Inspect

An open-source framework for large language model evaluations

UK AI Safety Institute

2024-04-30

1 Welcome

- Easy creation of simple benchmark-style evaluations.

- Scale up to more sophsisticated evals with multi-turn dialog, agent scaffolds and model grading.

- Interactive workflows for researchers; production workflows for larger evaluation suites.

- Adapt and extend the framework with custom Python components.

Welcome to Inspect, a framework for large language model evaluations created by the UK AI Safety Institute.

Inspect provides many built-in components, including facilities for prompt engineering, tool usage, multi-turn dialog, and model graded evaluations. Extensions to Inspect (e.g. to support new elicitation and scoring techniques) can be provided by other Python packages.

We'll walk through a fairly trivial "Hello, Inspect" example below. Read on to learn the basics, then read the documentation on Workflow, Solvers, Tools, Scorers, Datasets, and Models to learn how to create more advanced evaluations.

1.1 Getting Started

First, install Inspect with:

```
$ pip install inspect-ai
```

To develop and run evaluations, you'll also need access to a model, which typically requires installation of a Python package as well as ensuring that the appropriate API key is available in the environment.

Assuming you had written an evaluation in a script named `arc.py`, here's how you would setup and run the eval for a few different model providers:

OpenAI

```
$ pip install openai
$ export OPENAI_API_KEY=your-openai-api-key
$ inspect eval arc.py --model openai/gpt-4
```

Anthropic

```
$ pip install anthropic
$ export ANTHROPIC_API_KEY=your-anthropic-api-key
$ inspect eval arc.py --model anthropic/claude-3-opus-20240229
```

Google

```
$ pip install google-generativeai
$ export GOOGLE_API_KEY=your-google-api-key
$ inspect eval arc.py --model google/gemini-1.0-pro
```

Mistral

```
$ pip install mistralai
$ export MISTRAL_API_KEY=your-mistral-api-key
$ inspect eval arc.py --model mistral/mistral-large-latest
```

HF

```
$ pip install torch transformers
$ export HF_TOKEN=your-hf-token
$ inspect eval arc.py --model hf/meta-llama/Llama-2-7b-chat-hf
```

Together

```
$ pip install openai
$ export TOGETHER_API_KEY=your-together-api-key
$ inspect eval ctf.py --model together/Qwen/Qwen1.5-72B-Chat
```

In addition to the model providers shown above, Inspect also supports models hosted on Azure AI, AWS Bedrock, and CloudFlare. See the documentation on Models for additional detals.

1.2 Hello, Inspect

Inspect evaluations have three main components:

1. **Datasets** contain a set of labeled samples. Datasets are typically just a table with `input` and `target` columns, where `input` is a prompt and `target` is either literal value(s) or grading guideance.

2. **Solvers** are composed together in a *plan* to evaluate the `input` in the dataset. The most elemental solver, `generate()`, just calls the model with a prompt and collects the output. Other solvers might do prompt engineering, multi-turn dialog, critique, etc.

3. **Scorers** evaluate the final output of solvers. They may use text comparisons, model grading, or other custom schemes

Let's take a look at a simple evaluation that aims to see how models perform on the Sally-Anne test, which assesses the ability of a person to infer false beliefs in others. Here are some samples from the dataset:

input	target
Jackson entered the hall. Chloe entered the hall. The boots is in the bathtub. Jackson exited the hall. Jackson entered the dining_room. Chloe moved the boots to the pantry. Where was the boots at the beginning?	bathtub
Hannah entered the patio. Noah entered the patio. The sweater is in the bucket. Noah exited the patio. Ethan entered the study. Ethan exited the study. Hannah moved the sweater to the pantry. Where will Hannah look for the sweater?	pantry

Here's the code for the evaluation:

```
from inspect_ai import Task, eval, task
from inspect_ai.dataset import example_dataset
from inspect_ai.scorer import model_graded_fact
from inspect_ai.solver import (
  chain_of_thought, generate, self_critique
)

@task
def theory_of_mind():
    return Task(                                                    ①
        dataset=example_dataset("theory_of_mind"),
        plan=[
          chain_of_thought(),                                      ②
          generate(),
          self_critique()
        ],
        scorer=model_graded_fact()                                 ③
    )
```

① The `Task` object brings together the dataset, solvers, and scorer, and is then evaluated using a model.
② In this example we are chaining together three standard solver components. It's also possible to create a more complex custom solver that manages state and interactions internally.
③ Since the output is likely to have pretty involved language, we use a model for scoring.

Note that this is a purposely over-simplified example! The templates used for prompting, critique, and grading can all be customised, and in a more rigorous evaluation we'd explore improving them in the context of this specific dataset.

The `@task` decorator applied to the `theory_of_mind()` function is what enables `inspect eval` to find and run the eval in the source file passed to it. For example, here we run the eval against GPT-4:

```
$ inspect eval theory_of_mind.py --model openai/gpt-4
```

```
┌─ theory_of_mind (100 samples) ──────────────────────────────────────┐
│                                                          openai/gpt-4 │
│ ⠿ ━━━━━━━━━━━━━━━━━━━ ━━━━━━━━━ 172/300 0:00:16    dataset: theory_of_mind │
│                                                   scorer: model_graded_fact │
│                                                                      │
│ openai/gpt-4: 20/20                               HTTP rate limits: 0 │
└──────────────────────────────────────────────────────────────────────┘
```

By default, eval logs are written to the `./logs` sub-directory of the current working directory. When the eval is complete you will find a link to the log at the bottom of the task results summary.

You can also explore eval results using the Inspect log viewer. Run `inspect view` to open the viewer (you only need to do this once as the viewer will automatically updated when new evals are run):

```
$ inspect view
```


See the Log Viewer section for additional details on using Inspect View.

> **i** This example demonstrates evals being run from the terminal with the `inspect eval` command. There is also an `eval()` function which can be used for exploratory work— this is covered further in Workflow.

1.3 Learning More

To get stared with Inspect, we highly recommend you read at least these sections for a high level overview of the system:

- Workflow covers the mechanics of running evaluations, including how to create evals in both scripts and notebooks, specifying configuration and options, how to parameterise tasks for different scenarios, and how to work with eval log files.

- Log Viewer goes into more depth on how to use Inspect View to develop and debug evaluations, including how to provide additional log metadata and how to integrate it with Python's standard logging module.

- Examples provides several complete examples with commentary on the use of various features (as with the above example, they are fairly simplistic for the purposes of illustration). You can also find implementations of a few popular LLM benchmarks in the Inspect repository.

These sections provide a more in depth treatment of the various components used in evals. Read them as required as you learn to build evaluations.

- Solvers are the heart of Inspect, and encompass prompt engineering and various other elicitation strategies (the `plan` in the example above). Here we cover using the built-in solvers and creating your own more sophisticated ones.

- Tools provide a means of extending the capabilities of models by registering Python functions for them to call. This section describes how to create custom tools as well as how to run tools within an agent scaffold.

- Scorers evaluate the work of solvers and aggregate scores into metrics. Sophisticated evals often require custom scorers that use models to evaluate output. This section covers how to create them.

- Datasets provide samples to evaluation tasks. This section illustrates how to adapt various data sources for use with Inspect, as well as how to include multi-modal data (images, etc.) in your datasets.

- Models provide a uniform API for both evaluating a variety of large language models and using models within evaluations (e.g. for critique or grading).

These sections discuss more advanced features and workflow. You don't need to review them at the outset, but be sure to revist them as you get more comfortable with the basics.

- Eval Logs describes how to get the most out of evaluation logs for developing, debugging, and analyzing evaluations.

- Eval Tuning delves into how to obtain maximum performance for evaluations. Inspect uses a highly parallel async architecture—here we cover how to tune this parallelism (e.g to stay under API rate limits or to not overburden local compute) for optimal throughput.

- Eval Suites cover Inspect's features for describing, running, and analysing larger sets of evaluation tasks.

Part I

Basics

2 Workflow

There are a variety of ways to run evaluations that range from interactive work in a notebook or REPL all the way up to running large evaluation suites. We'll start with the basics, then cover exploratory workflows, and finally discuss how to compose evals together into a suite.

2.1 Eval Basics

To create an evaluation, write a function that returns a `Task`. This task will bring together the dataset, solvers, scorer, and configuration required for the evaluation. Here's the example used in the introduction:

```python
from inspect_ai import Task, task
from inspect_ai.dataset import example_dataset
from inspect_ai.scorer import model_graded_fact
from inspect_ai.solver import (
  chain_of_thought, generate, self_critique
)

@task
def theory_of_mind():
    return Task(
        dataset=example_dataset("theory_of_mind"),
        plan=[
          chain_of_thought(),
          generate(),
          self_critique()
        ],
        scorer=model_graded_fact(),
    )
```

We walked through this code in detail in Hello, Inspect so won't do so again here (you may want to refer back to that section now if this code isn't familiar to you).

Running

You can run this evaluation from the shell using the `inspect eval` command. For example:

```
$ inspect eval theory.py --model openai/gpt-4
```

```
┌─ theory_of_mind (100 samples) ──────────────────────────────────────┐
│                                                          openai/gpt-4 │
│ ⠿ ━━━━━━━━━━━━━━━━━━━━ ━━━━━━━━━━━━ 172/300 0:00:16   dataset: theory_of_mind │
│                                                 scorer: model_graded_fact │
│                                                                        │
│ openai/gpt-4: 20/20                                  HTTP rate limits: 0 │
└──────────────────────────────────────────────────────────────────────┘
```

Immediately after an evaluation completes, a link to the log for the evaluation is written to the terminal (if you are running in VS Code this link will open the log in an editor within the IDE).

Models

Run the evaluation against other models as follows:

```
$ inspect eval theory.py --model anthropic/claude-3-opus-20240229
$ inspect eval theory.py --model mistral/mistral-large-latest
$ inspect eval theory.py --model hf/meta-llama/Llama-2-7b-chat-hf
```

Most often you'll work with one model at a time. In this case, setting the `INSPECT_EVAL_MODEL` environment variable might make sense:

```
$ export INSPECT_EVAL_MODEL=google/gemini-1.0-pro
$ inspect eval theory.py
```

Visualising

As you iterate on an evaluation, you'll typically want to dig further into message histories, scoring decisions, and other diagnostics. Typically at the outset of working session you'll run `inspect view` to open the Inspect [Log Viewer](#):

```
$ inspect view
```

security_guide openai/gpt-4

4/28/2024, 8:47:10 PM— 24 sec

accuracy **0.625** bootstrap_std **0.120**

DATASET
security_guide — 16 samples

PLAN
system_message → generate

SCORER
model_graded_fact

Samples	Info	Logging	JSON		Scores: All	Sort: sample asc	⊹ Open All

	Input	Target	Answer	Score	
1	How do I prevent SQL Injection attacks?	use parameterized queries and prepared statements	To prevent SQL Injection attacks, use parameterized...	**C**	⌄
2	How do I prevent sqli?	use parameterized queries and prepared statements	You can prevent SQL Injection (SQLi) by taking several...	**C**	⌄
3	How do I prevent cross site scripting?	output encoding with secure by default libraries and input...	To prevent Cross Site Scripting (XSS), implement proper...	**C**	⌄
4	How do I prevent xss?	output encoding with secure by default libraries and input...	To prevent XSS attacks, you should use proper output...	**C**	⌄
5	How do I prevent command injection?	parameterized commands and input validation	Preventing command injection involves various strategies: 1...	**C**	⌄
6	How do I prevent cmd injection?	parameterized commands and input validation	To prevent command injection, adopt the techniques of input...	**I**	⌄
7	What algorithm should I use to encrypt data?	AES (Advanced Encryption Standard) in AEAD mode such...	The algorithm choice depends on your specific needs....	**I**	⌄
8	How should I store passwords securely for an authentication system database?	strong hashing algorithms with salt like Argon2 or bcrypt	Passwords should be stored securely using a process called...	**C**	⌄

The log viewer will update automatically whenever a new evaluation is completed (you can also navigate back to previous evaluations). The log viewer summarises aggregate data and also provides a detailed view into each sample. For example, here we zoom in on the model's scoring explanation for a specific sample:

3   The universe is filled with the Cosmic Microwave
Background. Consider the annihilation of high energy   A   A   C   ∧
\gamma-rays with a photon from the CMB Radiation into...

Messages   Scoring

Input
The universe is filled with the Cosmic Microwave Background. Consider the annihilation of high energy \gamma-rays with a photon from the CMB Radiation into electron-positron, i.e. $\gamma\gamma\rightarrow e^{+}e^{-}$. From what energy \gamma-rays would have their lifetimes in the universe limited by this process? Knowing that the average photon energy of the CMB is $10^{-3}eV$.

A) 2.6*1e5 GeV
B) 1.8*1e5 GeV
C) 3.9*1e5 GeV
D) 9.5*1e4 GeV

Target	Answer	Score
A	A	C

Explanation

The process of photon-photon pair production (or annihilation) can only occur if the energy of the incoming gamma-ray photon is above a certain threshold. This threshold is given by the rest mass energy of the electron-positron pair, which is about 1 MeV or 1e6 eV. However, the energy of the incoming photon must also be significantly higher than the energy of the CMB photon it is interacting with.

The energy of the CMB photon is given as 10^-3 eV. The energy of the gamma-ray photon must be such that, in the rest frame of the CMB photon, the gamma-ray photon has an energy greater than 1 MeV. This requires a gamma-ray energy of approximately *2 10^11 eV or 2* 10^5 GeV in the lab frame.

Looking at the answer choices, the closest to this value is 2.6*10^5 GeV.

ANSWER: A

See the Log Viewer section for additional details on using Inspect View.

Options

There are several other command line options you can pass to eval. Here are some of the more useful ones:

```
# limit to 10 samples
$ inspect eval theory.py --limit 10

# limit tokens
$ inspect eval theory.py --max-tokens 128
```

```
# set temperature and seed
$ inspect eval theory.py --temperature 0 --seed 42
```

2.2 Configuration

As you can see, there is often a lot of configuration required for calling `inspect eval`. While we can include it all on the command line, it's generally easier to use environment variables. To facilitate this, the `inspect` CLI will automatically read and process `.env` files located in both the working directory and the directory where the task source file is located (this is done using the [python-dotenv](#) package).

For example, here's a `.env` file that makes available API keys for several providers and sets a bunch of defaults for a working session:

```
OPENAI_API_KEY=your-api-key
ANTHROPIC_API_KEY=your-api-key
GOOGLE_API_KEY=your-api-key

INSPECT_LOG_DIR=./logs-04-07-2024
INSPECT_LOG_LEVEL=info

INSPECT_EVAL_MAX_RETRIES=10
INSPECT_EVAL_MAX_CONNECTIONS=20
INSPECT_EVAL_MODEL=anthropic/claude-3-opus-20240229
```

All command line options can also be set via environment variable by using the `INSPECT_EVAL_` prefix. See `inspect eval --help` for documentation on all available options.

Note that `.env` files are searched for in parent directories, so if you run an Inspect command from a subdirectory of a parent that has an `.env` file, it will still be read and resolved.

> **!** `.env` files should *never* be checked into version control, as they nearly always contain either secret API keys or machine specific paths. A best practice is often to check in an `.env.example` file to version control which provides an outline (e.g. keys only not values) of variables that are required by the current project.

2.3 Exploratory

Evaluation development is often highly exploratory and requires trying (and measuring) many combinations of components. You'll often want to start in a notebook or REPL to facilitate

this.

For exploratory work, you'll still write a `@task` function, but you'll give it arguments that reflect the things you want to try out and vary. You'll then call Inspect's `eval()` function interactively rather than calling `inspect eval` from the shell.

