

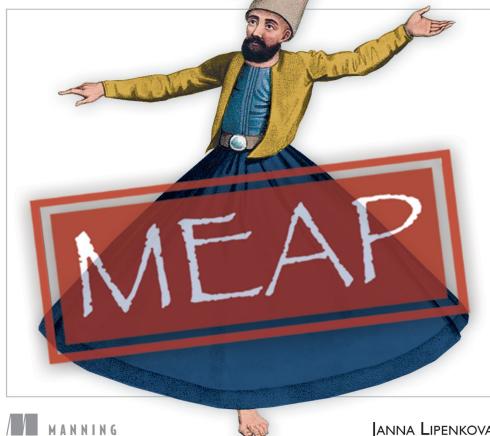
GenAI/LLM Myths and Misconceptions

Ryan Wesslen

Read these books!

the art of

AI PRODUCT MANAGEMENT



MANNING

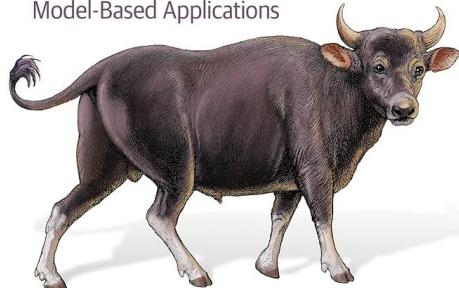
JANNA LIPENKOVA

<https://www.manning.com/books/the-art-of-ai-product-management>

O'REILLY®

Prompt Engineering for LLMs

The Art and Science of Building Large Language Model-Based Applications



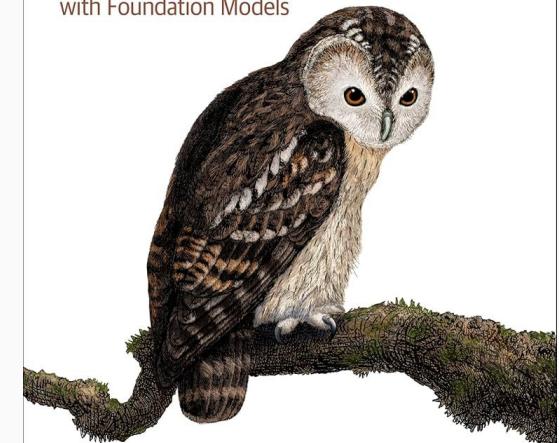
John Berryman
& Albert Ziegler

<https://www.amazon.com/Prompt-Engineering-LLMs-Model-Based-Applications/dp/1098156153>

O'REILLY®

AI Engineering

Building Applications
with Foundation Models



Chip Huyen

<https://www.amazon.com/AI-Engineering-Building-Applications-Foundation/dp/1098166302>

Highly recommend Alex Strick van Linschoten's study notes

Alex Strick van Linschoten - mllops.systems

Alex Strick van Linschoten

TIL About 🐧 🌐 🌐 🌐 🌐 🌐

MLOps.systems Blog

Feb 9, 2025
Alex Strick van Linschoten

BOOKS-I-READ LLM LLMPS EVALUATION

AI Engineering Architecture and User Feedback

My notes on chapter 10 of Chip Huyen's 'AI Engineering', an exploration of modern AI system architecture patterns and user feedback mechanisms, covering the evolution from simple API integrations to complex agent-based systems, including practical implementations of RAG, guardrails, caching strategies, and systematic approaches to gathering and utilizing user feedback for continuous improvement.

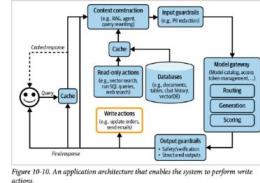


Figure 10-10. An application architecture that enables the system to perform write actions.

Categories

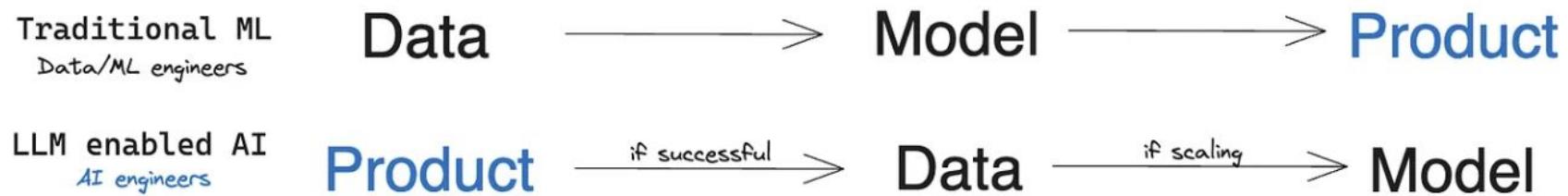
- RAG afghanistan agents
- appearances balochi
- balochi-language-model
- books-i-read calmcode
- computervision databases
- datalabelling datasets
- datavalidation debugging
- deep-learning deeplearning devops
- docker dockerinamonthoflunches
- emotions ethics evaluation fastai
- finetuning git go google hardware
- inference isafpr jupyter links llm
- llmops llms mathematics
- miniproject mllops mu123 nlp
- notation openai papers-i-read
- partone parttwo podcast

Myth 1: LLM Projects Start With Running Hugging Face Models in Jupyter notebooks

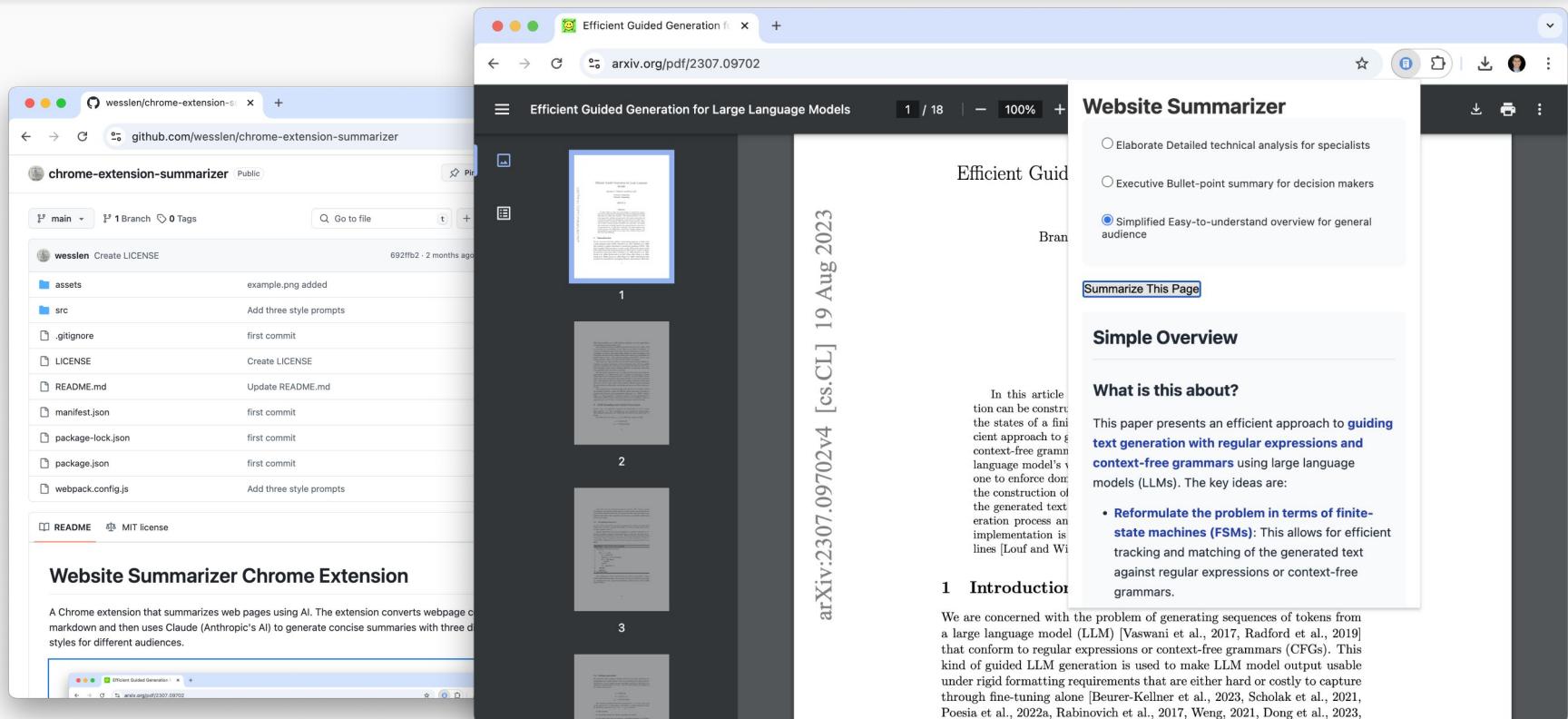
```
# Load model directly
from transformers import AutoModelForCausalLM
model = AutoModelForCausalLM.from_pretrained("deepseek-ai/DeepSeek-R1")
```

The New AI Engineering rewards those who can iterate fast

Fire, ready, aim. Instead of requiring *data scientists/ML engineers* do a laborious data collection exercise before training a single domain specific model that is then put into production, a *product manager/software engineer* can prompt an LLM, and build/validate a product idea, before getting specific data to finetune.



Example: Fun weekend project with Claude's help from zero to POC!

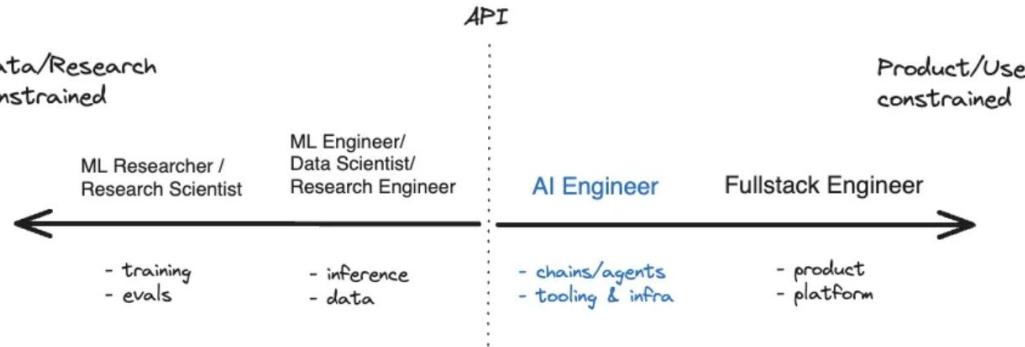


<https://github.com/wesslen/chrome-extension-summarizer>

Rise of the AI Engineer by Shawn Wang

We are observing a once in a generation “shift right” of applied AI, fueled by the emergent capabilities and open source/API availability of Foundation Models.

