Assignment 18 — Solutions

Python & ML Essentials for GenAI Engineers

Part E — Engineering Essentials · 16 questions with answers

Q1. What is the difference between multithreading and multiprocessing in Python? What is the GIL, and how does it affect GenAI backends? ★ · Asked: TCS, Infosys

Answer

- **GIL (Global Interpreter Lock):** in CPython, only **one thread executes Python bytecode at a time** per process. It simplifies memory management but limits CPU parallelism with threads. (Python 3.13+ offers an experimental free-threaded build without the GIL.)
- **Multithreading:** multiple threads in one process, shared memory. Because the GIL is **released during I/O** (network calls, file reads), threads work well for **I/O-bound** tasks – e.g. calling LLM APIs, vector DB, HTTP requests in parallel. Not useful for CPU-heavy Python code.
- **Multiprocessing:** separate processes, each with its own interpreter and GIL → **true parallelism on multiple cores** for **CPU-bound** work – PDF parsing, OCR, chunking, local embedding computation, data processing. Costs: more memory, inter-process communication overhead.

GenAI backend practice: most work is I/O-bound (waiting for LLM/API responses) → use **async (asyncio/FastAPI)** or threads; push CPU-heavy ingestion to process pools or separate workers; heavy model inference runs in optimized C++/CUDA code (vLLM, PyTorch) which releases the GIL.

Q2. Concurrency vs parallelism: why use async programming in an AI application? If you must call three independent APIs, would you call them one by one? ★ · Asked: Infosys

Answer

- **Concurrency** – handling many tasks **in overlapping time** (switching while waiting); one core can do it. **Parallelism** – tasks literally **running at the same time** on multiple cores.
- LLM apps spend most time **waiting on network I/O**. With **async**, while one request waits, the event loop serves others → much higher throughput with fewer resources; FastAPI is built on this.
- **Three independent API calls: don't call them sequentially** (total = sum of latencies, e.g. 2 + 3 + 1 = 6 s). Run them **concurrently** (total ≈ slowest call, ~3 s):

```
import asyncio, httpx

async def fetch(client, url):
    r = await client.get(url, timeout=10)
    r.raise_for_status()
    return r.json()

async def main():
    async with httpx.AsyncClient() as client:
        weather, news, stock = await asyncio.gather(
            fetch(client, "https://api.example.com/weather"),
            fetch(client, "https://api.example.com/news"),
            fetch(client, "https://api.example.com/stock"),
            return_exceptions=True)          # one failure doesn't cancel the others
```

```
    return weather, news, stock

asyncio.run(main())
```

Only call sequentially when one call depends on another's result.

Q3. What are generators and decorators in Python? How do generators relate to streaming LLM responses? ★ · Asked: TCS, Google

Answer

- **Generator:** a function that uses `**yield**` to produce values **one at a time, lazily**, keeping its state between calls – memory-efficient for large/infinite data.
- **Decorator:** a function that **wraps another function** to add behaviour (logging, timing, retries, caching, auth) without changing its code; applied with `@decorator`.
- **Streaming:** LLM APIs return tokens as a stream; a generator **yields each chunk as it arrives**, and a web framework (FastAPI `StreamingResponse`, SSE) sends it to the user immediately → low perceived latency.

```
import time, functools

def timed(func):                        # decorator
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        result = func(*args, **kwargs)
        print(f"{func.__name__} took {time.perf_counter() - start:.2f}s")
        return result
    return wrapper

def stream_answer(llm, question):       # generator
    for chunk in llm.stream(question): # LangChain chat model streaming
        yield chunk.content

# FastAPI streaming endpoint
from fastapi import FastAPI
from fastapi.responses import StreamingResponse
app = FastAPI()

@app.get("/chat")
def chat(q: str):
    return StreamingResponse(stream_answer(llm, q), media_type="text/plain")
```

Q4. How do you process very large data (e.g. millions of rows or huge files) in Python without Out-Of-Memory errors? · Asked: TCS

