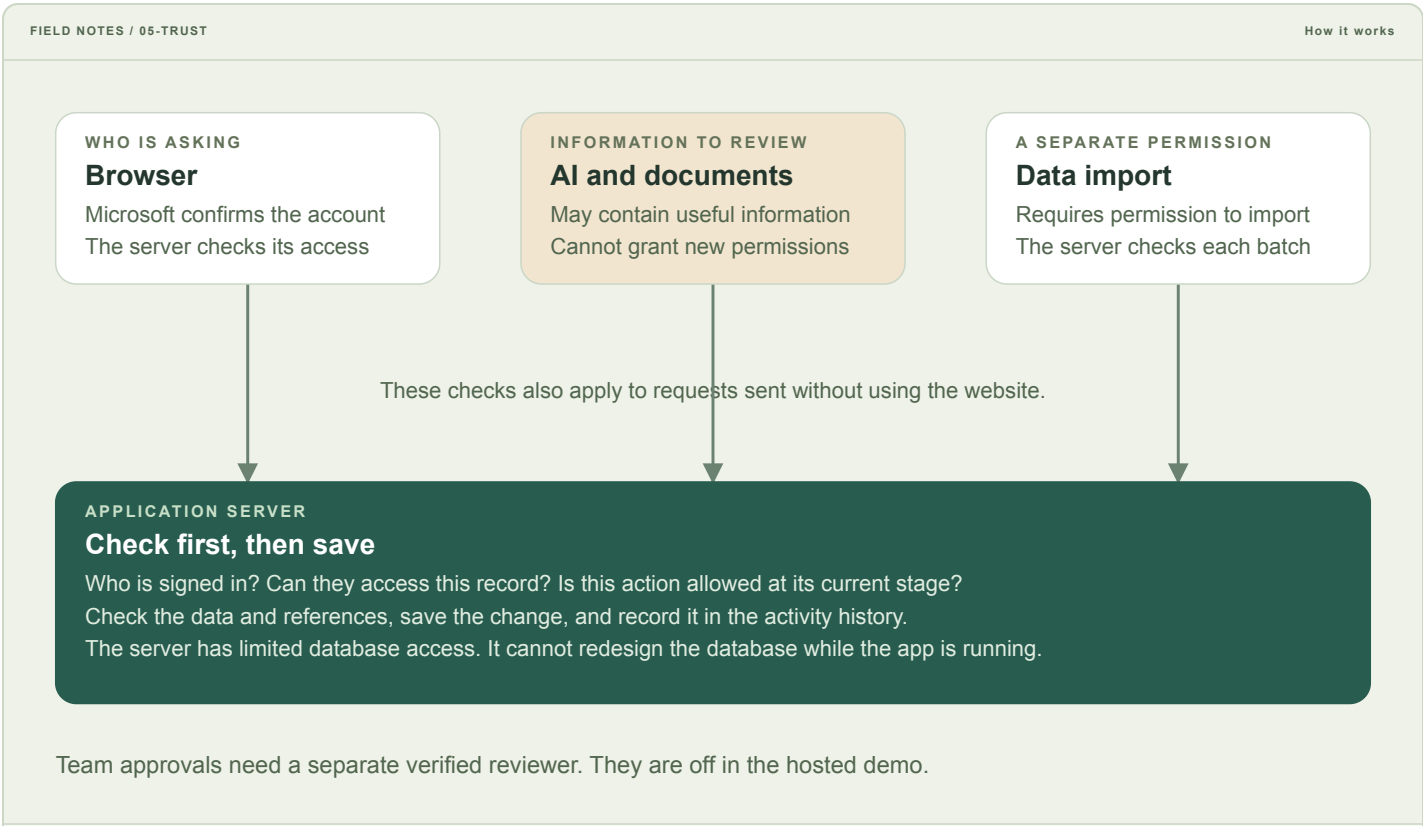
Each question has limits: up to 25 search matches, details for 12 wells, and six document excerpts per lookup. The AI can make up to 16 tool calls over four response rounds, within 90 seconds. The 12-well limit controls how much detail one question can request; the directory still contains 58,673 wells.