A wide range of AI tasks that used to take **5 years and a research team** to accomplish in **2013**, now just require API docs and a spare afternoon in **2023**.



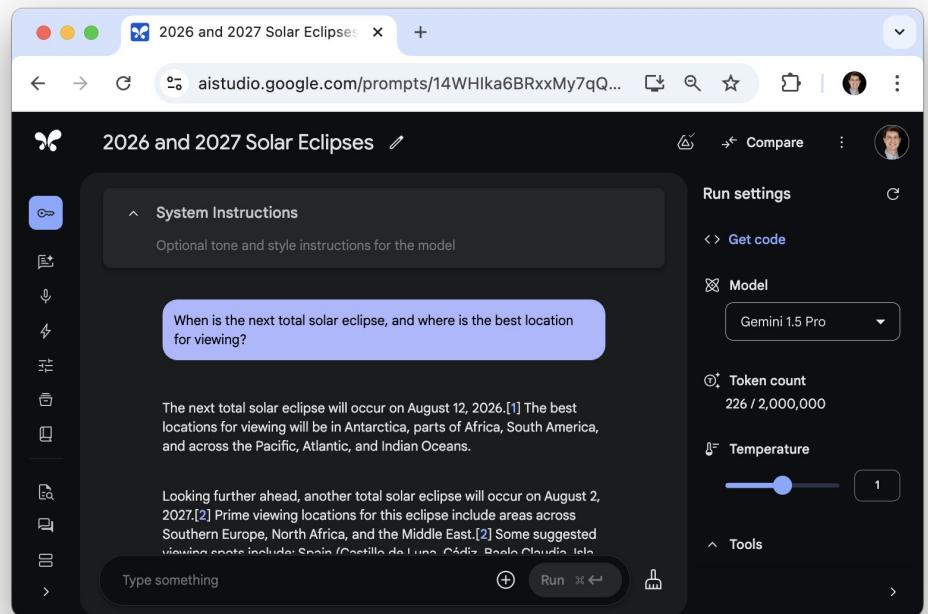
Important: the API line is permeable – AI Engineers can go left to finetune/host models and Research Engineers go right to build atop APIs too! This diagram has also been criticized for the placement of evals and data; we certainly agree evals are an important part of the job! MLR/MLEs handle foundation model concerns – aka *pretrain scale data and general benchmark evals*; but AI Engineers should certainly view *product-specific data and evals* as their job.

LLM API Endpoints

```
import os
from openai import OpenAI

client = OpenAI(api_key=os.environ.get("OPENAI_API_KEY"))

chat_completion = client.chat.completions.create(
    messages=[
        {
            "role": "user",
            "content": "Say this is a test",
        }
    ],
    model="gpt-4o",
)
```



LLM API Endpoints: Think Like A Developer

```
import os
from openai import OpenAI

client = OpenAI(api_key=os.environ.get("OPENAI_API_KEY"))

chat_completion = client.chat.completions.create(
    messages=[
        {
            "role": "user",
            "content": "Say this is a test",
        }
    ],
    model="gpt-4o",
)
```

- API Keys / Security (handling environmental variables)
- On Prem vs Off Prem
- Costs / Rate Limits
- Handling JSON
- Different Designs (OpenAI Compatibility vs others)
- Libraries (Clients, SDKs, or Connectors)

API Endpoints: Prompt Engineering

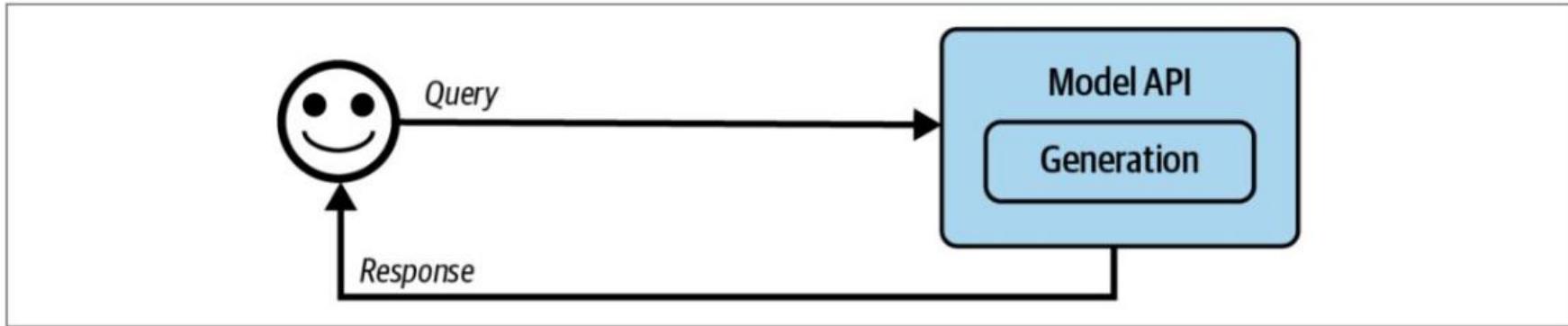


Figure 10-1. The simplest architecture for running an AI application.

Query

```
import os
from openai import OpenAI

client = OpenAI(api_key=os.environ.get("OPENAI_API_KEY"))

chat_completion = client.chat.completions.create(
    messages=[
        {
            "role": "user",
            "content": "Say this is a test",
        }
    ],
    model="gpt-4o",
)
```



Response

```
ChatCompletion(id='chatcmpl-AzCzF7eDhjwlpidJvaTrZGhwhNqf',
    choices=[Choice(finish_reason='stop', index=0, logprobs=None,
        message=ChatCompletionMessage(content='This is a test.', refusal=None, role='assistant',
        audio=None, function_call=None, tool_calls=None)), created=1739151293,
    model='gpt-4o-2024-08-06', object='chat.completion', service_tier='default',
    system_fingerprint='fp_4691090a87', usage=CompletionUsage(completion_tokens=6,
    prompt_tokens=12, total_tokens=18,
    completion_tokens_details=CompletionTokensDetails(accepted_prediction_tokens=0,
    audio_tokens=0, reasoning_tokens=0, rejected_prediction_tokens=0),
    prompt_tokens_details=PromptTokensDetails(audio_tokens=0, cached_tokens=0)))
```

API Endpoints: Context construction

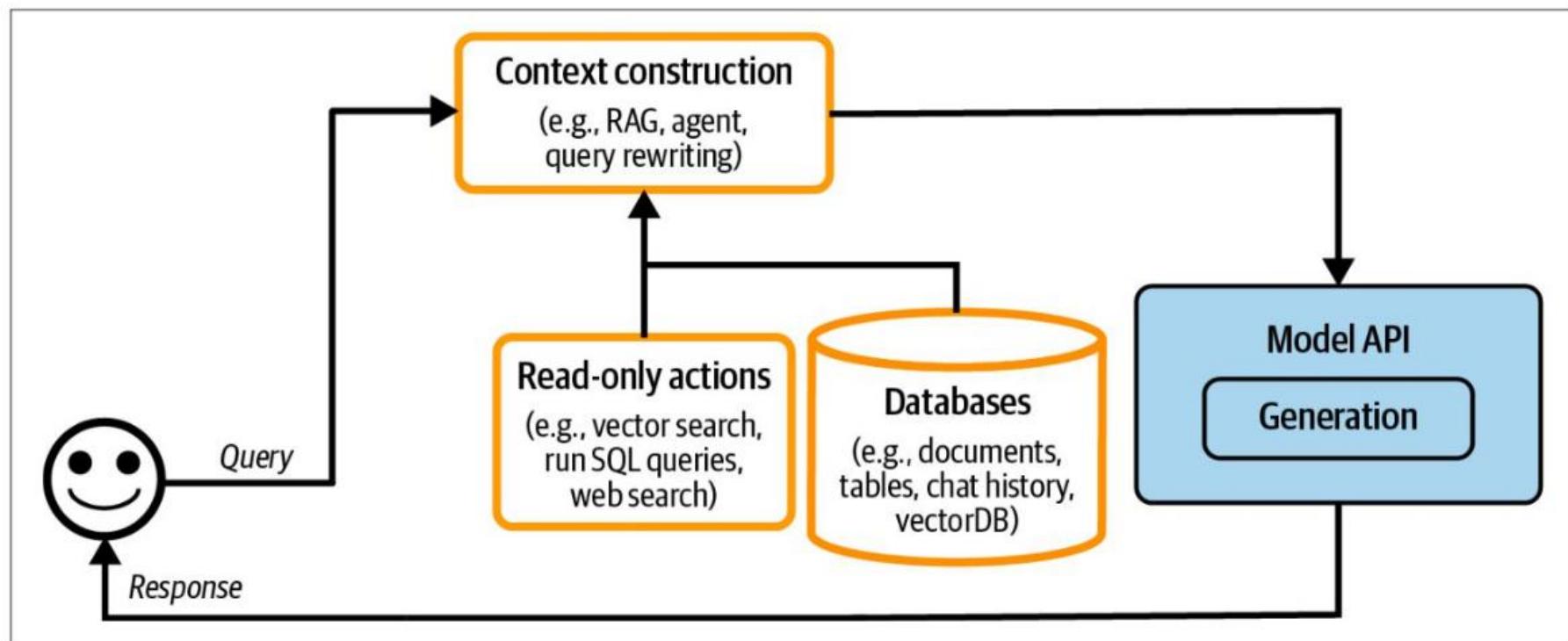


Figure 10-2. A platform architecture with context construction.

