From Guesswork to Confidence with Evals:

A How-To Guide to Evaluations in AI Development

FOCUSED

/LAY OF THE LAND

Table Of Contents

/INTRODUCTION

Why Evals Matter for AI Development

Quality software is driven by strong testing practices.

High quality tests give development teams confidence that their code is working as designed, clear and self-documenting definitions for expected system behavior at each layer of the application, and the freedom to iterate quickly without worrying that their changes will cause unintended negative consequences for their production users.

Evals are to AI development what tests are to traditional software development. They define what success looks like for your AI application, validate that your success criteria are being satisfied, and allow you to track how changes to your application affect the quality of your results over time.

Unlike with traditional software, developing **LLM-powered applications** means dealing with the inherently nondeterministic results that come from LLM invocations. This inherent unpredictability presents unique challenges and makes continuous, systematic evaluation of system behavior and performance critically important to the success of your product. Without a robust eval set, AI development becomes a game of guesswork and whack-a-mole that rarely gets past the demo phase.

Evaluation Driven Development (EDD) is the anchor you can use to steady yourself in a sea of non-deterministic LLM calls. Committing to EDD means grabbing hold of that anchor and building a ship yourself, eval by eval, that will carry you over the abyss of abandoned prototypes and wasted investment to the land of production value. Beyond building confidence, EDD will unlock a deeper understanding of the performance of your application. It will yield the data you need to make informed decisions when presented with cost, latency, and quality tradeoffs so that you can feel comfortable in the knowledge that you are maximizing the value your application delivers with every deployment.

What you'll find in this guide

This guide will take you from uncertainty to confidence by showing you how to think about evals, describe the tools that LangSmith provides for evaluation, and get you on your way to creating a comprehensive evaluation framework for your AI applications.

We'll focus on practical approaches using LangSmith, though the principles apply broadly to any AI development workflow.

/CHAPTER ONE

Understanding the Evals Landscape

/CHAPTER ONE

Understanding the Evals Landscape

What are Evals?

Evaluation is the process of assessing the performance and effectiveness of your LLM-powered applications. Evals are the tests that enable that assessment, tools that help you to systematically measure and track key metrics by comparing real results from your application against pre-defined desired results and performance benchmarks.

They help you answer questions like:

- Is my application producing accurate outputs?
- Are the outputs my application produces relevant to the task at hand and the user input?
- How are response quality and latency impacted when I modify prompts or switch models
- Where are the most common failure modes in my system?
- Am I ready to deploy this to production?
- When and why did my customer service chatbot start telling my customers that our retail stores are open at 3am?

Concepts, Tools, and Terms

LangSmith

LangSmith is an observability and evaluation platform from LangChain. It allows you to monitor your application in the real world and to design, manage, and run evaluations against your application to track how it is performing as you iterate.

LangSmith is framework agnostic so you can use it to monitor and evaluate your AI application even if you didn't use LangChain and/or LangGraph to build it (though using LangChain or LangGraph provides some nice conveniences with integration). It is Focused's preferred platform for monitoring and evaluating AI applications and it is an indispensable tool if you're serious about getting to production.

Datasets

Datasets are collections of examples that can be used to test your application. Each example typically contains a user input and a reference output that shows what a good quality response should look like for the user input.

Datasets should be representative of real world scenarios and should contain both examples for your expected happy path and examples that force the evaluation of edge cases and non-standard inputs.

Datasets can be manually curated, synthetically generated, or captured from real production traces or logs. LangSmith makes it extremely easy to create and manage datasets using any one of these methods or a combination of all three either programmatically or through the LangSmith user interface.

Concepts, Tools, and Terms *(continued)*

Metrics

Metrics are quantifiable measures of performance and/or response quality from your application. When you enable tracing on your application with LangSmith you'll get some simple, but important metrics on every call to your application out of the box without any further configuration - latency and token usage.

The metrics you capture during evaluation are totally up to you and may vary depending on the type of application you want to build. Evaluation metrics should be derived from your success criteria and should allow you to easily understand how close your application is to meeting that criteria.