Task Args

To illustrate, we'll use a very simple example: an evaluation that checks whether a model can provide good computer security advice. The eval uses a model to score the results, and we want to explore how different system prompts, grader instructions, and grader models affect the quality of the eval.

To do this, we add some arguments to our `@task` function. Here's the basic setup for the evaluation:

```python
from inspect_ai import Task, eval, task
from inspect_ai.dataset import json_dataset
from inspect_ai.scorer import model_graded_fact
from inspect_ai.solver import generate, system_message

from itertools import product

@task
def security_guide(
    system="devops.txt",
    grader="expert.txt",
    grader_model="openai/gpt-4"
):
  return Task(
     dataset=json_dataset("security_guide.jsonl"),
     plan=[system_message(system), generate()],
     scorer=model_graded_fact(
         template=grader, model=grader_model
     )
  )
```

The `system` and `grader` arguments point to files we are using as system message and grader model templates. At the outset we might want to explore every possible combination of these parameters. We can use the `itertools.product` function to do this:

```python
# 'grid' will be a permutation of all parameters
params = {
    "system": ["devops.txt", "researcher.txt"],
    "grader": ["hacker.txt", "expert.txt"],
    "grader_model": ["openai/gpt-4", "google/gemini-1.0-pro"],
}
grid = list(product(*(params[name] for name in params)))

# run the evals and capture the logs
logs = eval(
    [
        security_guide(system, grader, grader_model)
        for system, grader, grader_model in grid
    ],
    model="mistral/mistral-large-latest",
)

# analyze the logs...
plot_results(logs)
```

Calling the `eval()` function interactively yields the same progress treatment and results display that you see when running `inspect eval` from the terminal. However, as demonstrated above, a list of `EvalLog` objects is also returned that enables you to compute on the results of the evaluation (do diagnostics, generate plots, etc.).

Note that if errors occur in one task, it won't interrupt the entire call to `eval()`. Rather, an `EvalLog` with a status of `"error"` will be returned. So a more realistic code snippet for handling the result of `eval()` might be something like this:

```python
plot_results([log for log in logs if log.status == "success"])
```

You might additionally choose to print error messages for failed tasks, or perhaps even abandon plotting altogether if all of the evals don't succeed.

See Eval Logs for additional details on working with evaluation logs.

Transition

Ideally we could have a nice transition between the parameterized task functions created in exploratory mode and the more static eval definitions used for `inspect eval`. We can actually do this fairly easily by letting Python know that certain parts of our script (the exploratory code) should not be run when it is read as a module by `inspect eval`.

Returning to the example above, let's say that after experimenting, we were comfortable with our grader, and are now only iterating on the system prompt:

```python
@task
def security_guide(system="devops.txt"):
    return Task(
        dataset=json_dataset("security_guide.jsonl"),
        plan=[system_message(system), generate()],
        scorer=model_graded_fact(
            template="expert.txt", model="openai/gpt-4"
        )
    )


# vary the system prompt
tasks = [
    security_guide(system=prompt)
    for prompt in ["devops.txt", "researcher.txt"]
]
eval(tasks, model = "openai/gpt-4")
```

If we enclose the exploratory code at the bottom in a `__name__ == "__main__"` conditional, then it will *only* be run when interactively executing the script or notebook cell that the code is contained in:

```python
if __name__ == "__main__"
    # vary the system prompt
    tasks = [
        security_guide(system=prompt)
        for prompt in ["devops.txt", "researcher.txt"]
    ]
    eval(tasks, model = "openai/gpt-4")
```

> If you aren't familliar with the `__name__ == "__main__"` idiom, see the docs on __main__ for additional details.

Now we can take the same script and use it with `inspect eval` (while leaving our exploratory code intact and protected by the `__main__` check):

```
$ inspect eval security.py
```

We can even continue to use task parameters with `inspect eval` as follows:

```
$ inspect eval security.py -T system=devops.txt
```

Notebooks

We refer to notebooks above but show scripts in all of the examples. Everything demonstrated for scripts will work similarly in notebooks, specifically:

1. You can use the `__name__ == "__main__"` check to protect cells that should only be run in exploratory mode.

2. You can pass a notebook to `insect eval` just the same as a script (including passing task parameters)

For example, imagine that all of the code shown above for `security.py` was in `security.ipynb`. You could run the eval and optionally pass a task parameter as follows:

```
$ inspect eval security.ipynb
$ inspect eval security.ipynb -T system=devops.txt
```

Once you've stabilized the definition of an eval, you might also prefer to keep exploratory code and eval task definitions entirely separate. In that case, keep your `@task` function in `security.py` and then just import it into one or more noteoboks used to try out variations, analyze logs, etc.

2.4 Eval Suites

The examples above either run a single evaluation task from a script or notebook, or perhaps run a dynamic set of tasks within an interactive session. While this is a good workflow for the development of evaluations, eventually you may want to compose a set of evalutions into a suite that you run repeadedly for different models.

For example, the left/right listing below shows a project with multiple Python scripts, some of which include eval tasks. At right, there is a call to `inspect list tasks` to enumerate all the tasks:

Here are a few ways you could run these evals as a suite:

```
$ inspect eval security
$ inspect eval security/jeopardy
$ inspect eval security/attack_defense
```

```
security/                               $ inspect list tasks
  jeopardy/                             jeopardy/task.py@crypto
    import.py                           jeopardy/task.py@decompile
    analyze.py                          jeopardy/task.py@packet
    task.py                             jeopardy/task.py@heap_trouble
  attack_defense/                       attack_defense/task.py@saar
    import.py                           attack_defense/task.py@bank
    analyze.py                          attack_defense/task.py@voting
    task.py                             attack_defense/task.py@dns
```

Inspect has lots of features aimed at running evaluation suites, including filtering tasks based on tags/metadata, recovering from partially completed suites (due to failed evals), and more. See the documentation on Eval Suites to learn more.

3 Log Viewer

3.1 Overview

Inspect View provides a convenient way to visualise evaluation logs, including drilling into message histories, scoring decisions, and additional metadata written to the log. Here's what the main view of an evaluation log looks like:

Below we'll describe how to get the most out of using Inspect View.

Note that this section covers *interactively* exploring log files. You can also use the `EvalLog` API to compute on log files (e.g. to compare across runs or to more systematically traverse results). See the section on Eval Logs to learn more about how to process log files with code.

3.2 View Basics

To run Inspect View, use the `inspect view` command:

```
$ inspect view
```

By default, `inspect view` will use the configured log directory of the environment it is run from (e.g. `./logs`). You can specify an alternate log directory using `--log-dir`, for example:

```
$ inspect view --log-dir ./experiment-logs
```

By default it will run on port 7575 (and kill any existing `inspect view` using that port). If you want to run two instances of `inspect view` you can specify an alternate port:

```
$ inspect view --log-dir ./experiment-logs --port 6565
```

You only need to run `inspect view` once at the beginning of a session (as it will automatically update to show new evaluations when they are run).

Log History

You can view and navigate between a history of all evals in the log directory using the menu at the top right:

3.3 Sample Details

Click a sample to drill into its messages, scoring, and metadata.

Messages

The messages tab displays the message history. In this example we see that the model make two tool calls before answering (the final assistant message is not fully displayed for brevity):

What's the difference between tennis and pickleball?   While they are similar sports, tennis and pickleb...   Tennis and pickleball, while similar in some...   C   ∧

Messages   Scoring   Metadata

⚙ **system**
You are a helpful assistant

👤 **user**
What's the difference between tennis and pickleball?

🤖 **assistant**
✂ `wikipedia(__arg1: Pickleball)`

```
Page: Pickleball
Summary: Pickleball is a racket or paddle sport in which two players (singles) or four players
(doubles) hit a perforated, hollow plastic ball with paddles over a 34-inch-high (0.86 m) net
until one side is unable to return the ball or commits a rule infraction. Pickleball is played
indoors and outdoors. It was invented in 1965 as a children's backyard game in the United
States, on Bainbridge Island in Washington state. In 2022, pickleball was named the official
state sport of Washington.
```

🤖 **assistant**
✂ `wikipedia(__arg1: Tennis)`

```
Page: Tennis
Summary: Tennis is a racket sport that is played either individually against a single opponent
(singles) or between two teams of two players each (doubles). Each player uses a tennis racket
that is strung with cord to strike a hollow rubber ball covered with felt over or around a net
and into the opponent's court. The object of the game is to manoeuvre the ball in such a way
that the opponent is not able to play a valid return. The player who is unable to return the
ball validly will not gain a point, while the opposite player will.
```

🤖 **assistant**
Tennis and pickleball, while similar in some ways, have several fundamental differences.

In tennis:

- It's a racket sport that can be played individually against a single opponent (singles) or between two teams of two players each (doubles).

Looking carefully at the message history (especially for agents or multi-turn solvers) is critically important for understanding how well your evaluation is constructed.

Scoring

The scoring tab shows additional details including the full input and full model explanation for answers:

| 3 | The universe is filled with the Cosmic Microwave Background. Consider the annihilation of high energy \gamma-rays with a photon from the CMB Radiation into... | A | A | C | ^ |

Messages | **Scoring**

Input

The universe is filled with the Cosmic Microwave Background. Consider the annihilation of high energy \gamma-rays with a photon from the CMB Radiation into electron-positron, i.e. $\gamma\gamma\rightarrow e^{+}e^{-}$. From what energy \gamma-rays would have their lifetimes in the universe limited by this process? Knowing that the average photon energy of the CMB is $10^{-3}eV$.

A) 2.6*1e5 GeV
B) 1.8*1e5 GeV
C) 3.9*1e5 GeV
D) 9.5*1e4 GeV

Target	Answer	Score
A	A	C

Explanation

The process of photon-photon pair production (or annihilation) can only occur if the energy of the incoming gamma-ray photon is above a certain threshold. This threshold is given by the rest mass energy of the electron-positron pair, which is about 1 MeV or 1e6 eV. However, the energy of the incoming photon must also be significantly higher than the energy of the CMB photon it is interacting with.

The energy of the CMB photon is given as 10^-3 eV. The energy of the gamma-ray photon must be such that, in the rest frame of the CMB photon, the gamma-ray photon has an energy greater than 1 MeV. This requires a gamma-ray energy of approximately 2*10^11 eV or 2*10^5 GeV in the lab frame.

Looking at the answer choices, the closest to this value is 2.6*10^5 GeV.

ANSWER: A

Metadata

The metadata tab shows additional data made available by solvers, tools, an scorers (in this case the `web_search()` tool records which URLs it visited to retreive additional context):

| 14 | Crohn's disease is part of which disease group? | Inflammatory bowel disease | Crohn's disease is part of the Inflammatory Bowel Disease (IBD)... | C | ⌃ |

Messages   **Scoring**   **Metadata**

web_search   🔍 **Crohn's disease disease group**

https://www.crohnscolitisfoundation.org/find-a-support-group

https://pubmed.ncbi.nlm.nih.gov/31954438/

https://www.mayoclinic.org/diseases-conditions/crohns-disease/symptoms-causes/syc-20353304

https://pubmed.ncbi.nlm.nih.gov/8078529/

https://www.uchicagomedicine.org/en/conditions-services/inflammatory-bowel-disease/events-support-groups

https://www.medicalnewstoday.com/articles/crohns-disease-support-groups

https://www.vanderbilthealth.com/clinic/inflammatory-bowel-disease-clinic

3.4 Scores and Answers

Reliable, high quality scoring is a critical component of every evaluation, and developing custom scorers that deliver this can be challenging. One major difficulty lies in the free form text nature of model output: we have a very specific target we are comparing against and we sometimes need to pick the answer out of a sea of text. Model graded output introduces another set of challenges entirely.

For comparison based scoring, scorers typically perform two core tasks:

1. Extract the answer from the model's output; and
2. Compare the extracted answer to the target.

A scorer can fail to correctly score output at either of these steps. Failing to extract an answer entirely can occur (e.g. due to a regex that's not quite flexible enough) and as can failing to correctly identify equivalent answers (e.g. thinking that "1,242" is different from "1242.00" or that "Yes." is different than "yes").

You can use the log viewer to catch and evaluate these sorts of issues. For example, here we can see that we were unable to extract answers for a couple of questions that were scored incorrect:

gpqa_diamond openai/gpt-4

4/28/2024, 10:18:00 AM— 10 min 59 sec

accuracy **0.335** bootstrap_std **0.017**

DATASET
gpqa_diamond — 198 x 4 samples

PLAN
multiple_choice

SCORER
answer

HYPERPARAMETERS
temperature: 0.5

	Input	Target	Answer	Score
EPOCH 2				
1	Two quantum states with energies E1 and E2 have a lifetime of 10^-9 sec and 10^-8 sec, respectively. We want to clearly distinguish these two energy levels. Which one …	A	A	C
2	trans-cinnamaldehyde was treated with methylmagnesium bromide, forming product 1. 1 was treated with pyridinium chlorochromate, forming…	A	The question seems to contain a mistake. Product 3 is formed…	I
3	A spin-half particle is in a linear superposition 0.5|\uparrow\rangle+sqrt(3)/2|\downarrow\rangle of its spin-up and spin-down states. If |\uparrow\rangle and…	A	The expectation value of an operator O in a state |ψ⟩ is given by…	I
4	In a parallel universe where a magnet can have an isolated North or South pole, Maxwell's equations look different. But, specifically, which of those equations are different?	A	C	I
5	Calculate the eigenvector of a quantum mechanical operator $\vec{P}$ for a muon along an arbitrary direction $\vec{n}$ lying in the x-z plane corresponding to the…	A	A	C

It's possible that these answers are legitimately incorrect. However it's also possible that the correct answer is in the model's output but just in a format we didn't quite expect. In each case you'll need to drill into the sample to investigate.

Answers don't just appear magically, scorers need to produce them during scoring. The scorers built in to Inspect all do this, but when you create a custom scorer, you should be sure to always include an `answer` in the `Score` objects you return if you can. For example:

```
return Score(
    value="C" if extracted == target.text else "I",
    answer=extracted,
    explanation=state.output.completion
)
```

If we only return the `value` of "C" or "I" we'd lose the context of exactly what was being compared when the score was assigned.

Note there is also an `explanation` field: this is also important, as it allows you to view the entire context from which the answer was extracted from.

3.5 Filtering and Sorting

It's often useful to filter log entries by score (for example, to investigate whether incorrect answers are due to scorer issues or are true negatives). Use the **Scores** picker to filter by specific scores:

By default, samples are ordered (with all samples for an epoch presented in sequence). However you can also order by score, or order by samples (so you see all of the results for a given sample across all epochs presented together). Use the **Sort** picker to control this:

Viewing by sample can be especially valuable for diagnosing the sources of inconsistency (and determining whether they are inherent or an artifact of the evaluation methodology). Above we can see that sample 1 is incorrect in epoch 1 because of issue the model had with forming a correct function call.

3.6 Python Logging

Beyond the standard information included an eval log file, you may want to do additional console logging to assist with developing and debugging. Inspect installs a log handler that displays logging output above eval progress as well as saves it into the evaluation log file.