API Endpoints: With Guardrails

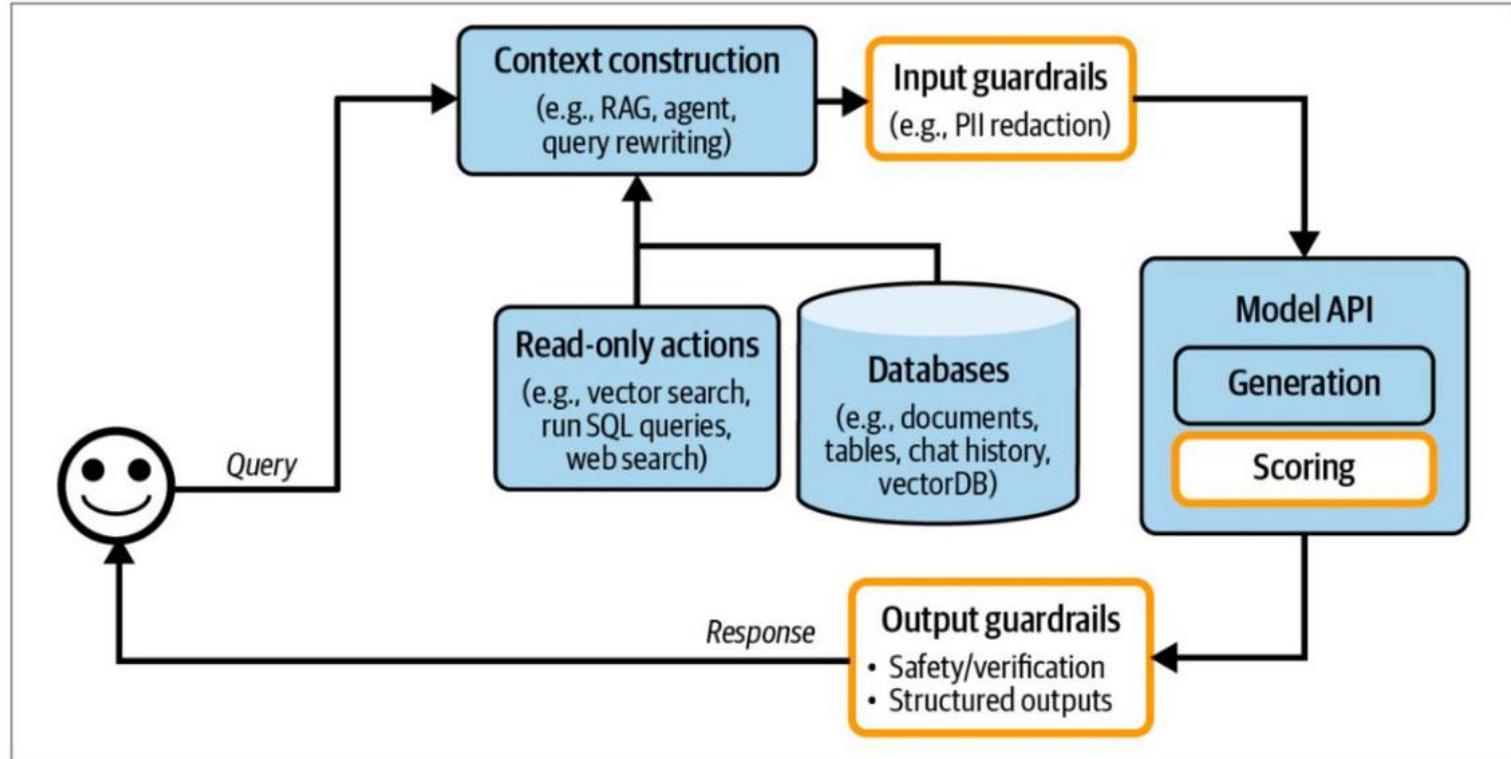


Figure 10-4. Application architecture with the addition of input and output guardrails.

Input/output guardrails

User query

I got 403 error on this code. What did I do wrong?

```
pat = "secret_token_that_shouldn't_be_leaked"  
url = "https://api.github.com/repos/{repo}/issues&page=30"  
response = get(url, access_token=pat)
```

Masked query

I got 403 error on this code. What did I do wrong?

```
pat = [ACCESS_TOKEN]  
url = "https://api.github.com/repos/{repo}/issues&page=30"  
response = get(url, access_token=pat)
```

Reversible PII map

[ACCESS_TOKEN]

"secret_token_that_sho
uldn't_be_leaked"

Model response

The URL you provided contains a syntax error. The correct URL should use ? to denote query parameters instead of &.

Here's a corrected version of your code:

```
pat = [ACCESS_TOKEN]  
url = "https://api.github.com/repos/{repo}/issues?page=30"  
response = get(url, access_token=pat)
```

Unmasked response

The URL you provided contains a syntax error. The correct URL should use ? to denote query parameters instead of &.

Here's a corrected version of your code:

```
pat = "secret_token_that_shouldn't_be_leaked"  
url = "https://api.github.com/repos/{repo}/issues?page=30"  
response = get(url, access_token=pat)
```

pydantic: Structured Output with Gemini 2.0 (Visual LLM) new PDF Capabilities

Invoice no: 27301261

Date of issue: 10/09/2012

Seller:

Williams LLC
72077 Plains Suite 342
West Alexandria, AR 97978

Tax Id: 922-88-2832
IBAN: GB70FTNR64199348221780

Client:

Hernandez-Anderson
084 Carter Lane Apt. 846
South Ronaldsbury, AZ 91030

Tax Id: 959-74-5868

ITEMS

No.	Description	Qty	UM	Net price	Net worth	VAT [%]	Gross worth
1.	Lilly Pulitzer dress Size 2	5,00	each	45,00	225,00	10%	247,50
2.	New ERIN Erin Fertherston Straight Dress White Sequence Lining Sleeveless SZ 10	1,00	each	59,99	59,99	10%	65,99
3.	Sequence dress Size Small	3,00	each	35,00	105,00	10%	115,50
4.	fire los angeles dress Medium	3,00	each	6,50	19,50	10%	21,45
5.	Eileen Fisher Women's Long Sleeve Fleece Lined Front Pockets Dress XS Gray	3,00	each	15,99	47,97	10%	52,77
6.	Lularoe Nicole Dress Size Small Light Solid Grey/ White Ringer Tee Trim	2,00	each	3,75	7,50	10%	8,25
7.	J.Crew Collection Black & White sweater Dress sz S	1,00	each	30,00	30,00	10%	33,00

SUMMARY

VAT [%]	Net worth	VAT	Gross worth
10%	494,96	49,50	544,46
Total	\$ 494,96	\$ 49,50	\$ 544,46

```
from pydantic import BaseModel, Field

class Item(BaseModel):
    description: str = Field(description="The description of the item")
    quantity: float = Field(description="The Qty of the item")
    gross_worth: float = Field(description="The gross worth of the item")

class Invoice(BaseModel):
    """Extract the invoice number, date and all list items with description, quantity and gross worth and the total gross worth"""
    invoice_number: str = Field(description="The invoice number e.g. 1234567890")
    date: str = Field(description="The date of the invoice e.g. 2024-01-01")
    items: list[Item] = Field(description="The list of items with description, quantity and gross worth")
    total_gross_worth: float = Field(description="The total gross worth of the invoice")

result = extract_structured_data("invoice.pdf", Invoice)
print(type(result))
print(f"Extracted Invoice: {result.invoice_number} on {result.date} with total gross worth {result.total_gross_worth}")
for item in result.items:
    print(f"Item: {item.description} with quantity {item.quantity} and gross worth {item.gross_worth}")

<class '__main__.Invoice'>
Extracted Invoice: 27301261 on 10/09/2012 with total gross worth 544.46
Item: Lilly Pulitzer dress Size 2 with quantity 5.0 and gross worth 247.5
Item: New ERIN Erin Fertherston Straight Dress White Sequence Lining Sleeveless SZ 10 with quantity 1.0 and gross worth 65.99
Item: Sequence dress Size Small with quantity 3.0 and gross worth 115.5
Item: fire los angeles dress Medium with quantity 3.0 and gross worth 21.45
Item: Eileen Fisher Women's Long Sleeve Fleece Lined Front Pockets Dress XS Gray with quantity 3.0 and gross worth 52.77
Item: Lularoe Nicole Dress Size Small Light Solid Grey/ White Ringer Tee Trim with quantity 2.0 and gross worth 8.25
Item: J.Crew Collection Black & White sweater Dress sz S with quantity 1.0 and gross worth 33.0
```

https://github.com/wesslen/llm-myths/blob/main/notebooks/invoice_gemini_structured_data.ipynb

<https://www.philschmid.de/gemini-pdf-to-data>

Instructor Structured Output

Using `pydantic` you can **add more context** to the model to **make it more accurate** and data validation.

Adding a comprehensive description can significantly improve the performance of the model.

Libraries like `instructor` added **automatic retries based on validation errors**, which can be a great help, but come at the **cost of additional requests**.