Evaluators

Evaluators are functions that measure and capture metrics on your application's outputs. When you run an experiment against one of your datasets, your evaluators analyze the outputs produced by your application and produce metrics that describe how your application performed.

There are three main types of evaluators:

1	**Custom Code Evaluators**	Simple, deterministic functions that validate some concrete aspect of your outputs like checking that the structured output of an LLM call is valid JSON and contains some number of required fields.
2	**LLM-as-Judge Evaluators**	Functions that use LLMs to allow for more flexible, dynamic judgement of the quality of your application's output based on an evaluation prompt and the reference outputs defined in your dataset examples.
3	**Composite Evaluators**	These allow you to combine two or more evaluators of different types into a single composite score using either a weighted average or sum

Experiments

An Experiment represents the execution of a set of evals. When you run an Experiment against a Dataset with LangSmith, each of the inputs from your Dataset examples will be run through your application, your evaluators will analyze the outputs produced by your application, capture the metrics you've defined, and publish those metrics to LangSmith.

LangSmith allows you to view the results of your Experiments and to compare results and metrics across different Experiments to understand how and why your application performed better or worse in one experiment vs another.

/CHAPTER TWO

Building Your Foundation - Datasets

/CHAPTER TWO

Building Your Foundation - Datasets

Creating Effective Datasets

Your evaluation is only as good as your test data. Here's how to build datasets that *actually* drive results.

Start Early, Start Small

When you're starting out it's more important to establish a practice of evaluation than to worry about exhaustive coverage. Iteration is key and your evals will evolve over time so laying the groundwork for continuous evaluation is more valuable at the beginning than generating thousands of mediocre examples.

Start with 10-30 high-quality, manually curated examples. Each example should represent a realistic use case or edge case that you care about. Think about your success criteria and design examples that will help to determine how your application performs against that criteria.

Even if you start with just 5 examples, having a few high quality evals can mean the difference between laying the foundation for success and getting lost in the weeds down the road wondering what caused your application to start returning irrelevant results. This gives you a foundation to build on.

The importance of getting started early with evals is something that we see reaffirmed over and over again at Focused.

Consult with Domain Experts

Getting started with some amount of evaluation is the most important thing, but how can we ensure that our Dataset examples are high quality? Developers don't always have the best insight into what is valuable to users or what a high quality response looks like, especially in situations where a high level of domain knowledge is needed, so we shouldn't expect them to have the final word on dataset design.

Consult with product managers, designers, or, if you have a small set of actual users who are willing to help improve your product, actual users to design example inputs that are valuable and represent common, real-world use cases.

Consult with domain experts to design and iterate on reference outputs for your examples. These team members usually have the best idea of what a valuable, accurate response looks like. The more we can leverage that expertise to guide the behavior of our application, the better it will perform.

If you don't have these experts available to you, it is still important to start evaluating your application so don't let a lack of true expertise stop you from starting your eval journey. Do your best to determine what quality examples look like for your application and know that they can be improved over time as you gather more data.

Creating Effective Datasets *(continued)*

Don't Forget the Edge Cases

It is important to evaluate your application along the happy path, that is what we are ultimately optimizing for, but it would be a mistake to ignore edge cases and unexpected input.

Make sure to include examples in your datasets that will exercise these unhappy paths in your application. This is critical to validating how your application handles and recovers from errors as well as mitigating attack vectors and preventing malicious users from successfully "jailbreaking" your system prompts.

Iterate On Your Datasets

Once you have your first dataset(s) created, where do you go from there? Your datasets, metrics, and evals should be first class citizens in your SDLC. This means you should keep them top of mind and iterate on them as you iterate on your application.

Adding a new feature? Adopt the EDD mindset and design a dataset that will allow you to understand how that feature is performing before implementing the feature. You will be able to have an idea of how your first pass performs and have a foundation to build off of and drive success.

Continuously audit your datasets and examples. Periodically work with your experts to answer the question: does this example or this dataset provide value? If it doesn't, redesign or remove it. LangSmith's Annotation Queues can help with this.

