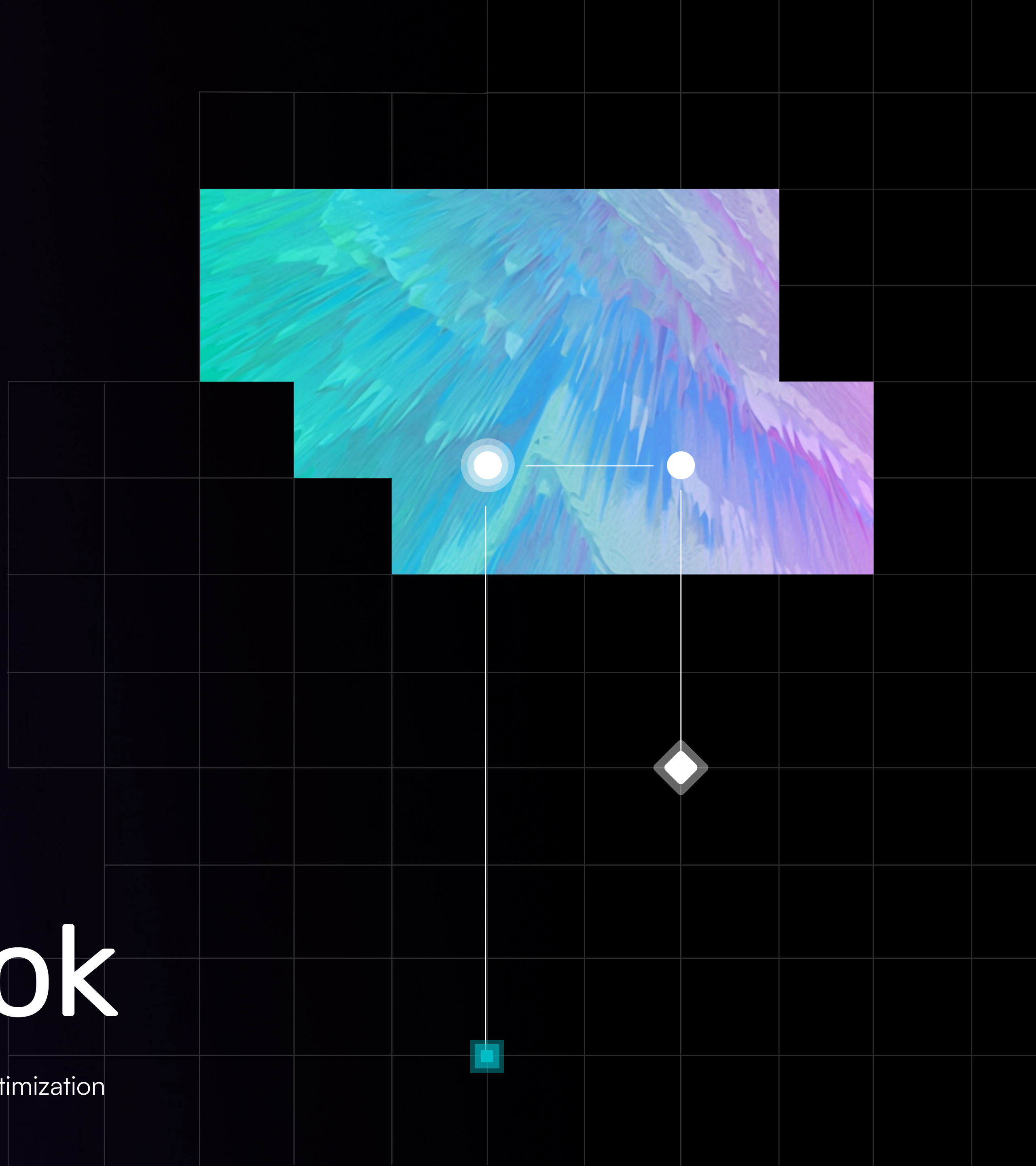


Prompt Learning Playbook

A new reinforcement learning-inspired approach to improving prompts for agent optimization



Contents

01	—————→	INTRODUCTION
02	—————→	UNDERSTANDING PROMPT LEARNING AND ITS BASIS IN RESEARCH
07	—————→	A CONCRETE EXAMPLE: BOOSTING CLINE ACCURACY
16	—————→	APPLY IT
16	→	PROMPT LEARNING IN ARIZE AX
22	→	PROMPT LEARNING SDK

Introduction

As more agents get deployed, a question emerges: can reinforcement learning control systems be built only in prompts? After all, reinforcement learning is all about using real-world feedback to optimize toward a goal, traditionally by adjusting model weights. But prompts themselves are the primary interface for guiding large language models.

Prompt learning (PL) is a new approach and algorithm developed by Arize AI. Unlike traditional optimization methods that rely on numerical scores, prompt learning uses natural language feedback to iteratively improve prompts. The roots of this approach are in the Voyager [paper](#) by Jim Fan's team at NVIDIA.

Like reinforcement learning, prompt learning follows an action → evaluation → improvement loop — but instead of gradients, it uses meta prompting: feeding a prompt into an LLM and asking it to make it better. Arize added a key twist: LLM-generated feedback explaining why outputs were right or wrong, giving the optimizer richer signal to refine future prompts.

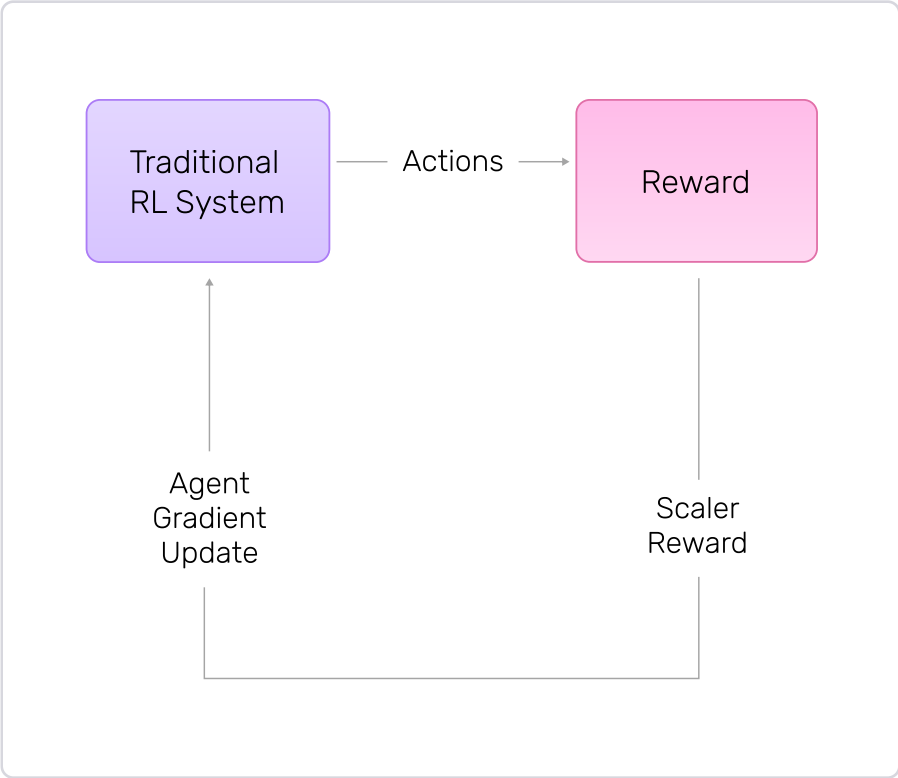
Understanding Prompt Learning

Instead of something like gradient descent, the prompt optimization technique that prompt learning uses is called **meta prompting**. Meta prompting is the simple concept of feeding a prompt into an LLM, and asking it to improve the prompt. You can also provide the meta prompt LLM with context that shows how the original prompt performs. This often includes input/output pairs, along with labels/scores on those outputs, such as whether those outputs were correct or incorrect.

This approach has one major limitation: the labels/scores/scalar reward provides very low information content to the meta prompt LLM about the individual data points. The LLM does not know WHY certain outputs were correct/incorrect, where improvement is needed, or any specific nuance about those datapoints.

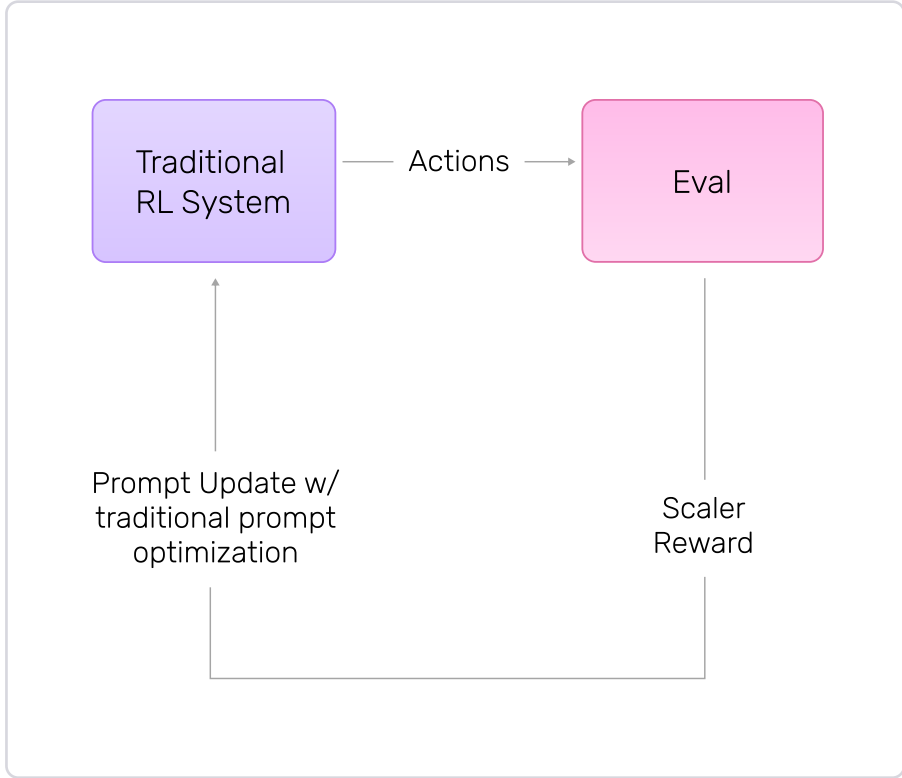
Prompt learning completes the algorithm by using LLMs to generate rich english feedback for the meta prompt. This helps guide the meta prompt LLM to producing better prompts.