[Jason Liu's Pydantic is all you need](#)

```
import instructor
import google.generativeai as genai
from pydantic import BaseModel

class ExtractUser(BaseModel):
    name: str
    age: int

client = instructor.from_gemini(
    client=genai.GenerativeModel(
        model_name="models/gemini-1.5-flash-latest",
    ),
    mode=instructor.Mode.GEMINI_JSON,
)

# note that client.chat.completions.create will also work
resp = client.messages.create(
    messages=[
        {
            "role": "user",
            "content": "Extract Jason is 25 years old.",
        }
    ],
    response_model=ExtractUser,
)

assert isinstance(resp, ExtractUser)
assert resp.name == "Jason"
assert resp.age == 25
```

More advanced GenAI Platforms: Routing and Model Gateway

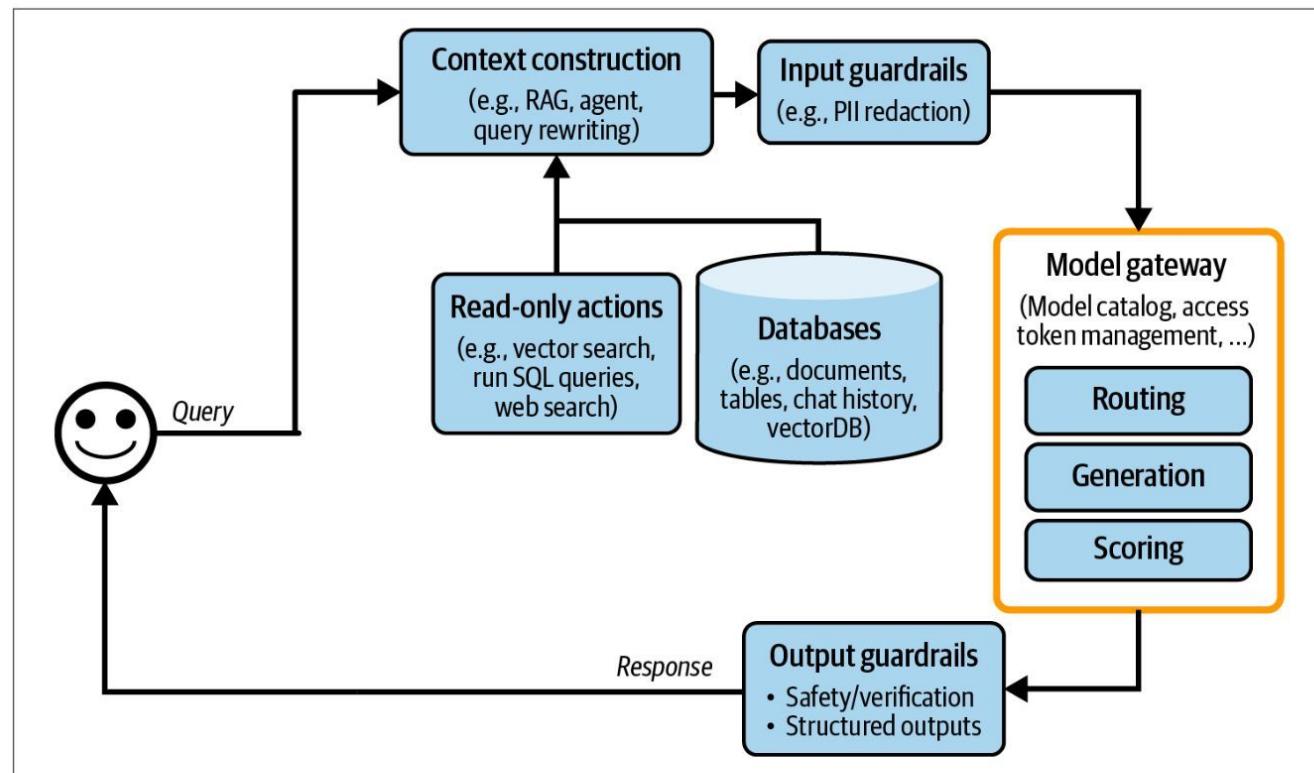


Figure 10-7. The architecture with the added routing and gateway modules.

Myth 2: Prompting is trivial; just fine tune instead.

Reality: Prompting matters

3.3.1 Prompt engineering comes first

Start with prompt engineering. Use all the techniques we discussed in the tactics section before. Chain-of-thought, n-shot examples, and structured input and output are almost always a good idea. Prototype with the most highly capable models before trying to squeeze performance out of weaker models.

Only if prompt engineering cannot achieve the desired level of performance should you consider finetuning. This will come up more often if there are non-functional requirements (e.g., data privacy, complete control, cost) that block the use of proprietary models and thus require you to self-host. Just make sure those same privacy requirements don't block you from using user data for finetuning!

[What We've Learned From A Year of Building with LLMs](#)

1.2.3 Prefer RAG over finetuning for new knowledge

Both RAG and finetuning can be used to incorporate new information into LLMs and increase performance on specific tasks. However, which should we prioritize?

Recent research suggests RAG may have an edge. [One study](#) compared RAG against unsupervised finetuning (aka continued pretraining), evaluating both on a subset of MMLU and current events. They found that RAG consistently outperformed finetuning for knowledge encountered during training as well as entirely new knowledge. In [another paper](#), they compared RAG against supervised finetuning on an agricultural dataset. Similarly, the performance boost from RAG was greater than finetuning, especially for GPT-4 (see Table 20).

Crawl (prompt engineering) before running your first marathon (fine tuning, agents+)

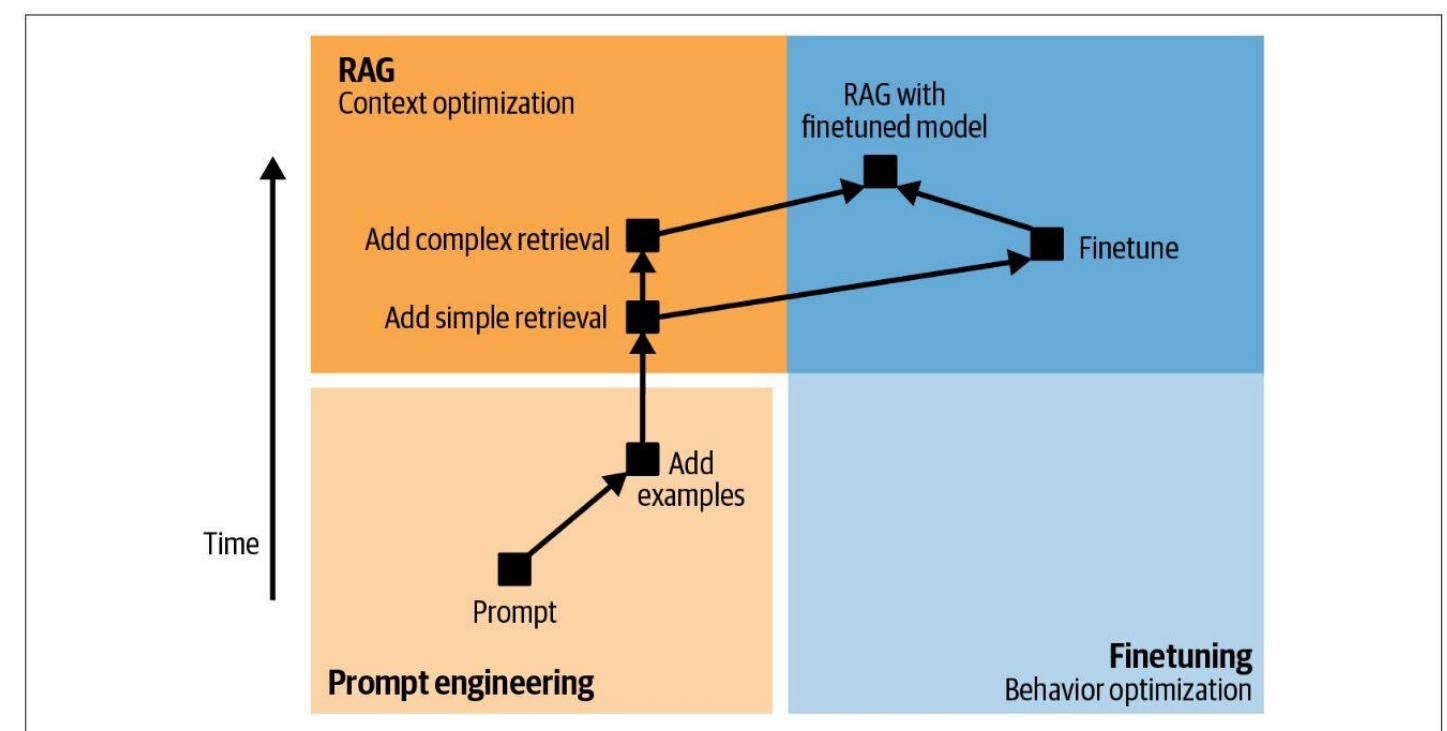
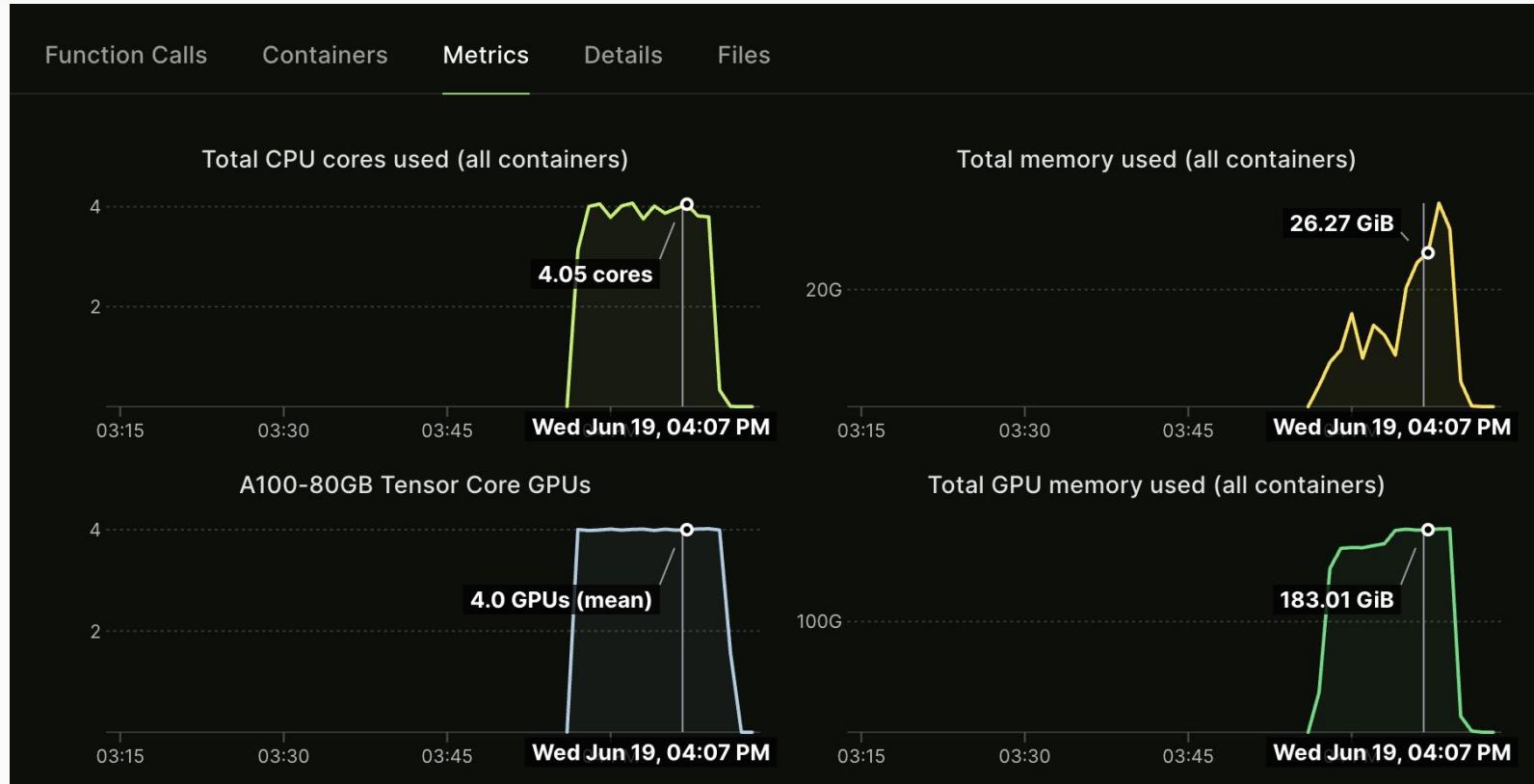


