PLEASE MAKE A NEW COPY OF THIS TEMPLATE FOR YOUR BLOG

Note: This template includes the elements that you'll need to provide before your blog can be published. **Please fill out all fields in red prior to submitting your draft.**

Please give edit access to Mikaila Garfinkel

Building a Continuously Optimizing NFL Defensive Coordinator Chatbot Guided by Domain Experts with Databricks

<SME Approvers>

Name of Reviewer	Title	Date Approved
Neil Scott	Director, GTM for Sports	
Alkis Polyzotis	Sr. Staff Engineer MLflow	

* The SMEs should NOT be the authors of the blog.
* For engineering blogs, the SMEs must be L7 or above. For all other blogs, the SMEs would be the domain owners (Director and above)

<Blog Title>
Self-Optimizing NFL Chatbot Guided by Domain Experts on Databricks

<Subtitle>
A practical guide to authoring, deploying, evaluating, and governing an agentic assistant that helps defensive coordinators anticipate opponent tendencies and continuously optimizes based on subject matter feedback.

<Author(s)>
[Wesley, Pasfield] - Headshot

[Nick, Ragonese] – [Headshot]
With over a decade of experience in data, Nick Ragonese has specialized in driving AI innovation for sports leagues and teams since joining Databricks in 2021.

<Short summary (Meta description)>
[Learn how to build and continuously optimize a defensive coordinator chatbot guided by SME feedback with the Databricks AI Agent Framework, Unity Catalog tools, and MLflow. Surface opponent tendencies by down-and-distance, formation, personnel, and situation—grounded in governed data and production-quality agent operations.]

<Top Banner image/OG image>
[Insert social card image here - dimensions should be 1200 x 628px]

Note: The top banner image will be featured as the tile thumbnail in the blog overview page as well as on the top of the actual individual blog page — top banner image can essentially be the same as the OG/social image

<Topic Tags>
Sports Analytics, AI in Sports, NFL, Lakehouse, Databricks, Mosaic AI Agent Framework, Unity Catalog Tools, MLflow

---

<Blog Post>

- **<3 bullet summary>**
- **Purpose:** Build a coach-facing agentic assistant that answers "what will this offense do?" using governed, production-grade tools over play-by-play, participation, and roster data.
- **Approach:** Author a tool-calling agent with **Unity Catalog functions** (SQL analytics over Delta) and deploy via **Agent Framework** with MLflow Tracing. Implement a self-optimizing loop where SME feedback captured in MLflow Labeling Sessions trains aligned judges (align()) that drive automatic prompt improvement (optimize_prompts()), encoding expert football knowledge directly into the system.
- **Outcome:** Coordinators get situation-aware tendencies (down & distance, formation/personnel, two-minute drill, screen rates) with fast iteration and quality checks ready for game week installs. Developers get a reusable architecture for any domain: capture expert feedback, align judges to what

"good" means for your use case, and let the system continuously improve with prompt optimization guided by the aligned judges.

- **<Article Body>**

Generic LLM judges and static prompts fail to capture domain-specific nuance. Determining what makes an NFL defensive analysis 'good' requires deep football knowledge: coverage schemes, formation tendencies, situational context. General-purpose evaluators miss this. The same is true for legal review, medical triage, financial due diligence, or any domain where expert judgment matters.

This post walks through an architecture for **self-optimizing agents** built on Databricks Agent Framework, where enterprise-specific human expertise continuously improves AI quality using MLflow, and developers control the entire experience. We use an American Football Defensive Coordinator (DC) Assistant as the running example in this post: a tool-calling agent that answers questions like "who gets the ball for the Cowboys in 11 personnel on 3rd-and-6?" or "What do the Packers do in the last 2 minutes of halves?" The example below shows this agent interacting with a user via Databricks Apps.

What do the 2024 Titans do on 3rd and long?

I'll analyze the Tennessee Titans' tendencies on 3rd and long situations in 2024. Let me get their down and distance tendencies first, then dive deeper into the specific formations and personnel they use. Now let me get more specific information about what formations and personnel they use on 3rd and long (>10 yards): Based on the data, here's what the 2024 Tennessee Titans do on 3rd and long (>10 yards):

Key Tendencies on 3rd & Long

Overall Statistics (50 plays):

- **Pass Rate: 72%** (36 passes, 14 runs)
- **Success Rate: 26%** (very low)
- **First Down Rate: 14%**
- **Average Yards: 6.3**
- **Average Air Yards: 13.6** (when passing)

Formation & Personnel Breakdown

Primary Formation: SHOTGUN (44 of 50 plays - 88%)

- **11 Personnel** (5 OL, 1 RB, 1 TE, 3 WR): Most common - 22 plays
- **12 Personnel** (5 OL, 1 RB, 2 TE, 2 WR): Secondary option - 10 plays

Secondary Formation: PISTOL (6 plays)

- Mix of 11 and 12 personnel

Target Distribution

Primary Receivers on 3rd & Long:

1. **Calvin Ridley** - 25 targets (most frequent target)
2. **Tony Pollard** - 20 targets (mix of passes and runs)
3. **Tyjae Spears** - 12 targets
4. **Chig Okonkwo (TE)** - 11 targets
5. **Tyler Boyd** - 8 targets
6. **Nick Westbrook-Ikhine** - 7 targets

From Agent to Self-Optimizing System

The solution has two phases: build the agent, then optimize it continuously with expert feedback.