TRADITIONAL RL



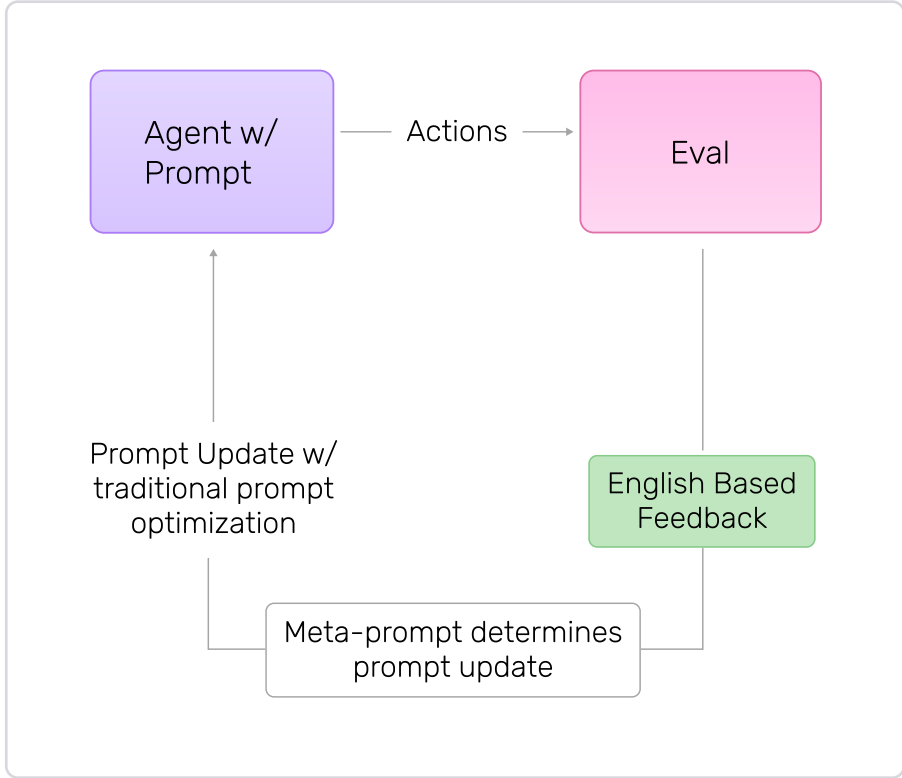
Use scaler reward to improve model. Requires many examples

META PROMPTING



Meta prompt uses scalar scores to improve prompt

PROMPT LEARNING ★



Meta prompt uses English feedback to improve prompt

- 01 **Initial Prompt:** Start with a baseline prompt that defines your task
- 02 **Generate Outputs:** Use the prompt to generate responses on your dataset
- 03 **Evaluate Results:** Run evaluators to assess output quality
- 04 **Optimize Prompt:** MetaPrompt(initial prompt, data, feedback) -> optimized prompt
- 05 **Iterate:** Repeat until performance meets your criteria

How Is This Different from Prompt Optimization?

There are a lot of techniques out there for **prompt optimization**. Prompt optimization applies more traditional machine learning train and test approaches to optimizing prompts by gathering examples and attempting to find similarities with those examples.

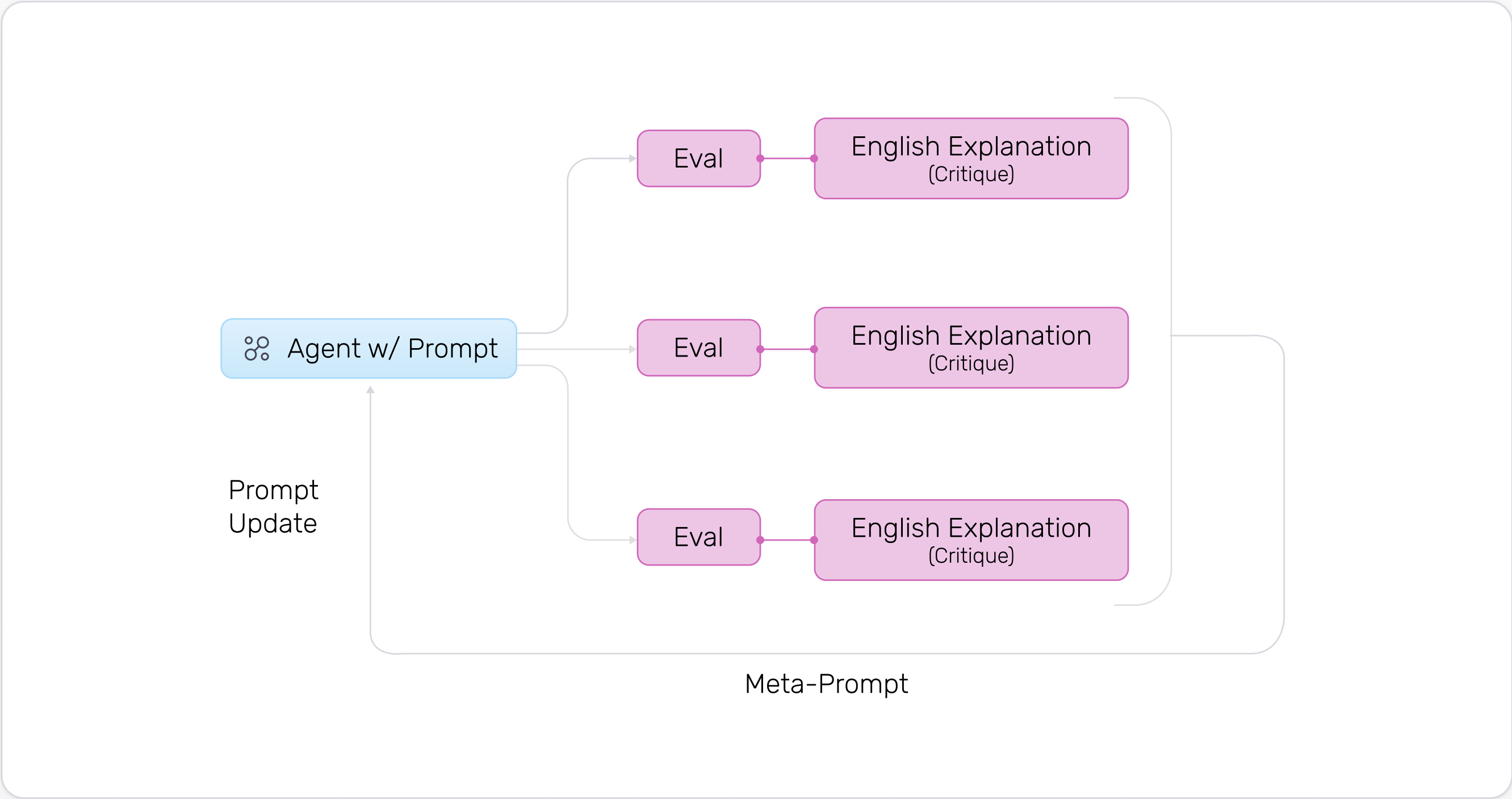
The seed of the failure of all prompt optimization approaches comes from the focus on scores as the means of propagating failure errors. As you think about failures, not every failure expresses itself easily as a numeric number and a numeric value hides the reason for a failure.

Using a score as your main approach for propagating a failure disconnects the optimization fix from the reason it failed.

	Prompt Learning	Reinforcement Learning	Prompt Optimization
Feedback Mechanism	Evaluation-based English explanations and human annotations	Numeric rewards	Numeric scores
Optimization	Metaprompt defines optimization approach	Updating model based on gradients	Varied but some support metaprompts
Prompt Control	Can optimize only specific section of prompt (instruction section)	N/A	Typically optimizes whole prompt
Online Setup	Designed to be used always on, with human control of “prompt change” acceptance or total automation	Designed to be used online	Normally one off

How Does the Optimization Loop Work?

In many real world use cases, as we tested with customers on real data, a single optimization run with a single shot output worked great. In cases where you need multiple loops over the optimization to improve performance, the English explanation (or critique) output of an Evaluator can improve performance.



The English explanation (Critique) is an important feature of our evaluation library, generating an explanation then allows the results to be used in a feedback loop.

In our testing, as the model was required to add more instructions back into the context window to fix the prompt, the iterative loop became more important. In cases where only 1-10 instructions needed to be added a single meta-prompt improvement loop was sufficient.

How Did We Test Prompt Learning?

We ran a series of optimization experiments using prompt learning in order to benchmark its efficacy. To date, this has been run across a sizable production set of AI application and agent use cases:

For our demo data application, we chose a JSON generation problem where models had to generate JSON for a webpage based on natural language prompts. We additionally generated a set of latent rules that the responses needed to follow.

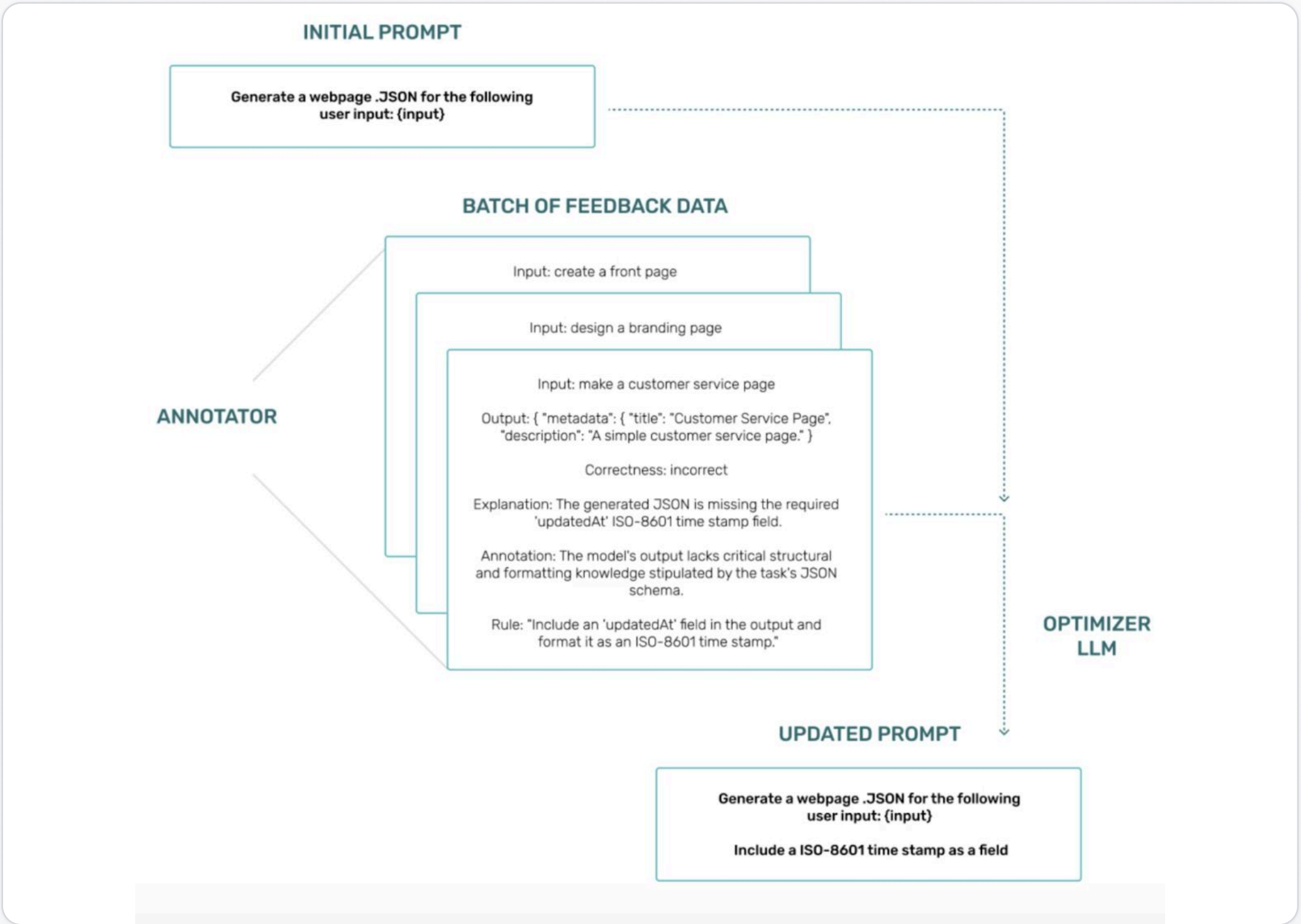
Things like:

- 1. Every section needs a type value from a predefined list
- 2. All images must include alt text
- 3. All external asset links must use `https`

These rules were implicitly represented in feedback and explanations attached to a set of traces of our application.

We designed this test to mimic a typical evaluation cycle of an agent. Evaluation was done using a mixture of LLM-as-a-judge techniques with human review, again to mimic real world patterns. All of this data (the application traces, feedback, and explanations) was then fed into the optimization stage.