Figure 7-3. Example application development flows. After simple retrieval (such as term-based retrieval), whether to experiment with more complex retrieval (such as hybrid search) or finetuning depends on each application and its failure modes.

How much GPU RAM do you need to LoRA Fine Tune Llama 3 8B?



https://wesslen.github.io/modal-examples/13_finetuning.html#train-lora-on-llama-3-8b

3.1.2 Don't finetune until you've proven it's necessary

For most organizations, finetuning is driven more by FOMO than by clear strategic thinking.

Organizations invest in finetuning too early, trying to beat the “just another wrapper” allegations. In reality, finetuning is heavy machinery, to be deployed only after you’ve collected plenty of examples that convince you other approaches won’t suffice.

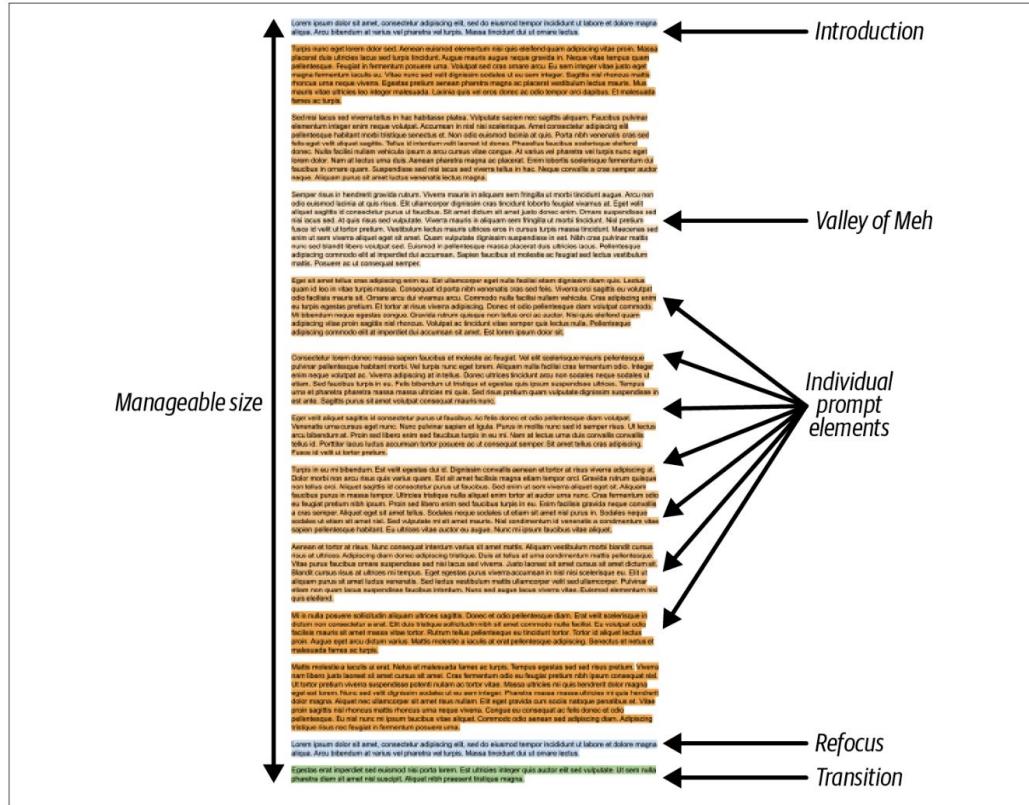
Some argue small prompts are all you need

1.1.3 Have small prompts that do one thing, and only one thing, well

A common anti-pattern / code smell in software is the “[God Object](#)”, where we have a single class or function that does everything. The same applies to prompts too.

A prompt typically starts simple: A few sentences of instruction, a couple of examples, and we’re good to go. But as we try to improve performance and handle more edge cases, complexity creeps in. More instructions. Multi-step reasoning. Dozens of examples. Before we know it, our initially simple prompt is now a 2,000 token Frankenstein. And to add injury to insult, it has worse performance on the more common and straightforward inputs! GoDaddy shared this challenge as their [No. 1 lesson from building with LLMs](#).

Others argue you can extend with proper planning



The closer a piece of information is to the **end of the prompt**, the **more impact** it has on the model

The **model often struggles** with the information **stuffed in the middle of the prompt**

Figure 6-1. Anatomy of a well-constructed prompt

One approach to prompt construction: additive greedy

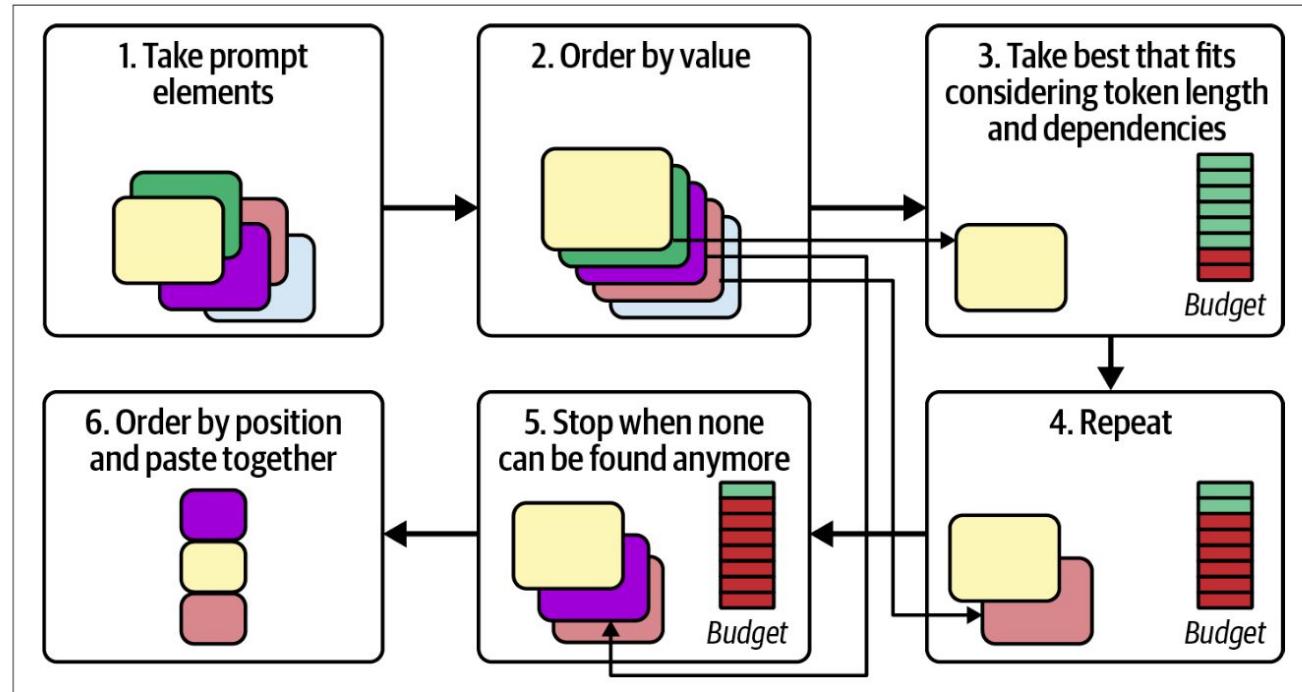


Figure 6-8. Additive greedy approach, in which the prompt crafter iteratively adds high-value elements to the prompt until the token budget is filled up, then re-sorts the elements according to position

Prompts are very sensitive! Don't necessarily work across models

2.2.2 Migrating prompts across models is a pain in the ***

Sometimes, our carefully crafted prompts work superbly with one model but fall flat with another. This can happen when we're switching between various model providers, as well as when we upgrade across versions of the same model.

Thus, if we have to migrate prompts across models, expect it to take more time than simply swapping the API endpoint. Don't assume that plugging in the same prompt will lead to similar or better results. Also, having reliable, automated evals helps with measuring task performance before and after migration, and reduces the effort needed for manual verification.

And be sure to version prompts and setup appropriate experiments!

Are prompts sensitive to small changes? Yes!

Table 2: Examples of atomic changes' impact on accuracy using probability ranking (prefix matching shown in Table 4). {} represents a text field; p_2 yields higher accuracy than p_1 for all tasks.