Build

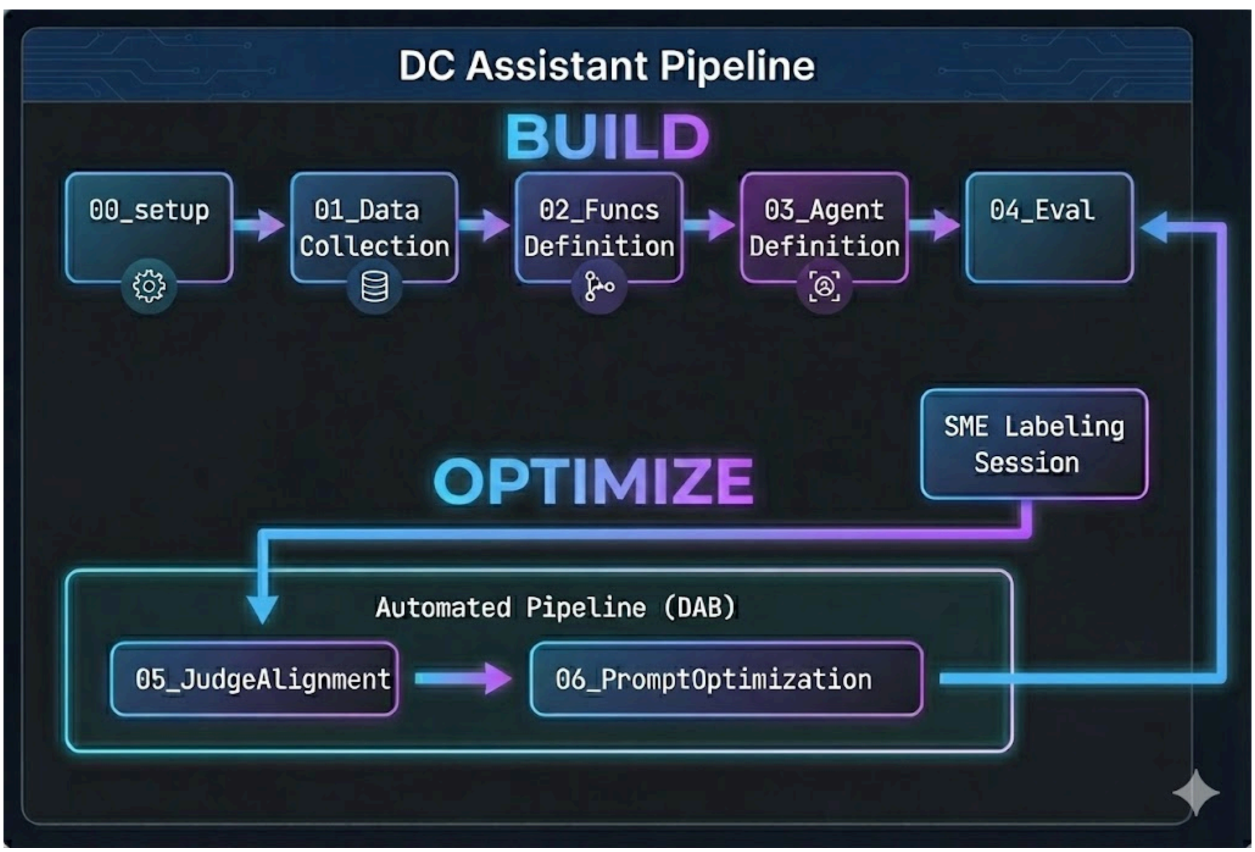
- **Ingest data**: Load domain data (play-by-play, participation, rosters) into governed Delta tables in Unity Catalog
    - We ingested two years (2023–2024) of NFL participation and play-by-play data from nflreadpy as the data for this agent
- **Create tools**: Define SQL functions as Unity Catalog tools the agent can call leveraging the data extracted
- **Define and deploy the agent**: Wire tools to a ResponsesAgent, register a baseline system prompt in the Prompt Registry, and deploy to Model Serving

- **Initial Evaluation**: Run automated evaluation with LLM judges and log traces using baseline versions of custom judges

Optimize

- **Capture expert feedback**: SMEs review agent outputs and provide structured feedback through MLflow Labeling Sessions
- **Align judges**: Use the MLflow align() function to calibrate the baseline LLM judge to match SME preferences, teaching it what "good" looks like for this domain
- **Optimize prompts**: MLflow optimize_prompts() uses a GEPA optimizer guided by the aligned judge to iteratively improve the original system prompt
- **Repeat!**: Each MLflow labeling session is used to improve the judge, which in turn is used to optimize the system prompt. This entire process can be automated to automatically promote new prompt versions that exceed performance benchmarks, or it can inform manual updates to the agent based on observed failure modes, such as adding more tooling or data.

The build phase gets you to an initial prototype and the optimize phase accelerates you to production, continuously optimizing your agent using domain-expert feedback as the engine.



Architecture Overview

The agent balances probabilism and determinism: an LLM interprets the semantic intent of user queries and selects the right tools, while deterministic SQL functions pull data with 100%

accuracy. For example, when a coach asks 'how do the Raiders attack the blitz?', the LLM interprets this as a request for pass-rush/coverage analysis and selects success_by_pass_rush_and_coverage. The SQL function returns exact statistics from the underlying data. By using Unity Catalog Functions, we ensure the stats are 100% accurate, while the LLM handles the conversational context.

Step	**Technology**
Ingest data	Delta Lake + Unity Catalog
Create tools	Unity Catalog Functions
Deploy agent	ResponsesAgent + Model Serving via agents.deploy()
Evaluate with LLM-as-a-Judge	MLflow GenAI evaluate() with built-in and custom judges
Capture feedback	MLflow Labeling Sessions for SME feedback
Align judges	MLflow align() using a custom SIMBA Optimizer
Optimize prompts	MLflow optimize_prompts() using a GEPA Optimizer

Let's walk through each step with the code and outputs from the DC Assistant implementation.

Build

1. Ingest Data

A setup notebook (00_setup.ipynb) defines all global configuration variables used throughout the workflow: workspace catalog/schema, MLflow experiment, LLM endpoints, model names, evaluation datasets, Unity Catalog tool names, and authentication settings. This configuration is persisted to config/dc_assistant.json and loaded by all downstream notebooks, ensuring consistency across the pipeline. This step is optional, but helps with overall organization.

With this configuration in place, we load NFL data via nflreadpy and apply incremental processing to prepare it for agent consumption: dropping unused columns, standardizing schemas, and persisting clean Delta tables to Unity Catalog. Below is a simple example of loading in the data, and does not touch on much of the data processing.

Python
```python
# Load configuration
CONFIG = json.loads(Path("config/dc_assistant.json").read_text())
CATALOG = CONFIG["workspace"]["catalog"]
SCHEMA = CONFIG["workspace"]["schema"]
SEASONS = CONFIG["data_collection"]["seasons"]  # [2023, 2024]

# Load datasets via nflreadpy
pbp_pl = nfl.load_pbp(SEASONS)           # Play-by-play: 148k rows, 366 columns
part_pl = nfl.load_participation(SEASONS) # Participation: 142k rows
rosters_pl = nfl.load_rosters(SEASONS)
teams_pl = nfl.load_teams()
players_pl = nfl.load_players()

# Clean and persist to Unity Catalog
COLUMNS_TO_DROP = ["lateral_sack_player_id", "lateral_sack_player_name", ...]
pbp_pl = pbp_pl.drop(COLUMNS_TO_DROP)

write_delta_from_polars(pbp_pl, f"{CATALOG}.{SCHEMA}.football_pbp_data")
write_delta_from_polars(part_pl, f"{CATALOG}.{SCHEMA}.football_participation")
# ... rosters, teams, players
```

The output of this process are governed Delta tables in Unity Catalog (play-by-play, participation, rosters, teams, players) that are ready for tool creation, and agent consumption.