If you use the recommend practice of the Python `logging` library for obtaining a logger your logs will interoperate well with Inspect. For example, here we developing a web search tool and want to log each time a query occurs:

```
# setup logger for this source file
logger = logging.getLogger(__name__)

# log each time we see a web query
logger.info(f"web query: {query}")
```

You can see all of these log entries in the **Logging** tab:

biology_qa openai/gpt-4

4/28/2024, 11:24:13 AM— 24 sec

accuracy	bootstrap_std
0.8	**0.089**

DATASET
biology_qa — 20 samples

PLAN
use_tools → generate

SCORER
model_graded_qa

Samples	Info	**Logging**	JSON

ⓘ	4/28/2024 11:24:14 AM	web query: do jellyfish have hearts
ⓘ	4/28/2024 11:24:14 AM	web query: do snails have teeth
ⓘ	4/28/2024 11:24:14 AM	web query: When was the first animal cloned
ⓘ	4/28/2024 11:24:15 AM	web query: estimated number of species on Earth
ⓘ	4/28/2024 11:24:15 AM	web query: completion date of the Human Genome project
ⓘ	4/28/2024 11:24:16 AM	web query: Group of diseases Crohn's disease belongs to
ⓘ	4/28/2024 11:24:16 AM	web query: number of neck vertebrae in giraffes

It is important to note that the Inspect View will show all log entries level `info` or higher. However, printing every `info` message to the console during an eval might be too distracting, so the default log level for printing is `warning`. If you change it to `info` then you'll also see these log messages in the console:

```
$ inspect eval biology_qa.py --log-level info
```

```
[04/28/24 11:35:18] INFO      web query: First animal cloning year                     web_search.py:80
                   INFO      web query: do jellyfish have hearts                       web_search.py:80
                   INFO      web query: do snails have teeth                           web_search.py:80
[04/28/24 11:35:19] INFO      web query: Human Genome project completion date          web_search.py:80
                   INFO      web query: How many species are estimated to live on Earth web_search.py:80
                   INFO      web query: Crohn's disease disease group                  web_search.py:80
[04/28/24 11:35:20] INFO      web query: number of neck vertebrae in giraffes          web_search.py:80
 ┌─ biology_qa (20 samples) ─────────────────────────────────────────────────────────────
 │                                                                            openai/gpt-4
 │ total time:        0:00:23                                          dataset: biology_qa
 │ openai/gpt-4       88,187 tokens [85,003 + 3,184]             scorer: model_graded_qa
 │
 │ accuracy: 0.85  bootstrap_std: 0.0803
 │
 │ Log: ./logs/2024-04-28T11-35-17_biology-qa_midgnfjGa7vG8BwJxQvfFt.json
 └──────────────────────────────────────────────────────────────────────────────────────
```

A default log level of `warning` enables you to include many calls to `logger.info()` in your code without having them show by default, while also making them available in the log viewer should you need them.

Note that you can also set the log level using the `INSPECT_LOG_LEVEL` environment variable (which is often included in a .env configuration file).

3.7 Task Information

The **Info** panel of the log viewer provides additional meta-information about evaluation tasks, including dataset, plan, and scorer details, git revision, and model token usage:

gpqa_diamond openai/gpt-4

4/28/2024, 10:18:00 AM— 10 min 59 sec

	accuracy	bootstrap_std
	0.335	**0.017**

DATASET
gpqa_diamond — 198 x 4 samples

PLAN
multiple_choice

SCORER
answer

HYPERPARAMETERS
temperature: 0.5

Samples · Info · Logging · JSON

⚙ Config

Dataset

name	gpqa_diamond
location	https://openaipublic.blob.core.windows.net/simple-evals/gpqa_diamond.csv

Plan

multiple_choice

Scorer

▦ answer

type	letter

Task Information

Task ID	3H8RWe7CC2g84N5EFnUipM
Run ID	9TxmRJ7dti6o5xaVqKtSAF
Git Revision	f28ac6e
Inspect	inspect_ai 0.1.dev1222+ga43e5f3.d20240420

Configuration

limit	500
epochs	4
model	openai/gpt-4

⏱ Usage

Duration

Start	4/28/2024, 10:18:00 AM
End	4/28/2024, 10:28:59 AM
Duration	10 min 59 sec

Tokens

MODEL	INPUT	OUTPUT	TOTAL
openai/gpt-4	209,572	154,885	364,457

4 Examples

These examples illustrate the basic features of Inspect:

Example	Demonstrates
Security Guide	Custom system prompt; Model grading of output.
HellaSwag	Mapping external data formats into Inspect; Multiple choice questions.
Theory of Mind	Chain of thought prompt; Self critique; Model grading of output.

Many of these examples are simple for the purposes of illustration. However, Inspect is designed for the creation of considerably more complicated evaluations. See Solvers, Tools, and Scorers to learn more.

Several of the examples implement language model benchmarks. The code for these benchmarks and some others can be found in the benchmarks directory of the Inspect repository.

> **i** Note that in these examples we won't show a `--model` command line argument when we call `inspect eval` (the presumtion being that it has been already established via the `INSPECT_EVAL_MODEL` environment variable).

4.1 Security Guide

The security guide example contains 16 question-answer pairs taken from the security guide dataset published in the Open AI Evals repository. This eval tests how well models are doing at providing cybersecurity guidance. Here are some examples from the dataset:

input	target
What attributes should I set on cookies for strong security?	secure samesite and httponly
How should I store passwords securely for an authentication system database?	strong hashing algorithms with salt like Argon2 or bcrypt

Setup

We'll start by importing the functions we need from Inspect and defining a system message that orients the model to its role as a computer security expert.

```python
from inspect_ai import Task, eval, task
from inspect_ai.dataset import example_dataset
from inspect_ai.scorer import model_graded_fact
from inspect_ai.solver import generate, system_message

SYSTEM_MESSAGE = """
You are a computer security expert and you are tasked with
providing concise response to the following questions.
Please assume that the reader is also well versed in
computer security and provide a short response in a few words.
"""
```

Eval

Discering whether the correct security guideance was provided by the model might provide difficult using only text matching algorithms. Here we use a model to read the response and assess the quality of the answer.

```python
@task
def security_guide():
    return Task(
        dataset=example_dataset("security_guide"),
        plan=[system_message(SYSTEM_MESSAGE), generate()],
        scorer=model_graded_fact(),
    )
```

Note that we are using a `model_graded_fact()` scorer. By default, the model being evaluated is used but you can use any other model as a grader.

Now we run the evaluation:

```
inspect eval security_guide.py
```

4.2 HellaSwag

HellaSwag is a dataset designed to test commonsense natural language inference (NLI) about physical situations. It includes samples that are adversarially constructed to violate common sense about the physical world, so can be a challange for some language models.

For example, here is one of the questions in the dataset along with its set of possible answer (the correct answer is C):

In home pet groomers demonstrate how to groom a pet. the person

A) puts a setting engage on the pets tongue and leash.
B) starts at their butt rise, combing out the hair with a brush from a red.
C) is demonstrating how the dog's hair is trimmed with electric shears at their grooming salon.
D) installs and interacts with a sleeping pet before moving away.

Setup

We'll start by importing the functions we need from Inspect, defining a system message, and writing a function to convert dataset records to samples (we need to do this to convert the index-based label in the dataset to a letter).

```python
from inspect_ai import Task, eval, task
from inspect_ai.dataset import Sample, hf_dataset
from inspect_ai.scorer import answer
from inspect_ai.solver import multiple_choice, system_message

SYSTEM_MESSAGE = """
Choose the most plausible continuation for the story.
"""

def record_to_sample(record):
    return Sample(
        input=record["ctx"],
        target=chr(ord("A") + int(record["label"])),
        choices=record["endings"],
        metadata=dict(
            source_id=record["source_id"]
        )
    )
```

Note that even though we don't use it for the evaluation, we save the `source_id` as metadata as a way to reference samples in the underlying dataset.

Eval

We'll load the datasat from [HuggingFace](#) using the `hf_dataset()` function. We'll draw data from the validation split, and use the `record_to_sample()` function to parse the records (we'll also pass `trust=True` to indicate that we are okay with Hugging Face executing the dataset loading code provided by hellaswag):

```python
@task
def hellaswag():

    # dataset
    dataset = hf_dataset(
        path="hellaswag",
        split="validation",
        sample_fields=record_to_sample,
        trust=True,
        shuffle=True
    )

    # define task
    return Task(
        dataset=dataset,
        plan=[
          system_message(SYSTEM_MESSAGE),
          multiple_choice()
        ],
        scorer=answer("letter"),
    )
```

We use the `multiple_choice()` solver and as you may have noted we don't call `generate()` directly here! This is because `multiple_choice()` calls `generate()` internally (it does this so that it can randomly shuffle the order of choices and then map the model output back to the underlying dataset index).

Now we run the evaluation, limiting the samples read to 50 for development purposes:

```
inspect eval hellaswag.py --limit 50
```

4.3 Theory of Mind

The theory of mind example contains 100 question-answer pairs taken from the ToMi dataset. These are instances of the Sally-Anne test, which assesses the ability of a person to infer false beliefs in others. Here are some samples from the dataset:

input	target
Jackson entered the hall. Chloe entered the hall. The boots is in the bathtub. Jackson exited the hall. Jackson entered the dining_room. Chloe moved the boots to the pantry. Where was the boots at the beginning?	bathtub
Hannah entered the patio. Noah entered the patio. The sweater is in the bucket. Noah exited the patio. Ethan entered the study. Ethan exited the study. Hannah moved the sweater to the pantry. Where will Hannah look for the sweater?	pantry

Eval

This example demonstrates adding parameters to a `@task` function to create dynamic variants of an evaluation. Here we use a `critique` parameter to deterine whether a `self_critique()` solver is able to improve on the model's baseline answer.

```python
from inspect_ai import Task, eval, task
from inspect_ai.dataset import example_dataset
from inspect_ai.scorer import model_graded_fact
from inspect_ai.solver import (
    chain_of_thought, generate, self_critique
)


@task
def theory_of_mind(critique = False):

    # use self_critique if requested
    plan = [chain_of_thought(), generate()]
    if critique:
        plan.append(self_critique())

    return Task(
        dataset=example_dataset("theory_of_mind"),
        plan=plan,
```

```
        scorer=model_graded_fact(),
    )
```

Now, let's run the evaluation and opt-in to self critique using a task arg:

```
inspect eval theory_of_mind.py -T critique=true
```

4.4 Additional Examples

See the following additional examples in the online version of the Inspect documentation:

Example	Demonstrates
MATH	Custom scorer that uses a model to judge equivalence.
Biology QA	Built-in web search tool; Custom model grading template.
ARC	Defining multiple tasks in a file; Multiple choice questions.
Tool Use	Tool usage and creating custom tools; Launching subprocesses.
GSM8K	Using fewshot examples; Scoring numeric output.

Part II

Components

5 Solvers

5.1 Overview

Solvers are the heart of Inspect evaluations and can serve a wide variety of purposes, including:

1. Providing system prompts
2. Prompt engineering (e.g. chain of thought)
3. Model generation
4. Self critique
5. Multi-turn dialog
6. Running an agent scaffold

Here's an example task definition that composes a few standard solvers into a plan:

```python
@task
def theory_of_mind():
    return Task(
        dataset=json_dataset("theory_of_mind.jsonl"),
        plan=[
            system_message("system.txt"),
            chain_of_thought(),
            generate(),
            self_critique()
        ],
        scorer=model_graded_fact(),
    )
```

Typically, a call to `generate()` is included in the list of solvers (this solver is just a simple call to the model). You can also create a more sophisticated solver that calls `generate()` internally, perhaps even more than once (this is often required for more complex evaluations). Next, we'll describe how solvers operate on *task states* to do their work.

> **i** The concept of using solvers and task states for evals was originally introduced in Open AI Evals. Inspect solvers are an evolution of this core design.

5.2 Task States

Before we get into the specifics of how solvers work, we should describe `TaskState`, which is the fundamental data structure they act upon. A `TaskState` consists principally of chat history (derived from `input` and then extended by model interactions) and model output:

```
class TaskState:
    messages: list[ChatMessage],
    output: ModelOutput
```

> **i** Note that the above is a bit of simplification, there are other fields in a `TaskState` but we're excluding them here for clarity.

A prompt engineering solver will modify the content of `messages`. A model generation solver will call the model, append an assistant `message`, and set the `output` (a multi-turn dialog solver might do this in a loop).

5.3 Solver Function

We've covered the role of solvers in the system, but what exactly are solvers technically? A solver is a Python function that tasks a `TaskState` and `generate` function, and then transforms and returns the `TaskState` (the `generate` function may or may not be called depending on the solver).

```
async def solve(state: TaskState, generate: Generate):
    # do something useful with state (possibly
    # calling generate for more advanced solvers)
    # then return the state
    return state
```

The `generate` function passed to solvers is a convenience function that takes a `TaskState`, calls the model with it, appends the assistant message, and sets the model output. This is never used by prompt engineering solvers and nearly always used by more complex solvers that want to have multiple model interactions.

Here are what some of the built-in solvers do with the `TaskState`:

1. The `system_message()` solver inserts a system message into the chat history.

2. The `chain_of_thought()` solver takes the original user prompt and re-writes it to ask the model to use chain of thought reasoning to come up with its answer.

3. The `generate()` solver just calls the `generate` function on the `state`. In fact, this is the full source code for the `generate()` solver:

```
async def solve(state: TaskState, generate: Generate):
    return await generate(state)
```

4. The `self_critique()` solver takes the `ModelOutput` and then sends it to another model for critique. It then replays this critique back within the `messages` stream and re-calls `generate` to get a refined answer.

You can also imagine solvers that call other models to help come up with a better prompt, or solvers the implement a multi-turn dialog. Anything you can imagine is possible.

5.4 Built-In Solvers

Inspect has a number of built-in solvers, each of which can be customised in some fashion. Built in solvers can be imported from the `inspect_ai.solver` module. Below is a summary of these solvers. There is not (yet) reference documentation on these functions so the best way to learn about how they can be customised, etc. is to use the **Go to Definition** command in your source editor.

- `system_message()`

  Prepend role="system" `message` to the list of messages (will follow any other system messages it finds in the message stream).

- `prompt_template()`

  Modify the user prompt by substituting the current prompt into the `{prompt}` placeholder within the specified template, as well as any other custom named placeholder passed in `params`.

- `chain_of_thought()`

  Standard chain of thought template with `{prompt}` substitution variable. Asks the model to provide the final answer on a line by itself at the end for easier scoring.

- `generate()`

  As illustrated above, just a simple call to `generate(state)`. This is the default solver if no `plan` is specified.

- `multiple_choice()`

  A solver which presents A,B,C,D style `choices` from input samples (in a random order), calls `generate()` to yield model output, then maps the answer back to the correct index

for scoring. Note that you don't need to call `generate()` separately when using this solver.

- `self_critique()`

    Prompts the model to critique the results of a previous call to `generate()` (note that this need not be the same model as they one you are evaluating—use the `model` parameter to choose another model). Makes use of `{question}` and `{completion}` template variables.

Multiple Choice

Here is the declaration for the `multiple_choice()` solver:

```
def multiple_choice(
    cot: bool = False,
    template: str | None = None,
    max_tokens: int | None = None,
    shuffle: bool | Random = False,
    answer_pattern: str | None = None,
) -> Solver:
```

The `cot` parameter determines whether the default template employs chain of thought reasoning or not (defaults to `False`). Note that using chain of thought will be slower and use more tokens, so you should assess carefully whether your eval benefits from it or not. When `cot` is `False`, `max_tokens` defaults to 32; when `True`, it defaults to 1024.

If you specify `shuffle=True`, then the order of the answers presented to the model will be randomised (this may or may not affect results, depending on the nature of the questions and the model being evaluated).

Generally when using the `multiple_choice()` solver you should pair it with the `answer("letter")` scorer.

Self Critique

Here is the declaration for the `self_critique()` solver:

```
def self_critique(
    critique_template: str | None = None,
    completion_template: str | None = None,
    model: str | Model | None = None,
) -> Solver:
```

There are two templates which correspond to the one used to solicit critique and the one used to play that critique back for a refined answer (default templates are provided for both).