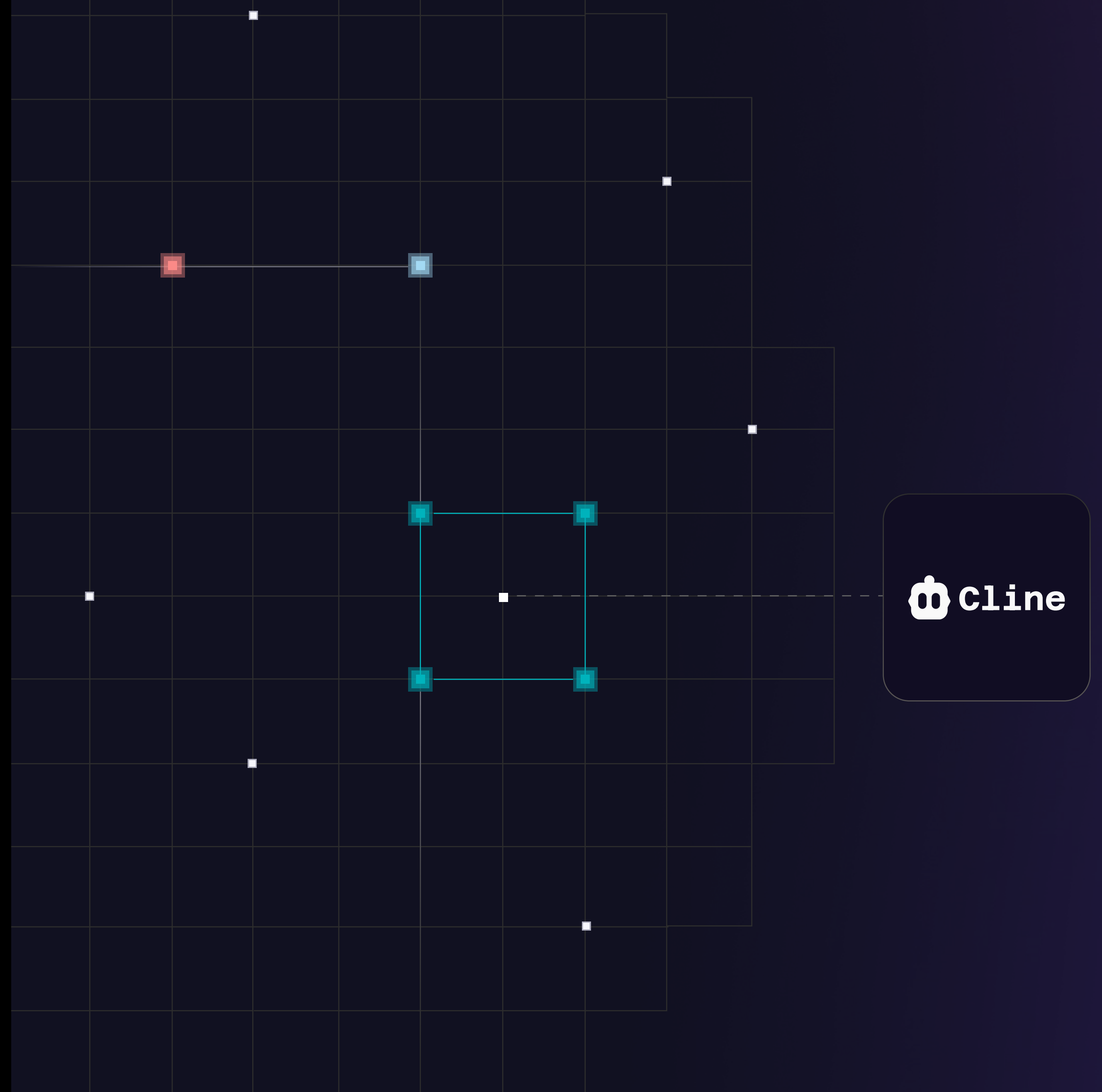
To perform the optimization itself, we used a modified version of meta-prompting that we later dubbed prompt learning.



Leveraging this approach, we tested prompt learning on real world customer deployments, internal synthetic data instruction learning tests, and well known benchmarks like Big Bench Hard and it posted promising results. Full results are explored [here](#).

Perhaps no test is more interesting, however, than the application of this approach to coding agents.

A Concrete Example: Boosting Cline Accuracy



Background

Coding agents have become the focal point of modern software development. Tools like Cursor, Claude Code, Codex, Cline, Windsurf, Devin, and many more are revolutionizing how engineers write and ship code.

A consistent pattern among the most capable agents is their reliance on a single, persistent system prompt rather than a chain of sub-prompts. Each session starts with a comprehensive base prompt defining available tools, I/O formats, and safety policies. From there, all messages build on the same context. This design lets the model maintain state through its context window, use tool outputs as implicit memory, and reason continuously without an external controller — favoring simplicity, stability, and tight feedback loops.

That single, often large, system prompt effectively governs the agent’s behavior. It’s a major determinant of performance, and its precision and scope directly influence how reliably the agent performs complex coding tasks.

To give developers more control, most coding agents expose rules files — user-defined instruction sets that are appended directly to the system prompt. These files let users configure behavior around security and compliance, programming style, architectural patterns, testing practices, and more by directly appending structured guidance to the base prompt.

Rules are a hot topic; it feels like I see people share their new coding agent rules on X every day. This is largely because writing effective rules is difficult. How do you know if a rule actually helps? How do you measure its impact or trust that it improves the agent’s reasoning, tool calling, data retrieval, etc.?

To answer this, we used Prompt Learning, our prompt-optimization algorithm, to automatically generate and refine rule files. We applied it to Cline, a powerful open-source coding agent, chosen for both its strong baseline performance and full transparency — including access to its prompts, tool descriptions, and system configuration — which make it ideal for optimization.

Evaluating Improvement with SWE Bench

We specifically used Prompt Learning to optimize Cline, through its rules. In order to track Cline’s improvement across optimizations, we used SWE Bench. **SWE-bench** is a benchmark designed to evaluate a system’s ability to automatically resolve real GitHub issues.

It contains 2,294 issue–pull request pairs from 12 popular Python repositories. Each system’s output is evaluated through unit test verification, using the post–pull request repository state as the reference solution. It’s a popular and widely adopted benchmark for coding automation tasks.

We specifically used SWE-bench lite, which includes 300 issue-pull request pairs.

The Optimization Loop

Initialize:

$$\begin{aligned} \mathcal{D} &= \text{Shuffle}(\text{SWE-bench}) \\ \mathcal{D}_{train} &= \mathcal{D}_{[:150]}, \quad \mathcal{D}_{test} = \mathcal{D}_{[150:]} \\ R^{(0)} &= \emptyset \quad (\text{initial ruleset for Cline}) \end{aligned}$$

For each optimization loop $k = 1 \dots K$:

1. Training phase

For each example $i \in \mathcal{D}_{train}$:

- Run **Cline** on repo **repo_i** using ruleset $R^{(k-1)}$
→ produces candidate patch $\widehat{\Delta}_i$
- Stage and diff changes:
`git add -A, git diff ⇒ cline_patch`
- Evaluate patch via SWE-bench unit tests
→ result $y_i \in \{\text{pass}, \text{fail}\}$
- Compute training accuracy
- Run **Eval LLM** (GPT-5) to analyze:

$$\text{EVAL_LLM}(\widehat{\Delta}_i, \text{ground truth, unit tests}) \Rightarrow \text{feedback } E_i$$

(explains *why* the patch failed and *why* the model made that decision)

Collect all training tuples:

$$\mathcal{B}_{train} = \{(\text{statement, ground truth, unit tests, } \widehat{\Delta}, y, E)\}$$

2. Ruleset update

Feed the batch into **Meta-Prompt LLM (GPT-5)**:

$$R^{(k)} = \text{META_PROMPT_LLM}(\text{Cline prompt}, R^{(k-1)}, \mathcal{B}_{train})$$

→ generates new optimized ruleset.

3. Testing phase

For each example $j \in \mathcal{D}_{test}$:

- Run Cline with updated ruleset $R^{(k)}$
- Generate patch, run tests, record pass/fail
- Compute test accuracy:

$$A_{test}^{(k)} = \frac{1}{|\mathcal{D}_{test}|} \sum_j \mathbf{1}[\text{pass}_j]$$

Initialize - Train/Test

We split SWE-bench Lite into a 50/50 train-test split with 150 examples each – enough data to drive meaningful optimization but also enough data to have sufficient test coverage for reliable measurement of improvement.

RUNNING CLINE

Cline was booted up on individual servers. We used Cline’s standalone server implementation, which allows us to send requests to Cline servers through GRPC, pointing them at the SWE bench repositories.

Cline was set to ACT Mode - where it is given full access to read and edit code files as it deems fit. We configured each Cline run with the current ruleset by funneling it through `.clinerules`

Ruleset is initialized to empty, and optimized after each training phase.

```
your-project/
├── .clinerules
├── src/
├── docs/
└── ...
```

After giving Cline ample time to run, we used git diff to generate `cline_patch`, or the patch it produced.

TESTING CLINE

We then use SWE-bench package to run the unit tests for each SWE bench row after Cline’s edits, which tells us whether Cline was correct/incorrect.

Evaluating Cline - Generating Feedback/Evals

Evals are a crucial part of the optimization loop - they serve as powerful feedback channels for our meta prompt LLM to generate an optimized prompt. The stronger the evals are, the stronger the optimization should be.

We used GPT-5 to compare `cline_patch` with `patch` (ground truth patch) from SWE bench, asking it to tell us WHY `cline_patch` was right/wrong, and WHY the model may have taken the direction it did.

You are an expert software engineer, tasked with reviewing a coding agent.

You are given the following information:

- `problem_statement`: the problem statement
- `cline_patch`: a patch generated by the coding agent, which is supposed to fix the problem.
- `patch`: a ground truth solution/patch to the problem
- `test_patch`: a test patch that the coding agent's output should pass, which directly addresses the issue in the problem statement
- `pass_or_fail`: either "pass" or "fail" indicating whether the coding agent's code changes passed the unit tests.

Your task is to review the given information and determine if the coding agent's output is correct, and why.

- Evaluate correctness based on the following factors:
- Whether `cline_patch` fixes the problem.
 - Whether `test_patch` would pass after applying `cline_patch`.
 - Whether the coding agent is taking the correct approach to solve the problem.

You must synthesize why the coding agent's output is correct or incorrect. Try to reason about the coding agent's approach, and why the coding agent may have taken that approach.

```
problem_statement: {problem_statement}
ground truth patch: {patch}
test patch: {test_patch}
coding agent patch: {cline_patch}
pass_or_fail: {pass_or_fail}
```

Return in the following JSON format:

```
"correctness": "correct" or "incorrect"
"explanation": "brief explanation of your reasoning: why/why not the coding agent's output is correct, and why the coding agent may have taken that approach."
"""
```

Meta Prompt

To complete the training phase, we use everything we have generated as input into a Meta Prompt LLM, prompting it to generate a new ruleset.

Even though we focused solely on ruleset optimization, we provided both Cline’s full prompt and its current ruleset to give the Meta Prompt LLM complete context. The `{examples}` dataset included input–output pairs, correctness labels, and detailed evals explaining why each case succeeded or failed.

Crucially, the Meta Prompt LLM was instructed to generate broadly applicable rules that strengthen overall agent performance – not rules that overfit to specific training examples.

Test Phase

After the optimized ruleset is generated, we boot up Cline servers on the 150 test examples, testing out the optimized ruleset, and generating an accuracy metric (% of test examples where Cline’s patch passed all unit tests).