These checks help keep answers tied to the records, but they cannot prove that an explanation is correct. A person still needs to judge the conclusion. Saving a draft does not approve field work, and the assistant cannot approve work or close an incident.

> **The demo has a shared AI budget**
>
> Before an AI call, the app sets aside part of its spending allowance. If a call is interrupted and its cost is unclear, that amount stays reserved until the outcome is checked.

Signing in is the first check.

Microsoft handles sign-in. FieldBrief then checks which records and actions that account can access.



The server checks every request. The server checks the account, the requested action, and the records involved. An AI answer or a document cannot grant access. Team approval actions are off in the hosted demo.

The home page, sign-in page, and this article are public. Opening the workspace requires an account. Access checks happen on the server, so hiding or showing a button is never the only protection.

The demo login is shared. Everyone using it can see that account's saved chats and work. The welcome message explains this along with the sample data and AI limits. Once dismissed, it stays hidden in that browser unless its storage is cleared.

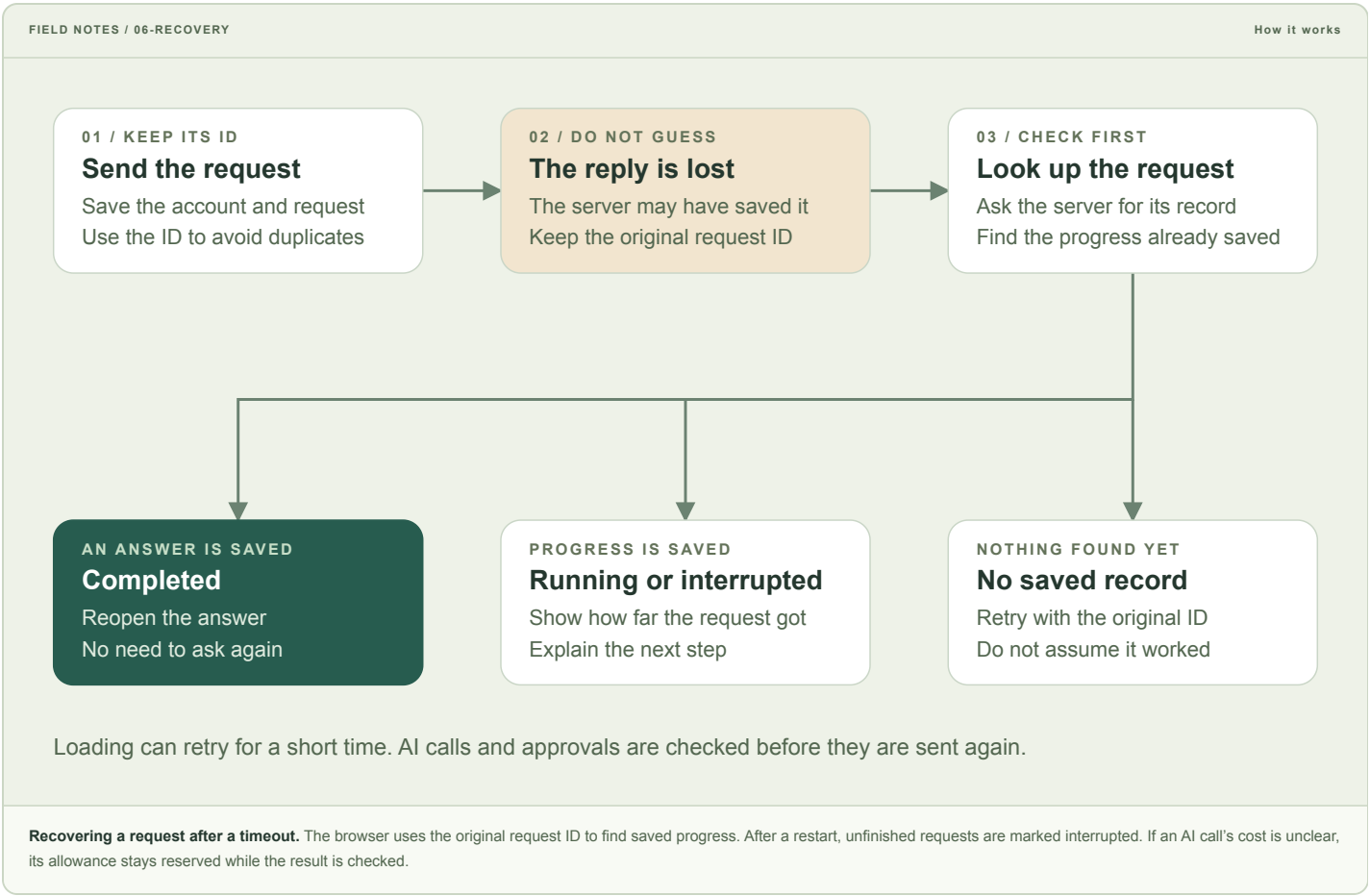
Database credentials stay off the website. Azure gives the server its own identity and limits what it can do in the database. Text from a document or an AI answer cannot grant extra permissions.