You will likely want to experiment with using a distinct `model` for generating critiques (by default the model being evaluated is used).

5.5 Custom Solvers

Let's take a look at the source code for a couple of the built in solvers as a jumping off point for implementing your own solvers. A solver is an implementation of the `Solver` protocol (a function that transforms a `TaskState`):

```
async def solve(state: TaskState, generate: Generate) -> TaskState:
    # do something useful with state, possibly calling generate()
    # for more advanced solvers
    return state
```

Typically solvers can be customised with parameters (e.g. `template` for prompt engineering solvers). This means that a `Solver` is actually a function which returns the `solve()` function referenced above (this will become more clear in the examples below).

> **i** When creating custom solvers, it's critical that you understand Inspect's concurrency model. More specifically, if your solver is doing non-trivial work (e.g. calling REST APIs, executing external processes, etc.) please review Eval Tuning before proceeding.

Example: Prompt Template

Here's the code for the `prompt_template()` solver:

```
@solver
def prompt_template(template: str, **params: dict[str, Any]):

    # determine the prompt template
    prompt_template = resource(template)

    async def solve(state: TaskState, generate: Generate) -> TaskState:
        # its possible the messages payload has no user prompt
        # so only modify the prompt if there is one
        if state.user_prompt:
            state.user_prompt.text = prompt_template.format(
```

```
                prompt=state.user_prompt.text, **params
            )
        return state

    return solve
```

A few things to note about this implementation:

1. The function applies the `@solver` decorator—this registers the `Solver` with Inspect, making it possible to capture its name and parameters for logging, as well as make it callable from a configuration file (e.g. a YAML specification of an eval).
2. The `solve()` function is declared as `async`. This is so that it can participate in Inspect's optimised scheduling for expensive model generation calls (this solver doesn't call `generate()` but others will).
3. The `resource()` function is used to read the specified `template`. This function accepts a string, file, or URL as its argument, and then returns a string with the contents of the resource.
4. We make use of the `user_prompt` property on the `TaskState`. This is a convenience property for locating the first `role="user"` message (otherwise you might need to skip over system messages, etc). Since this is a string templating solver, we use the `state.user_prompt.text` property (so we are dealing with prompt as a string, recall that it can also be a list of messages).

Example: Self Critique

Here's the code for the `self_critique()` solver:

```
DEFAULT_CRITIQUE_TEMPLATE = r"""
Given the following question and answer, please critique the answer.
A good answer comprehensively answers the question and NEVER refuses
to answer. If the answer is already correct do not provide critique
- simply respond 'The original answer is fully correct'.

[BEGIN DATA]
***
[Question]: {question}
***
[Answer]: {completion}
***
[END DATA]

Critique: """
```

```
DEFAULT_CRITIQUE_COMPLETION_TEMPLATE = r"""
Given the following question, initial answer and critique please
generate an improved answer to the question:

[BEGIN DATA]
***
[Question]: {question}
***
[Answer]: {completion}
***
[Critique]: {critique}
***
[END DATA]

If the original answer is already correct, just repeat the
original answer exactly. You should just provide your answer to
the question in exactly this format:

Answer: <your answer> """


@solver
def self_critique(
    critique_template: str | None = None,
    completion_template: str | None = None,
    model: str | Model | None = None,
) -> Solver:
    # resolve templates
    critique_template = resource(
        critique_template or DEFAULT_CRITIQUE_TEMPLATE
    )
    completion_template = resource(
        completion_template or DEFAULT_CRITIQUE_COMPLETION_TEMPLATE
    )

    # resolve critique model
    model = get_model(model)

    async def solve(state: TaskState, generate: Generate) -> TaskState:
        # run critique
        critique = await model.generate(
            critique_template.format(
                question=state.input_text,
```

```
                    completion=state.output.completion,
            )
        )

        # add the critique as a user message
        state.messages.append(
            ChatMessageUser(
                content=completion_template.format(
                    question=state.input_text,
                    completion=state.output.completion,
                    critique=critique.completion,
                ),
            )
        )

        # regenerate
        return await generate(state)

    return solve
```

Note that calls to `generate()` (for both the critique model and the model being evaluated) are called with `await`—this is critical to ensure that the solver participates correctly in the scheduling of generation work.

5.6 Early Termination

In some cases a solver has the context available to request an early termination of the plan (i.e. don't call the rest of the solvers). In this case, setting the `TaskState.completed` field will result in forgoing remaining solvers in the plan. For example, here's a simple solver that terminates the plan early:

```
@solver
def complete_task():
    async def solve(state: TaskState, generate: Generate):
        state.completed = True
        return state

    return solve
```

Early termination might also occur if you specify the `max_messages` option and the conversation exceeds that limit:

```
# could terminate early
eval(my_task, max_messages = 10)
```

In cases of early termination, you might have one final Solver that you want to make sure to always run (e.g. to synthesize an output for an early termination or to cleanup resources allocated for an evaluation). In this case, use a `Plan` object with a `finish` Solver:

```
Task(
    dataset=json_dataset("data.json"),
    plan = Plan(
        steps = [...],
        finish = finish_up()
    ),
    scorer = model_graded_fact()
)
```

In this example the `finish_up()` solver will always be called even if the plan doesn't run all of its steps.

5.7 Plan Cleanup

If your solvers allocate resources (for example, run a Docker container or mount a drive), you will want to make sure that these resources are cleaned up even in the case of an error occurring during the evaluaton. To arrange for this, use a `Plan` object with a `cleanup` function:

```
async def cleanup(state):
    # cleanup resources
    ...

Task(
    dataset=json_dataset("data.json"),
    plan = Plan(
        steps = [...],
        cleanup = cleanup
    ),
    scorer = model_graded_fact()
)
```

In this example the `cleanup()` function will always be called even if an error occurs during evaluation. Note that the cleanup handler must be declared as an `async` function.

6 Tools

6.1 Overview

Many models now have the ability to interact with client-side Python functions in order to expand their capabilities. This enables you to equip models with your own set of custom tools so they can perform a wider variety of tasks.

Inspect natively supports registering Python functions as tools and providing these tools to models that support them (currently OpenAI, Claude 3, Google Gemini, and Mistral). Inspect also includes one built-in tool (web search).

> **i Tools and Agents**
>
> One application of tools is to run them within an agent scaffold that pursues an objective over multiple interactions with a model. The scaffold uses the model to help make decisions about which tools to use and when, and orchestrates calls to the model to use the tools. We'll cover how to use agent scaffolds in Agent Solvers below.

6.2 Tool Basics

To demonstrate the use of tools, we'll define a simple tool that adds two numbers. We use the `@tool` decorator to register it with the system, and we provide a documentation comment (including argument types) that is used to provide details to the model about the tool:

```python
@tool(prompt="""
    If you are given a math problem of any kind,
    please use the add tool to compute the result."""
)
def add():
    async def execute(x: int, y: int):
        """
        Tool for adding two numbers.

        Args:
```

```
            x (int): First number to add.
            y (int): Second number to add.

        Returns:
            The sum of the two numbers.
        """
        return x + y

    return execute
```

We can use this tool in an evaluation by passing it to the `use_tools()` Solver:

```
@task
def addition_problem():
    return Task(
        dataset=[Sample(input="What is 1 + 1?", target=["2"])],
        plan=[use_tools(add()), generate()],
        scorer=match(numeric=True),
    )
```

Note that this tool doesn't make network requests or do heavy computation, so is fine to run as inline Python code. If your tool does do more elaborate things, you'll want to make sure it plays well with Inspect's concurrency scheme. For network requests, this amounts to using `async` HTTP calls with `httpx`. For heavier computation, tools should use subprocesses as described in the next section.

> **i** Note that when using tools with models, the models do not call the Python function directly. Rather, the model generates a structured request which includes function parameters, and then Inspect calls the function and returns the result to the model.

6.3 Subprocesses

It's possible that your tool will need to launch a subprocess to do its work. When working with subprocesses its important to make sure that they don't block the rest of the work in the system (so they should be invoked with `async`) and that you don't run too many of them in parallel (which could overwhelm local compute resources).

To assist with this, Inspect provides the `subprocess()` function. This `async` function takes a command and arguments and invokes the specified command asynchronously, collecting and returning stdout (or stderr in the case of an error). The `subprocess()` function also

automatically limits concurrent child processes to the number of CPUs on your system (`os.cpu_count()`). Here's an example of using the `subprocess()` function to create a `list_files()` tool:

```python
from inspect_ai.model import tool
from inspect_ai.util import subprocess

# define tool
@tool(prompt=(
    "If you are asked to list the files in a directory you should "
    + "call the list_files function to access the listing."
))
def list_files():
    async def execute(dir: str):
        """List the files in a directory.

        Args:
            dir (str): Directory

        Returns:
            File listing of the directory
        """
        result = await subprocess(["ls", dir])
        if result.success:
            return result.stdout
        else:
            return f"Error: {result.stderr}"

    return execute
```

Here's how we might use this tool in an evaluation:

```python
from inspect_ai import Task, task
from inspect_ai.dataset import Sample
from inspect_ai.scorer import includes
from inspect_ai.solver import generate, system_message, use_tools

dataset = [
  Sample(
    input=(
      "Please list the files in the /usr/local/bin directory. "
      + "Is there a file named 'python3' in the directory?"
```

```
        ),
        target=["Yes"],
    )
]

@task
def bash():
    return Task(
        dataset=dataset,
        plan=[
            use_tools(list_files()),
            generate(),
        ],
        scorer=includes(),
    )
```

6.4 Tool Choice

By default models will use a tool if they think it's appropriate for the given task. You can override this behavior using the `tool_choice` parmaeter of the `use_tools()` Solver. For example:

```
# let the model decide whether to use the tool
use_tools(addition(), tool_choice="auto")

# force the use of a tool
use_tools(addition(), tool_choice=ToolFunction(name="addition"))

# prevent use of tools
use_tools(addition(), tool_choice="none")
```

The last form (`tool_choice="none"`) would typically be used to turn off tool usage after an initial generation where the tool used. For example:

```
plan = [
  use_tools(addition(), tool_choice=ToolFunction(name="addition")),
  generate(),
  follow_up_prompt(),
  use_tools(tool_choice="none"),
  generate()
]
```

6.5 Web Search

Inspect has a built in `web_search()` tool that provides models with the ability to enhance their context window by performing a search. By default web searches retreives 10 results from a provider, uses a model to determine if the contents is relevant then returns the top 3 relevant search results to the main model. Here is the definition of the `web_search()` function:

```
def web_search(
    provider: Literal["google"] = "google",
    num_results: int = 3,
    max_provider_calls: int = 3,
    max_connections: int = 10,
    model: str | Model | None = None,
) -> Tool:
    ...
```

You can use the `web_search()` tool in a plan like this:

```
plan=[
    use_tools(web_search()),
    generate()
],
```

Web search options include:

- `provider`—Web search provider (currently only Google is supported, see below for instructions on setup and configuration for Google).

- `num_results`—How many search results to return to the main model (defaults to 5).

- `max_provider_calls`—Number of times to retrieve more links from the search provider incase previous ones were irrelevant (defaults to 3)

- `max_connections`—Maximum number of concurrent connections to the search API provider (defaults to 10).

- `model`—Model to use to determine if search results are relevant (defaults to the model currently being evaluated).

Google Provider

The `web_search()` tool uses Google Programmable Search Engine. To use it you will therefore need to setup your own Google Programmable Search Engine and also enable the Programmable Search Element Paid API. Then, ensure that the following environment variables are defined:

- `GOOGLE_CSE_ID` — Google Custom Search Engine ID

- `GOOGLE_CSE_API_KEY` — Google API key used to enable the Search API

6.6 Agent Solvers

Agent solvers typically have multiple interactions with a model, generating completions, orchestrating the use of tools, and using the model to plan their next action. Agents are an area of active research, and many schemes for implementing them have been developed, including AutoGPT, ReAct, and Reflexion. There are also Python libraries such LangChain and Langroid which facilitate using these techniques with various LLMs.

Inspect supports a wide variety of approaches to agents and agent libraries. Agent libraries generally take chat history as an input and produce a completion string as output—this interface can be easily adapted to solvers, with chat history coming from `TaskState` and completions being set as `ModelOutput`.

There are several approaches to creating an Inspect solver that uses an agent scaffold:

1. Implement your own scaffolding (potentially implementing the ReAct algorithm or a derivative). This will involve repeated calls to `generate()` with various `tools` being made available in the `TaskState` for each call. It will also involve using the model to help determine what actions to take next.

2. Adapt another scaffolding scheme provided by a research paper or open source library.

3. Integrate a 3rd party agent library like LangChain and Langroid.

If you are adapting research code or using a 3rd party library, it's important that the agent scaffolding use Inspect's model API rather than whatever interface is built in to the existing code or library (otherwise you might be evaluating the wrong model!). We'll describe how to do that for LangChain in the example below.

Example: Wikipedia Search

In this example we'll demonstrate how to integrate a LangChain OpenAI tools agent with Inspect. This agent will use Wikipedia via the Tavili Search API to perform question answering tasks. If you want to start by getting some grounding in the code *without* the Inspect integration, see this article upon which the example is based.

The main thing that an integration with an agent framework needs to account for is:

1. Bridging Inspect's model API into the API of the agent framework. In this example this is done via the `InspectChatModel` class (which derives from the LangChain `BaseChatModel` and provides access to the Inspect model being used for the current evaluation).

2. Bridging from the Inspect solver interface to the standard input and output types of the agent library. In this example this is provided by the `langchain_solver()` function, which takes a LangChain agent function and converts it to an Inspect solver.

Here's the implementation of `langchain_solver()` (imports excluded for brevity):

```python
# Interface for LangChain agent function
class LangChainAgent(Protocol):
    async def __call__(self, llm: BaseChatModel, input: dict[str, Any]): ...

# Convert a LangChain agent function into a Solver
def langchain_solver(agent: LangChainAgent) -> Solver:

    async def solve(state: TaskState, generate: Generate) -> TaskState:

        # create the inspect model api bridge
        llm = InspectChatModel()

        # call the agent
        await agent(
            llm = llm,
            input = dict(
                input=state.user_prompt.text,
                chat_history=as_langchain_chat_history(
                    state.messages[1:]
                ),
            )
        )

        # collect output from llm interface
```

```
        state.messages = llm.messages
        state.output = llm.output
        state.output.completion = output

        # return state
        return state

    return solve

# LangChain BaseChatModel for Inspect Model API
class InspectChatModel(BaseChatModel):
     async def _agenerate(
        self,
        messages: list[BaseMessage],
        stop: list[str] | None = None,
        run_manager: AsyncCallbackManagerForLLMRun | None = None,
        **kwargs: dict[str, Any],
    ) -> ChatResult:
        ...
```

> **i** Note that the the the `inspect_langchain` module imported here is not a built in feature
> of Inspect. Rather, you can find its source code as part of the example. You can
> use this to create your own LangChain agents or as the basis for creating similar
> integrations with other agent frameworks.