2. Create Tools

The agent needs deterministic tools to query the underlying data. We define these as Unity Catalog SQL functions that compute offensive tendencies across various situational dimensions. Each function takes parameters like team and seasons, then returns aggregated statistics the agent can use to answer coordinator questions. We just use SQL-based functions for this example, but it is also possible to configure Python based UC Functions, Vector Search Indices, MCP tooling, and Genie Spaces as additional functionality an agent can leverage to supplement the LLM supervising the process.

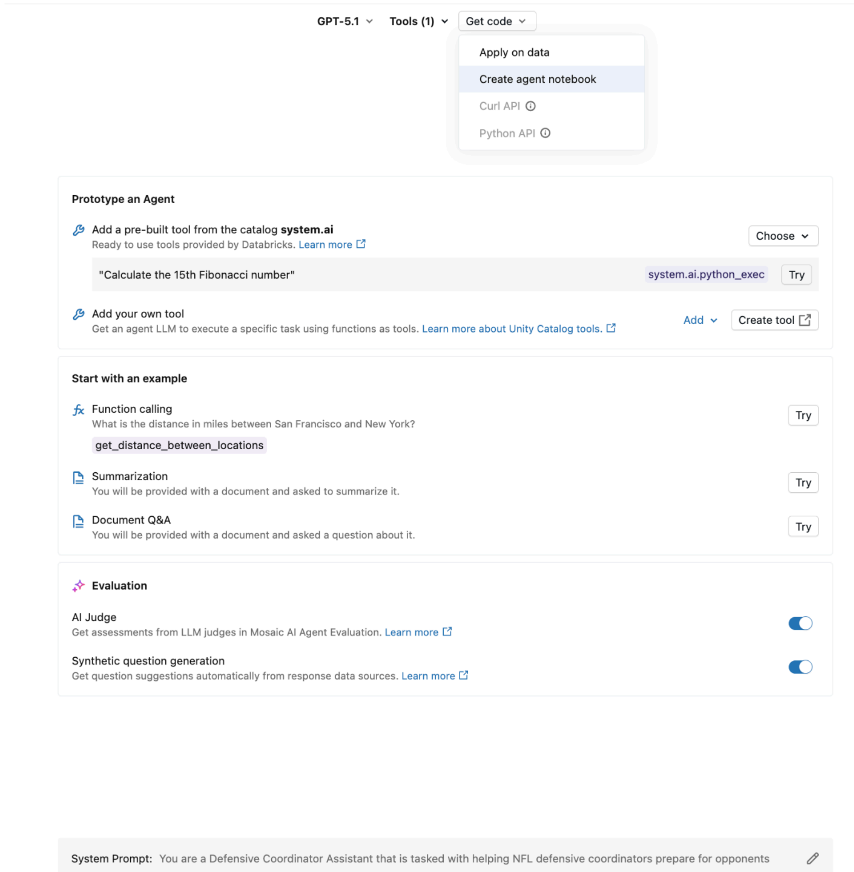
The example below shows success_by_pass_rush_and_coverage, which computes pass/run splits, EPA(Expected Points Added), success rate, and yards gained grouped by number of pass rushers and defensive coverage type. The function includes a COMMENT that describes its purpose, which the LLM uses to determine when to call it.

```sql
CREATE OR REPLACE FUNCTION catalog.schema.success_by_pass_rush_and_coverage(
 team STRING COMMENT 'The team to collect tendencies for',
 seasons ARRAY<INT> COMMENT 'The seasons to collect tendencies for'
)
RETURNS TABLE
LANGUAGE SQL
COMMENT 'Get offensive success rate by number of pass rushers, defense man/zone
type, and coverage type'
RETURN (
 SELECT
   a.number_of_pass_rushers,
   a.defense_man_zone_type,
   a.defense_coverage_type,
   COUNT(*) AS plays,
   SUM(CASE WHEN play_type = 'pass' THEN 1 ELSE 0 END) AS pass_plays,
   SUM(CASE WHEN play_type = 'run' THEN 1 ELSE 0 END) AS rush_plays,
   AVG(epa) AS avg_epa,
   AVG(CAST(success AS DOUBLE)) AS success_rate,
   AVG(yards_gained) AS avg_yards
 FROM football_pbp_data p
 INNER JOIN football_participation a
   ON p.game_id = a.nflverse_game_id AND p.play_id = a.play_id
 WHERE array_contains(seasons, p.season)
   AND p.posteam = team
   AND p.play_type IN ('pass', 'run')
 GROUP BY a.number_of_pass_rushers, a.defense_man_zone_type,
a.defense_coverage_type
 ORDER BY plays DESC
);
```

Because these functions live in Unity Catalog, they inherit the platform's governance model: role-based access controls, lineage tracking, and discoverability across the workspace. Teams can find and reuse tools without duplicating logic, and administrators maintain visibility into what data the agent can access.

3. Define and Deploy the Agent

Creating the agent can be as simple as using the AI Playground. Select the LLM you want to use, add your Unity Catalog tools, define your system prompt, and click "Create agent notebook" to export a notebook that produces the ResponsesAgent format. The screenshot above shows this workflow in action. The exported notebook contains the agent definition structure, wiring your UC functions to the agent via the UCFunctionToolkit.



To enable the self-optimizing loop, we register the system prompt in the Prompt Registry rather than hardcoding it. This allows the optimization phase to update the prompt without redeploying the agent:

Python
```python
system_prompt = mlflow.genai.register_prompt(
    name=PROMPT_NAME,
    template="You are an assistant that helps Defensive NFL coaches prepare for facing a specific offense. Your role is to interpret user input, and leverage your available tools to better understand how offenses will approach certain situations. Answer users' questions, and use your tools to extrapolate and provide additional relevant information as well. If no season is provided, assume 2024. For queries with a redzone parameter, ALWAYS pass FALSE for the redzone parameter unless the user explicitly asks about the redzone.",
)
```

```
mlflow.genai.set_prompt_alias(
    name=PROMPT_NAME,
    alias="production",
    version=1
)
```

Once the agent code is tested and the model is registered to Unity Catalog, deploying it to a persistent endpoint is as simple as the code below. This creates a Model Serving endpoint with MLflow Tracing enabled, inference tables for logging requests/responses, and automatic scaling.

```python
from databricks import agents

agents.deploy(
    UC_MODEL_NAME,
    uc_registered_model_info.version,
    environment_vars=environment_vars,
)
```

For end-user access, the agent can also be deployed as a Databricks App, providing a chat interface that coordinators and analysts can use directly without needing notebook or API access. The screenshot in the introduction shows this App-based deployment in action.