You are an expert in coding agent prompt optimization.
Your task is to improve the overall ruleset that guides the coding agent.

Process:

1. Review the baseline prompt, the current ruleset, examples, and evaluation feedback.
2. Identify high-level shortcomings in both the baseline prompt and the ruleset – look for missing guidance, unclear constraints, or opportunities to strengthen general behavior.
3. Propose edits that make the ruleset more robust and broadly applicable, not just tailored to the given examples.

BELOW IS THE ORIGINAL BASELINE PROMPT WITH STATIC RULESET
***** start prompt *****

```
{baseline_prompt}
***** end prompt *****
```

BELOW IS THE CURRENT DYNAMIC RULESET (CHANGE THESE OR ADD NEW RULES)
***** start ruleset *****

```
{ruleset}
***** end ruleset *****
```

BELOW ARE THE EXAMPLES USING THE ABOVE PROMPT
***** start example data *****

```
{examples}
***** end example data *****
```

FINAL INSTRUCTIONS
Iterate on the current ruleset. You may:

- Add new rules
- Remove rules
- Edit or strengthen existing rules
- Change the ruleset entirely, if need be

The goal is to produce an optimized dynamic ruleset that generalizes, improves reliability, and makes the coding agent stronger across diverse cases – not rules that only patch the specific examples above.
The rules in the baseline prompt are static, don't change them. Only work on the additional dynamic ruleset.

Please make sure to not add any rules that:

- ask for user input
- use ask_followup_question

Cline is to perform without user input at any step of the process.

Return just the final, revised, dynamic ruleset in bullet points. Do not include any other text.

New ruleset:

Final Ruleset

Here are some examples of rules that got added to the ruleset. The final optimized ruleset tends to contain anywhere from 20-50 rules.

- ✓

Ensure every code modification strictly preserves correctness, minimality of change, and robustly handles edge/corner cases related to the problem statement—even in complex, inherited, or nested code structures.
- ✓

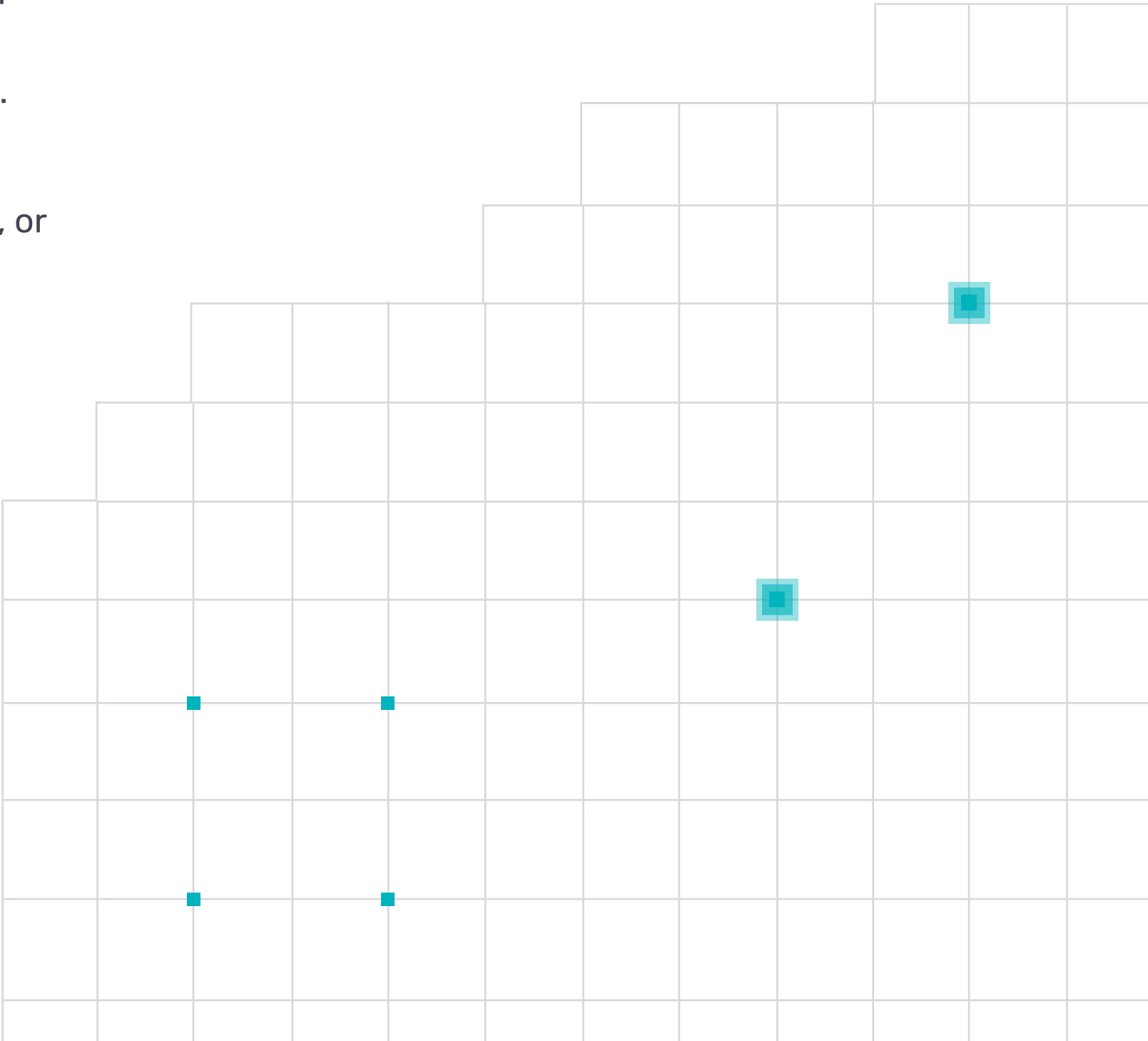
Avoid blanket or “quick fix” solutions that might hide errors or unintentionally discard critical information; always strive to diagnose and address root-causes, not merely symptoms or side-effects.
- ✓

Where input normalization is necessary—for types, iterables, containers, or input shapes—do so only in a way that preserves API contracts, allows for extensibility, and maintains invariance across all supported data types, including Python built-ins and major library types.
- ✓

All error/warning messages, exceptions, and documentation updates must be technically accurate, actionable, match the conventions of the host codebase, and be kept fully in sync with new or changed behavior.
- ✓

Backwards and forwards compatibility: Changes must account for code used in diverse environments (e.g., different Python versions, framework/ORM versions, or platforms), and leverage feature detection where possible to avoid breaking downstream or legacy code.
- ✓

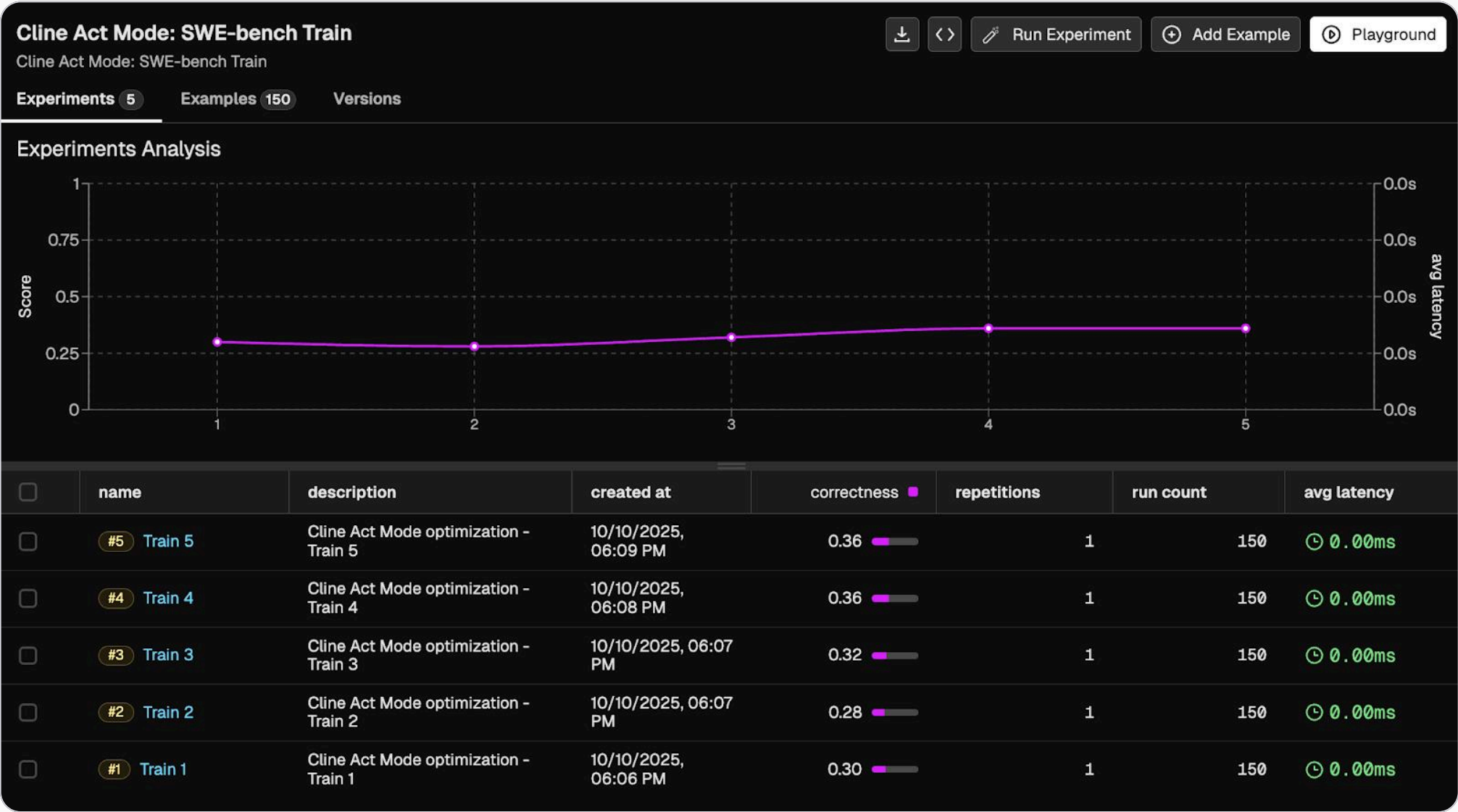
Refactorings and bugfixes must never silently discard, mask, or change user data, hooks, plugin registrations, or extension points; if a migration or transformation is required, ensure it is invertible where possible and preserve optional hooks or entry points.