Answer

- **Stream/iterate instead of loading everything:** read files line by line, use **generators**; `pd.read_csv(..., chunksize=100_000)` and process chunk by chunk.
- **Batch processing** for embeddings/LLM calls (e.g. 100 documents per batch) and write results incrementally to disk/DB.
- **Efficient dtypes** (category, int32, float32), load only needed columns (`usecols`).
- **Columnar/lazy tools:** Polars (lazy mode), DuckDB, PyArrow/Parquet, Dask or Spark for distributed data.
- **Memory-mapped files** / NumPy memmap for large arrays; vector DBs instead of in-memory embedding arrays.
- Free memory (`del`, avoid unnecessary copies), and profile with `memory_profiler`/`tracemalloc`.
- Scale out with queues and workers rather than one giant process.

Q5. Write a Python function that calls an LLM API with exponential backoff retries (with jitter) for transient errors. [CODE] · Asked: Wipro, EPAM

Answer

```
import random, time

class TransientError(Exception):
    '''Timeouts, HTTP 429 or 5xx errors.'''

def call_with_backoff(fn, *args, max_retries=5, base_delay=1.0, max_delay=30.0, **kwargs):
    for attempt in range(max_retries + 1):
        try:
            return fn(*args, **kwargs)
        except TransientError as e:
            if attempt == max_retries:
                raise                                   # give up after last attempt
            delay = min(max_delay, base_delay * (2 ** attempt))   # 1, 2, 4, 8, 16...
            delay = random.uniform(0, delay)            # "full jitter" avoids thundering herd
            print(f"Attempt {attempt + 1} failed ({e}); retrying in {delay:.1f}s")
            time.sleep(delay)

# Usage with the OpenAI SDK
# from openai import OpenAI, RateLimitError, APITimeoutError, InternalServerError
# def ask(prompt):
#     try:
#         return client.chat.completions.create(model="gpt-4o-mini",
#                     messages=[{"role": "user", "content": prompt}], timeout=30)
#     except (RateLimitError, APITimeoutError, InternalServerError) as e:
#         raise TransientError(e)
# call_with_backoff(ask, "Hello")
```

Key points for the interviewer: retry **only transient errors** (not 400/401), cap the attempts and the delay, add **jitter**, respect `Retry-After`, set timeouts, and make the operation idempotent. In production use the `tenacity` library or the SDK's built-in `max_retries`.

Q6. Implement an in-memory cache (with TTL) for LLM queries so identical requests don't call the API again. [CODE] · Asked: Wipro

Answer

```
import hashlib, json, time

class LLMCache:
    def __init__(self, ttl_seconds=3600, max_size=10_000):
        self.ttl, self.max_size, self.store = ttl_seconds, max_size, {}

    def _key(self, model, prompt, **params):
        raw = json.dumps({"model": model, "prompt": prompt, **params}, sort_keys=True)
        return hashlib.sha256(raw.encode()).hexdigest()

    def get(self, model, prompt, **params):
        k = self._key(model, prompt, **params)
        item = self.store.get(k)
        if item and time.time() - item[1] < self.ttl:
            return item[0]
        self.store.pop(k, None)                  # expired or missing
        return None

    def set(self, model, prompt, response, **params):
        if len(self.store) >= self.max_size:     # simple eviction: drop oldest
            oldest = min(self.store, key=lambda k: self.store[k][1])
            del self.store[oldest]
        self.store[self._key(model, prompt, **params)] = (response, time.time())

cache = LLMCache(ttl_seconds=600)

def cached_llm(prompt, model="gpt-4o-mini", temperature=0):
    hit = cache.get(model, prompt, temperature=temperature)
    if hit is not None:
        return hit
    response = call_llm(prompt, model=model, temperature=temperature)   # your API call
    cache.set(model, prompt, response, temperature=temperature)
    return response
```

Notes: the key includes model and parameters; only cache deterministic (temperature 0) requests; in production use **Redis** (shared across servers, built-in TTL) and consider semantic caching for similar questions. `functools.lru_cache` works for simple cases but has no TTL.