Receive a bug report from an end-user or spot a non-optimal response in your production traces? Add it as an example to one of your datasets or create a new dataset for it. LangSmith makes this incredibly easy - if you have a trace of the interaction you can add it to a dataset for further evaluation with a couple of clicks in the UI.

Managing and Organizing Datasets in LangSmith

As your datasets grow in both size and number it will be useful to be aware of some strategies and some features available in LangSmith that will make managing and using your datasets easier.

Targeted Datasets

Datasets can be organized in many different ways in LangSmith. You can have one giant dataset full of examples that are tagged in different ways to allow for filtering the examples into subsets that cover different success criteria, you can create many different datasets targeted towards evaluating these different aspects of your application, or you can do some combination of both. It's common for AI developers to use a combination of both.

A commonly used method at Focused for organizing datasets is to create separate datasets for use in evaluating different success criteria. For example, if you are building a RAG chatbot you may care about the following criteria: retrieval relevance, answer relevance, groundedness, correctness and tone. It's common to create datasets that target each of these criteria separately. This keeps things organized and isolates concerns for each dataset and the evals that execute against each dataset.

You might also consider creating a dataset to capture all of the production traces that you find where a user either intentionally or unintentionally provided input that caused a negative experience or attempted prompt injection. Every time you find an example of this in your production traces, you add it to the dataset which allows you to hone in on these types of queries and build guardrails around this type of input that can be evaluated in isolation.

Version Control

Datasets in LangSmith are versioned for you. Any time you add, update or delete examples in a dataset, a new version of the dataset will be created. This allows you to track how your datasets change over time and revert to earlier versions if you find yourself several changes down a path that proves less than valuable.

Metadata

Dataset examples can be tagged with metadata. This metadata takes the form of simple key-value pairs and you can use them to tag and organize the examples within your dataset. When you're running evaluations, this metadata can be used to filter the examples used in the evaluation run.

Splits

Splits are another way to organize examples within a dataset. In the LangSmith UI you can do this by selecting one or more examples in your dataset and clicking the "Add to split" button. This logically partitions the examples of the dataset into different subgroups. Splits can also be used to filter examples when running evaluations and are commonly used to keep training and test examples separate when there is a dataset containing both.

Importing and Exporting Datasets

LangSmith allows you to import datasets from JSONL or CSV files and allows you to export datasets in the same formats. This can be very useful for sharing datasets for audit by experts who don't have access to your LangSmith account or for reusing general purpose datasets that might be applicable to multiple different AI applications you may have under development.

/CHAPTER THREE

Choosing and Implementing Evaluators

/CHAPTER THREE

Choosing and Implementing Evaluators

High quality datasets are the cornerstone of your eval process, but what do you do with them once you have them? Well, you use them to evaluate the performance of your application. That's where evaluators come in.

Evaluators are functions that analyze the response your application generates when given an example input in your dataset and determine if that response meets the criteria you've defined for success. There are three primary types of evaluators to consider in LangSmith: custom code evaluators, LLM-as-judge evaluators, and composite evaluators.

Custom Code Evaluators

Custom code evaluators are simple, deterministic functions that are useful for assessing concrete, closed-ended metrics about your application's outputs.

For example, say you have a node in your agent graph that should output valid JSON with a set of fields that are required by a subsequent node in your graph. This is a great time to reach for a code validator like the one shown below.

```python
def validate_json_response(output):
    """Check if output is valid JSON with required fields""" try:
        data = json.loads(output) required_fields = ['status', 'message',
        'data'] return all(field in data for field in required_fields)
    except:
        return False
```

Other examples where a custom code evaluator might be a good fit:

Length constraints
for evaluating the conciseness of a generated response

Keyword presence/absence
for evaluating that key content is mentioned in a generated response

Numerical range checks
for evaluating things like the number of retries used in an error recovery scenario or ensuring that the number of generated tool calls doesn't exceed a pre-defined limit