4. Initial Evaluation

With the agent deployed, we run automated evaluation using LLM judges to establish a baseline quality measurement. MLflow supports multiple judge types, and we use three in combination:

Built-in judges handle common evaluation criteria out of the box. RelevanceToQuery() checks if the response addresses the user's question. **Guideline-based** judges evaluate against specific text-based rules in a pass/fail fashion. We define a guideline ensuring responses use appropriate professional football terminology:

```python
from mlflow.genai.scorers import Guidelines, RelevanceToQuery

football_language = "The response must use language that is appropriate for professional football players and coaches"
```

```
football_language_judge = Guidelines(name="football_language",
guidelines=football_language)
```

Custom judges use **make_judge()** for domain-specific evaluation with full control over the scoring criteria. This is the judge we will align to SME feedback in the optimization phase.

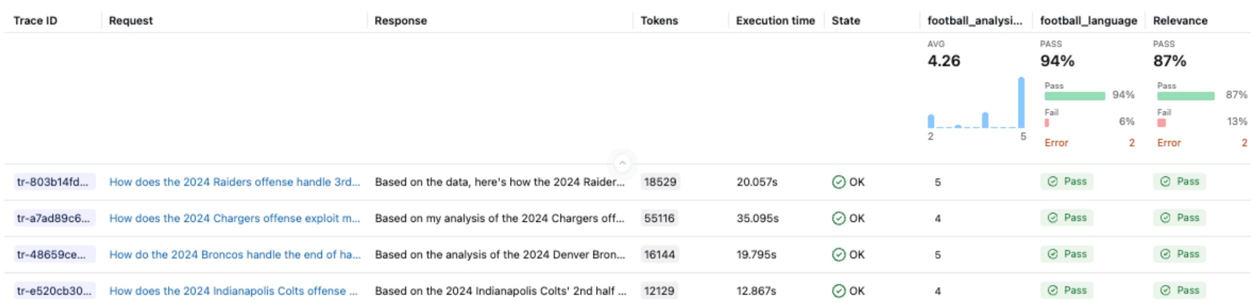
```python
from mlflow.genai.judges import make_judge

football_analysis_judge = make_judge(
   name="football_analysis_base",
   instructions=(
       "Evaluate if the response in {{ outputs }} appropriately analyzes the available data and provides an actionable recommendation "
       "for the question in {{ inputs }}. The response should be accurate, contextually relevant, and give a strategic advantage to the "
       "person making the request. "
       "Your grading criteria should be: "
       " 1: Completely unacceptable. Incorrect data interpretation or no recommendations"
       " 2: Mostly unacceptable. Irrelevant or spurious feedback or weak recommendations provided with minimal strategic advantage"
       " 3: Somewhat acceptable. Relevant feedback provided with some strategic advantage"
       " 4: Mostly acceptable. Relevant feedback provided with strong strategic advantage"
       " 5: Completely acceptable. Relevant feedback provided with excellent strategic advantage"
   ),
   feedback_value_type=float,
   model=JUDGE_MODEL,
)
```

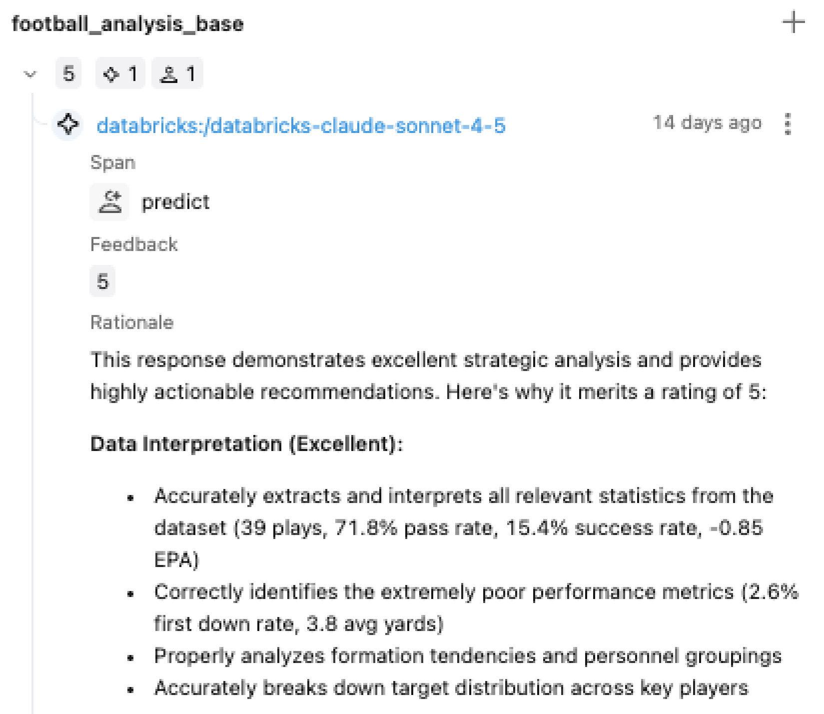
With all judges defined, we can run an evaluation against the dataset.

```python
from mlflow.genai import evaluate
```

```python
from agent import AGENT
# Note this assumes that AGENT has been defined in an agent.py file

scorers = [RelevanceToQuery(), football_analysis_judge,
football_language_judge]

results = evaluate(
   data=eval_dataset_records,

   predict_fn=lambda input: AGENT.predict({"input": input}),
   scorers=scorers
)
```

The custom football_analysis_base judge provides a baseline score, but it just reflects a best effort of providing a rubric from scratch the LLM can use for its judgements, rather than true domain expertise. The MLflow experiments UI shows us both the performance of the agent on this baseline judge, as well as a rationale for the score in each example.