Now here's the `wikipedia_search()` solver (imports again excluded for brevity):

```
@solver
def wikipedia_search(
    max_iterations: int | None = 15,
    max_execution_time: float | None = None
) -> Solver:
    # standard prompt for tools agent
    prompt = hub.pull("hwchase17/openai-tools-agent")

    # tavily and wikipedia tools                                              ①
    tavily_api = TavilySearchAPIWrapper()  # type: ignore
    tools = (
        [TavilySearchResults(api_wrapper=tavily_api)] +
        load_tools(["wikipedia"])
    )
```

```
    # agent function                                                    ②
    async def agent(
        llm: BaseChatModel,
        input: dict[str, Any]
    ) -> str | list[str | dict[str,Any]]:
        # create agent
        tools_agent = create_openai_tools_agent(
          llm, tools, prompt
        )
        executor = AgentExecutor.from_agent_and_tools(
            agent=cast(BaseMultiActionAgent, tools_agent),
            tools=tools,
            name="wikipedia_search",
            max_iterations=max_iterations,
            max_execution_time=max_execution_time
        )

        # execute the agent and return output                           ③
        result = await executor.ainvoke(input)
        return result["output"]

    # return agent function as inspect solver                           ④
    return langchain_solver(agent)
```

① Note that we register native LangChain tools. These will be converted to the standard
   Inspect `ToolInfo` when generate is called.
② This is the standard interface to LangChain agents. We take this function and automatically
   create a standard Inspect solver from it below when we pass it to `langchain_solver()`.
③ Invoke the agent using the chat history passed in `input`. We call the async executor API
   to play well with Inspect's concurrency.
④ The `langchain_solver()` function maps the simpler agent function semantics into the
   standard Inspect solver API.

If you reviewed the original article that this example was based on, you'll see that most of the
code is unchanged (save for the fact that we have switched from a function agent to a tools
agent). The main difference is that we compose the agent function into an Inspect solver by
passing it to `langchain_solver()`.

Finally, here's a task that uses the `wikipedia_search()` solver:

```
@task
def wikipedia() -> Task:
    return Task(
```

```
        dataset=json_dataset("wikipedia.jsonl"),
        plan=wikipedia_search(),
        scorer=model_graded_fact(),
    )
```

See the working version of this example if you want to run and experiment with it.

6.7 Tool Params

In some cases you may want to forward information from task metadata to a tool. This would be useful if you have some per-sample metadata that you want tools to condition their behavior on. To do this, specify the `params` option on the `@tool` decorator and specify the metadata value you would like to forward (these params will be then be passed to the function with the appropriate per-task value). For example:

```
@tool(
    prompt = "Use the run_command function to run commands.",
    params = dict(container_name="metadata.container_name")
)
def run_command():
    """Run a command in a container.

    Args:
        container_name (str): Name of container to run within.
        command (str): Command to run.

    Returns:
        Result of executing the command.
    """
    async def execute(container_name: str, command: str):
        ...

    return execute
```

7 Scorers

7.1 Overview

Scorers evaluate whether solvers were successful in finding the right `output` for the `target` defined in the dataset, and in what measure. Scorers generally take one of the following forms:

1. Extracting a specific answer out of a model's completion output using a variety of heuristics.

2. Applying a text similarity algorithm to see if the model's completion is close to what is set out in the `target`.

3. Using another model to assess whether the model's completion satisfies a description of the ideal answer in `target`.

4. Using another rubric entirely (e.g. did the model produce a valid version of a file format, etc.)

Scorers also define one or more metrics which are used to aggregate scores (e.g. `accuracy()` which computes what percentage of scores are correct, or `mean()` which provides an average for scores that exist on a continuum).

7.2 Built-In Scorers

Inspect includes some simple text matching scorers as well as a couple of model graded scorers. Built in scorers can be imported from the `inspect_ai.scorer` module. Below is a summary of these scorers. There is not (yet) reference documentation on these functions so the best way to learn about how they can be customised, etc. is to use the **Go to Definition** command in your source editor.

- `includes()`

  Determine whether the `target` from the `Sample` appears anywhere inside the model output. Can be case sensitive or insensitive (defaults to the latter).

- `match()`

  Determine whether the `target` from the `Sample` appears at the beginning or end of model output (defaults to looking at the end). Has options for ignoring case, white-space, and punctuation (all are ignored by default).

- `pattern()`

  Extract the answer from model output using a regular expression.

- `answer()`

  Scorer for model output that preceded answers with "ANSWER:". Can extract letters, words, or the remainder of the line.

- `model_graded_qa()`

  Have another model assess whether the model output is a correct answer based on the grading guidance contained in `target`. Has a built-in template that can be customised.

- `model_graded_fact()`

  Have another model assess whether the model output contains a fact that is set out in `target`. This is a more narrow assessment than `model_graded_qa()`, and is used when model output is too complex to be assessed using a simple `match()` or `pattern()` scorer.

Scorers provide one or more built-in metrics (each of the scorers above provides `accuracy` as a metric). You can also provide your own custom metrics in `Task` definitions. For example:

```
Task(
    dataset=dataset,
    plan=[
        system_message(SYSTEM_MESSAGE),
        multiple_choice()
    ],
    scorer=match(),
    metrics=[custom_metric()]
)
```

Model Graded

Model graded scorers are well suited to assessing open ended answers as well as factual answers that are embedded in a longer narrative. The built-in model graded scorers can be customised in several ways—you can also create entirely new model scorers (see the model graded example below for a starting point).

Here is the declaration for the `model_graded_qa()` function:

```
@scorer(metrics=[accuracy(), bootstrap_std()])
def model_graded_qa(
    template: str | None = None,
    instructions: str | None = None,
    grade_pattern: str | None = None,
    partial_credit: bool = False,
    model: str | Model | None = None,
) -> Scorer:
    ...
```

The default model graded QA scorer is tuned to grade answers to open ended questions. The default `template` and `instructions` ask the model to produce a grade in the format `GRADE: C` or `GRADE: I`, and this grade is extracted using the default `grade_pattern` regular expression. The grading is by default done with the model currently being evaluated. There are a few ways you can customise the default behaviour:

1. Provide alternate `instructions`—the default instructions ass the model to use chain of thought reasoning and provide grades in the format `GRADE: C` or `GRADE: I`. Note that if you provide instructions that ask the model to format grades in a different way, you will also want to customise the `grade_pattern`.
2. Specify `partial_credit = True` to prompt the model to assign partial credit to answers that are not entirely right but come close (metrics by default convert this to a value of 0.5). Note that this parameter is only valid when using the default `instructions`.
3. Specify an alternate `model` to perform the grading (e.g. a more powerful model or a model fine tuned for grading).
4. Specify a different `template`—note that templates are passed these variables: `question`, `criterion`, `answer`, and `instructions`.

The `model_graded_fact()` scorer works identically to `model_graded_qa()`, and simply provides an alternate `template` oriented around judging whether a fact is included in the model output.

If you want to understand how the default templates for `model_graded_qa()` and `model_graded_fact()` work, see their source code.

7.3 Custom Scorers

Custom scorers are functions that take a `TaskState` and `Target`, and yield a `Score`.

```
async def score(state: TaskState, target: Target):
    # Compare state / model output with target
    # to yield a score
    return Score(value=...)
```

First we'll talk about the core `Score` and `Value` objects, then provide some examples of custom scorers to make things more concrete.

> **i** Note that `score()` above is declared as an `async` function. When creating custom scorers, it's critical that you understand Inspect's concurrency model. More specifically, if your scorer is doing non-trivial work (e.g. calling REST APIs, executing external processes, etc.) please review Eval Tuning before proceeding.

Score

The components of `Score` include:

Field	Type	Description
value	Value	Value assigned to the sample (e.g. "C" or "I", or a raw numeric value).
answer	str	Text extracted from model output for comparison (optional).
explanation	str	Explanation of score, e.g. full model output or grader model output (optional).
metadata	dict[str,Any]	Additional metadata about the score to record in the log file (optional).

For example, the following are all valid `Score` objects:

```
Score(value="C")
Score(value="I")
Score(value=0.6)
Score(
    value="C" if extracted == target.text else "I",
    answer=extracted,
    explanation=state.output.completion
)
```

If you are extracting an answer from within a completion (e.g. looking for text using a regex pattern, looking at the beginning or end of the completion, etc.) you should strive to *always* return an `answer` as part of your `Score`, as this makes it much easier to understand the details of scoring when viewing the eval log file.

Value

`Value` is union over the main scalar types as well as a `list` or `dict` of the same types:

```
Value = Union[
    str | int | float | bool,
    list[str | int | float | bool],
    dict[str, str | int | float | bool],
]
```

The vast majority of scorers will use `str` (e.g. for correct/incorrect via "C" and "I") or `float` (the other types are there to meet more complex scenarios). One thing to keep in mind is that whatever `Value` type you use in a scorer must be supported by the metrics declared for the scorer (more on this below).

Next, we'll take a look at the source code for a couple of the built in scorers as a jumping off point for implementing your own scorers. If you are working on custom scorers, you should also review the Scorer Workflow section below for tips on optimising your development process.

Example: Includes

Here is the source code for the built-in `includes()` scorer:

```
@scorer(metrics=[accuracy(), bootstrap_str()])                              ①
def includes(ignore_case: bool = True):

    async def score(state: TaskState, target: Target):                       ②

        # check for correct
        answer = state.output.completion
        target = target.text                                                  ③
        if ignore_case:
            correct = answer.lower().rfind(target.lower()) != -1
        else:
            correct = answer.rfind(target) != -1
```

```
        # return score
        return Score(
            value = CORRECT if correct else INCORRECT,        ④
            answer=answer                                     ⑤
        )

    return score
```

① The function applies the `@scorer` decorator and registers two metrics for use with the scorer.

② The `score()` function is declared as `async`. This is so that it can participate in Inspect's optimised scheduling for expensive model generation calls (this scorer doesn't call a model but others will).

③ We make use of the `text` property on the `Target`. This is a convenience property to get a simple text value out of the `Target` (as targets can technically be a list of strings).

④ We use the special constants `CORRECT` and `INCORRECT` for the score value (as the `accuracy()` and `bootstrap_std()` metrics know how to convert these special constants to float values (1.0 and 0.0 respectively).

⑤ We provide the full model completion as the answer for the score (`answer` is optional, but highly recommended as it is often useful to refer to during evaluation development).

Example: Model Grading

Here's a somewhat simplified version of the code for the `model_graded_qa()` scorer:

```
@scorer(metrics=[accuracy(), bootstrap_std()])
def model_graded_qa(
    template: str = DEFAULT_MODEL_GRADED_QA_TEMPLATE,
    instructions: str = DEFAULT_MODEL_GRADED_QA_INSTRUCTIONS,
    grade_pattern: str = DEFAULT_GRADE_PATTERN,
    model: str | Model | None = None,
) -> Scorer:

    # resolve grading template and instructions,
    # (as they could be file paths or URLs)
    template = resource(template)
    instructions = resource(instructions)

    # resolve model
    grader_model = get_model(model)
```

```python
    async def score(state: TaskState, target: Target) -> Score:
        # format the model grading template
        score_prompt = template.format(
            question=state.input_text,
            answer=state.output.completion,
            criterion=target.text,
            instructions=instructions,
        )

        # query the model for the score
        result = await grader_model.generate(score_prompt)

        # extract the grade
        match = re.search(grade_pattern, result.completion)
        if match:
            return Score(
                value=match.group(1),
                answer=match.group(0),
                explanation=result.completion,
            )
        else:
            return Score(
                value=INCORRECT,
                explanation="Grade not found in model output: "
                + f"{result.completion}",
            )

    return score
```

Note that the call to `model_grader.generate()` is done with `await`—this is critical to ensure that the scorer participates correctly in the scheduling of generation work.

Note also e use the `input_text` property of the `TaskState` to access a string version of the original user input to substitute it into the grading template. Using the `input_text` has two benefits: (1) It is guaranteed to cover the original input from the dataset (rather than a transformed prompt in `messages`); and (2) It normalises the input to a string (as it could have been a message list).

7.4 Metrics

Each scorer provides one or more built-in metrics (typically `accuracy` and `bootstrap_std`). In addition, you can specify other metrics (either built-in or custom) to compute when defining a `Task`:

```
Task(
    dataset=dataset,
    plan=[
        system_message(SYSTEM_MESSAGE),
        multiple_choice()
    ],
    scorer=match(),
    metrics=[custom_metric()]
)
```

Built-In Metrics

Inspect includes some simple built in metrics for calculating accuracy, mean, etc. Built in metrics can be imported from the `inspect_ai.scorer` module. Below is a summary of these metrics. There is not (yet) reference documentation on these functions so the best way to learn about how they can be customised, etc. is to use the **Go to Definition** command in your source editor.

- `accuracy()`

  Compute proportion of total answers which are correct. For correct/incorrect scores assigned 1 or 0, can optionally assign 0.5 for partially correct answers.

- `mean()`

  Mean of all scores.

- `var()`

  Variance over all scores.

- `bootstrap_std()`

  Standard deviation of a bootstrapped estimate of the mean. 1000 samples are taken by default (modify this using the `num_samples` option).

Custom Metrics

You can also add your own metrics with `@metric` decorated functions. For example, here is the implementation of the variance metric:

```python
import numpy as np

from inspect_ai.scorer import Metric, Score, metric

def var() -> Metric:
    """Compute variance over all scores."""

    def metric(scores: list[Score]) -> float:
        return np.var([score.as_float() for score in scores]).item()

    return metric
```

Note that the `Score` class contains a `Value` that is a union over several scalar and collection types. As a convenience, `Score` includes a set of accessor methods to treat the value as a simpler form (e.g. above we use the `score.as_float()` accessor).

7.5 Workflow

Score Command

By default, model output in evaluations is automatically scored. However, you can separate generation and scoring by using the `--no-score` option. For example:

```
inspect eval popularity.py --model openai/gpt-4 --no-score
```

You can score an evaluation previously run this way using the `inspect score` command:

```
# score last eval
inspect score popularity.py

# score specific log file
inspect score popularity.py ./logs/2024-02-23_task_gpt-4_TUhnCn473c6.json
```

> 💡 **Tip**
>
> Using a distinct scoring step is particularly useful during scorer development, as it bypasses the entire generation phase, saving lots of time and inference costs.

Log Overwriting

By default, `inspect score` overwrites the file it scores. If don't want to overwrite target files, pass the `--no-overwrite` flag:

```
inspect score popularity.py --no-overwrite
```

When specifying `--no-overwrite`, a `-scored` suffix will be added to the original log file name:

```
./logs/2024-02-23_task_gpt-4_TUhnCn473c6-scored.json
```

Note that the `--no-overwrite` flag does not apply to log files that already have the `-scored` suffix—those files are always overwritten by `inspect score`. If you plan on scoring multiple times and you want to save each scoring output, you will want to copy the log to another location before re-scoring.

Python API

If you are exploring the performance of different scorers, you might find it more useful to call the `score()` function using varying scorers or scorer options. For example:

```python
log = eval(popularity, model="openai/gpt-4")[0]

grader_models = [
    "openai/gpt-4",
    "anthropic/claude-3-opus-20240229",
    "google/gemini-1.0-pro",
    "mistral/mistral-large-latest"
]

scoring_logs = [score(log, model_graded_qa(model=model))
                for model in grader_models]

plot_results(scoring_logs)
```

8 Datasets

8.1 Overview

Inspect has native support for reading datasets in the CSV, JSON, and JSON Lines formats, as well as from Hugging Face. In addition, the core dataset interface for the evaluation pipeline is flexible enough to accept data read from just about any source.

If your data is already in a format amenable for direct reading as an Inspect `Sample`, reading a dataset is as simple as this:

```python
from inspect_ai.dataset import csv_dataset, json_dataset
dataset1 = csv_dataset("dataset1.csv")
dataset2 = json_dataset("dataset2.json")
```

Of course, many real-world datasets won't be so trivial to read. Below we'll discuss the various ways you can adapt your datasets for use with Inspect.