Q7. Why is FastAPI preferred for GenAI backends, and how does Pydantic fit in? ★ · Asked: EPAM, Google

Answer

FastAPI:

- **Async-native** (ASGI) → handles many concurrent I/O-bound LLM/API calls efficiently.
- Easy **streaming** (StreamingResponse/SSE) for token-by-token output.
- **Automatic validation and OpenAPI/Swagger docs** from type hints; very fast; dependency injection for auth, DB sessions, rate limiting.

Pydantic: defines **typed data models** used for request/response validation (rejects bad input with clear errors), settings management, and – very important in GenAI – **validating LLM structured outputs** (JSON → Pydantic object, retry if invalid). LangChain `with_structured_output` and tool schemas use Pydantic.

```
from fastapi import FastAPI
from pydantic import BaseModel, Field

class AskRequest(BaseModel):
    question: str = Field(min_length=3, max_length=2000)
    session_id: str

class AskResponse(BaseModel):
```

```
    answer: str
    sources: list[str]

app = FastAPI()

@app.post("/ask", response_model=AskResponse)
async def ask(req: AskRequest):
    result = await rag_chain.ainvoke(req.question)
    return AskResponse(answer=result["answer"],
                       sources=[d.metadata["source"] for d in result["docs"]])
```

Q8. Quick-fire Python: list vs tuple, shallow vs deep copy, and what is a context manager? · Asked: Google

Answer

- **List vs tuple:** list is **mutable** (append/modify), tuple is **immutable**, slightly faster and smaller, **hashable** (usable as dict keys/set elements if its items are hashable). Use tuples for fixed records.
- **Shallow vs deep copy:** a **shallow copy** (`copy.copy`, `list.copy()`, slicing) creates a new outer object but **shares the nested objects**; a **deep copy** (`copy.deepcopy`) recursively copies everything.

```
import copy
a = [[1, 2], [3]]
s = copy.copy(a); d = copy.deepcopy(a)
a[0].append(99)
print(s[0], d[0])   # [1, 2, 99] [1, 2]
```

- **Context manager:** an object that sets up and **guarantees cleanup** of a resource using `with` (implements `__enter__`/`__exit__`, or `@contextlib.contextmanager`). Examples: files, DB connections, locks, `httpx.AsyncClient`, timing blocks. Cleanup runs even if an exception occurs.

Q9. Pandas: given a DataFrame with columns 'Student' and 'Marks', mark students with marks below 30 as 'Fail' (others 'Pass') in a new column 'Result', and list the failed students. [CODE] · Asked: PwC

Answer

```
import pandas as pd
import numpy as np

df = pd.DataFrame({"Student": ["Asha", "Ravi", "Neha", "Kiran"],
                   "Marks": [72, 25, 29, 30]})

df["Result"] = np.where(df["Marks"] < 30, "Fail", "Pass")
# alternative: df.loc[df["Marks"] < 30, "Result"] = "Fail"

failed = df[df["Marks"] < 30]
print(df)
#   Student  Marks Result
# 0    Asha     72   Pass
# 1    Ravi     25   Fail
# 2    Neha     29   Fail
# 3   Kiran     30   Pass
print(failed["Student"].tolist())   # ['Ravi', 'Neha']
```

Mention vectorized operations (`np.where`, boolean masks) instead of row loops (`iterrows`) for performance.

Q10. SQL: find the second-highest salary, and find duplicate records in a table.

[CODE] · Asked: Google

Answer

```
-- Second-highest distinct salary
SELECT MAX(salary) AS second_highest
FROM employees
WHERE salary < (SELECT MAX(salary) FROM employees);

-- Using a window function (works for Nth highest; handles ties)
SELECT DISTINCT salary
FROM (SELECT salary, DENSE_RANK() OVER (ORDER BY salary DESC) AS rnk
      FROM employees) t
WHERE rnk = 2;

-- Duplicate records (same email)
SELECT email, COUNT(*) AS cnt
FROM customers
GROUP BY email
HAVING COUNT(*) > 1;

-- Show the full duplicate rows
SELECT * FROM (
  SELECT c.*, ROW_NUMBER() OVER (PARTITION BY email ORDER BY id) AS rn
  FROM customers c) t
WHERE rn > 1;
```

Know the difference: INNER JOIN returns only matching rows; LEFT JOIN returns all rows of the left table with NULLs where there is no match.