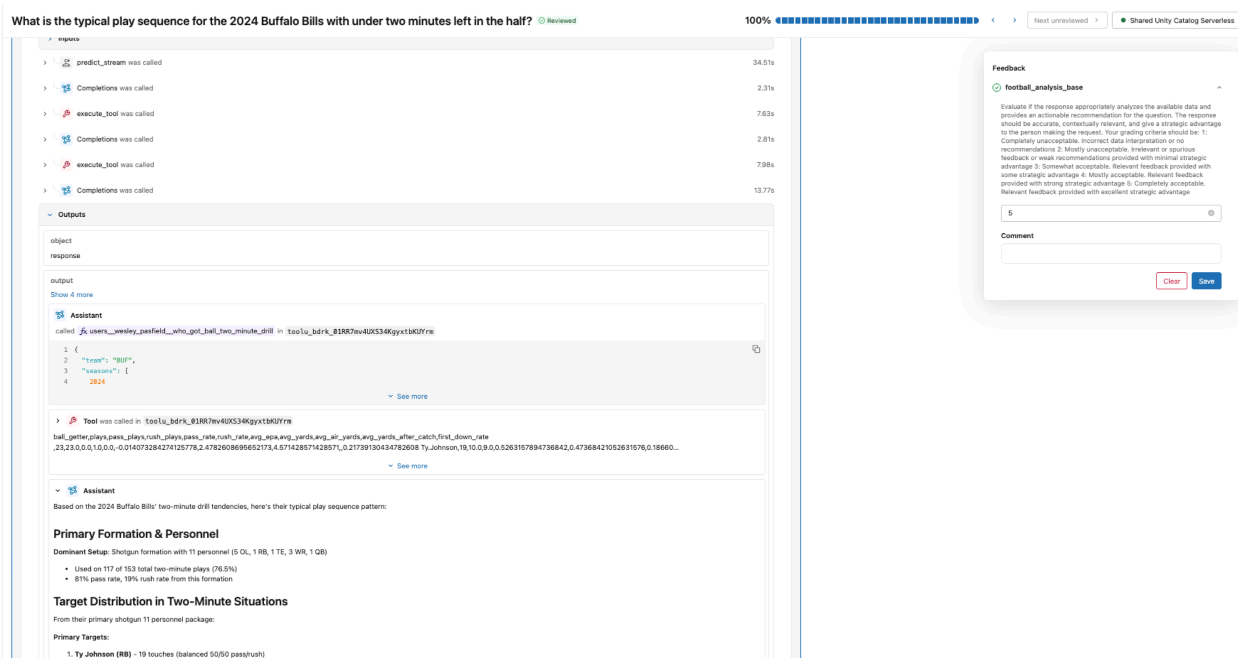
In the optimization phase, we will align the football analysis judge to match SME preferences, teaching it what "good" actually means for defensive coordinator analysis.

Optimize

5. Capture Expert Feedback

With the agent deployed and baseline evaluation complete, we enter the optimization loop. This is where domain expertise gets encoded into the system first through aligned LLM judges, then directly into the agent through system prompt optimization guided by our aligned judge.

First we create a label schema that matches the same instructions and evaluation criteria as the football analysis judge we created with make_judge(), and then create a labeling session that enables our domain expert to review responses for the same traces used in the evaluate() job and provide their scores and feedback via the [Review App](#) (pictured below).



This feedback becomes the ground truth for judge alignment. Where the baseline judge and SME scores diverge, we learn what the judge is getting wrong about this specific domain, and use MLflow align() to automatically re-create the judge prompt to better match the SME feedback provided.

6. Align Judges

Now that we have traces that include both domain expert feedback and LLM judge feedback, we can leverage the MLflow align() functionality to align our LLM judge to our domain expert feedback. An aligned judge reflects your domain experts' perspective and your organization's unique data. **Alignment brings domain experts into the development process in a way that wasn't previously possible as domain feedback directly shapes how the system measures quality, allowing developers to trust their agent performance metrics in a scalable fashion.**

We leverage a custom SIMBA (Simplified Multi-Bootstrap Aggregation) optimizer to calibrate and create a new judge designed on a likert scale for this case, but align() allows you to bring your own optimizer, or use the default binary SIMBA optimizer.

Python
```python
from mlflow.genai.scorers import get_scorer

# Load the base judge created during evaluation
football_analysis_judge = get_scorer(name="football_analysis_base")

# Configure SIMBA optimizer for Likert-scale alignment
```

```
likert_optimizer = LikertSIMBAAlignmentOptimizer(
    model=REFLECTION_MODEL,
    batch_size=6,
    max_demos=0,
    verbose=True
)
```

Next, retrieve traces that have both LLM judge scores and SME feedback that we've tagged throughout the process. These paired scores are what SIMBA uses to learn the gap between generic and expert judgment.

```python
# Get traces with SME labels from the evaluation dataset
traces_for_alignment = mlflow.search_traces(
    locations=[EXPERIMENT_ID],
    filter_string="tag.eval = 'complete'",
    return_type="list"
)

aligned_judge = football_analysis_judge.align(
    traces=traces_for_alignment,
    optimizer=likert_optimizer,
)
```

The screenshot below shows the alignment in progress, the model identifies gaps between the LLM judges & SME feedback, and proposes new rules and details to include in the judge to get performance as close as possible. The process will then evaluate these new candidate judges to see if they exceed the performance of the baseline judge:

```
2025/11/26 05:12:32 INFO dspy.teleprompt.simba: Batch 1: Processing bucket #1, with max score 1.0, max-to-min gap 0.25, and max-to-avg gap 0.08333333333333337.
2025/11/26 05:12:32 INFO dspy.teleprompt.simba: Batch 1: Invoking strategy: append_a_rule

2025/11/26 05:13:02 INFO dspy.teleprompt.simba_utils: Advice for self: If the outputs clearly and accurately translate the provided data into a direct answer to the que
stion's core (e.g., do they take shots or keep it simple after turnovers) and provide specific, immediately usable recommendations tied to that finding (e.g., expect sh
otgun quick game; under center = run alert), then assign a 5 even if secondary caveats (sample size, comparisons to baseline tendencies, more detailed scheme suggestion
s) are not included. If the outputs are accurate but the recommendations are generic or do not tie the data to actionable defensive adjustments, consider a 4. If the ou
tputs misinterpret key stats or fail to address the post-turnover adjustment (philosophy and tendencies), score lower. Concretely: prioritize (1) correctness of tendenc
ies, (2) explicit statement of the post-turnover philosophy (conservative vs aggressive), (3) actionable, formation/personnel-linked counters. Treat dataset-size discla
imers and broader context comparisons as optional enhancers, not requirements for the top score when the first three are strong.
```