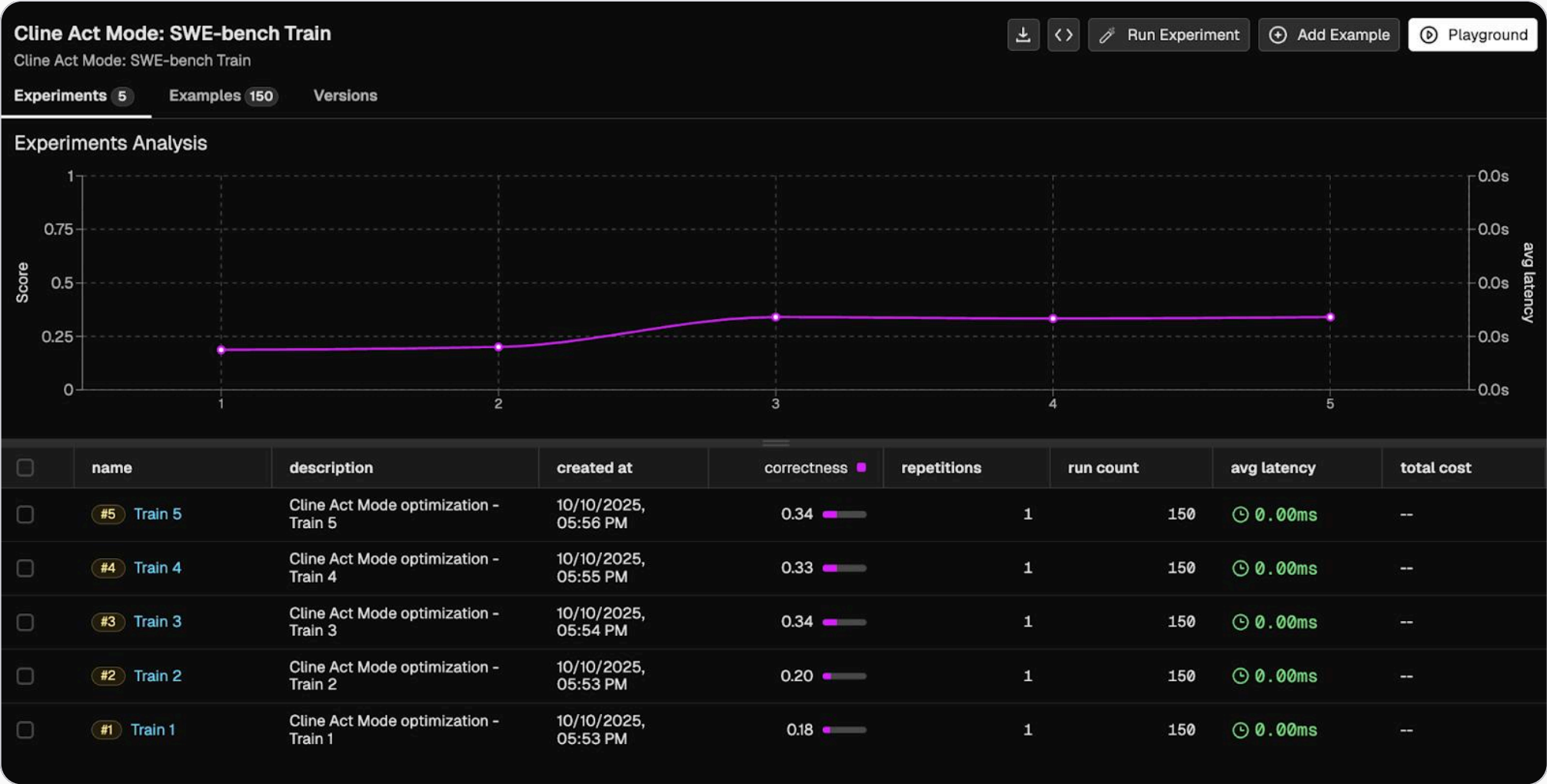
Results Overview

We used Arize Phoenix Experiments to track our Cline runs at each level of optimization. Phoenix Experiments are a great way to test out your LLM applications over datasets, run evals on those experiments, and track your experiments in one central location over time.

CLAUDE SONNET 4-5



GPT 4.1



Detailed Performance Metrics

Sonnet 4-5 is already near the ceiling of coding-agent performance on SWE-bench – a state-of-the-art model that leaves little headroom for further gains. GPT-4.1, on the other hand, had more room to improve.

Through ruleset optimization alone, Prompt Learning was able to close much of that gap, bringing GPT-4.1’s performance close to Sonnet-level accuracy without any model retraining, architectural changes, or additional tools.

This result underscores the power of prompt optimization: major performance gains can be achieved purely through better instructions. It demonstrates that improving the prompt – not the model – can deliver measurable, system-level advancements. And perhaps most impressively, these improvements emerged within just two optimization loops, showing how data-efficient and fast Prompt Learning can be compared to traditional reinforcement-learning-based methods.

CLAUDE SONNET 4-5

Optimization Loop	Train Accuracy	Train Delta	Test Accuracy	Test Delta
0	0.3000	-	0.3533	-
1	0.2800	-0.0200	0.3333	-0.0200
2	0.3200	+0.0200	0.3533	0.0000
3	0.3600	+0.0600	0.3600	+0.0067
4	0.3600	+0.0600	0.3600	+0.0067

Sonnet 4-5 saw a 6% boost in training accuracy and a 0.67% gain in test accuracy. As one of the strongest coding models today—used by most SWE-bench leaderboard leaders—it’s likely already saturated within Cline’s setup, leaving limited room for improvement. Still, the 6% training gain shows some success in eval based prompt optimization, even if ground truths are provided.

GPT 4.1

Optimization Loop	Train Accuracy	Train Delta	Test Accuracy	Test Delta
0	0.1867	-	0.1733	-
1	0.2000	+0.0133	0.2133	+0.0400
2	0.3400	+0.1533	0.3133	+0.1400
3	0.3333	+0.1466	0.2800	+0.1067
4	0.3400	+0.1533	0.3000	+0.1267

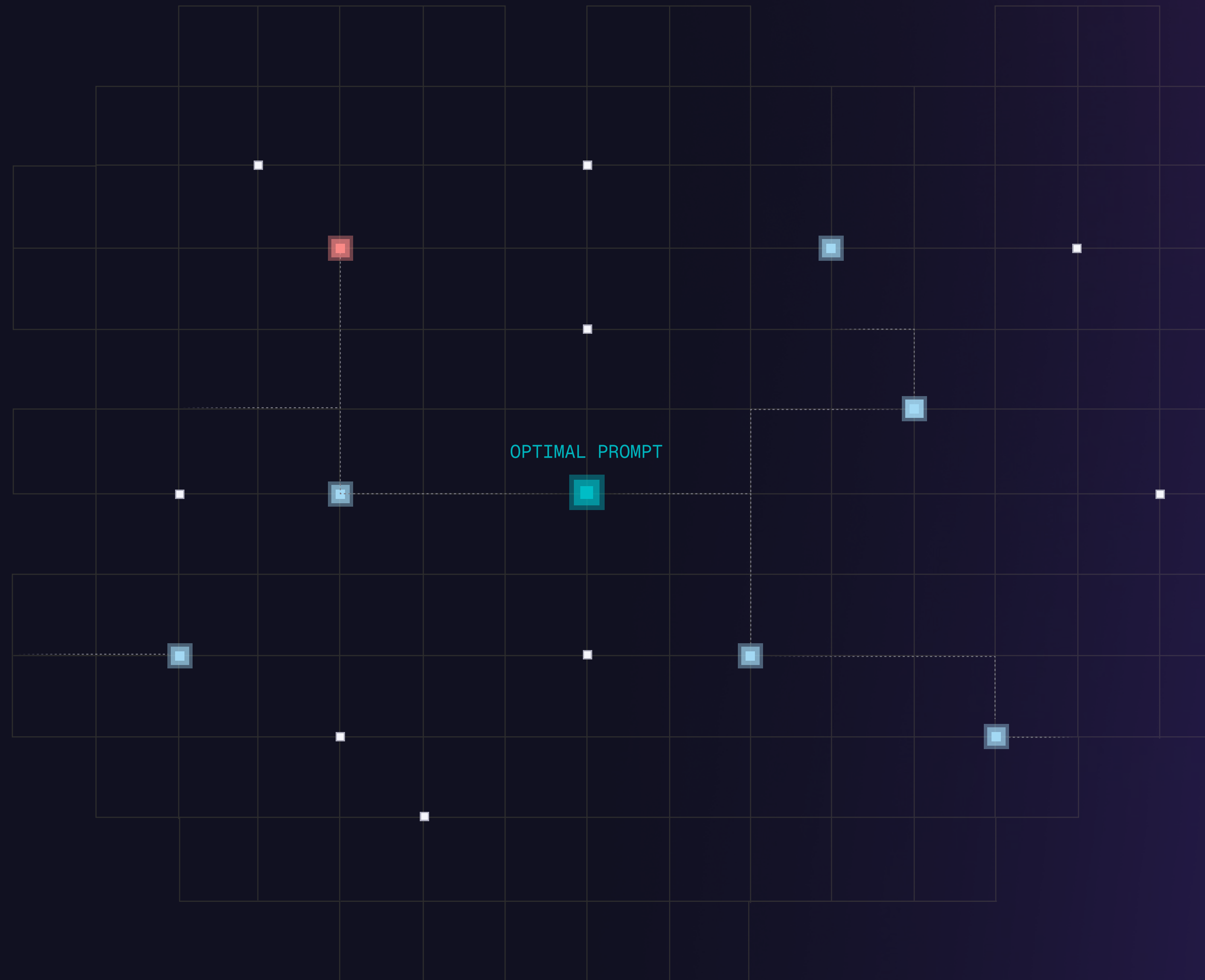
GPT-4.1 saw significant gains in both training and test accuracy, reaching near Sonnet 4-5 performance through ruleset optimization alone – and notably, it achieved this in just two optimization loops.

Prompt Learning in Arize AX

With Prompt Optimization tasks in Arize AX, you can optimize prompts in a few clicks using human or automated feedback loops and versioned releases. This brings prompt engineering into a reproducible, CI-friendly workflow instead of trial-and-error. Each auto-generated version of the prompt is committed to the Prompt Hub, so you can A/B test different versions and safely promote the winner.

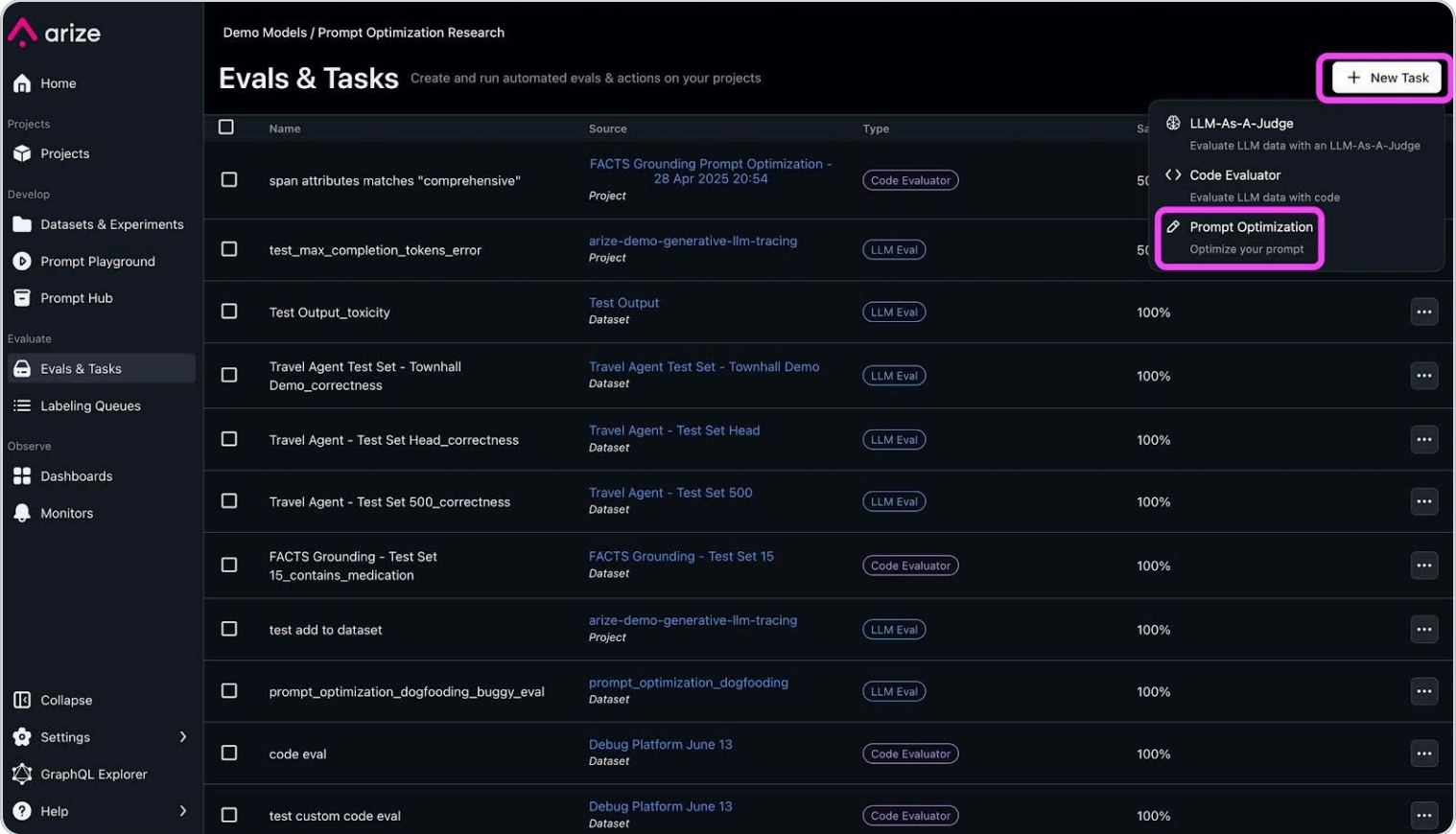
KEY FEATURES

- Auto-generate the best prompt from your labeled **dataset**
- Promote the best auto-generated prompt in the **prompt hub** as the production version
- Evaluate the auto-generated prompt against the original using a side-by-side comparison on the **experiments** page