Successfully compiled
Whether or not generated code can be successfully compiled

LLM-as-Judge Evaluators

While custom code evaluators excel at closed-ended evaluations, LLM-as-Judge evaluators are what you should reach for when you need more open-ended evaluation. Because LLMs generate non-deterministic natural language responses, it can be extremely difficult or impossible to use deterministic tests to evaluate those responses. You can't validate that the output string is an exact match to your expected response if the generated response changes slightly every time you run the same input through your application

As the name implies, LLM-as-Judge evaluators address this problem by utilizing LLMs to judge non-deterministic output from your application. This allows you to evaluate your application in a more flexible, dynamic way. If your application's response always contains core information that is accurate and satisfies the desired output criteria, but varies slightly in how it is structured, you don't need to waste time worrying about those subtle differences.

ALLM-as-judge correctness evaluator could look something like this:

```python
# Evaluation result Schema class

class
CorrectnessGrade(TypedDict):
    explanation: Annotated[str, ..., "Explain your reasoning for the
    score"] correct: Annotated[bool, ..., "True if the answer is correct,
    False
otherwise."
]

correctness_instructions = """You are a teacher grading a
quiz.

You will be given a QUESTION, the GROUND TRUTH (correct) ANSWER, and the
STUDENT ANSWER.

Here is the grade criteria to follow: (1) Grade the student answers based ONLY
on their factual accuracy relative to the ground truth answer. (2) Ensure that
the student answer does not contain any conflicting statements. (3) It is OK
if the student answer contains more or less information than the ground truth
answer, as long as it is factually accurate relative to the ground truth
answer.

Correctness: A correctnessvalueofTruemeansthatthestudent'sanswermeetsallofthe
criteria. A correctnessvalueofFalsemeansthatthestudent'sanswerdoesnotmeetall
of thecriteria.

Explain your reasoning in a step-by-step manner to ensure your reasoning and
conclusion are correct.

Avoid simplystatingthecorrectanswerattheoutset."""

correctness_llm = ChatAnthropic(model="claude-3-5-sonnet-20240620",
temperature=0).with_structured_output(
    CorrectnessGrade,method="json_schema",strict=True
)

def correctness(inputs: dict, outputs: dict, reference_outputs: dict) -> bool:
    """An evaluator for RAG answer accuracy"""
    answers = f"""\
QUESTION: {inputs['question']} GROUND TRUTH
ANSWER: {reference_outputs['answer']} STUDENT
ANSWER: {outputs['answer']}"""

    grade = correctness_llm.invoke(
        [
            {"role": "system", "content": correctness_instructions},
            {"role": "user", "content": answers},
        ]
    )
    returngrade["correct"]
```

Use LLMs to evaluate subjective qualities that can't be captured easily with programmatic rules like:

Relevance
Does the answer address the question?

Coherence
Is the response logically consistent?

Groundedness
Is the answer supported by provided context?

Helpfulness
Does the response solve the user's problem?

Hallucination
Does the response contain hallucinated facts?

LangSmith conveniently provides pre-built LLM-as-judge evaluators out of the box for you to get started with, including evaluators for correctness, conciseness, hallucination, and code generation. These pre-built evaluators are a great place to start, but you can also define your own LLM-as-judge evaluators to measure the metrics you care about most.

LLM-as-Judge Evaluators *(continued)*

LLMs Judging LLMs

You might be thinking: "Wait, you want me to use LLMs to evaluate my LLMs? I thought LLMs were non-deterministic. How can I trust them to be good evaluators? Who evaluates the evaluators?"

You would be correct to be leery of using LLMs to judge your application's LLM-generated responses, but with appropriate caution and guard rails they can be excellent tools for judging the quality of your application's generated responses. There are a few things to keep in mind when using LLM-as-judge evaluators.

Calibrating and Auditing LLM Evaluators

It is important to continuously audit the results of your LLM-as-judge evaluators. A great way to go about this is to design a feedback process that keeps your domain experts in the evaluation loop. LangSmith provides a feature called Annotation Queues that allow team members to provide feedback on traces from your application. These could be production traces from real users or evaluation traces generated from running your evals.