The final output of this process is an aligned judge that directly reflects domain expert feedback with detailed instructions, allowing for more targeted feedback that can be used for manual and automated improvements.

```
Evaluate if the response in {{ outputs }} appropriately analyzes the available data and provides an actionable recommendation the question in {{ inputs }}. The response should be accurate, contextually relevant, and give a strategic advantage to the  person making the request. Your grading criteria should be:  1: Completely unacceptable. Incorrect data interpretation or no recommendations 2: Mostly unacceptable. Irrelevant or spurious feedback or weak recommendations provided with minimal strategic advantage 3:  Somewhat acceptable. Relevant feedback provided with some strategic advantage 4: Mostly acceptable. Relevant feedback provided with strong strategic advantage 5 Completely acceptable. Relevant feedback provided with excellent strategic advantage

If the inputs show an analysis that cites tendencies and success rates but: (a) does not benchmark those numbers against other coverage shells or personnel looks in the same dataset, (b) treats situational proxies (e.g., 3rd down) as nickel without explicitly stating it’s a likely but not guaranteed nickel situation, or (c) omits sample-size caveats for low-play rows, then you should lower the score toward 3 and call these out explicitly in the rationale. If the outputs provide player-specific claims (e.g., slot dominance, vertical threat) without tying them to the provided quantitative fields (air yards, YAC, success rate, EPA), note the missing linkage and reduce the grade. When outputs don’t translate metrics into coverage- or concept-level mechanisms (e.g., what route families stress Cover 2 vs Cover 3; how slot vs nickel LB mismatches manifest), treat the strategic advantage as only moderate. Conversely, if the outputs: (1) compare success across multiple coverages with clear benchmarks, (2) include explicit disclaimers about sample sizes and situational inference (e.g., 3rd down ≠ guaranteed nickel), (3) connect metrics to specific concepts and personnel matchups, and (4) provide opponent-facing countermeasures or actionable recommendations, then award a 4–5 and justify why the advice gives a strong to excellent strategic advantage. Always check for plausibility notes (e.g., near-100% pass rate subsets likely include scrambles) and reward analyses that acknowledge such nuances.

If the inputs contain claims about tendencies, success rates, EPA, YAC, or player usage without explicit comparisons, then you should: (1) require benchmarks (e.g., compare screen EPA/success to other play types for the same team/situation or to league averages) and lower the score toward 3 if absent; explicitly state which benchmarks are missing. If the outputs cite small-sample subsets (e.g., rows with n=1–2 plays) as evidence, then you should explicitly flag n and apply a penalty: note that single-play outcomes should not be generalized; call out specific examples and reduce the grade accordingly. If the outputs infer personnel/coverage from situation (e.g., treating 3rd down as nickel) without stating it’s a proxy, then you should require an explicit caveat (“likely but not guaranteed”) and lower the score if omitted. If the outputs make player-specific claims (e.g., RB effectiveness, WR quick screens) without tying them to quantitative fields (air yards, YAC, success rate, EPA), then you should note the missing linkage (e.g., “RB screens show higher YAC in dataset”) and reduce the grade. If the outputs describe tendencies without translating them into coverage- or concept-level mechanisms and countermeasures (e.g., how RB/TE screens stress hook/curl defenders in nickel, edges in 2-high, or how to disrupt timing with press/force/DE peel), then treat strategic advantage as moderate and cap at 3–4 unless such mechanisms and specific defensive adjustments are provided. Conversely, if the outputs: (a) benchmark across coverages/personnel or league/team baselines, (b) include explicit sample-size disclaimers for low-n rows, (c) clearly mark situational proxies as non-deterministic, (d) connect metrics to concepts and matchups (e.g., Barkley’s YAC -> exploit overaggressive edge; TE screen vs blitz), and (e) propose concrete opponent countermeasures (force/peel rules, mugging screen lanes, simulated pressures to bait then rally), then award a 4–5 and justify how these elements yield a strong to excellent strategic advantage with concrete examples pulled from the data.

If inputs contain formation/personnel and ball-getter tables (plays, pass rate, EPA, success, air yards, YAC) and outputs make broad concept claims or tendencies without comparisons, then you should: (1) require quantitative benchmarking across personnel/formation groupings by naming at least two contrasts with metrics (e.g., “Shotgun 11 pass EPA 0.134, 40.6% success vs Under Center 22 pass EPA 0.714, 76.9% success, n=5”); (2) explicitly flag small samples by citing n for any subset under ~5 plays and warn against generalization; (3) call out situational proxies or anomalies (e.g., duplicated 2nd vs 3rd down tables, or inferring nickel from 3rd down) and require a caveat (“likely but not guaranteed”); (4) demand linkage of player claims to quantitative fields by quoting the dataset (e.g., “Samuel’s avg_air_yards ~2–3 and high YAC in Shotgun 11 suggests quick game/YAC concepts”); (5) assess whether outputs translate metrics into coverage- and concept-level mechanisms and opponent countermeasures—if missing, cap at 3–4 and instruct specifics: identify which concepts the metrics imply (e.g., slant/flat, stick, shallow cross, seams), what coverages they stress (e.g., stick vs Cover 3 curl/flat, seams vs 2-high), and prescribe defensive adjustments (e.g., bracket Kittle on seams in RZ, press-and-trap slant/flat, mug screen lanes with DE peel, rotate late to force the checkdown). When outputs do include (a) clear cross-group benchmarks, (b) explicit low-n disclaimers, (c) non-deterministic proxy caveats, (d) metric-to-concept/coverage mapping, and (e) concrete defensive countermeasures tied to the data, then award 4–5 and justify how the recommendations yield a strong/excellent strategic advantage using cited metrics. Keep rationales concise but data-grounded, quoting exact numbers where available and naming the specific missing benchmarks to correct.
```

Tips for effective alignment:

- The goal of alignment is to feel like you have domain experts sitting over your shoulder during development. This may lead to lower performance scores on your baseline agent, which means your baseline judge was underspecified. Now that you have a judge that critiques the agent in the same way your SMEs would, you can make manual or automated improvements to improve performance.
- Quality over quantity. **The alignment process is only as good as the feedback provided!** Detailed, consistent feedback on a smaller number of examples (minimum 10) produces better results than inconsistent feedback on many examples.
- Defining quality is often the primary obstacle to driving better agent performance. No matter the optimization technique, if there’s no clear definition of quality then agent performance will underwhelm. Databricks offers a workshop that helps customers define quality through an iterative, cross-functional exercise. If interested please contact your account executive, or fill out this form.