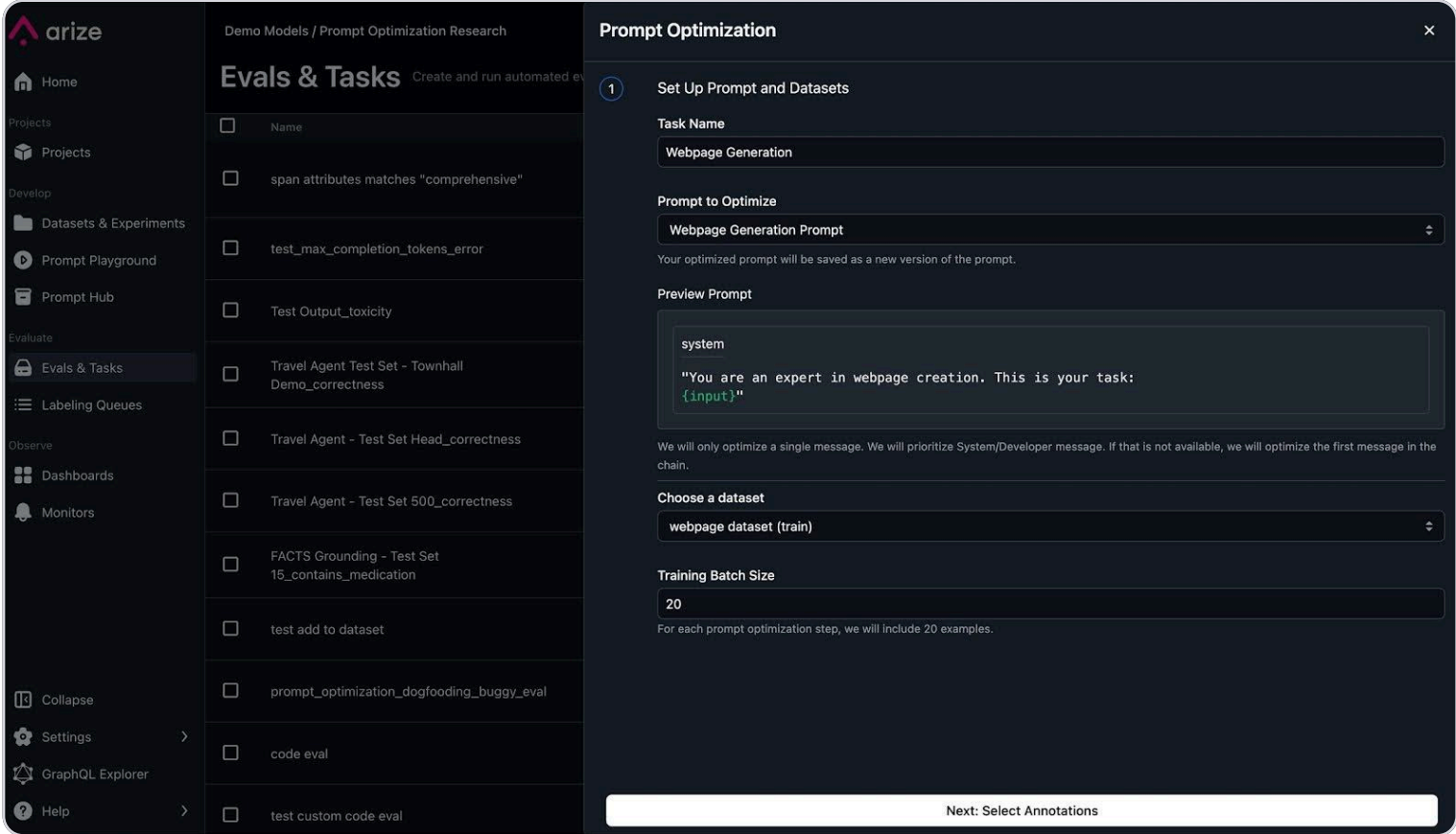


Quick Setup

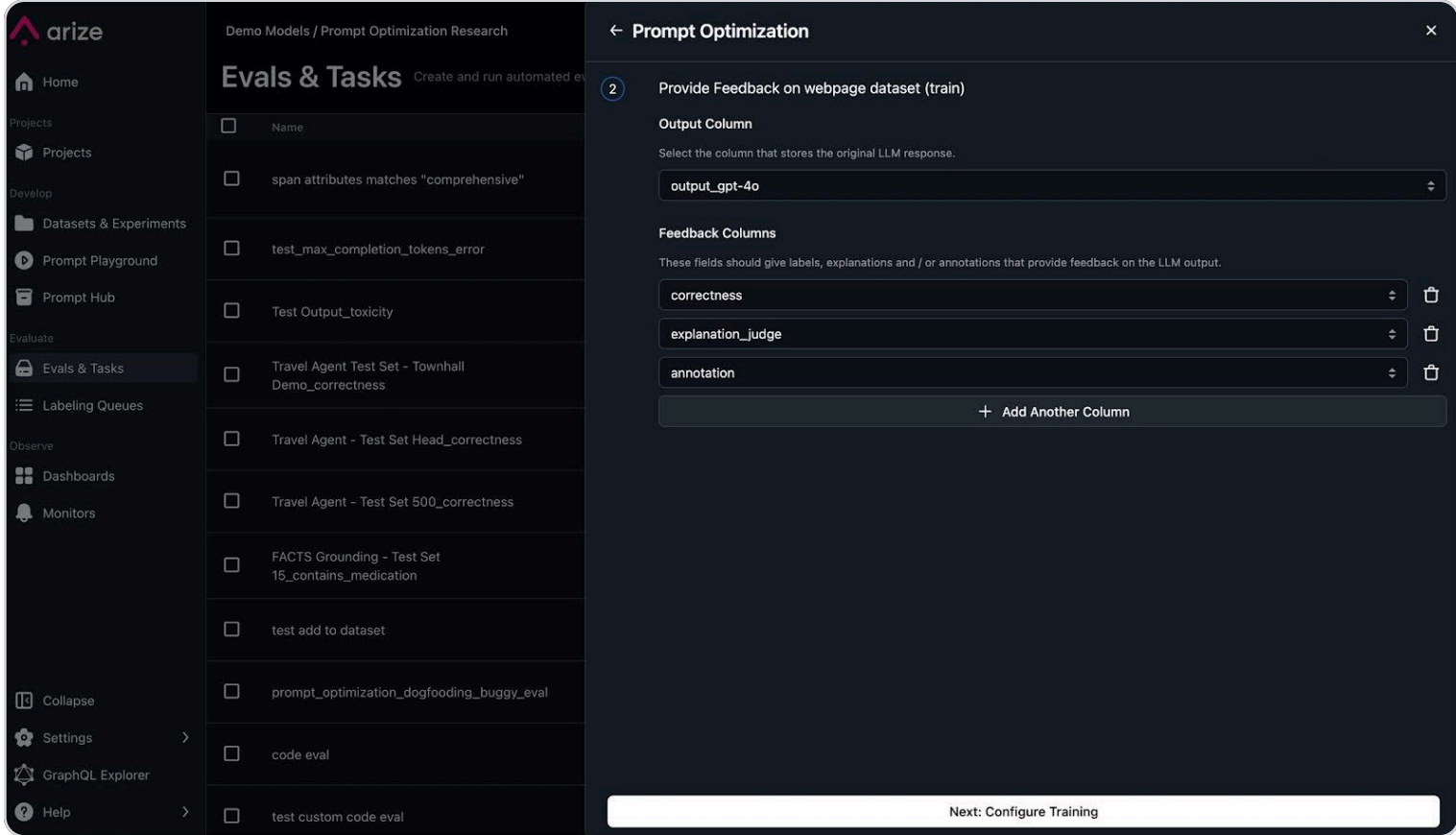
Prompt optimization uses a similar task builder as the one used for Online Evals, so you can get started quickly. Pick the prompt you want to optimize from **Prompt Hub**. If you don't have one yet, click **Create New Prompt** to add it. Then, choose a **Training Dataset** and set a **Batch Size** (defaults to 10). Finally, select one or more **Feedback Columns** that contain evaluation signals, which can be labels from human annotators or LLM-as-a-Judge evaluators, plus optional explanations.



From the Evals & Tasks page, click "+ New Task" and select "Prompt Optimization".



Select a prompt from the prompt hub, then choose a training dataset that contains LLM input, output and feedback labels to guide the optimization process.