Building a process where domain experts are presented with a queue of traces to review and annotate to provide detailed feedback on the generated response from your application can be extremely beneficial when using LLM-as-judge evaluators. This allows developers to take that feedback and adjust system prompts or context to drive convergence between your application's responses and the ideal responses as defined by the domain expert(s). Traces can be added to annotation queues manually or via LangSmith Automations that use a filter to automatically add traces that meet the filter criteria to an annotation queue.

Have your LLM evaluators provide an explanation for their judgement. This can be helpful when you are trying to determine whether or not your evaluator prompt needs adjustment or a system prompt in your application needs adjustment. If your evaluator scores a response negatively, but the reasoning doesn't make sense, you don't want to change your application to fit the evaluator's flawed criteria. You want to adjust your evaluator's prompt so that it is driving the results you are looking for.

Scoring

A common approach with LLM-as-judge evaluators is to have them score a generated response on a scale. For example, you might prompt your LLM evaluator to rate the correctness of a response on a 0-1 or a 1-5 scale. If this scale is well defined and your team and your LLM judges have a good understanding of what the difference is between a 3 and a 4 on a 1-5 scale, then this can be a good way to understand how close you are to successful results.

Without a clearly defined scale, however, this approach can cause confusion and loose requirements on quality. If there is no clear definition for a 3 vs a 4, then what do the results actually mean? What should you change to try and increase the score? Keep this in mind as you design your evaluator prompts so that you don't end up with a lot of data that you're not
sure how to interpret. Consider enforcing pass/fail results or be sure to have a clear definition around the scale you use so that when your evals produce a lot of 2s and 3s, you and your team all understand exactly what that means. Keep your evaluations actionable.

LLM Bias

LLMs have been shown to display bias when assessing LLM generated output. For example, a given model may tend to prefer answers that were generated using the same model and score them more highly than answers generated by other models.

It is important to be aware of this and keep an eye out for it in your results. It may be beneficial to choose models that are different from the models you use for your application's response generation for your LLM judges. It may also be helpful to periodically swap out the models used for your judges to see how it affects your metrics and analyze if those differences are due to bias in the LLM judges.

Cost

Cost is important to keep in mind when designing your eval set and in determining when to use LLM-as-judge evaluators. Investing in robust evaluation is a strong step towards success, but you also need to keep in mind that LLM-as-judge evaluators can get expensive as your eval set grows and you are continuously running evaluation, sometimes more expensive than running your actual application in production.

The balance between cost and LLM evaluation is something that is dependent on your project and budget. The important thing here is to be aware of the cost that they can introduce and to be intentional about when you use them if cost is a limiting factor. Keep in mind what LLM-as-judge evaluators are good at - qualitative and subjective evaluation. Use them for those types of evaluations and use code evaluators for quantitative evaluations. You might also consider limiting how frequently your LLM-as-judge evaluators execute if their cost is becoming prohibitive. Run your less costly evaluations frequently and more costly evaluations on major changes or when code merges to detect regressions in quality.

Composite Evaluators

Composite evaluators allow you to combine multiple evaluators into a single overall score using a weighted sum or average.

These can be useful to get an at-a-glance composite score of how your application is performing across multiple metrics that have varying importance in overall success.

/CHAPTER FOUR

Evaluation Strategies for Different Application Types

/CHAPTER FOUR

Evaluation Strategies for Different Application Types

This chapter includes some basic examples for strategies on how to approach evaluation for different types of applications. These are not exhaustive and your success criteria or key metrics may differ, but these examples can give you an idea of how to get started.