Changes are recorded in an activity history. The app can detect some changes to that history, but it does not claim the record is impossible for a database administrator to alter.

> **Team approvals are off in the hosted demo**
>
> You can investigate wells and prepare briefs. Team approvals, member changes, and task assignments are disabled. Those workflows have local tests, but still need to be checked with separate accounts on the live site.

A lost reply should not mean starting over.

Sometimes the server saves a request, but the reply never reaches your browser. FieldBrief checks for that saved request before trying it again.



Recovering a request after a timeout. The browser uses the original request ID to find saved progress. After a restart, unfinished requests are marked interrupted. If an AI call's cost is unclear, its allowance stays reserved while the result is checked.

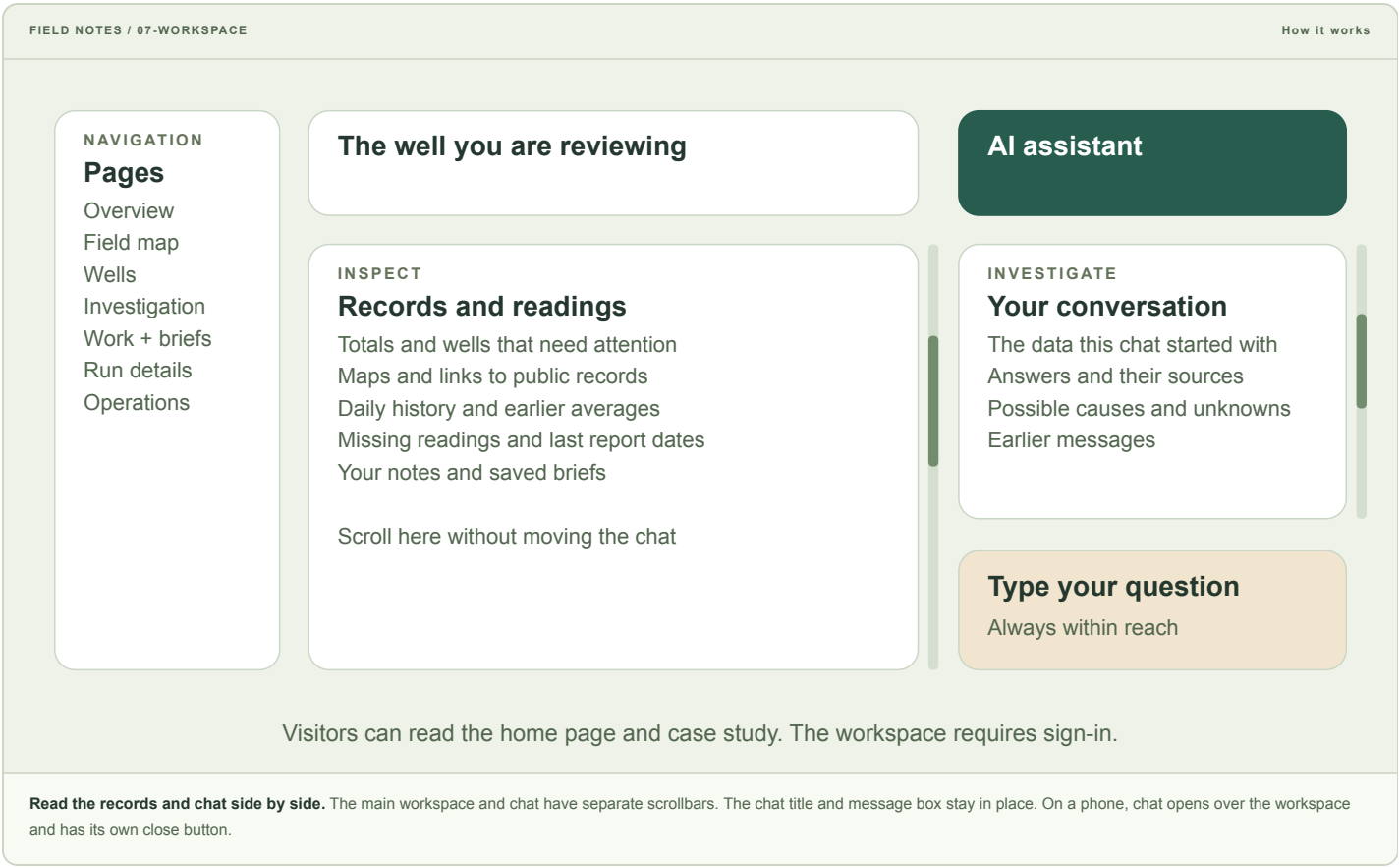
Each question or change gets a unique request ID. Think of it as a receipt number. If the connection drops, the browser uses that number to find the original result. It can reopen a finished answer or show that the request is still running or was interrupted.