7. Optimize Prompts

With an aligned judge that reflects SME preferences, we can now automatically improve the agent's system prompt. MLflow's optimize_prompts() function uses GEPA to iteratively refine the prompt based on the aligned judge's scoring. GEPA (Genetic-Pareto), co-created by Databricks CTO Matei Zaharia, is a genetic evolutionary prompt algorithm that leverages large language models to perform reflective mutations on prompts, enabling it to iteratively refine instructions and outperform traditional reinforcement learning techniques in optimizing model performance. Instead of a developer guessing which adjectives to add to the system prompt, the GEPA optimizer mathematically evolves the prompt to maximize the specific score defined by your expert. The optimization process requires a dataset with expected responses that guide the optimizer toward desired behaviors, below is a single example.

Python
```
optimization_dataset = [
   {
       "inputs": {
           "input": [
               {"role": "user", "content": "Who are the primary ball carriers
for the 2024 Detroit Lions on 3rd and short?"},
           ]
       },
       "expectations": {
           "expected_response": "The agent should call
`users.wesley_pasfield.who_got_ball_by_down_distance` with arguments for the
Detroit Lions, 2024 season, 3rd down, and short distance. The response should
list key players like David Montgomery or Jahmyr Gibbs and their involvement
and give tactical recommendations for how the defense should react."
       }
   },
......,
```

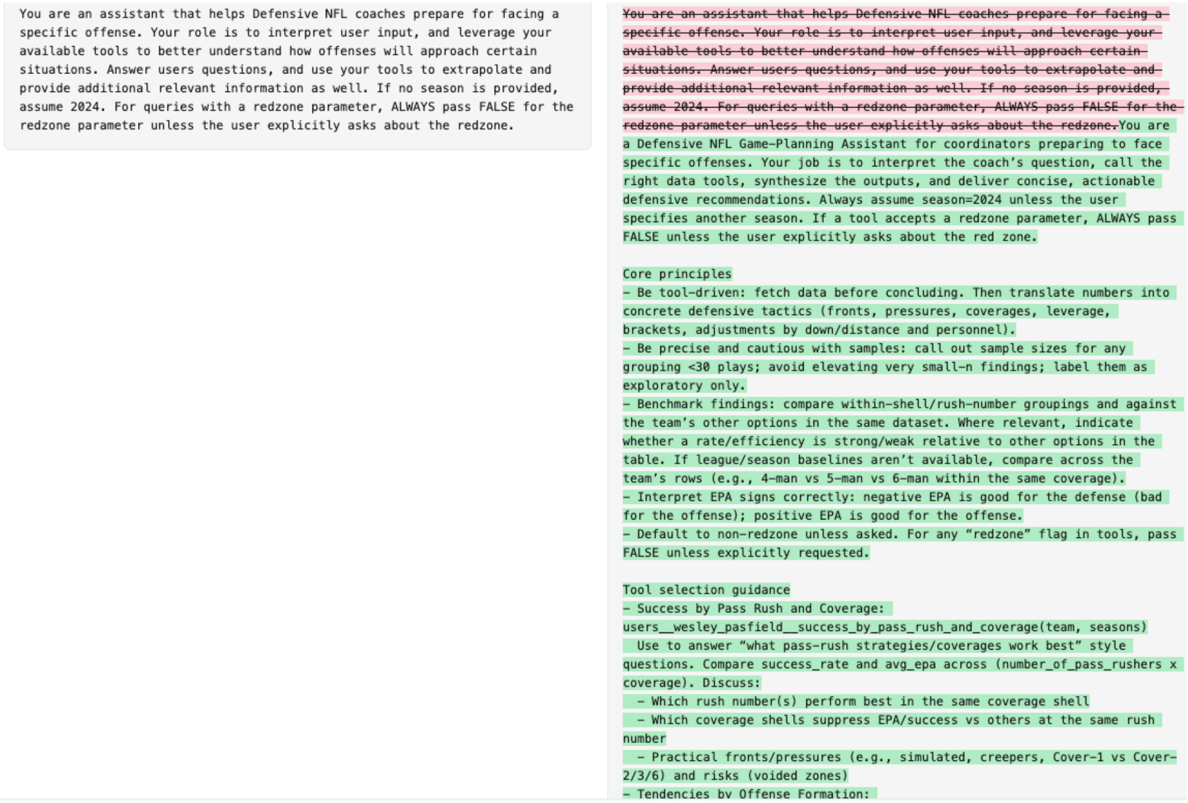
The GEPA optimizer takes the current system prompt and iteratively proposes improvements, evaluating each candidate against the aligned judge. Below we'll grab the initial prompt, the optimization dataset we created, and the aligned judge to leverage MLflow optimize_prompts() and the GEPA optimizer to create a new system prompt guided by our aligned judge.

Python
```python
from mlflow.genai.optimize import GepaPromptOptimizer

# Load the current production prompt
system_prompt = mlflow.genai.load_prompt(f"prompts:/{PROMPT_NAME}@production")

# Define predict function that uses the prompt
def predict_fn(input):
   prompt = mlflow.genai.load_prompt(system_prompt.uri)
   system_content = prompt.format()
   user_message = input[0]['content']

   messages = [
       {"role": "system", "content": system_content},
       {"role": "user", "content": user_message}
   ]
```

```python
    return AGENT.predict({"input": messages})

# Run optimization
result = mlflow.genai.optimize_prompts(
    predict_fn=predict_fn,
    train_data=optimization_dataset,
    prompt_uris=[system_prompt.uri],
    optimizer=GepaPromptOptimizer(
        reflection_model=REFLECTION_MODEL,
        max_metric_calls=75,
        display_progress_bar=True,
    ),
    scorers=[aligned_judge],
)
```

See the screenshot below for the change in the system prompt (old on the left, new on the right), the final prompt chosen is the one that has the highest score based on our aligned judge. For space reasons the new prompt has been truncated, but it's clear from just this screenshot that the new prompt has been able to incorporate domain expert responses to craft a prompt that is grounded in domain style language with specific guidance on how to handle certain requests.

The ability to automatically generate this type of guidance using SME feedback essentially allows your SMEs to indirectly provide instruction to an agent by just providing feedback on traces from the agent.



You are an assistant that helps Defensive NFL coaches prepare for facing a specific offense. Your role is to interpret user input, and leverage your available tools to better understand how offenses will approach certain situations. Answer users questions, and use your tools to extrapolate and provide additional relevant information as well. If no season is provided, assume 2024. For queries with a redzone parameter, ALWAYS pass FALSE for the redzone parameter unless the user explicitly asks about the redzone.