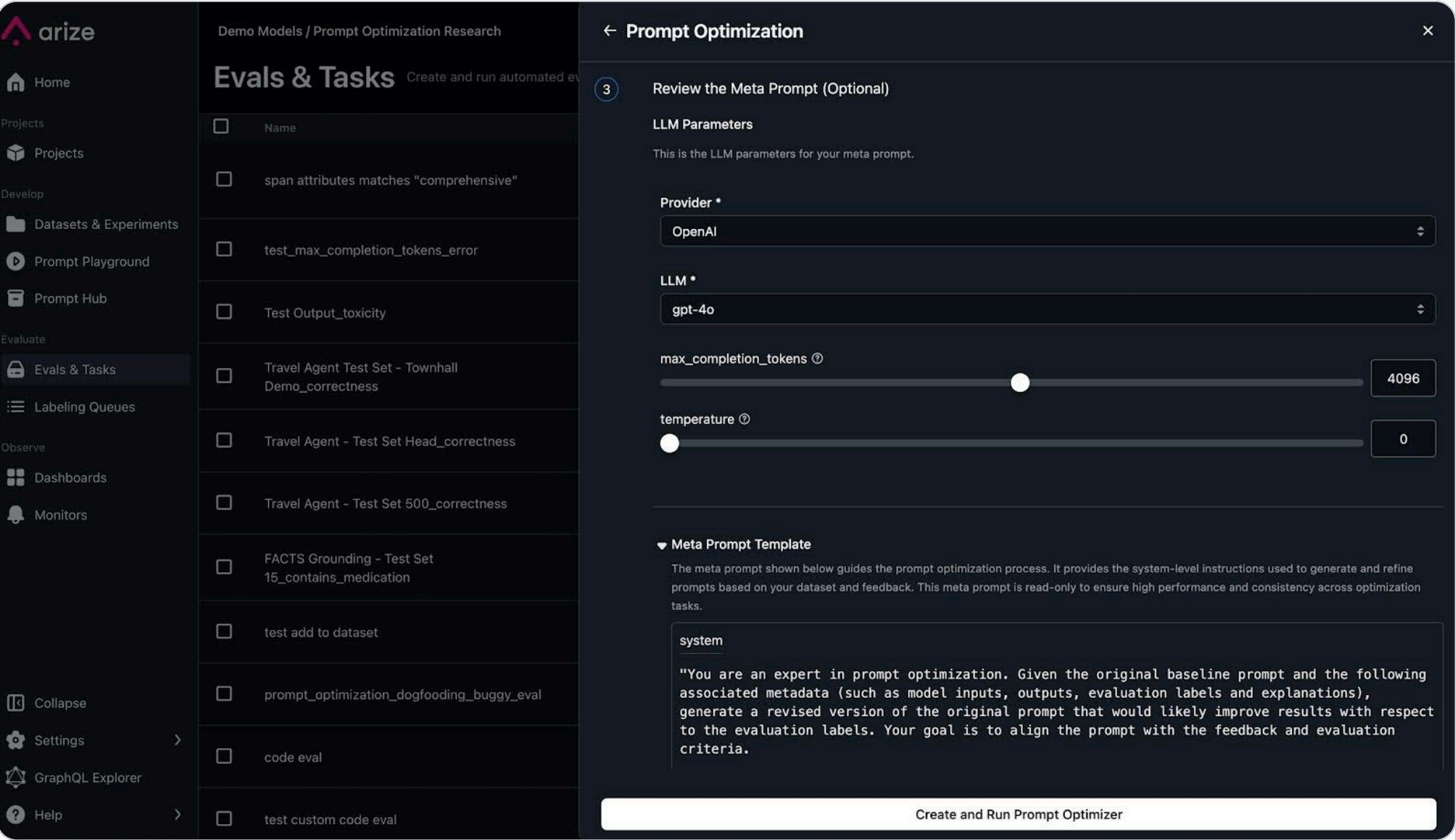


Select your LLM "output" and feedback columns, which contain evaluation signals. You can include multiple feedback fields if you're optimizing for more than one metric (e.g., "correctness" and "toxicity").

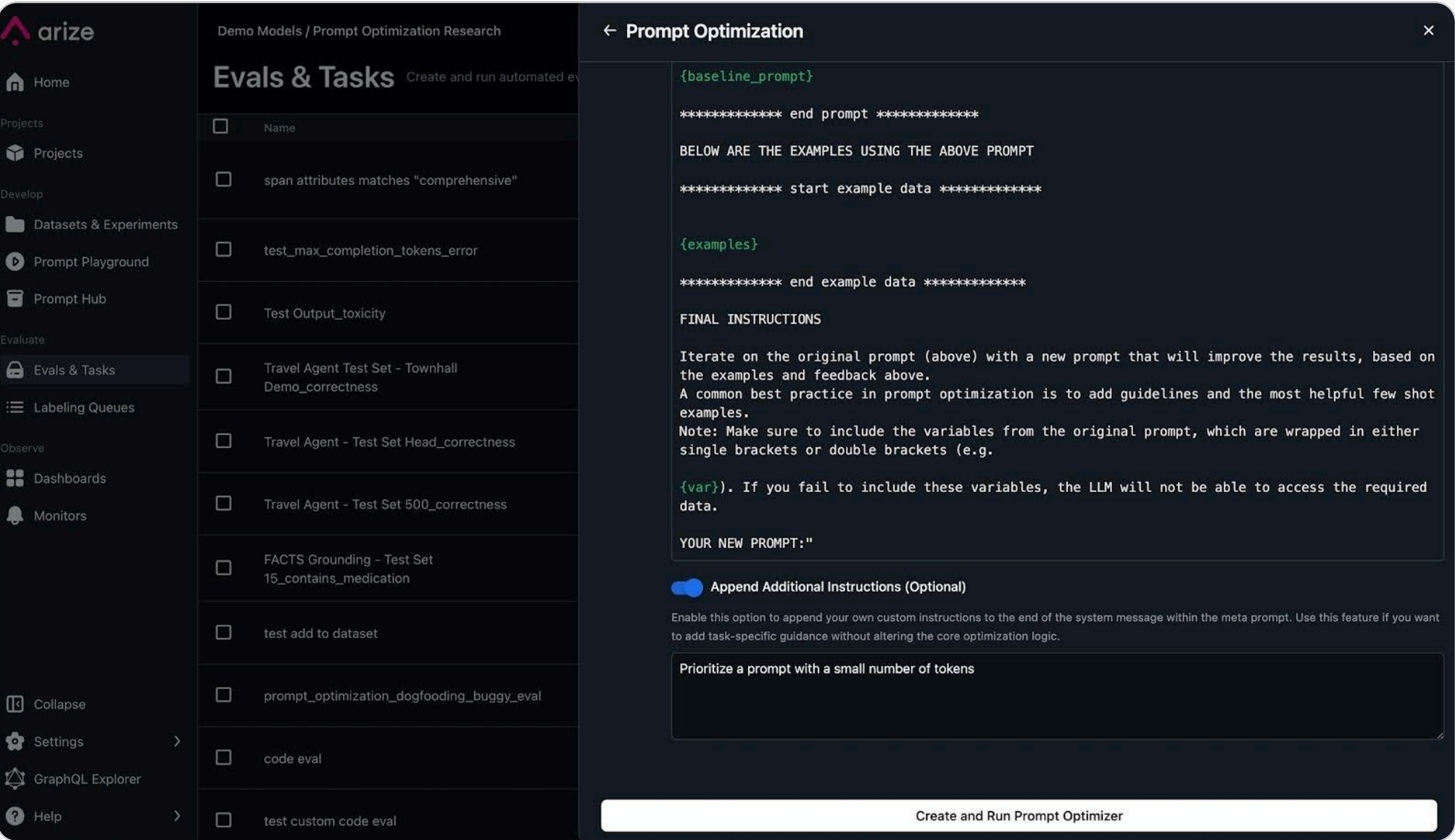
Customize Your Meta Prompt

Before launching, Arize shows you the full meta prompt that will steer optimization. This meta prompt has been tested and optimized internally for a variety of customer applications.

To customize this, you can add optional **Advanced Instructions**, such as “Use ≤150 tokens” or domain-specific constraints. You can also specify the LLM model and parameters, such as the temperature setting.



Select your LLM model and parameters.



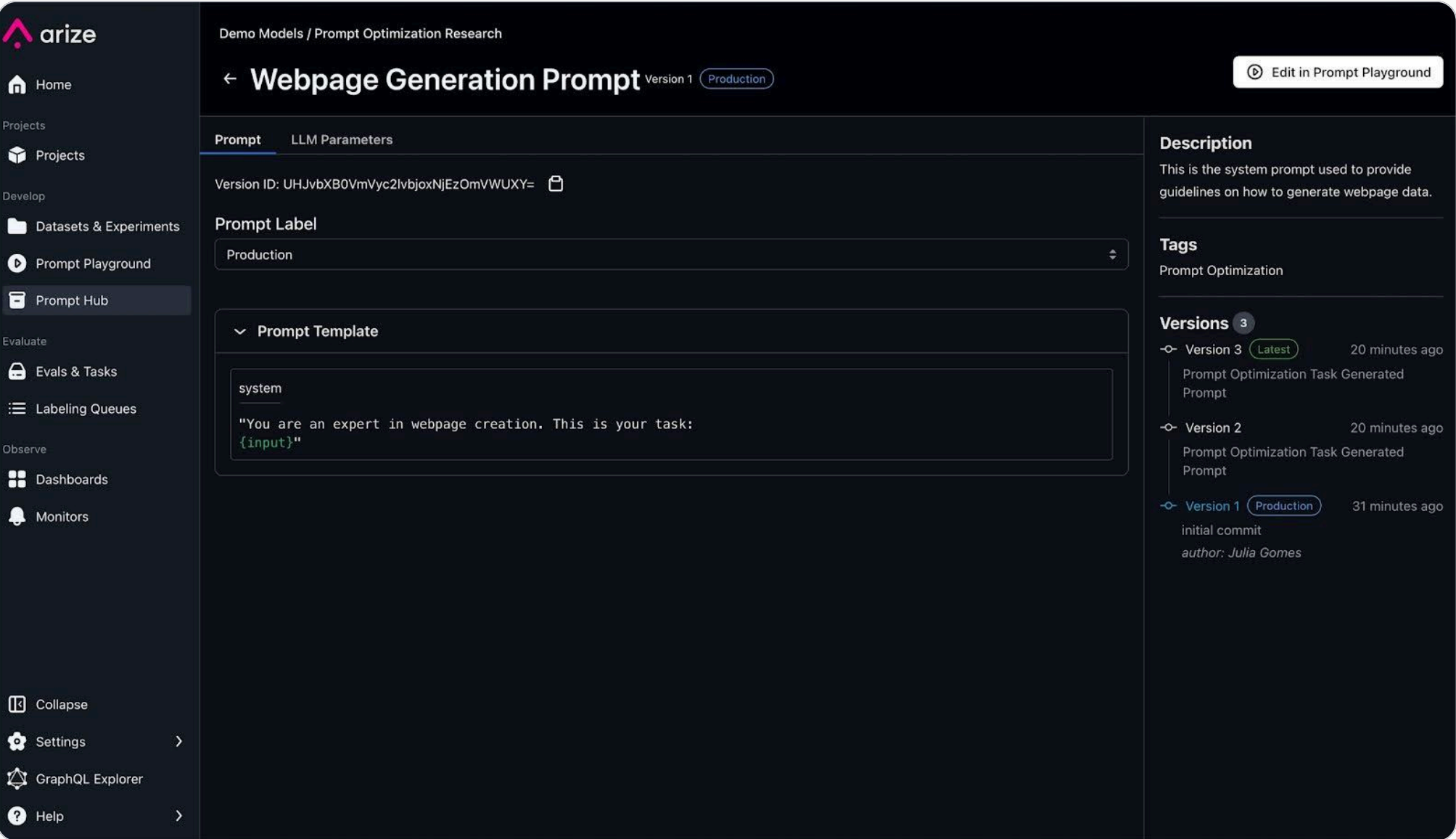
Append custom instructions to the meta prompt template.