If no saved record exists, a retry keeps the same ID. This helps prevent a second AI run or a duplicate change. If the server restarts, unfinished runs are marked interrupted so they do not appear to be working forever.

Signing in can also succeed while the workspace is still waking up. The first load waits up to 90 seconds, then offers a retry while keeping the sign-in. The visitor gets a next step instead of an endless loading screen.

Keep the records beside the conversation.

You should be able to read a long answer without losing your place in a well’s history.



Read the records and chat side by side. The main workspace and chat have separate scrollbars. The chat title and message box stay in place. On a phone, chat opens over the workspace and has its own close button.

On a desktop, the records fill the main part of the screen and the assistant sits on the right. Each has its own scrollbar. The chat title and message box stay in place as you read earlier messages.

On a phone, chat opens over the workspace and has a close button that takes you back to the records. Maps and tables load only what the current view needs, keeping thousands of well records from landing in the browser at once.

The home page and sign-in page are separate from the workspace. Opening a workspace link while signed out sends you home. If you close the Microsoft sign-in window, the sign-in button returns so you can try again.

> **What the maps can tell you**
>
> Map points show reported well locations. Lease names are references from public records; they do not establish legal boundaries or prove a well is operating today. The local New Mexico examples show public lease outlines only where the lease reference matches.

Start small, then connect the pieces.

The project began with a small set of wells that was easy to check by hand. The larger dataset and live services came later.

01

Get the calculations right.

The first local demo used 12 New Mexico wells with sample operations. That made it easier to check declines, missing readings, and recovery before adding more data.

02

Make the work saveable.

Chats, questions, and investigation progress were saved so someone could leave and return. Fixed example replies made it possible to test failures without paying for AI calls.

03

Add the Texas well directory.

The next step brought in 58,673 public well records and generated 90 days of readings for each. The import checks made sure incomplete data could not become the active dataset.

04

Connect the live services.

Microsoft sign-in, Azure SQL, Databricks, and Foundry were connected and checked separately. A local example reply was never counted as proof that live AI worked.

05

Try it in the browser and fix the rough edges.

Testing found slow startup, map selection, and scrolling problems. Later changes added ten sample well problems so visitors could investigate visible declines and missing readings.

> **A fresh local setup starts smaller**
>
> The local overview starts with the original 12 New Mexico wells. Its directory and field map also include the Texas wells. The larger Texas overview and ten sample problems need a separate data import.

Why it was built this way.

We chose a separate web app and a modular monolith for the backend: one API with clear sections for calculations, imports, AI, and saved work. That keeps each job understandable and gives us one backend to deploy.

Give the website and API different jobs.

Next.js handles the pages, maps, and chat interface. The C# API owns calculations, access checks, and saved work. This keeps one set of rules behind every screen and AI tool. The cost is maintaining two apps and checking that the messages they exchange still match.

Prepare data in Databricks, then serve it from SQL.