Task Id	Prompt Format 1 (p_1)	Prompt Format 2 (p_2)	Acc p_1	Acc p_2	Diff.
task280	passage:{}\n answer:{}	passage {}\\n answer {}	0.043	0.826	0.783
task317	Passage::{} Answer::{}	Passage:: {} Answer:: {}	0.076	0.638	0.562
task190	Sentence[I]- {}Sentence[II]- {} -- Answer\\t{}}	Sentence[A]- {}Sentence[B]- {} -- Answer\\t{}}	0.360	0.614	0.254
task904	input:: {} \\n output:: {}	input::{} \\n output::{}	0.418	0.616	0.198
task320	target - {} \\n{} \\nanswer - {}	target - {}; \\n{}; \\nanswer - {}	0.361	0.476	0.115
task322	COMMENT: {} ANSWER: {}	comment: {} answer: {}	0.614	0.714	0.100
task279	Passage : {}. Answer : {}	PASSAGE : {}. ANSWER : {}	0.372	0.441	0.069

[Quantifying Language Models' Sensitivity to
Spurious Features in Prompt Design or: How I
learned to start worrying about prompt formatting](#)

Myth 3: Metrics = LLM Evaluation

Reality: Metrics \subset LLM Evaluation

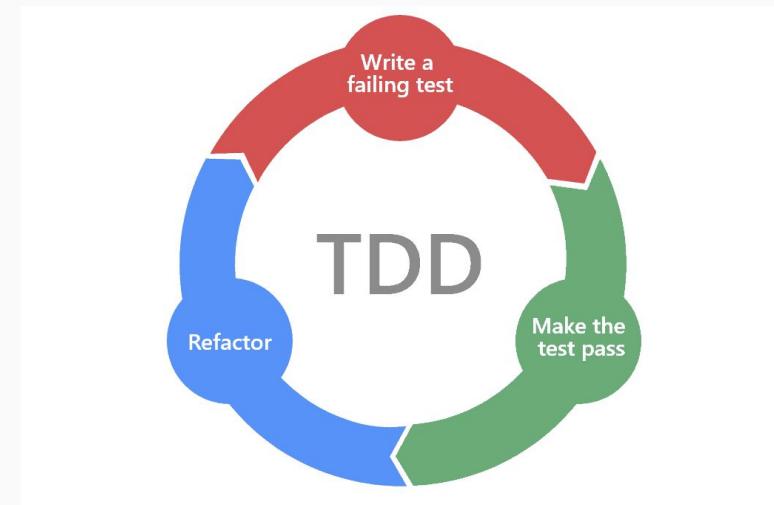
What makes LLM development different?

Category	Building with Traditional ML	Building with LLMs
AI Interface	Less important	Important
Prompt engineering	Not applicable	Important
Evaluation	Important	More important

Evaluation-Driven Development

A methodology where **AI application development begins with explicit evaluation criteria**, similar to how **test-driven development** starts with test cases.

However, EDD encompasses a broader range of metrics and considerations specific to AI systems.



Test-driven development

Evaluation Harness

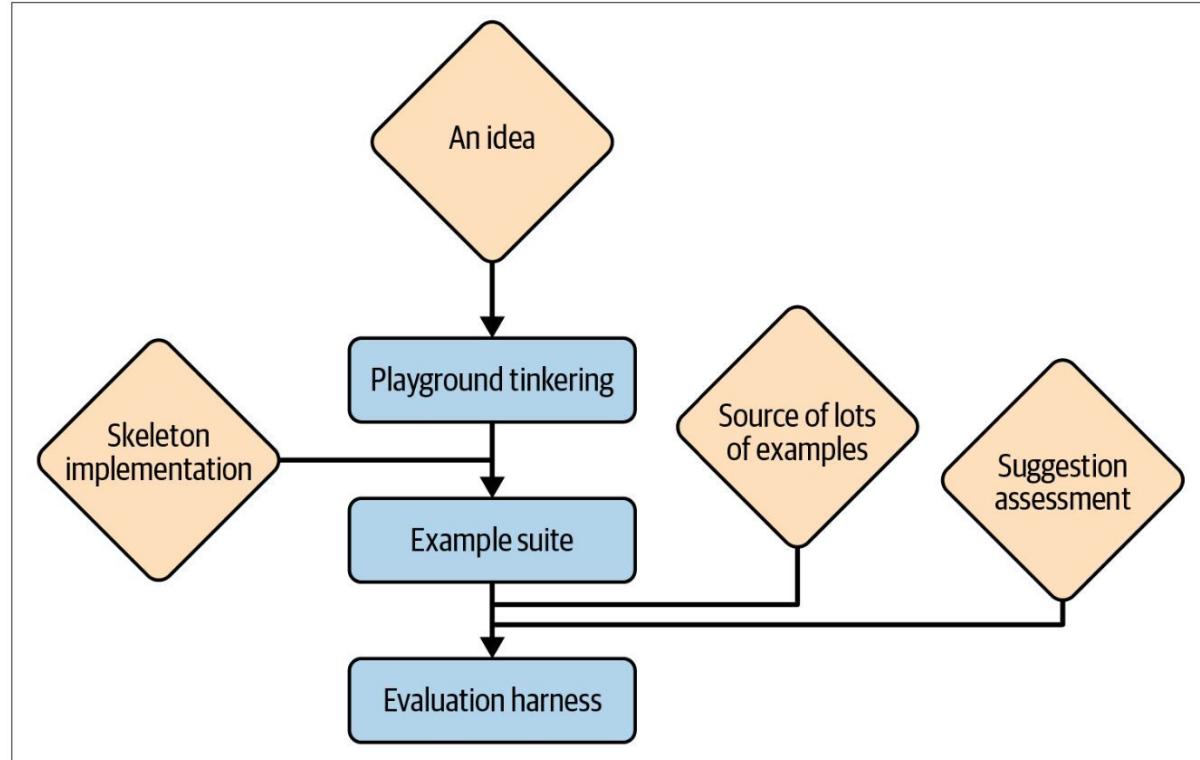


Figure 10-1. The tech tree of offline evaluations

Prompt sensitivity experiment: Suite of examples using DSPy on Claude Opus

The screenshot shows a Jupyter Notebook interface with the following details:

- Title:** llm-experiments / notebooks / prompt_sensitivity / claude.ipynb
- Code Tab:** Preview, Code, Blame, 2085 lines (2085 loc) · 195 KB
- Code Content:** A Python class definition for generating prompt variations. It includes methods for initialization, generating variations based on different types (spacing, punctuation, capitalization, word boundaries), and returning the variations.
- Test Cases:** A list of dictionaries defining test cases for the prompt variations. Each entry contains a "prompt" and a "context" string.

```
def __init__(self, base_prompt: str):
    self.base_prompt = base_prompt
    self.results = []

def generate_variations(self) -> List[Dict]:
    """Generate various prompt modifications"""
    variations = [
        # Original prompt
        {"type": "original", "prompt": self.base_prompt},

        # Extra spacing variations
        {"type": "extra_space_start", "prompt": f" {self.base_prompt}"}, 
        {"type": "extra_space_end", "prompt": f"{self.base_prompt} "}, 
        {"type": "double_space", "prompt": self.base_prompt.replace(" ", " ")}),

        # Punctuation variations
        {"type": "add_period", "prompt": f"{self.base_prompt}.}, 
        {"type": "add_question", "prompt": f"{self.base_prompt}?"}, 
        {"type": "add_exclamation", "prompt": f"{self.base_prompt}!"}, 

        # Capitalization variations
        {"type": "all_caps", "prompt": self.base_prompt.upper()}, 
        {"type": "all_lower", "prompt": self.base_prompt.lower()}, 

        # Word boundary variations
        {"type": "no_spaces", "prompt": self.base_prompt.replace(" ", "")}, 
        {"type": "extra_newline", "prompt": f"\n{self.base_prompt}\n"}
    ]
    return variations

# Test cases
test_cases = [
    {
        "prompt": "Explain how photosynthesis works",
        "context": "You are a biology teacher explaining concepts to high school students."
    },
    {
        "prompt": "Write a short poem about the moon",
        "context": "You are a creative writing instructor helping students with poetry."
    },
    {
        "prompt": "What are the key features of Python",
        "context": "You are a programming instructor teaching beginners about Python."
    },
    {
        "prompt": "Summarize",
        "context": "You are a research assistant helping to condense complex papers into key points."
    },
    {
        "prompt": "Explain the difference between mitochondrial DNA and nuclear DNA",
        "context": "You are a genetics researcher explaining concepts to undergraduate students."
    },
    {
        "prompt": "Design a database schema for a library management system",
        "context": "You are a database architect helping junior developers."
    },
    {
        "prompt": "Debug why my SELECT query with JOIN is slow",
        "context": "You are a SQL performance tuning expert helping optimize database queries."
    }
]
```

https://github.com/wesslen/llm-experiments/blob/main/notebooks/prompt_sensitivity/claude.ipynb

What's the impact of adding double white space to a prompt? A lot

- Original (original): What are the key features of Python

Some of the key features of Python include:

1. Easy to learn and read syntax – Python uses simple, English-like syntax that is easy for beginners to pick up. The code is very readable compared to many other programming languages.

2. Interpreted language – Python code is executed line by line at runtime by the interpreter. You don't need to compile the entire program first. This allows for faster development.

3. Dynamically typed – You don't need to declare variable types in Python. The interpreter infers types at runtime. This makes the code more concise.

4. Extensive standard library – Python comes with a large standard library that supports many common programming tasks like connecting to web servers, reading and modifying files, and working with data.

5. Third party packages – There is a rich ecosystem of third-party Python packages available that extend the language for specific domains like web development, data science, machine learning and more.

6. Portability – Python programs can run on many different computers and operating systems with little to no modifications required.

7. Supports multiple programming paradigms – Python supports procedural, object-oriented, and to some extent functional programming. This allows you to choose the paradigm that best fits the problem.

These are some of the key reasons Python has become one of the most popular programming languages in the world, used for everything from web development to data analysis to DevOps.

+ Modified (double_space): What are the key features of Python

Some of the key features of Python include:

1. Easy to learn and read syntax – Python uses a clean, English-like syntax that is easy for beginners to understand and learn. The code is very readable.