RAG Applications

Success Criteria

- **Accuracy** - 95% of responses contain factually correct information
- **Retrieval relevance** - retrieved context must contain information pertinent to the user's input in 90% of cases (changed from 100%—that's unrealistic)
- **Groundedness** - 98% of factual claims in responses must be supported by retrieved documents
- **Response time** P95 < 2 seconds
- **Safety** - 0% of responses include PII or confidential information
- **Tone** - 100% of responses use appropriate, professional language

Key Metrics

- **Retrieval precision** (% of retrieved documents that are relevant)
- **Retrieval recall** (% of relevant documents that were retrieved)
- **Answer accuracy** (% of responses that correctly answer the question)
- **Groundedness score** (% of claims supported by retrieved context)
- **Context relevance** (% of retrievals that contain pertinent information)
- **Latency** (P50, P95, P99)
- **PII detection rate** (% of responses flagged for sensitive information)
- **Tone appropriateness** (% of responses meeting professional language standards)

Evaluation Approach

1. **Test retrieval independently** from generation using precision/recall metrics
2. **Evaluate answer accuracy** by comparing responses to reference answers
3. **Check for hallucinations** by verifying claims against source documents (groundedness)
4. **Test** with queries that should return no relevant results (measure how system handles insufficient context)
5. **Include adversarial examples** attempting to extract confidential information

Agents and Multi-Step Workflows

Success Criteria

- **Task completion** - 85% of tasks successfully completed end-to-end
- **Efficiency** - average task completion in ≤ 8 steps (or whatever is reasonable for your agent)
- **Tool accuracy** - 95% of tool calls use correct tools with valid parameters
- **Error recovery** - 70% of recoverable errors successfully handled without human intervention
- **Response time** - P95 latency < 5 seconds per step

Key Metrics

- **Task completion rate** (% of tasks that reach successful end state)
- **Average steps to completion**
- **Tool selection accuracy** (% of correct tool choices)
- **Tool parameter validity** (% of tool calls with valid parameters)
- **Error recovery rate** (% of errors successfully recovered from)
- **Unnecessary step rate** (% of steps that don't contribute to task completion)
- **Latency per step** and total task latency

Evaluation Approach

1. **Test individual tool calls** in isolation to validate parameter generation and response handling
2. **Evaluate decision-making logic** by examining which tools are selected for different scenarios
3. **Measure end-to-end success rates** with realistic task examples
4. **Test error handling** by introducing failed tool calls and invalid responses
5. **Analyze agent traces** to identify inefficient paths or loops
6. **Compare multi-step performance** against human baseline or simpler approaches

Conversational AI

Success Criteria

- **Response appropriateness** - 95% of responses are contextually appropriate and helpful
- **Context retention** - 90% of responses correctly reference information from earlier in conversation
- **Consistency** - 98% of responses remain consistent with previous statements in the conversation
- **Response time** - P95 < 1.5 seconds
- **Safety** - 0% of responses contain harmful, offensive, or inappropriate content
- **Conversation flow** - 85% of multi-turn conversations maintain natural coherence

Key Metrics

- **Response appropriateness score** (% of contextually suitable responses)
- **Context retention accuracy** (% of responses correctly using conversation history)
- **Consistency rate** (% of responses that don't contradict earlier statements)
- **Conversation coherence score** (multi-turn flow quality)
- **Safety violation rate** (% of responses flagged for harmful content)
- **User intent recognition** (% of user inputs correctly understood)
- **Latency** (P50, P95, P99)

Evaluation Approach

1. **Test single-turn responses** to establish baseline quality
2. **Evaluate multi-turn context handling** with conversations of varying length (3, 5, 10+ turns)
3. **Check for consistency** by asking questions that reference earlier conversation context
4. **Test edge cases** including topic switches, ambiguous references, and clarification requests
5. **Include adversarial inputs testing** for jailbreaks, prompt injection, and inappropriate requests
6. **Measure conversation abandonment rate** (if applicable) as a proxy for user satisfaction

/CHAPTER FIVE

Best Practices and Common Pitfalls

/CHAPTER FIVE

Best Practices and Common Pitfalls

Best Practices

01. Start Simple

- Start with small, high-quality datasets
- Focus on your most critical use cases first