Databricks gives us a place to keep the original input, check it, and publish a complete dataset. Azure SQL then serves that dataset to the app. We chose this split so browsing does not depend on a running Databricks warehouse. Fresh data needs a separate import.

Put Foundry behind the API.

Foundry hosts the model and a versioned set of agent instructions. The API supplies approved lookups, checks access, and records AI usage. This lets the assistant help with open-ended questions while the application controls which records it can read and how much it can spend.

Use cloud storage and a local stand-in.

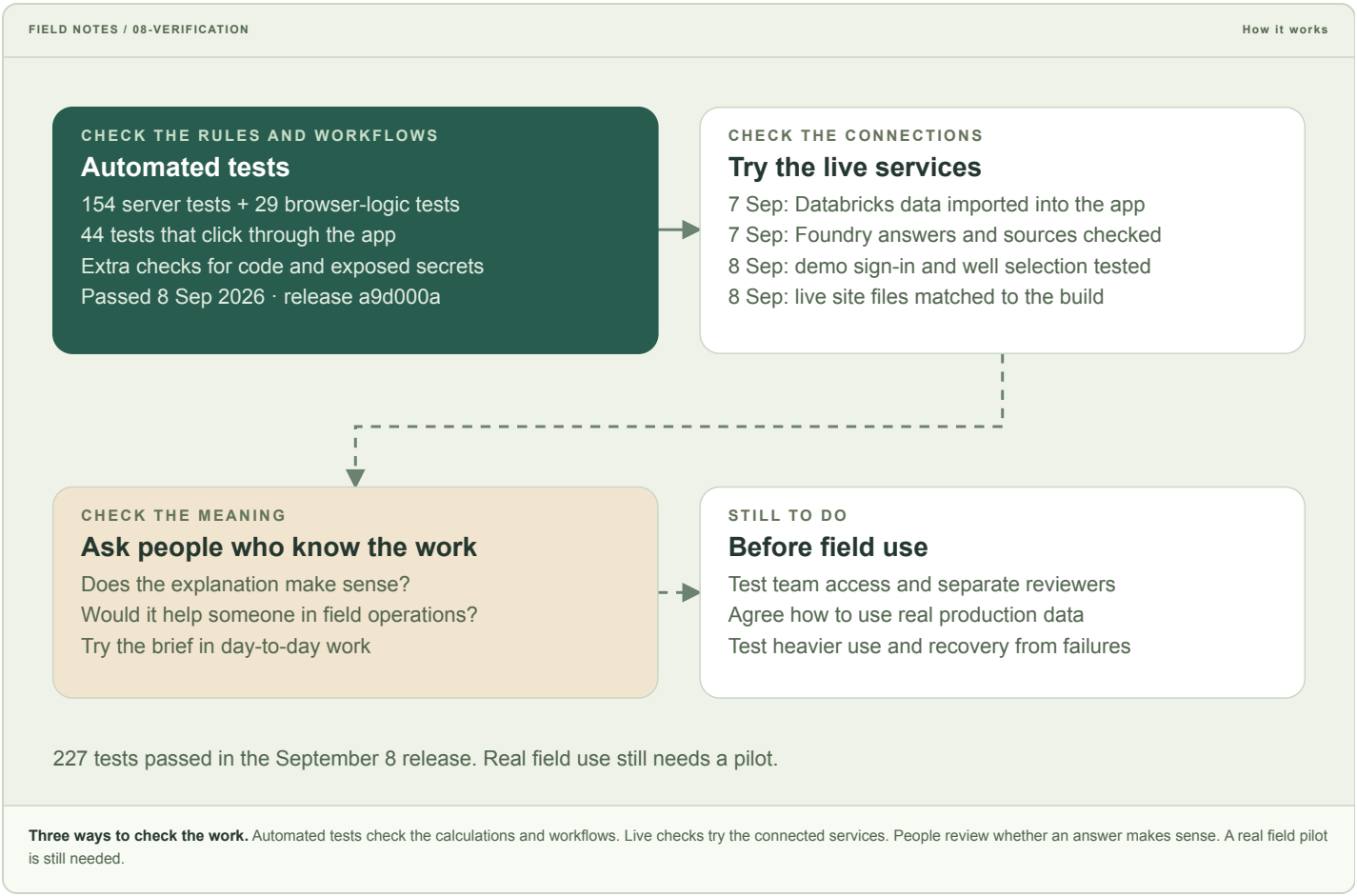
Azure SQL keeps the hosted workspace’s saved records across app restarts. SQLite supports local development, alongside fixed example AI replies. Both sit behind storage interfaces, so the calculation rules can be tested without cloud services. Live checks still have to catch differences between the two environments.