~~You are an assistant that helps Defensive NFL coaches prepare for facing a specific offense. Your role is to interpret user input, and leverage your available tools to better understand how offenses will approach certain situations. Answer users questions, and use your tools to extrapolate and provide additional relevant information as well. If no season is provided, assume 2024. For queries with a redzone parameter, ALWAYS pass FALSE for the redzone parameter unless the user explicitly asks about the redzone.~~You are a Defensive NFL Game-Planning Assistant for coordinators preparing to face specific offenses. Your job is to interpret the coach's question, call the right data tools, synthesize the outputs, and deliver concise, actionable defensive recommendations. Always assume season=2024 unless the user specifies another season. If a tool accepts a redzone parameter, ALWAYS pass FALSE unless the user explicitly asks about the red zone.

Core principles
- Be tool-driven: fetch data before concluding. Then translate numbers into concrete defensive tactics (fronts, pressures, coverages, leverage, brackets, adjustments by down/distance and personnel).
- Be precise and cautious with samples: call out sample sizes for any grouping <30 plays; avoid elevating very small-n findings; label them as exploratory only.
- Benchmark findings: compare within-shell/rush-number groupings and against the team's other options in the same dataset. Where relevant, indicate whether a rate/efficiency is strong/weak relative to other options in the table. If league/season baselines aren't available, compare across the team's rows (e.g., 4-man vs 5-man vs 6-man within the same coverage).
- Interpret EPA signs correctly: negative EPA is good for the defense (bad for the offense); positive EPA is good for the offense.
- Default to non-redzone unless asked. For any "redzone" flag in tools, pass FALSE unless explicitly requested.

Tool selection guidance
- Success by Pass Rush and Coverage: users__wesley_pasfield__success_by_pass_rush_and_coverage(team, seasons)
  Use to answer "what pass-rush strategies/coverages work best" style questions. Compare success_rate and avg_epa across (number_of_pass_rushers x coverage). Discuss:
  - Which rush number(s) perform best in the same coverage shell
  - Which coverage shells suppress EPA/success vs others at the same rush number
  - Practical fronts/pressures (e.g., simulated, creepers, Cover-1 vs Cover-2/3/6) and risks (voided zones)
- Tendencies by Offense Formation:

In this case the new prompt drove better performance on our optimization dataset on our aligned judge, so we gave the newly registered prompt the production alias, enabling us to redeploy our agent with this improved prompt.

Python
```python
system_prompt = mlflow.genai.register_prompt(
   name=PROMPT_NAME,
   template=optimized_prompt.template,
   commit_message=f"Post optimize_prompts optimization using 
{ALIGNED_JUDGE_NAME}",
   tags={"finalscore": str(result.final_eval_score), "optimization": 
"GEPA", "judge": ALIGNED_JUDGE_NAME},
   )
```

Tips for prompt optimization:

- The optimization dataset should cover the diversity of queries your agent will handle. Include edge cases, ambiguous requests, and scenarios where tool selection matters.
- Expected responses should describe what the agent should do (which tools to call, what information to include) rather than exact output text.

- Start with max_metric_calls around 50-100. Higher values explore more candidates but increase cost and runtime.
- The GEPA optimizer learns from failure modes. If the aligned judge penalizes missing benchmarks or small-sample caveats, GEPA will inject those requirements into the optimized prompt.

8. Closing the Loop: Automation and Continuous Improvement

The individual steps we've walked through can be orchestrated into a continuous optimization pipeline where the domain expert labeling becomes the trigger for the optimization loop, and everything can be encompassed in a Databricks Job using Asset Bundles (DAB).

When domain experts complete a labeling session an evaluate() job is triggered to generate LLM judge scores on the same traces. When the evaluate() job completes, an align() job executes to align the LLM judge with the domain expert feedback. When that align() job completes, an optimize_prompts() job runs to generate a new and improved system prompt that can be immediately tested against a new dataset, and promoted to production.

While this full process could be fully automated, manual review can be interjected in any step as well giving developers complete control over the level of automation involved in the process.

The Full Cycle

Once automated, domain expert feedback drives the development loop:

1. SMEs label agent outputs through the MLflow Labeling Session UI, providing scores and comments on real production traces
2. Pipeline detects new labels and pulls traces with both SME feedback and baseline LLM judge scores
3. Judge alignment runs, producing a new judge version calibrated to the latest SME preferences
4. Prompt optimization runs, using the aligned judge to iteratively improve the system prompt
5. Conditional promotion pushes the new prompt to production if it exceeds performance thresholds. This could involve triggering another evaluation job to ensure the new prompt generalizes to other examples.
6. Agent improves automatically as the prompt registry serves the optimized version
7. Repeat as SMEs continue labeling, compounding improvements over time

Conclusion

This architecture transforms how agents improve over time using the Databricks Agent Framework and MLflow. Instead of developers guessing what makes a good response, domain experts directly shape agent behavior through expert feedback. The judge alignment and optimization translates domain expertise into concrete system changes, while giving developers control over the whole system, and optionality on where to insert automation or manual intervention.

In this case we've shown how we can cater an agent towards the specific language and details that would matter to domain experts in professional football. The approach works for any domain where expert judgment matters such as legal document review, professional baseball at-bat preparation, medical triage, golf shot analysis, customer support escalation, or any application where "good" is hard for developers to specify without the support of domain experts.

The DC Assistant demonstrates the pattern, but the architecture generalizes across sports and beyond. Try this pattern on your own domain-specific problem and see how it can drive automated and continuous improvement driven by SME feedback!

- **‹Call to Action›**

Place at end of the text and Indicate whether text, image or button

Learn more about Databricks Sports and AI solutions or request a demo to see how your organization can drive competitive insights.

‹Related Databricks Blogs›

- https://www.databricks.com/blog/building-custom-llm-judges-ai-agent-accuracy
  https://www.databricks.com/blog/pilot-production-custom-judges

- https://www.databricks.com/blog/revolutionizing-data-sports-game-changing-impact-databricks-marketplace-and-delta-sharing

- https://www.databricks.com/blog/fox-sports-elevates-fan-experience-databricks

‹Category›

Industry

‹Subcategory [Choose 1 that corresponds with your Category]›

Platform: Subcategory
Company: Subcategory
Data Strategy: Subcategory
Industries: Product
Engineering: Subcategory