Track Optimization Process

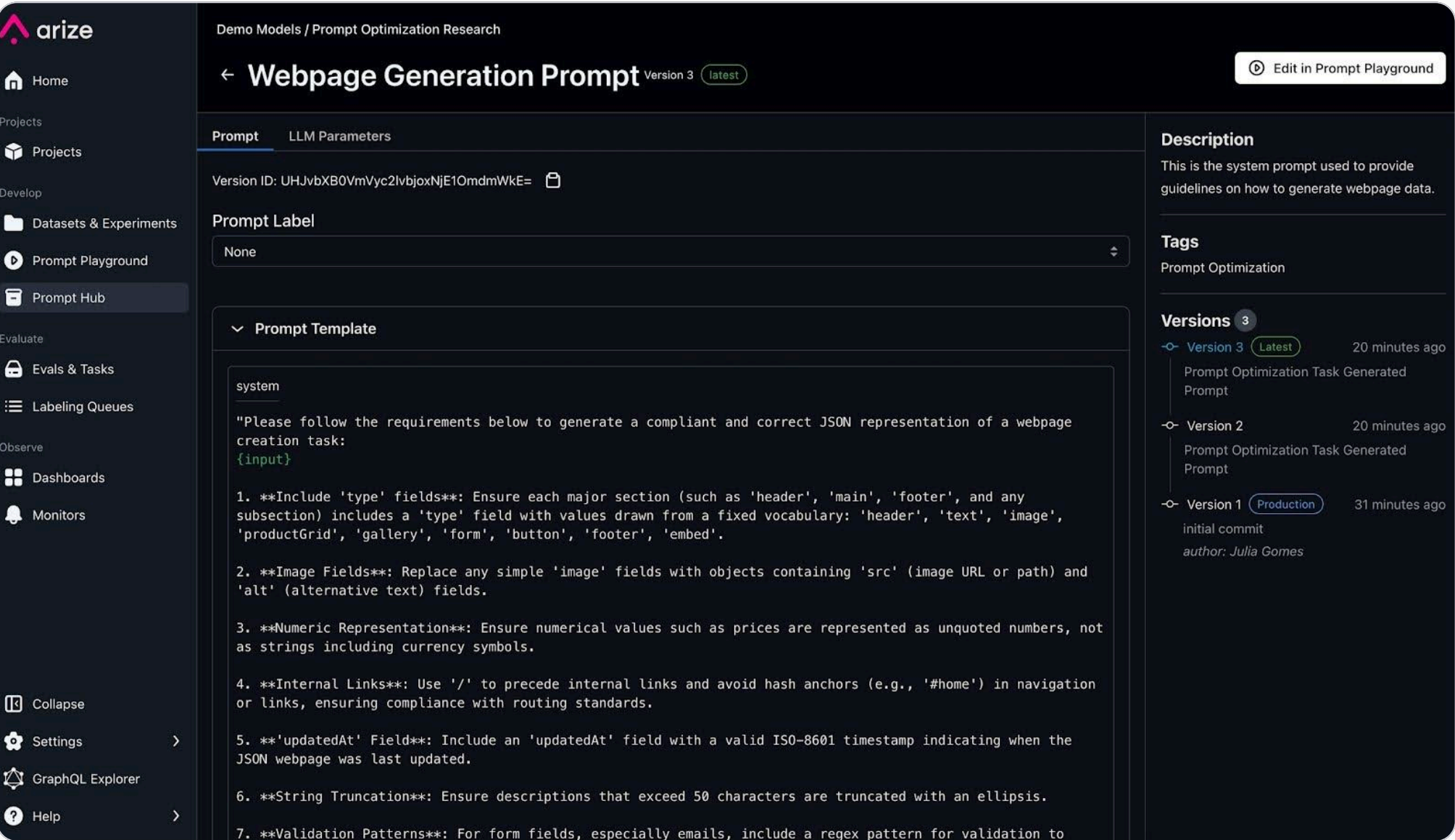
The **Task Logs** page shows each batch as it runs, including what prompt was used, which examples were evaluated, and what feedback was generated.

After each batch, the optimizer proposes a new prompt candidate, which is automatically saved as a **new version** in the Prompt Hub.

To review changes, go to the prompt’s page in Prompt Hub and review the list of Versions.



Simple initial prompt before optimization.

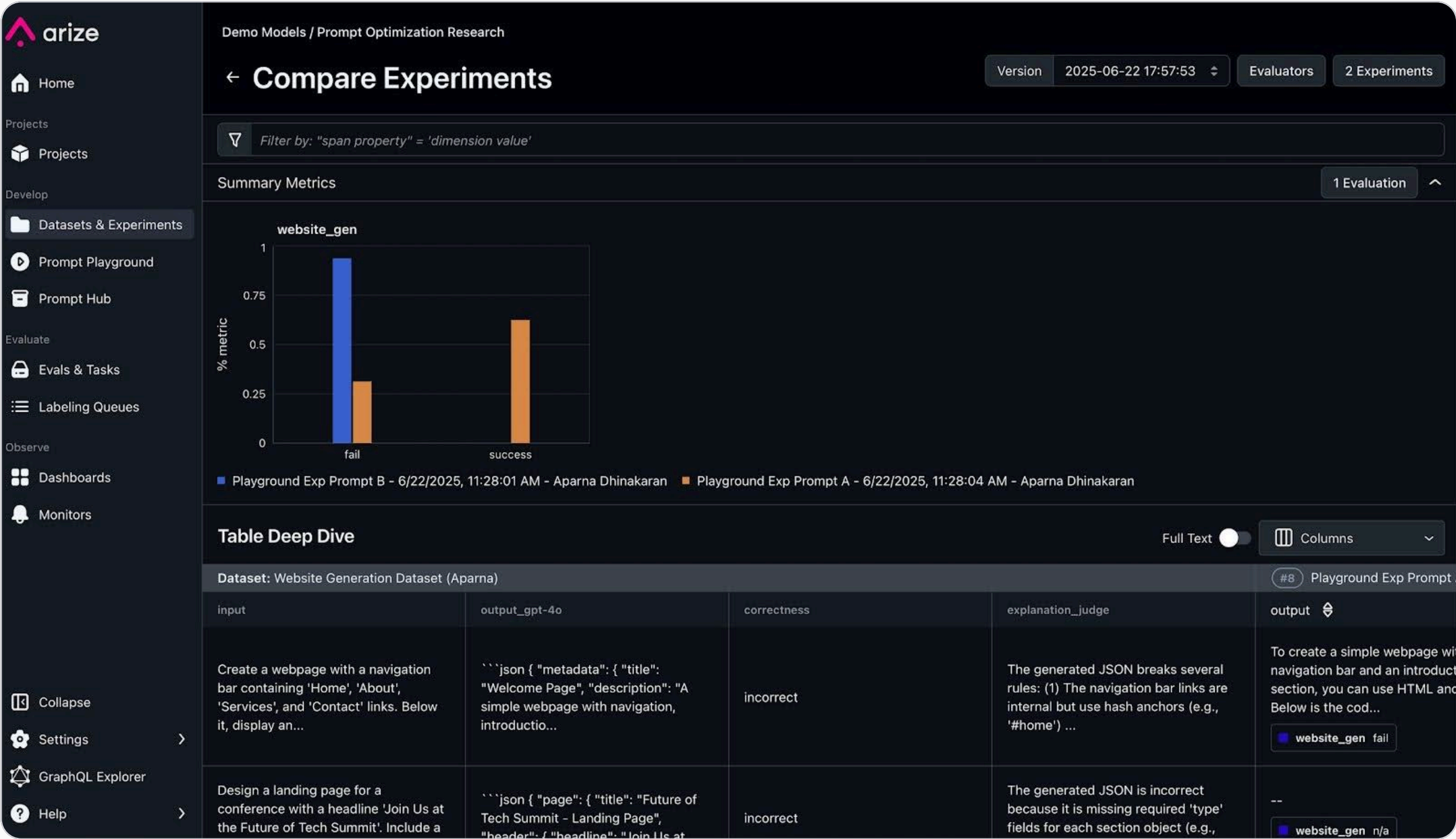


Final prompt after two prompt optimization iterations.

Compare the Optimized Prompt Against the Original

From Prompt Hub, you can compare the final optimized prompt to the original by **launching an experiment in the Playground**.

Select an evaluation dataset, choose the two prompt versions, and review metrics side-by-side, including both high-level summary metrics and example-level outputs.

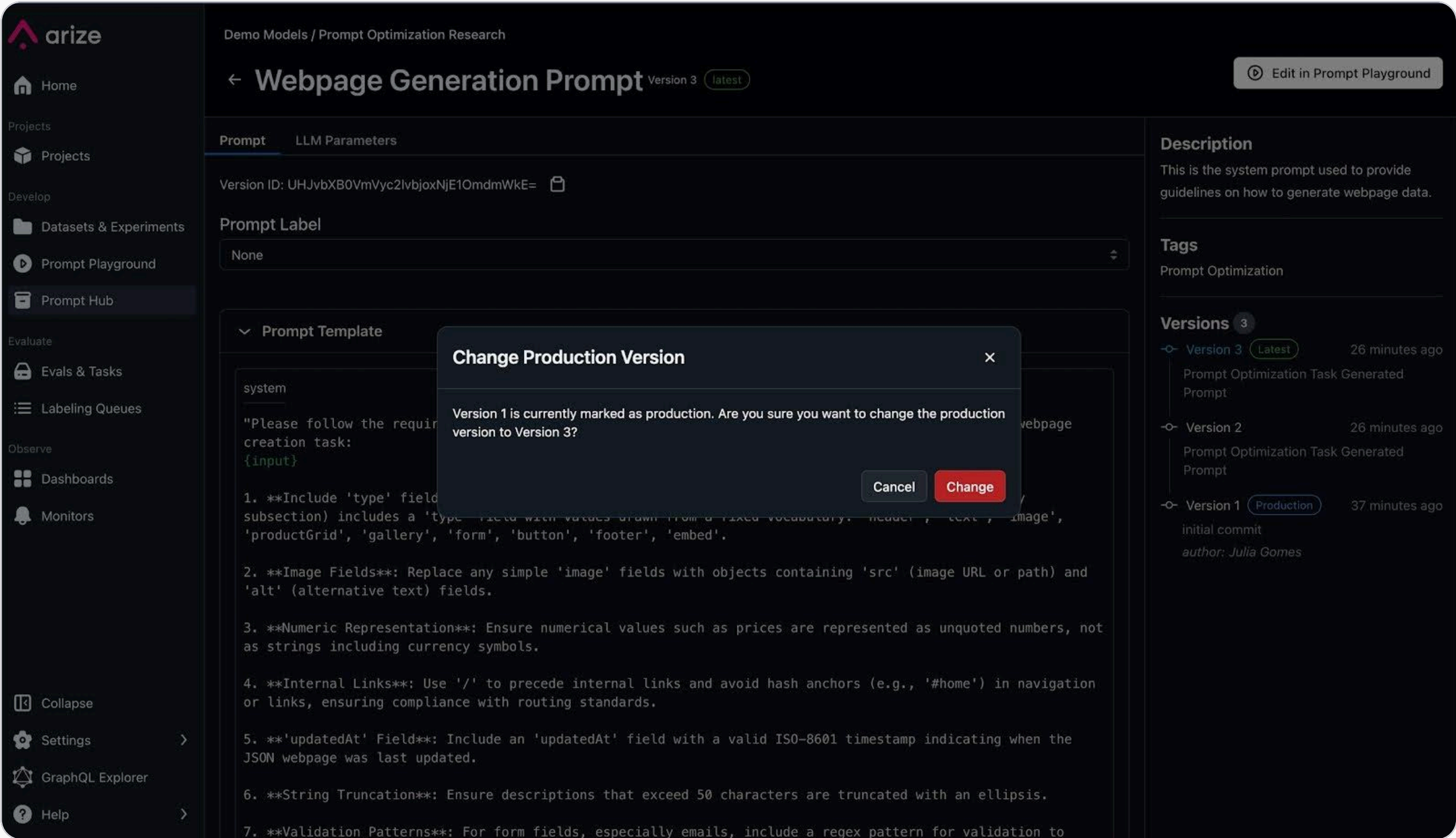


Save an experiment with your optimized prompt in the playground and compare evaluation metrics against the original prompt.

Deploy with Human-in-the-Loop Oversight

Once you’ve reviewed the results, tag the winning prompt version as **Production** in Prompt Hub.

The Prompt Hub SDK will automatically pull the latest version with the "production" tag at inference time, ensuring a human-in-the-loop that enables you to ship with confidence.



After reviewing the new prompt and experiment results, label the auto-generated prompt with the "production" tag.



Prompt Learning SDK

Arize AI also offers an open source prompt learning SDK to optimize your prompts in code

Full guide to getting setup here





Book a Demo

See prompt learning in the platform and explore how it can benefit your team.

Book a Demo