Keep deployment small enough to manage.

App Service hosts the web app and API without us managing virtual machines. One API process owns the database work and AI spending record. We chose that simpler arrangement for the demo’s size and budget. Free hosting limits availability, and adding API workers would require coordinated job ownership.

Check the code, then try the live app.

Automated tests check the rules and workflows. Live checks confirm that the hosted services work together. People still need to judge whether an AI answer is useful.



Three ways to check the work. Automated tests check the calculations and workflows. Live checks try the connected services. People review whether an answer makes sense. A real field pilot is still needed.

The September 8 release, `a9d000a`, passed 227 tests: 154 server tests, 29 browser-logic tests, and 44 tests that click through the app. They cover calculations, access rules, imports, recovery, maps, and user workflows. None failed or were skipped.

Local tests use fixed AI replies, so they are repeatable and do not spend the live allowance. Separate September 7 checks confirmed the Databricks import and completed Foundry investigations. Those AI checks used earlier data; they do not establish answer quality for the newer ten-problem examples.

September 8 checks used the demo account to try sign-in, well selection, source records, and investigation screens on desktop and mobile. Release checks also compare the files served by the live site with the files that were built.

> **Passing tests is one part of the picture**
>
> The free hosting plan can still be slow to wake up. Team workflows need live testing with separate users, and AI explanations need review by people who know field operations.

What the architecture taught us.

Building and testing the app showed us where these boundaries helped, and where connecting the parts needed more care.

Shared rules prevent conflicting answers.

The overview, well history, and AI all need to agree about a decline or a missing reading. Keeping that logic in C# gave us one place to check it. We learned to share calculated results across the app instead of rebuilding the same rules in each screen.

Saving an answer means saving its data version.

A conversation can outlive the dataset that was current when it began. Pinning it to a dataset version keeps its evidence stable when someone navigates elsewhere or new data arrives. This taught us to make data versions part of saved work from the start.

Recovery starts with what the database remembers.

A timeout does not tell us whether a request finished. Saved request IDs, import progress, and AI usage reservations let the app check before repeating work. We learned that a retry button depends on a storage design that can distinguish new work from an interrupted reply.

AI needs boundaries the application can enforce.

Instructions alone cannot decide what a user may access or whether another model call fits the budget. Those checks belong in the API and database. The architecture can enforce those limits, but judging whether an explanation makes sense still needs human review.

Local tests and live checks answer different questions.

SQLite and fixed AI replies let us test the core workflows repeatedly. Azure added separate concerns: database access, sign-in, service startup, and provider responses. We learned to test the rules in isolation, then check each connection and the complete hosted workflow.

A few places to look closer.

For readers who want the technical detail, these files and public guides explain the main parts of the build.

Where to look in the code

`apps/web`	The pages, maps, charts, sign-in screen, and chat.
`ProductionRules.cs`	The production comparisons and rules for missing readings.
`Dataset* / V2*`	Data imports, saved investigations, AI requests, and work records.
`DatabaseRequestGate`	Checks that the server can safely use the database and reconnects when needed.
`scripts / infra / tests`	Tools for loading data, deploying the app, and running tests.

Public guides

Texas Railroad Commission: downloadable data

The source of the Texas well records. FieldBrief generates its own sample production readings.

Databricks: preparing data in stages

The approach behind the original, checked, and ready-to-use data layers.

Microsoft Entra: how browser sign-in works

The sign-in process used by the website.

Microsoft Foundry: how AI agents work

The service behind the AI. FieldBrief's server manages its lookups and saves the conversation.

Next.js: deploying a website

How a Next.js website can be packaged and hosted.

Updated September 8, 2026. The demo's readings end on September 6 and do not change with today's date. Test results and live checks refer to the dates given above.