8.2 Dataset Samples

The core data type underlying the use of datasets with Inspect is the `Sample`. A sample has an `input`, a `target`, an optional `id`, and an optional collection of `metadata`.

Class `inspect_ai.dataset.Sample`

Field	Type	Description
input	str | list[ChatMessage]	The input to be submitted to the model.
choices	list[str] | None	Optional. Multiple choice answer list.
target	str | list[str] | None	Optional. Ideal target output. May be a literal value or narrative text to be used by a model grader.
id	str | None	Optional. Unique identifier for sample.

Field	Type	Description
`metadata`	`dict[str	Any]	None`	Optional. Arbitrary metadata associated with the sample.

So a CSV dataset with the following structure:

input	target
What cookie attributes should I use for strong security?	secure samesite and httponly
How should I store passwords securely for an authentication system database?	strong hashing algorithms with salt like Argon2 or bcrypt

Can be read directly with:

```
dataset = csv_dataset("security_guide.csv")
```

Note that samples from datasets without and `id` field will automatically be assigned ids based on an auto-incrementing integer starting with 1.

If your samples include `choices`, then the label should be a numeric index into the available `choices` rather a letter (this is an implicit assumption of the `multiple_choice()` solver).

8.3 Field Mapping

If your dataset contains inputs and targets that don't use `input` and `target` as field names, you can map them into a `Dataset` using a `FieldSpec`. This same mechanism also enables you to collect arbitrary additional fields into the `Sample metadata` bucket. For example:

```
from inspect_ai.dataset import FieldSpec, json_dataset

dataset = json_dataset(
    "popularity.jsonl",
    FieldSpec(
        input="question",
        target="answer_matching_behavior",
        id="question_id",
        metadata=["label_confidence"],
    ),
)
```

If you need to do more than just map field names and actually do custom processing of the data, you can instead pass a function which takes an `index` and `record` (represented as a `dict`) from the underlying file and returns a `Sample`. For example:

```python
from inspect_ai.dataset import Sample, json_dataset

def record_to_sample(record):
    return Sample(
        input=record["question"],
        target=record["answer_matching_behavior"].strip(),
        id=record["question_id"],
        metadata={
            "label_confidence": record["label_confidence"]
        }
    )

dataset = json_dataset("popularity.jsonl", record_to_sample)
```

8.4 Filter and Shuffle

The `Dataset` class includes `filter()` and `shuffle()` methods, as well as support for the slice operator.

To select a subset of the dataset, use `filter()`:

```python
dataset = json_dataset("popularity.jsonl", record_to_sample)
dataset = dataset.filter(
    lambda sample : sample.metadata["category"] == "advanced"
)
```

To select a subset of records, use standard Python slicing:

```python
dataset = dataset[0:100]
```

Shuffling is often helpful when you want to vary the samples used during evaluation development. To do this, either use the `shuffle()` method or the `shuffle` parameter of the dataset loading functions:

```
# shuffle method
dataset = dataset.shuffle()

# shuffle on load
dataset = json_dataset("data.jsonl", shuffle=True)
```

Note that both of these methods optionally support specifying a random seed for shuffling.

8.5 Hugging Face

Hugging Face Datasets is a library for easily accessing and sharing datasets for machine learning, and features integration with Hugging Face Hub, a repository with a broad selection of publicly shared datasets. Typically datasets on Hugging Face will require specification of which split within the dataset to use (e.g. train, test, or validation) as well as some field mapping. Use the `hf_dataset()` function to read a dataset and specify the requisite split and field names:

```
from inspect_ai.dataset import FieldSpec, hf_dataset

dataset=hf_dataset("openai_humaneval",
  split="test",
  sample_fields=FieldSpec(
    id="task_id",
    input="prompt",
    target="canonical_solution",
    metadata=["test", "entry_point"]
  )
)
```

Note that some HuggingFace datasets execute Python code in order to resolve the underlying dataset files. Since this code is run on your local machine, you need to specify `trust = True` in order to perform the download. This option should only be set to `True` for repositories you trust and in which you have read the code. Here's an example of using the `trust` option (note that it defaults to `False` if not specified):

```
dataset=hf_dataset("openai_humaneval",
  split="test",
  trust=True,
  ...
)
```

Under the hood, the `hf_dataset()` function is calling the [load_dataset()](#) function in the Hugging Face datasets package. You can additionally pass arbitrary parameters on to `load_dataset()` by including them in the call to `hf_dataset()`. For example `hf_dataset(..., cache_dir="~/my-cache-dir")`.

8.6 Amazon S3

Inspect has integrated support for storing datasets on [Amazon S3](#). Compared to storing data on the local file-system, using S3 can provide more flexible sharing and access control, and a more reliable long term store than local files.

Using S3 is mostly a matter of substituting S3 URLs (e.g. `s3://my-bucket-name`) for local file-system paths. For example, here is how you load a dataset from S3:

```
json_dataset("s3://my-bucket/dataset.jsonl")
```

S3 buckets are normally access controlled so require authentication to read from. There are a wide variety of ways to configure your client for AWS authentication, all of which work with Inspect. See the article on [Configuring the AWS CLI](#) for additional details

8.7 Chat Messages

The most important data structure within `Sample` is the `ChatMessage`. Note that often datasets will contain a simple string as their input (which is then internally converted to a `ChatMessageUser`). However, it is possible to include a full message history as the input via `ChatMessage`. Another useful application of `ChatMessage` is providing multi-modal input (e.g. images).

Class `inspect_ai.model.ChatMessage`

Field	Type	Description
role	`"system" | "user" | "assistant" | "tool"`	Role of this chat message.
content	`str | list[ChatContent]`	The content of the message. Can be a simple string or a list of content parts intermixing text and images.

An input with chat messages in your dataset might will look something like this:

```
"input": [
  {
    "role": "user",
    "content": "What cookie attributes should I use for strong security?"
  }
]
```

Note that for this example we wouldn't normally use a full chat message object (rather we'd just provide a simple string). Chat message objects are more useful when you want to include a system prompt or prime the conversation with "assistant" responses.

8.8 Image Input

To include an image, your dataset input would look like this:

```
"input": [
  {
    "role": "user",
    "content": [
        { "type": "text", "text": "What is this a picture of?"},
        { "type": "image", "image": "picture.png"}
    ]
  }
]
```

Where `"picture.png"` is located in the directory where your task runs. The image can be specified either as a URL (accessible to the model), a local file path, or a base64 encoded Data URL.

If you are constructing chat messages programmatically, then the equivalent to the above would be:

```
ChatMessageUser(content = [
    ContentText(text="What is this a picture of?"),
    ContentImage(image="picture.png")
])
```

> **i** Note that image input is currently only supported for Open AI vision models (e.g. gpt-4-vision-preview), Google Gemini vision models (e.g. gemini-pro-vision), and Anthropic Claude 3 models.

8.9 Custom Reader

You are not restricted to the built in dataset functions for reading samples. Since the `dataset` field of the `Task` class takes either a `Dataset` or a sequences of `Sample`, the following is also valid:

```python
from inspect_ai import Task, task
from inspect_ai.dataset import Sample
from inspect_ai.scorer import model_graded_fact
from inspect_ai.solver import generate, system_message

dataset=[
    Sample(
        input="What cookie attributes should I use for strong security?",
        target="secure samesite and httponly",
    )
]

@task
def security_guide():
    return Task(
        dataset=dataset,
        plan=[system_message(SYSTEM_MESSAGE), generate()],
        scorer=model_graded_fact(),
    )
```

So if the built in dataset functions don't meet your needs, you can create a custom function that yields a list of `Sample` instances and pass those directly to your `Task`.

9 Models

9.1 Overview

Inspect has built in support for a variety of language model API providers and can be extended to support arbitrary additions ones. Built-in model API providers, their dependencies, and environment variables required to use them are as follows:

Model API	Dependencies	Environment Variables
OpenAI	`pip install openai`	`OPENAI_API_KEY`
Anthropic	`pip install anthropic`	`ANTHROPIC_API_KEY`
Google	`pip install google-generativeai`	`GOOGLE_API_KEY`
Mistral	`pip install mistralai`	`MISTRAL_API_KEY`
Hugging Face	`pip install transformers`	`HF_TOKEN`
TogetherAI	`pip install openai`	`TOGETHER_API_KEY`
AWS Bedrock	`pip install boto3`	`AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_DEFAULT_REGION`
Azure AI	None required	`AZURE_API_KEY` and `INSPECT_EVAL_MODEL_BASE_URL`
CloudFlare	None required	`CLOUDFLARE_ACCOUNT_ID` and `CLOUDFLARE_API_TOKEN`

9.2 Using Models

To select a model for use in an evaluation task you specify it using a *model name*. Model names include their API provider and the specific model to use (e.g. `openai/gpt-4`) Here are the supported providers along with example model names and links to documentation on all available models:

Provider	Model Name	Docs
OpenAI	`openai/gpt-3.5-turbo`	OpenAI Models
Anthropic	`anthropic/claude-2.1`	Anthropic Models

Provider	Model Name	Docs
Google	`google/gemini-1.0-pro`	Google Models
Mistral	`mistral/mistral-large-latest`	Mistral Models
Hugging Face	`hf/openai-community/gpt2`	Hugging Face Models
TogetherAI	`together/lmsys/vicuna-13b-v1.5`	TogetherAI Models
AWS Bedrock	`bedrock/meta.llama2-70b-chat-v1`	AWS Bedrock Models
Azure AI	`azureai/azure-deployment-name`	Azure AI Models
CloudFlare	`cf/meta/llama-2-7b-chat-fp16`	CloudFlare Models

To select a model for an evaluation, pass it's name on the command line or use the `model` argument of the `eval()` function:

```
$ inspect eval security_guide --model openai/gpt-3.5-turbo
$ inspect eval security_guide --model anthropic/claude-instant-1.2
```

Or:

```
eval(security_guide, model="openai/opeangpt-3.5-turbo")
eval(security_guide, model="anthropic/claude-instant-1.2")
```

Alternatively, you can set the `INSPECT_EVAL_MODEL` environment variable (either in the shell or a `.env` file) to select a model externally:

```
INSPECT_EVAL_MODEL=google/gemini-1.0-pro
```

> **i** If are using Azure AI, AWS Bedrock, or Hugging Face, you should additionally consult the sections below on using the Azure AI, AWS Bedrock, and Hugging Face providers to learn more about available models and their usage and authentication requirements.

Model Base URL

Each model also can use a different base URL than the default (e.g. if running through a proxy server). The base URL can be specified with the same prefix as the `API_KEY`, for example, the following are all valid base URLs:

Provider	Environment Variable
OpenAI	`OPENAI_BASE_URL`

Provider	Environment Variable
Anthropic	`ANTHROPIC_BASE_URL`
Google	`GOOGLE_BASE_URL`
Mistral	`MISTRAL_BASE_URL`
TogetherAI	`TOGETHER_BASE_URL`
AWS Bedrock	`BEDROCK_BASE_URL`
Azure AI	`AZUREAI_BASE_URL`
CloudFlare	`CLOUDFLARE_BASE_URL`

In addition, there are separate base URL variables for running various frontier models on Azure and Bedrock:

Provider (Model)	Environment Variable
AzureAI (OpenAI)	`AZUREAI_OPENAI_BASE_URL`
AzureAI (Mistral)	`AZUREAI_MISTRAL_BASE_URL`
Bedrock (Anthropic)	`BEDROCK_ANTHROPIC_BASE_URL`

9.3 Generation Config

There are a variety of configuration options that affect the behaviour of model generation. There are options which affect the generated tokens (`temperature`, `top_p`, etc.) as well as the connection to model providers (`timeout`, `max_retries`, etc.)

You can specify generation options either on the command line or in direct calls to `eval()`. For example:

```
$ inspect eval --model openai/gpt-4 --temperature 0.9
$ inspect eval --model google/gemini-1.0-pro --max-connections 20
```

Or:

```
eval(security_guide, model="openai/gpt-4", temperature=0.9)
eval(security_guide, model="google/gemini-1.0-pro", max_connections=20)
```

Use `inspect eval --help` to learn about all of the available generation config options. |

Connections and Rate Limits

Inspect uses an asynchronous architecture to run task samples in parallel. If your model provider can handle 100 concurrent connections, then Inspect can utilise all of those connections to get the highest possible throughput. The limiting factor on parallelism is therefore not typically local parallelism (e.g. number of cores) but rather what the underlying rate limit is for your interface to the provider.

If you are experiencing rate-limit errors you will need to experiment with the `max_connections` option to find the optimal value that keeps you under the rate limit (the section on Eval Tuning includes additional documentation on how to do this). Note that the next section describes how you can set a model-provider specific value for `max_connections` as well as other generation options.

Model Specific Configuration

In some cases you'll want to vary generation configuration options by model provider. You can do this by adding a `model` argument to your task function. You can use the `model` in a pattern matching statement to condition on different models. For example:

```python
@task
def popularity(model):
    # condition temperature on model
    config = GenerateConfig()
    match model:
        case "gpt" | "gemini":
            config.temperature = 0.9
        case "claude":
            config.temperature = 0.8

    return Task(
        dataset=json_dataset("popularity.jsonl"),
        plan=[system_message(SYSTEM_MESSAGE), generate()],
        scorer=match(),
        config=config,
    )
```

9.4 Provider Notes

This section provides additional documentation on using the Azure AI, AWS Bedrock, and Hugging Face providers.

Azure AI

[Azure AI](#) provides hosting of models from OpenAI and Mistral as well as a wide variety of other open models. One special requirement for models hosted on Azure is that you need to specify a model base URL. You can do this using the `AZUREAI_OPENAI_BASE_URL` and `AZUREAI_MISTRAL_BASE_URL` environment variables or the `--model-base-url` command line parameter. You can find the model base URL for your specific deployment in the Azure model admin interface.

OpenAI

To use OpenAI models on Azure AI, specify an `AZUREAI_OPENAI_API_KEY` along with an `AZUREAI_OPENAI_BASE_URL`. You can then use the normal `openai` provider, but you'll need to specify a model name that corresponds to the [Azure Deployment Name](#) of your model. For example, if your deployed model name was `gpt4-1106-preview-ythre`:

```
$ export AZUREAI_OPENAI_API_KEY=key
$ export AZUREAI_OPENAI_BASE_URL=https://your-url-at.azure.com
$ inspect eval --model openai/gpt4-1106-preview-ythre
```

The complete list of environment variables (and how they map to the parameters of the `AzureOpenAI` client) is as follows:

- `api_key` from `AZUREAI_OPENAI_API_KEY`
- `azure_endpoint` from `AZUREAI_OPENAI_BASE_URL`
- `organization` from `OPENAI_ORG_ID`
- `api_version` from `OPENAI_API_VERSION`

Mistral

To use Mistral models on Azure AI, specify an `AZURE_MISTRAL_API_KEY` along with an `INSPECT_EVAL_MODEL_BASE_URL`. You can then use the normal `mistral` provider, but you'll need to specify a model name that corresponds to the [Azure Deployment Name](#) of your model. For example, if your deployment model name was `mistral-large-ctwi`:

```
$ export AZUREAI_MISTRAL_API_KEY=key
$ export AZUREAI_MISTRAL_BASE_URL=https://your-url-at.azure.com
$ inspect eval --model mistral/mistral-large-ctwi
```

Other Models

Azure AI supports many other model types, you can access these using the `azureai` model provider. As with OpenAI and Mistral, you'll need to specify an `AZUREAI_API_KEY` along with an `AZUREAI_BASE_URL`, as well as use the the [Azure Deployment Name](#) of your model as the model name. For example:

```
$ export AZUREAI_API_KEY=key
$ export AZUREAI_BASE_URL=https://your-url-at.azure.com
$ inspect eval --model azureai/llama-2-70b-chat-wnsnw
```

AWS Bedrock

[AWS Bedrock](#) provides hosting of models from Anthropic as well as a wide variety of other open models. Note that all models on AWS Bedrock require that you [request model access](#) before using them in a deployment (in some cases access is granted immediately, in other cases it could one or more days).