2. Interpreted language – Python code is executed line by line at runtime by the interpreter. There is no separate compilation step.

3. Dynamically typed – You don't need to declare variable types in Python. The interpreter infers types at runtime.

4. Supports multiple programming paradigms – Python supports procedural, object-oriented and functional programming styles.

5. Extensive standard library – Python comes with a large standard library that supports many common programming tasks.

6. Third party packages – There is a huge ecosystem of third-party packages available for Python that extend its capabilities.

7. Automatic memory management – Python automatically handles memory management and garbage collection. You don't have to manage memory manually.

8. Interactive mode – Python provides an interactive interpreter that allows you to execute code line by line for quick testing and debugging.

9. Portability – Python code can run on different platforms and operating systems with little to no modifications.

These features make Python a great language for beginners to learn programming concepts, while also being powerful enough for advanced users and real-world applications.

Running unit tests via CI/CD with GitHub Actions

The screenshot shows two browser windows side-by-side. The left window displays the GitHub code editor for the file `test_mathematics.py` in the `tests` directory of the `llm-evaluation-framework` repository. The right window shows the repository's README page, which includes a section titled "Repository Structure" listing the project's directory structure.

Code Editor Screenshot:

```
1  """Tests for mathematical capabilities."""
2
3  import pytest
4  import ast
5  import uuid
6
7  from src.llm_client import LLMClient
8  from src.database import get_session, UnitTestSuite, UnitTest
9
10 @pytest.mark.asyncio
11 async def test_arithmetic_operations(llm_client, db_session, make_test_suite):
12     """Test basic arithmetic operations."""
13     client = llm_client # Removed await
14
15     suite = make_test_suite(
16         name_prefix="arithmetic",
17         description="Tests for basic arithmetic operations",
18         category="mathematics"
19     )
20
21     test_cases = [
22         {
23             "prompt": "What is the result of multiplying 23.5 by 8.75?",
24             "expected": 205.625
25         },
26         {
27             "prompt": "What is 1234 divided by 56.5?",
28             "expected": 21.84070796460177
29         }
30     ]
31 
```

Repository Structure Screenshot:

```
llm-evaluation-framework/
├── .github/
│   └── workflows/
│       └── run_tests.yml
├── database/
│   ├── migrations/
│   │   └── 001_initial_schema.sql
│   └── init_db.py
└── tests/
    ├── __init__.py
    ├── conftest.py
    ├── test_summarization.py
    ├── test_reasoning.py
    ├── test_mathematics.py
    ├── test_code_generation.py
    └── test_few_shot.py
src/
├── __init__.py
├── config.py
├── database.py
├── llm_client.py
└── metrics.py
.gitignore
LICENSE
README.md
requirements.txt
datasette.yaml
```

<https://github.com/wesslen/llm-evaluation-framework>

.yml file for GH Actions reproduce LLM tests with specified endpoints credentials

The screenshot shows a GitHub Actions workflow named "run_tests.yml" in the "wesslen/llm-evaluation-framework" repository. The workflow has 20 runs listed, all triggered by the "workflow_dispatch" event. A modal window is open over the list of runs, prompting the user to "Use workflow from". The modal includes fields for "Branch: main", "API Base URL *" (set to "https://wesslen--vllm-openai-compatible-serv"), "API Key *" (containing placeholder text "xxxxxx"), "Model Name *" (set to "/models/NousResearch/Meta-Llama-3.1-8B-Int"), and "Minimum required coverage rate (%)" (set to 50). A "Run workflow" button is at the bottom of the modal. The right side of the screen shows the full content of the "run_tests.yml" file.

```
name: Set up Python
uses: actions/setup-python@v4
with:
  python-version: '3.9'

- name: Install dependencies
  run: |
    python -m pip install --upgrade pip
    pip install -r requirements.txt

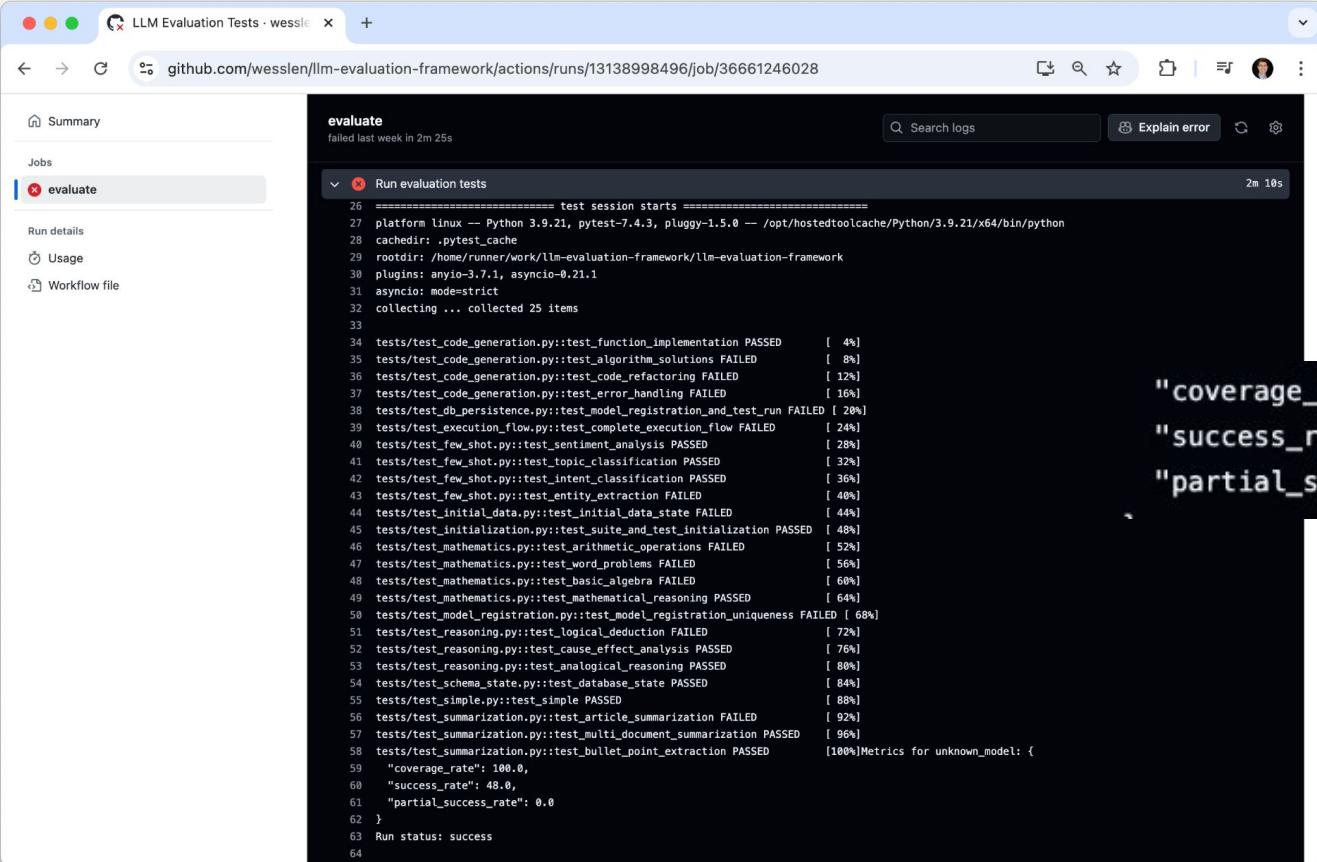
- name: Initialize database
  run: |
    mkdir -p database/migrations

# Initialize database with schema
echo "Initializing database..."
python database/init_db.py --drop

# Verify initial database structure
echo "Verifying initial database structure..."
sqlite3 database/llm_evaluation.db << EOF
.mode column
.headers on
SELECT name FROM sqlite_master
WHERE type='table' AND name NOT LIKE 'sqlite_%'
ORDER BY name;
EOF

- name: Run evaluation tests
  id: run_tests
  env:
    API_BASE_URL: ${{ inputs.api_base_url }}
    API_KEY: ${{ inputs.api_key }}
    MODEL_NAME: ${{ inputs.model_name }}
    DATABASE_URL: sqlite:///database/llm_evaluation.db
  run: |
    # Run tests and capture the exit code
    pytest tests/ -v
    test_exit_code=$?
```

Running Automated Tests on DeepSeek



The screenshot shows a browser window with the URL github.com/wesslen/llm-evaluation-framework/actions/runs/13138998496/job/36661246028. The page displays the results of a test run named "evaluate". The left sidebar shows "Jobs" and "evaluate" is selected. The main area shows the log output for "Run evaluation tests". The log includes test session details and a list of test cases with their outcomes and coverage percentages. At the bottom of the log, three metrics are listed: "coverage_rate": 100.0, "success_rate": 48.0, and "partial_success_rate": 0.0.

```
===== test session starts =====
platform linux -- Python 3.9.21, pytest-7.4.3, pluggy-1.5.0 -- /opt/hostedtoolcache/Python/3.9.21/x64/bin/python
cachedir: .pytest_cache
rootdir: /home/runner/work/llm-evaluation-framework/llm-evaluation-framework
plugins: anyio-3.7.1, asyncio-0.2.1
asyncio: mode=strict
collecting ... collected 25 items
33

34 tests/test_code_generation.py::test_function_implementation PASSED [ 4%]
35 tests/test_code_generation.py::test_algorithm_solutions FAILED [ 8%]
36 tests/test_code_generation.py::test_code_refactoring FAILED [ 12%]
37 tests/test_code_generation.py::test_error_handling FAILED [ 16%]
38 tests/test_db_persistence.py::test_model_registration_and_test_run FAILED [ 20%]
39 tests/test_execution_flow.py::test_complete_execution_flow FAILED [ 24%]
40 tests/test_few_shot.py::test_sentiment_analysis PASSED [ 28%]
41 tests/test_few_shot.py::test_topic_classification PASSED [ 32%]
42 tests/test_few_shot.py::test_intent_classification PASSED [ 36%]
43 tests/test_few_shot.py::test_entity_extraction FAILED [ 40%]
44 tests/test_initial_data.py::test_initial_data_state FAILED [ 44%]
45 tests/test_INITIALIZATION.py::test_suite_and_test_INITIALIZATION PASSED [ 48%]
46 tests/test_mathematics.py::test_arithmetic_operations FAILED [ 52%]
47 tests/test_mathematics.py::test_word_problems FAILED [ 56%]
48 tests/test_mathematics.py::test_basic_algebra FAILED [ 60%]
49 tests/test_mathematics.py::test_mathematical_reasoning PASSED [ 64%]
50 tests/test_model_registration.py::test_model_registration_uniqueness FAILED [ 68%]
51 tests/test_reasoning.py::test_logical_deduction FAILED [ 72%]
52 tests/test_reasoning.py::test_cause_effect_analysis PASSED [ 76%]
53 tests/test_reasoning.py::test_analogical_reasoning PASSED [ 80%]
54 tests/test_schema_state.py::test_database_state PASSED [ 84%]
55 tests/test_simple.py::test_simple PASSED [ 88%]
56 tests/test_summarization.py::test_article_summarization FAILED [ 92%]
57 tests/test_summarization.py::test_multi_document_summarization PASSED [ 96%]
58 tests/test_summarization.py::test_bullet_point_extraction PASSED [100%]Metrics for unknown_model: {
59 "coverage_rate": 100.0,
60 "success_rate": 48.0,
61 "partial_success_rate": 0.0
62 }
63 Run status: success
64
```