02. Build Evaluation into Your SDLC

- **Development Phase**
  - Design quality data sets with domain experts
  - Run evals as frequently as possible
  - Quick feedback loops with smaller datasets
  - Focus on building evals that represent your success criteria and will help you catch regressions as you progress through development
  - Automate evaluation runs by adding them to your CI/CD pipeline(s)
- **Pre-Production Phase**
  - Comprehensive evaluation on expanded datasets
  - Domain expert on datasets and evaluation results
  - A/B testing different approaches and models
  - Performance benchmarking
  - Audit LLM evaluators for bias
- **Production Phase**
  - Online evaluation of live traffic
  - User feedback collection
  - Continuous monitoring and alerting
  - Regular audits and updates to eval framework
  - Continued domain expert evaluation feedback as well as feedback on live traffic

03. Iterate on Your Datasets

- Update datasets as you discover new edge cases
- Sample production traffic for evaluation

04. Iterate on Your Evaluators

- Regularly audit evaluator accuracy
- Use human feedback to calibrate LLM judges

05. Balance Coverage and Cost

- Sample production traffic for evaluation
- Use cheaper models for high-volume evaluation
- Cache evaluation results when possible

06. Make Evals Actionable

- Set clear performance thresholds
- Automate regression detection
- Create feedback loops to improve weak areas

07. Evals Alongside Traditional Tests

- Evals do not completely replace traditional testing
- Continue to use traditional testing methods to validate code that doesn't involve LLM calls like helper and utility functions

Common Pitfalls to Avoid

01. Over-relying on LLM Judges

- LLM evaluators can be biased or inconsistent
- Always validate with human evaluation periodically
- Use multiple evaluators for critical metrics

02. Evaluation Data Leakage

- Keep test data separate from training/prompt examples
- Rotate test sets periodically
- Monitor for overfitting to evaluation metrics

03. Neglecting Edge Cases

- Don't just test happy paths
- Include adversarial examples
- Test failure modes and error handling

04. Ignoring Production Reality

- Test with realistic data distributions
- Include latency and cost in your metrics
- Evaluate under production-like conditions

/CHAPTER SIX

Advanced Techniques

/CHAPTER SIX

Advanced Techniques

Few-Shot Evaluation Improvement

Improve LLM evaluator accuracy with few-shot examples.

Few-shot prompt templates allow you to provide examples to the LLM you are using for generation. This is a good way to help guide your LLM-as-judge evaluators on what constitutes a good answer or a bad answer and how results should be scored.

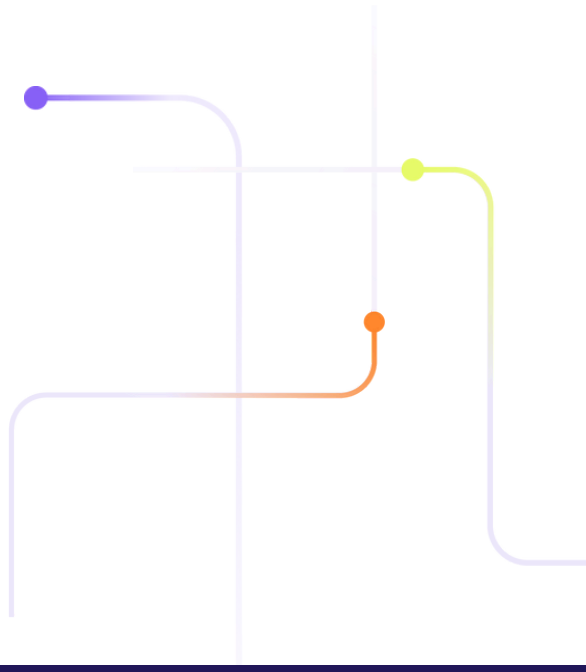
```python
few_shot_example
s = [
   {
      "input": "Is the sky blue?", "output":
      "Yes, the sky appears blue during clear
      days.",
      "evaluation": "Score: 1. The response
      directly answers
thequestion
."
   }, #More
   examples...
]
```

LangSmith Annotation Queues

LangSmith's Annotation Queues provide a way to mark certain traces from your application for further review.