You should be sure that you have the appropriate AWS credentials before accessing models on Bedrock. Once credentials are configured, use the `bedrock` provider along with the requisite Bedrock model name. For example, here's how you would access models from a variety of providers:

```
$ export AWS_ACCESS_KEY_ID=ACCESSKEY
$ export AWS_SECRET_ACCESS_KEY=SECRETACCESSKEY
$ export AWS_DEFAULT_REGION=us-east-1

$ inspect eval bedrock/anthropic.claude-3-haiku-20240307-v1:0
$ inspect eval bedrock/mistral.mistral-7b-instruct-v0:2
$ inspect eval bedrock/meta.llama2-70b-chat-v1
```

You aren't likely to need to, but you can also specify a custom base URL for AWS Bedrock using the `BEDROCK_BASE_URL` environment variable.

Hugging Face

The Hugging Face provider implements support for local models using the [transformers](#) package. You can use any Hugging Face model by specifying it with the `hf/` prefix. For example:

```
$ inspect eval popularity --model hf/openai-community/gpt2
```

Batching

Concurrency for REST API based models is managed using the `max_connections` option. The same option is used for `transformers` inference—up to `max_connections` calls to `generate()` will be batched together (note that batches will proceed at a smaller size if no new calls to `generate()` have occurred in the last 2 seconds).

The default batch size for Hugging Face is 32, but you should tune your `max_connections` to maximise performance and ensure that batches don't exceed available GPU memory. The Pipeline Batching section of the transformers documentation is a helpful guide to the ways batch size and performance interact.

Device

The PyTorch `cuda` device will be used automatically if CUDA is available (as will the Mac OS `mps` device). If you want to override the device used, use the `device` model argument. For example:

```
$ inspect eval popularity --model hf/openai-community/gpt2 -M device=cuda:0
```

This also works in calls to `eval()`:

```
eval(popularity, model="hf/openai-community/gpt2", model_args=dict(device="cuda:0"))
```

Or in a call to `get_model()`

```
model = get_model("hf/openai-community/gpt2", device="cuda:0")
```

Local Models

In addition to using models from the Hugging Face Hub, the Hugging Face provider can also use local model weights and tokenizers (e.g. for a locally fine tuned model). Use `hf/local` along with the `model_path`, and (optionally) `tokenizer_path` arguments to select a local model. For example, from the command line, use the `-M` flag to pass the model arguments:

```
$ inspect eval popularity --model hf/local -M model_path=./my-model
```

Or using the `eval()` function:

```
eval(popularity, model="hf/local", model_args=dict( model_path="./my-model"))
```

Or in a call to `get_model()`

```
model = get_model("hf/local", model_path="./my-model")
```

9.5 Helper Models

Often you'll want to use language models in the implementation of Solvers and Scorers. Inspect includes some critique solvers and model graded scorers that do this, and you'll often want to do the same in your own.

Helper models will by default use the same model instance and configuration as the model being evaluated, however this can be overridden using the `model` argument.

```
self_critique(model = "google/gemini-1.0-pro")
```

You can also pass a fully instantiated `Model` object (for example, if you wanted to override its default configuration) by using the `get_model()` function. For example, here we'll provide custom models for both critique and scoring:

```python
from inspect_ai import Task, task
from inspect_ai.dataset import json_dataset
from inspect_ai.model import GenerationConfig, get_model
from inspect_ai.scorer import model_graded_fact
from inspect_ai.solver import chain_of_thought, generate, self_critique

@task
def theory_of_mind():

  critique_model = get_model("google/gemini-1.0-pro")

  grader_model = get_model("anthropic/claude-2.1", config = GenerationConfig(
    temperature = 0.9,
    max_connections = 10
  ))

  return Task(
```

```
    dataset=json_dataset("theory_of_mind.jsonl"),
    plan=[
        chain_of_thought(),
        generate(),
        self_critique(model = critique_model)
    ],
    scorer=model_graded_fact(model = grader_model),
)
```

9.6 Model Args

The section above illustrates passing model specific arguments to local models on the command line, in `eval()`, and in `get_model()`. This actually works for all model types, so if there is an additional aspect of a modal you want to tweak that isn't covered by the `GenerationConfig`, you can use this method to do it. For example, here we specify the `transport` option for a Google Gemini model:

```
inspect eval popularity --model google/gemini-1.0-pro -M transport:grpc
```

The additional `model_args` are forwarded as follows for the various providers:

Provider	Forwarded to
OpenAI	`AsyncOpenAI`
Anthropic	`AsyncAnthropic`
Google	`genai.configure`
Mistral	`MistralAsyncClient`
Hugging Face	`AutoModelForCausalLM.from_pretrained`
TogetherAI	`AsyncOpenAI`
AzureAI	Chat HTTP Post Body
CloudFlare	Chat HTTP Post Body

See the OpenAI, Anthropic, Google, Mistral, Hugging Face, TogetherAI, Azure AI, and Cloud-Flare provider documentation for more information on the additional options available.

9.7 Custom Models

You can add a model provider by deriving a new class from `ModelAPI` and adding the `@modelapi` decorator to it. For example:

```python
@modelapi(name="custom")
class CustomModelAPI(ModelAPI):
    def __init__(
        self,
        model_name: str,
        base_url: str | None = None,
        config: GenerateConfig = GenerateConfig(),
        **model_args: dict[str,Any]
    ) -> None:
        super().__init__(model_name, base_url, config)

    async def generate(
        self,
        input: list[ChatMessage],
        tools: list[ToolInfo],
        tool_choice: ToolChoice,
        config: GenerateConfig,
    ) -> ModelOutput:
        ...
```

The `__init__()` method *must* call the `super().__init__()` method, and typically instantiates the model client library.

The `generate()` method handles interacting with the model. In addition, there are some optional methods you can override to specify various behaviours and constraints (default max tokens and connections, identifying rate limit errors, etc.)

Once you've created the class and decorated it with `@modelapi` as shown above, you can reference it as follows:

```python
# get a model instance
model = get_model("custom/name-of-model")

# run an eval with the model
eval(math, model = "custom/name-of-model")
```

In this example, the `model_name` argument passed to `__init__()` will be "name-of-model".

Part III

Advanced

10 Eval Logs

10.1 Overview

Every time you use `inspect eval` or call the `eval()` function, an evaluation log is written for each task evaluated. By default, logs are written to the `./logs` sub-directory of the current working directory (we'll cover how to change this below). You will find a link to the log at the bottom of the results for each task:

```
$ inspect eval security_guide.py --model openai/gpt-4
```

```
┌─ security_guide (16 samples) ─────────────────────────────────────┐
│                                                      openai/gpt-4  │
│ total time:     0:00:25                       dataset: security_guide │
│ openai/gpt-4    18,944 tokens [10,210 + 8,734]   scorer: model_graded_fact │
│                                                                   │
│ accuracy: 0.812                                                   │
│                                                                   │
│ Log: ./logs/2024-04-07T07:38:09_security-guide_kgzgj4aD9kvx7E8j2FvJTr.json │
└───────────────────────────────────────────────────────────────────┘
```

You can also use the Inspect log viewer for interactive exploration of logs. Run this command once at the beginning of a working session (the view will update automatically when new evaluations are run):

```
$ inspect view
```

security_guide openai/gpt-4

4/28/2024, 8:47:10 PM— 24 sec

	accuracy	bootstrap_std
	0.625	**0.120**

DATASET
security_guide — 16 samples

PLAN
system_message → generate

SCORER
model_graded_fact

Samples | Info | Logging | JSON

Scores: All   Sort: sample asc   ÷ Open All

	Input	Target	Answer	Score
1	How do I prevent SQL Injection attacks?	use parameterized queries and prepared statements	To prevent SQL Injection attacks, use parameterized...	C
2	How do I prevent sqli?	use parameterized queries and prepared statements	You can prevent SQL Injection (SQLi) by taking several...	C
3	How do I prevent cross site scripting?	output encoding with secure by default libraries and input...	To prevent Cross Site Scripting (XSS), implement proper...	C
4	How do I prevent xss?	output encoding with secure by default libraries and input...	To prevent XSS attacks, you should use proper output...	C
5	How do I prevent command injection?	parameterized commands and input validation	Preventing command injection involves various strategies: 1...	C
6	How do I prevent cmd injection?	parameterized commands and input validation	To prevent command injection, adopt the techniques of input...	I
7	What algorithm should I use to encrypt data?	AES (Advanced Encryption Standard) in AEAD mode such...	The algorithm choice depends on your specific needs....	I
8	How should I store passwords securely for an authentication system database?	strong hashing algorithms with salt like Argon2 or bcrypt	Passwords should be stored securely using a process called...	C

This section won't cover using `inspect view` though. Rather, it will cover the details of managing log usage from the CLI as well as the Python API for reading logs. See the Log Viewer section for details on interactively exploring logs.

10.2 Log Location

By default, logs are written to the `./logs` sub-directory of the current working directory You can change where logs are written using eval options or an environment variable

```
$ inspect eval popularity.py --model openai/gpt-4 --log-dir ./experiment-log
```

Or:

```
log = eval(popularity, model="openai/gpt-4", log_dir = "./experiment-log")
```

Note that in addition to logging the `eval()` function also returns an `EvalLog` object for programmatic access to the details of the evaluation. We'll talk more about how to use this object below.

The `INSPECT_LOG_DIR` environment variable can also be specified to override the default `./logs` location. You may find it convenient to define this in a `.env` file from the location where you run your evals:

```
INSPECT_LOG_DIR=./experiment-log
INSPECT_LOG_LEVEL=warning
```

If you define a relative path to `INSPECT_LOG_DIR` in a `.env` file, then its location will always be resolved as *relative to* that `.env` file (rather than relative to whatever your current working directory is when you run `inspect eval`).

> **i** If you are running in VS Code, then you should restart terminals and notebooks using Inspect when you change the `INSPECT_LOG_DIR` in a `.env` file. This is because the VS Code Python extension also reads variables from `.env` files, and your updated `INSPECT_LOG_DIR` won't be re-read by VS Code until after a restart.

See the Amazon S3 section below for details on logging evaluations to Amazon S3 buckets.

10.3 EvalLog

The `EvalLog` object returned from `eval()` provides programmatic interface to the contents of log files:

Class `inspect_ai.log.EvalLog`

Field	Type	Description
status	str	Status of evaluation ("started", "success", or "error").
eval	EvalSpec	Top level eval details including task, model, creation time, etc.
plan	EvalPlan	List of solvers and model generation config used for the eval.
samples	list[EvalSample]	Each sample evaluated, including its input, output, target, and score.

Field	Type	Description
results	EvalResults	Aggregate results computed by scorer metrics.
stats	EvalStats	Model usage statistics (input and output tokens)
logging	list[LoggingMessage]	Logging messages (e.g. from `log.info()`, `log.debug()`, etc.
error	EvalError	Error information (if `status == "error`) including traceback.

Before analysing results from a log, you should always check their status to ensure they represent a successful run:

```
log = log = eval(popularity, model="openai/gpt-4")
if log.status == "success":
    ...
```

In the section below we'll talk more about how to deal with logs from failed evaluations (e.g. retrying the eval).

You can enumerate, read, and write `EvalLog` objects using the following helper functions from the `inspect_ai.log` module:

Function	Description
list_eval_logs()	List all of the eval logs at a given location.
read_eval_log(log_file)	Read an `EvalLog` from a log file path.
write_eval_log(log, log_file)	Write an `EvalLog` to a log file path.

A common workflow is to define an `INSPECT_LOG_DIR` for running a set of evaluations, then calling `list_eval_logs()` to analyse the results when all the work is done:

```
# setup log dir context
os.environ["INSPECT_LOG_DIR"] = "./experiment-logs"

# do a bunch of evals
eval(popularity, model="openai/gpt-4")
eval(security_guide, model="openai/gpt-4")

# analyze the reuslts in the logs
logs = list_eval_logs()
```

Note that `list_eval_logs()` lists log files recursively. Pass `recursive=False` to list only the log files at the root level.

10.4 Errors and Retries

The example above isn't quite complete as it doesn't demonstrate checking the log for success status. This also begs the question of what to do with failed evaluation tasks. In some cases failed tasks need further debugging, but in other cases they may have failed due to connectivity or API rate limiting. For these cases, Inspect includes an `eval_retry()` function that you can pass a log to.

Here's an example of checking for logs with errors and retrying them with a lower number of max connections(the theory in this case being that too many concurrent connections may have caused a rate limit error:

```
logs = list_eval_logs(filter = lambda log: log.status == "error")
eval_retry(logs, max_connections = 3)
```

10.5 Amazon S3

Storing evaluation logs on S3 provides a more permanent and secure store than using the local filesystem. While the `inspect eval` command has a `--log-dir` argument which accepts an S3 URL, the most convenient means of directing inspect to an S3 bucket is to add the `INSPECT_LOG_DIR` environment variable to the `.env` file (potentially alongside your S3 credentials). For example:

```
INSPECT_LOG_DIR=s3://my-s3-inspect-log-bucket
AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
AWS_DEFAULT_REGION=eu-west-2
```

One thing to keep in mind if you are storing logs on S3 is that they will no longer be easily viewable using a local text editor. You will likely want to configure a FUSE filesystem so you can easily browse the S3 logs locally.

10.6 Log CLI Commands

We've shown a number of Python functions that let you work with eval logs from code. However, you may be writing an orchestration or visualisation tool in another language (e.g. Typescript) where its not particularly convenient to call the Python API. The Inspect CLI has a few commands intended to make it easier to work with Inspect logs from other languages.

Listing Logs

You can use the `inspect list logs` command to enumerate all of the logs for a given log directory. This command will utilise the `INSPECT_LOG_DIR` if it is set (alternatively you can specify a `--log-dir` directly). You'll likely also want to use the `--json` flag to get more granular and structured information on the log files. For example:

```
$ inspect list logs --json          # uses INSPECT_LOG_DIR
$ inspect list logs --json --log-dir ./security_04-07-2024
```

You can also use the `--status` option to list only logs with a `success` or `error` status:

```
$ inspect list logs --json --status success
$ inspect list logs --json --status error
```

Reading Logs

The `inspect list logs` command will return set of URIs to log files which will use a variety of protocols (e.g. `file://`, `s3://`, `gcs://`, etc.). You might be tempted to try to read these URIs directly, however you should always do so using the `inspect info log-file` command. This is because log files can be located on remote storage systems (e.g. Amazon S3) that users have configured read/write credentials for within their Inspect environment, and you'll want to be sure to take advantage of these credentials.

For example, here we read a local log file and a log file on Amazon S3:

```
$ inspect info log-file file:///home/user/log/logfile.json
$ inspect info log-file s3://my-evals-bucket/logfile.json
```

Log files are stored in JSON. You can get the JSON schema and Typescript type definitions for the log file format with the following calls to `inspect info`:

```
$ inspect info log-schema
$ inspect info log-types
```

> **❗ NaN and Inf**
>
> Because evaluation logs contain lots of numerical data and calculations, it is possible that some `number` values will be `NaN` or `Inf`. These numeric values are supported natively by Python's JSON parser, however are not supported by the JSON parsers built in to browsers and Node JS.
>
> To correctly read `Nan` and `Inf` values from eval logs in JavaScript, we recommend that you use the JSON5 Parser. For other languages, `Nan` and `Inf` may be natively supported (if not, see these JSON 5 implementations for other languages).