"coverage_rate": 100.0,
"success_rate": 48.0,
"partial_success_rate": 0.0

“[W]ith general-purpose models,
evaluation is not only about assessing a
model’s performance on known tasks
but also **about discovering new tasks**
that the model can do.”

- Chip Huyen p.115

Thank you!

VSCode - LLM developers must go beyond notebooks.

The screenshot shows a VS Code interface with the following details:

- File Explorer:** Shows a project structure for "MODAL-LLAMA-3-8B-SERVING". It includes a "notebooks" folder containing "dsba6010_openai_api_prompting_with_modal.ipynb" (selected), "api.py", and "client.py". Other files like ".env", ".gitignore", "LICENSE", "README.md", and "requirements.txt" are also listed.
- Code Editor:** Displays the content of "api.py". The code uses the `Modal` library to interact with a Llama-3-8B model. It defines a list of messages and creates a completion stream.

```
client.base_url = userdata.get("MODAL_BASE_URL")
model = "/models/NousResearch/Meta-Llama-3-8B-Instruct"

messages = [
    {
        "role": "system",
        "content": "You are a poetic assistant, skilled in writing satirical doggerel with creative flair.",
    },
    {
        "role": "user",
        "content": "Compose a limerick about baboons and racoons.",
    },
]

stream = client.chat.completions.create(
    model=model,
    messages=messages,
    stream=True,
)
```

- Terminal:** Shows deployment logs:
 - Created function download_model_to_image.
 - Created web function serve => <https://wesslen--vllm-openai-compatible-serve.modal.run>
 - App deployed in 258.941s!
- Bottom Status Bar:** Shows the file name "llama-3-1-8B*", line numbers 1 and 10, and a status icon.

Evaluation Workflow

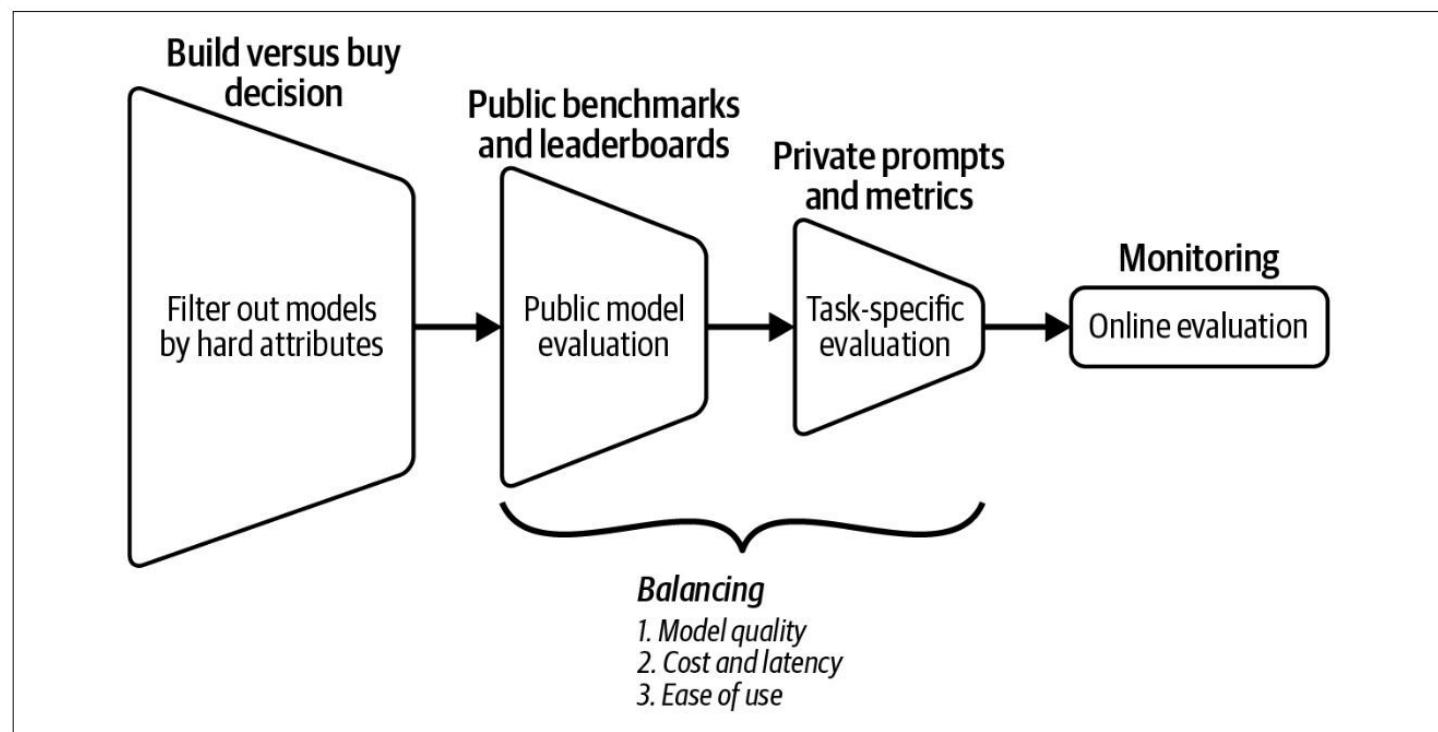


Figure 4-5. An overview of the evaluation workflow to evaluate models for your application.

Execution Flows

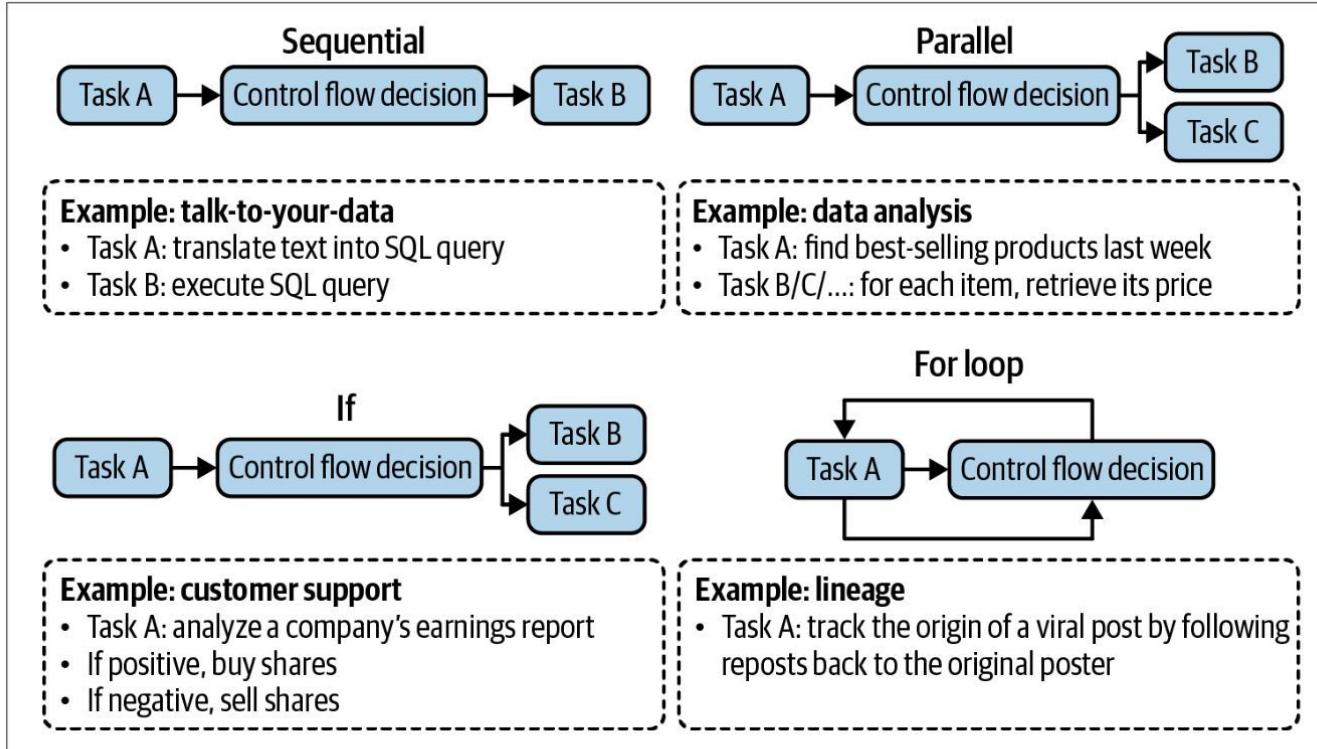


Figure 6-11. Examples of different orders in which a plan can be executed.