These traces could come from live production traffic, offline evaluation, or both. When a trace is added to an Annotation Queue, reviewers can access the trace information for that invocation and provide feedback on the result.

This is a great way to allow domain experts or other reviewers to assess a production result or evaluation and provide a better, more accurate answer that can be incorporated into datasets for further evaluation.



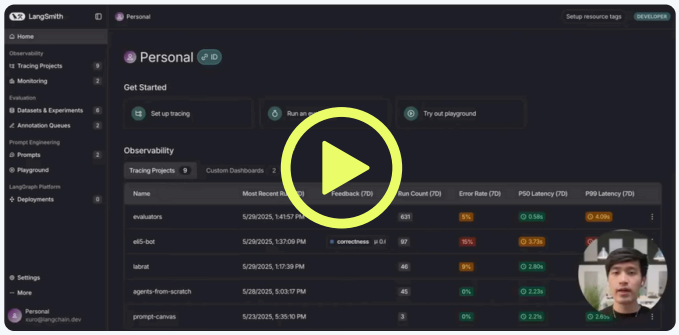
LangSmith Automation and Online Evaluations

Automations are rules that you can configure to run over every trace sent from your application.

Automations are made up of a filter, a sampling rate, and an action to perform on sampled traces that match the filter criteria. This is a powerful feature that allows you to automatically take action on traces from your application.

With LangSmith automations you can automatically add traces to a dataset for further evaluation, add traces to an annotation queue for review/domain expert feedback, trigger webhooks to trigger alerts or an action, or extend the data retention policy for the trace allowing developers more time to review failure modes.

Here is a great video on getting started with LangSmith automation and online evaluation:



https://www.youtube.com/watch?v=z69cBXTJFZ0&t=2s

Online Evaluations are a type of Automation that help to measure metrics on your application's output in production. Unlike offline evaluations, online evaluations are run on live customer interactions in production and can be used to gather metrics on how your application performs in the real world with real user input.

Self-Correcting Systems with Evals

Use evaluation in your application logic to add a quality check in your application's workflow.

This can be a great way to ensure that your application is producing the best possible results for your users. This can be as simple as retrying the LLM call if the initial answer doesn't meet evaluation criteria.

```python
class SelfCorrectingChain:
    def run(self, input):
        output = self.generate(input)
        evaluation = self.evaluate(output)

        if evaluation.score < threshold:
            # Retry with improved prompt or different approach
            output = self.regenerate_with_feedback(input,
evaluation.feedback)

        return output
```

/THE END

Conclusion

/THE END

Conclusion

Implementing a robust evaluation framework transforms AI development from an art to a science. Instead of guessing whether changes improve your system, you have quantifiable evidence and continuous validation.

The journey from guesswork to confidence requires:

1

Commitment

to building and maintaining evaluation infrastructure

2

Discipline

to run evals consistently and act on results

3

Iteration

to continuously improve both your application and evaluation methods

Start small, be consistent, and gradually expand your evaluation coverage.

With the approaches and tools covered in this guide you have everything needed to build AI applications you can trust.

The goal isn't perfect scores on every metric. It's understanding your system's capabilities and limitations, making informed decisions about trade-offs, and having confidence that your application will perform reliably in production.

Your next step? Pick one critical aspect of your AI application, create a small dataset of 20 test cases, implement a simple evaluator, and start measuring. From that foundation, you can build a comprehensive evaluation system that ensures your AI applications deliver real value with confidence.

If you find yourself unsure about how to proceed with evals and EDD, reach out to us at Focused!

We have lots of experience putting agents and AI applications into production and we'd love to help you on your journey.

Contact Us Today

The path from guesswork to confidence is clear. The tools are available.

The only question is: when will you start measuring what matters?

Resources and Further Learning

1. Getting Started with LangSmith YouTube Series
2. LangSmith Evaluations YouTube Series
3. Getting Started with LangSmith Documentation
4. LangChain Evaluation Concepts
5. LangChain Evaluation QuickStart

Build Agents that Work