11 Eval Suites

11.1 Overview

Most of the examples in the documentation run a single evaluation task by either passing a script name to `inspect eval` or by calling the `eval()` function directly. While this is a good workflow for developing evaluations, once you've settled on a group of evaluations you want to run frequently, you'll typically want to run them all together as an evaluation suite. Below we'll cover the various tools and techniques available to create eval suites.

11.2 Prerequisites

Before describing the various ways you can define and run eval suites, we'll cover some universal prerequisites related to logging and task definitions.

Logging Context

A precursor to running any evaluation suite is to establish an isolated logging context for it. This enables you to enumerate and analyse all of the eval logs in the suite as a cohesive whole (rather than having them intermixed with the results of other runs). Generally, you'll do this by setting the `INSPECT_LOG_DIR` prior to running the suite. For example:

```
export INSPECT_LOG_DIR = ./security-mistral_04-07-2024
export INSPECT_EVAL_MODEL = mistral/mistral-large-latest
inspect eval security
```

This will group all of the log files for the suite, enabling you to call `list_eval_logs()` to collect and analyse all of the tasks.

Task Definitions

Whether you are working on evaluations in Python scripts or Jupyter Notebooks, you likely have a lot of code that looks roughly like this:

```python
@task
def security_guide():
    return Task(
        dataset=example_dataset("security_guide"),
        plan=[
          system_message(SYSTEM_MESSAGE),
          generate()
        ],
        scorer=model_graded_fact(),
    )

eval(security_guide, model="google/gemini-1.0-pro")
```

This is a natural and convenient way to run evals during development, but in a task suite you'll want `inspect eval` to do the execution rather than direct calls to `eval()` (as this allows for varying the model, generation config, and task parameters dynamically). You can keep your existing code more or less as-is, but you'll just want to add one line above `eval()`:

```python
if __name__ == "__main__":
    eval(security_guide, model="google/gemini-1.0-pro")
```

Doing this allows your source file to be both a Python script that is convenient to run during development as well as be a Python module that tasks can be read from without executing the eval. There is no real downside to this, and it's a good way in general to write all of your eval scripts and notebooks (see the docs on ___main___ for additional details).

11.3 Use Cases

Multiple Tasks in a File

The simplest possible eval suite would be multiple tasks defined in a single source file. Consider this source file (`ctf.py`) with two tasks in it:

```
@task
def jeopardy():
  return Task(
    ...
  )

@task
def attack_defense():
  return Task(
    ...
  )
```

We can run both of these tasks with the following command (note for this and the remainder of examples we'll assume that you have let an `INSPECT_EVAL_MODEL` environment variable so you don't need to pass the `--model` argument explicitly).

```
$ inspect eval ctf.py
```

Note we could also run the tasks individually as follows (e.g. for development and debugging):

```
$ inspect eval ctf.py@jeopardy
$ inspect eval ctf.py@attack_defense
```

Multiple Tasks in a Directory

Next, let's consider a multiple tasks in a directory. Imagine you have the following directory structure, where `jeopardy.py` and `attack_defense.py` each have one or more `@task` functions defined:

```
security/
  import.py
  analyze.py
  jeopardy.py
  attack_defense.py
```

Here is the listing of all the tasks in the suite:

```
$ inspect list tasks security
jeopardy.py@crypto
jeopardy.py@decompile
jeopardy.py@packet
jeopardy.py@heap_trouble
attack_defense.py@saar
attack_defense.py@bank
attack_defense.py@voting
attack_defense.py@dns
```

You can run this eval suite as follows:

```
$ inspect eval security
```

Note that some of the files in this directory don't contain evals (e.g. `import.py` and `analyze.py`). These files are not read or executed by `inspect eval` (which only executes files that contain `@task` definitions).

If we wanted to run more than one directory we could do so by just passing multiple directory names. For example:

```
$ inspect eval security pursuasion
```

Eval Function

Note that all of the above example uses of `inspect eval` apply equally to the `eval()` function. in the context of the above, all of these statements would work as expected:

```
eval("ctf.py")
eval("ctf.py@jeopardy")
eval("ctf.py@attack_defense")

eval("security")
eval(["security", "pursuasion"])
```

11.4 Listing and Filtering

Recursive Listings

Note that directories or expanded globs of directory names passed to `eval` are recursively scanned for tasks. So you could have a very deep hierarchy of directories, with a mix of

task and non task scripts, and the `eval` command or function will discover all of the tasks automatically.

There are some rules for how recursive directory scanning works that you should keep in mind:

1. Sources files and directories that start with . or _ are not scanned for tasks.
2. Directories named `env`, `venv`, and `tests` are not scanned for tasks.

Attributes and Filters

Eval suites will sometimes be defined purely by directory structure, but there will be cross-cutting concerns that are also used to filter what is run. For example, you might want to define some tasks as part of a "light" suite that is less expensive and time consuming to run. This is supported by adding attributes to task decorators. For example:

```python
@task(light=True)
def jeopardy():
  return Task(
    ...
  )
```

Given this, you could list all of the light tasks in `security` and pass them to `eval()` as follows:

```python
light_suite = list_tasks(
  "security",
  filter = lambda task: task.attribs.get("light") is True
)
logs = eval(light_suite)
```

Note that the `inspect list tasks` command can also be used to enumerate tasks in plain text or JSON (use one or more `-F` options if you want to filter tasks):

```
$ inspect list tasks security
$ inspect list tasks security --json
$ inspect list tasks security --json -F light=true
```

> **!** One important thing to keep in mind when using attributes to filter tasks is that both `inspect list tasks` (and the underlying `list_tasks()` function) do not execute code when scanning for tasks (rather they parse it). This means that if you want to use a task attribute in a filtering expression it needs to be a constant (rather than the result of function call). For example:
>
> ```python
> # this is valid for filtering expressions
> @task(light=True)
> def jeopardy():
>    ...
>
> # this is NOT valid for filtering expressions
> @task(light=True and light_enabled("ctf"))
> def jeopardy():
>    ...
> ```

11.5 Errors and Retries

If a runtime error occurs during an evaluation, it is caught, logged, and reported, and then the `eval()` function returns as normal. The returned `EvalLog` has a `status` field on it which can checked for `"success"` or `"error"`.

This status can be used to see which tasks need to be retried, and the failed log file can be passed directly to `eval()`, for example:

```python
# list the security suite and run it
task_suite = list_tasks("security")
eval_logs = eval(task_suite)

# check for failed evals and retry (likely 'later')
error_logs = log in eval_logs if log.status == "error"]
eval_retry(error_logs)
```

Note that the code which checks for errors will often not be in the same script as that which kicks off the evals. You can handle this by using the log directory as the reference point rather than the logs returned from `eval()`. Returning to the example from the beginning of this article, we might do something like this:

```python
# setup log context
os.environ["INSPECT_LOG_DIR"] = "./security-mistral_04-07-2024"
```

```
# run the eval suite
eval("security", model="mistral/mistral-large-latest")

# ...later, in another process that also has access to INSPECT_LOG_DIR
error_logs = list_eval_logs(filter = lambda log: log.status == "error")
eval_retry(error_logs)
```

12 Eval Tuning

12.1 Overview

Inspect runs evaluations using a highly parallel async architecture. Rather than processing a batch at a time, all samples are processed concurrently. This is possible because evaluations generally use relatively little local compute, but rather spend most of their time waiting for model API calls and web requests to complete. Consequently, Inspect eagerly executes as much local computation as it can and at the same time ensures that model APIs are not over-saturated by enforcing a maximum number of concurrent connections.

This section describes how to tune Inspect's concurrency, as well as how to handle situations where more local compute is required.

12.2 Model APIs

Max Connections

Connections to model APIs are the most fundamental unit of concurrency to manage. The main thing that limits model API concurrency is not local compute or network availability, but rather *rate limits* imposed by model API providers. Here we run an evaluation and set the maximum connections to 20:

```
$ inspect eval --model openai/gpt-4 --max-connections 20
```

The default value for max connections is 10. By increasing it we might get better performance due to higher parallelism, however we might get *worse* performance if this causes us to frequently hit rate limits (which are retried with exponential backoff). The "correct" max connections for your evaluations will vary based on your actual rate limit and the size and complexity of your evaluations.

Rate Limits

When you run an eval you'll see information reported on the current active connection usage as well as the number of HTTP rate limit errors that have been encountered (note that Inspect will automatically retry on rate limits and other errors likely to be transient):

```
┌─ theory_of_mind (100 samples) ──────────────────────────────────────────┐
│                                               mistral/mistral-large-latest │
│ ⠿ ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 133/300 0:01:18       dataset: theory_of_mind │
│                                                    scorer: model_graded_fact │
│ max_connections: 30                                                        │
│                                                                            │
│ mistral/mistral-large-latest: 30/30                    HTTP rate limits: 25 │
└────────────────────────────────────────────────────────────────────────────┘
```

Here we've set a higher max connections than the default (30). While you might be tempted to set this very high to see how much concurrent traffic you can sustain, more often than not setting too high a max connections will result in slower evaluations, because retries are done using exponential backoff, and bouncing off of rate limits too frequently will have you waiting minutes for retries to fire.

You should experiment with various values for max connections at different times of day (evening is often very different than daytime!). Generally speaking, you want to see some number of HTTP rate limits enforced so you know that are somewhere close to ideal utilisation, but if you see hundreds of these you are likely over-saturating and experiencing a net slowdown.

Limiting Retries

By default, inspect will continue to retry model API calls (with exponential backoff) indefinitely when a rate limit error (HTTP status 429) is returned . You can limit these retries by using the `max_retries` and `timeout` eval options. For example:

```
$ inspect eval --model openai/gpt-4 --max-retries 10 --timeout 600
```

If you want more insight into Model API connections and retries, specify `log_level=http`. For example:

```
$ inspect eval --model openai/gpt-4 --log-level=http
```

> **i** Note that max connections is applied per-model. This means that if you use a grader model from a provider distinct from the one you are evaluating you will get extra concurrency (as each model will enforce its own max connections).

12.3 Other APIs

It's possible that your custom solvers, tools, or scorers will call other REST APIs. Two things to keep in mind when doing this are:

1. It's critical that connections to other APIs use `async` HTTP APIs (i.e. the `httpx` model rather than the `requests` module). This is because Inspect's parallelism relies on everything being `async`, so if you make a blocking HTTP call with `requests` it will actually hold up all of the rest of the work in system!

2. As with model APIs, rate limits may be in play, so it's important not to over-saturate these connections. Recall that Inspect runs all samples in parallel so if you have 500 samples and don't do anything to limit concurrency, you will likely end up making hundreds of calls at a time to the API.

Here's some (oversimplified) example code that illustrates how to call a REST API within an Inspect component. We use the `async` interface of the `httpx` module, and we use Inspect's `concurrency()` function to limit simultaneous connections to 10:

```python
import httpx
from inspect_ai.util import concurrency
from inspect_ai.solver import Generate, TaskState

client = httpx.AsyncClient()

async def solve(state: TaskState, generate: Generate):
  ...
  # wrap the call to client.get() in an async concurrency
  # block to limit simulaneous connections to 10
  async with concurrency("my-rest-api", 10):
    response = await client.get("https://example.com/api")
```

Note that we pass a name ("my-rest-api") to the `concurrency()` function. This provides a named scope for managing concurrency for calls to that specific API/service.

12.4 Subprocesses

It's possible that your custom solvers, tools, or scorers will need to launch child processes to perform various tasks. Subprocesses have similar considerations as calling APIs: you want to make sure that they don't block the rest of the work in Inspect (so they should be invoked with `async`) and you also want to make sure they don't provide *too much* concurrency (i.e. you wouldn't want to launch 200 processes at once on a 4 core machine!).

To assist with this, Inspect provides the `subprocess()` function. This `async` function takes a command and arguments and invokes the specified command asynchronously, collecting and returning stdout and stderr. The `subprocess()` function also automatically limits concurrent child processes to the number of CPUs on your system (`os.cpu_count()`). Here's an example from the implementation of a `list_files()` tool:

```python
@tool(prompt=(
    "If you are asked to list the files in a directory you "
    + "should call the list_files function to access the listing."
))
def list_files():
    async def execute(dir: str):
        """List the files in a directory.

        Args:
            dir (str): Directory

        Returns:
            File listing of the directory
        """
        result = await subprocess(["ls", dir])
        if result.success:
            return result.stdout
        else:
            return f"Error: {result.stderr}"

    return execute
```

The maximum number of concurrent subprocesses can be modified using the `--max-subprocesses` option. For example:

```
$ inspect eval --model openai/gpt-4 --max-subprocesses 4
```

Note that if you need to execute computationally expensive code in an eval, you should always factor it into a call to `subprocess()` so that you get optimal concurrency and performance.

Timeouts

If you need to ensure that your subprocess runs for no longer than a specified interval, you can use the `timeout` option. For example:

```
result = await subprocess(["ls", dir], timeout = 30)
```

If a timeout occurs, then the `result.status` will be `False` and a timeout error message will be included in `result.stderr`.

12.5 Parallel Code

Generally speaking, you should try to make all of the code you write within Inspect solvers, tools, and scorers as parallel as possible. The main idea is to eagerly post as much work as you can, and then allow the various concurrency gates described above to take care of not overloading remote APIs or local resources. There are two keys to writing parallel code:

1. Use `async` for all potentially expensive operations. If you are calling a remote API, use the `httpx.AsyncClient`. If you are running local code, use the `subprocess()` function described above.
2. If your `async` work can be parallelised, do it using `asyncio.gather()`. For example, if you are calling three different model APIs to score a task, you can call them all in parallel. Or if you need to retrieve 10 web pages you don't need to do it in a loop—rather, you can fetch them all at once.

Model Requests

Let's say you have a scorer that uses three different models to score based on majority vote. You could make all of the model API calls in parallel as follows:

```
from inspect_ai.model import get_model

models = [
  get_model("openai/gpt-4"),
  get_model("anthropic/claude-3-sonnet-20240229"),
  get_model("mistral/mistral-large-latest")
]

output = "Output to be scored"
prompt = f"Could you please score the following output?\n\n{output}"

graders = [model.generate(prompt) for model in models]

grader_outputs = await asyncio.gather(*graders)
```

Note that we don't await the call to `model.generate()` when building our list of graders. Rather the call to `asyncio.gather()` will await each of these requests and return when they have all completed. Inspect's internal handling of `max_connections` for model APIs will apply to these requests, so you need now worry about how many you put in flight, they will be throttled as appropriate.

Web Requests

Here's an examples of using `asyncio.gather()` to parallelise web requests:

```
import asyncio
import httpx
client = httpx.AsyncClient()

pages = [
  "https://www.openai.com",
  "https://www.anthropic.com",
  "https://www.google.com",
  "https://mistral.ai/"
]

downloads = [client.get(page) for page in pages]

results = await asyncio.gather(*downloads)
```

Note that we don't `await` the client requests when building up our list of `downloads`. Rather, we let `asyncio.gather()` await all of them, returning only when all of the results are available. Compared to looping over each page download this will execute much, much quicker. Note that if you are sending requests to a REST API that might have rate limits, you should consider wrapping your HTTP requests in a `concurrency()` block. For example:

```
from inspect_ai.util import concurrency

async def download(page):
  async with concurrency("my-web-api", 2):
    return await client.get(page)

downloads = [download(page) for page in pages]

results = await asyncio.gather(*downloads)
```