Q11. Precision vs recall vs F1 score – when is recall more important than precision? ★ · Asked: Google

Answer

- **Precision** = TP / (TP + FP) – of everything the model flagged, how much was correct. ("When I say fraud, am I right?")
- **Recall** = TP / (TP + FN) – of all actual positives, how many did we catch. ("Did I catch all frauds?")
- **F1** = 2 × P × R / (P + R) – harmonic mean; useful for imbalanced data when both matter.

Recall matters more when **missing a positive is costly**: fraud detection, cancer screening, security threats, **RAG retrieval** (missing the relevant chunk means a wrong answer).

Precision matters more when **false alarms are costly**: spam filtering (losing real emails), blocking customer transactions, recommending content.

Tune the decision **threshold** to trade them off; use PR-AUC for imbalanced problems; accuracy is misleading when 99% of transactions are legitimate.

Q12. What is overfitting, and what is data leakage? How do you prevent each? ★ · Asked: Google

Answer

- **Overfitting:** the model memorizes training data (including noise) – great training score, poor validation/test score. **Prevent:** more data, simpler model, regularization (L1/L2, dropout), early stopping, cross-validation, data augmentation, feature selection.
- **Data leakage:** information that would **not be available at prediction time** leaks into training, giving unrealistically good validation results that collapse in production. Examples: fitting the scaler/TF-IDF on the full dataset before splitting; features computed from the future (e.g. "chargeback flag" when predicting fraud at transaction time); duplicates across train and

test; random split on time-series data.

Prevent leakage: split first, then fit preprocessing on train only (sklearn **Pipelines**); time-based splits for temporal data; group-aware splits (same customer not in both); careful feature review.

- **Cross-validation (k-fold):** train/evaluate k times on different splits for a more reliable estimate.

Q13. How do you handle class imbalance (e.g. 1% fraud vs 99% normal transactions)? ★ · Asked: NLP interview rounds, Google

Answer

- **Right metrics:** precision, recall, F1, **PR-AUC**, confusion matrix – not accuracy (99% accuracy by predicting "not fraud" always).
- **Resampling:** undersample the majority, oversample the minority, **SMOTE** (synthetic samples) – only on the training set.
- **Class weights / cost-sensitive learning:** `class_weight="balanced"`, `scale_pos_weight` in XGBoost, focal loss for deep learning.
- **Threshold tuning** based on business cost of FP vs FN.
- **Better features** and anomaly-detection approaches (Isolation Forest, autoencoders).
- For text: data augmentation (back-translation, LLM paraphrasing), more labelled minority examples.
- Use **stratified** splits so each fold has minority examples.

Q14. How do you detect and handle outliers before feeding data into ML or embedding models? · Asked: PwC

Answer

Detect:

- Statistical: **IQR rule** (below Q1 − 1.5·IQR or above Q3 + 1.5·IQR), **z-score** > 3.
- Visual: box plots, histograms, scatter plots.
- Model-based: **Isolation Forest**, LOF, DBSCAN; for text/embeddings – points far from cluster centroids, unusually long/short documents, wrong language, garbage OCR text.

Handle (depends on cause):

- Data errors → **fix or remove**.
- Genuine extreme values → **cap/winsorize**, log-transform, or use robust models/scalers (RobustScaler, tree models).
- In fraud, outliers are often **the signal** – don't remove them; model them.

For RAG ingestion: filter empty/boilerplate chunks, duplicated pages and corrupted OCR before embedding.

Q15. Why are traditional ML concepts (classification vs regression, loss functions, metrics) still asked in GenAI interviews? · Asked: PwC

Answer

Because GenAI systems are built on and around them:

- LLMs are trained by minimizing a **loss function** (cross-entropy on next-token prediction); fine-tuning, RLHF/DPO and embedding training all rely on losses and optimization.
- Many GenAI products still contain **classical ML tasks**: intent classification, routing, re-ranking, fraud scoring, anomaly detection, evaluation classifiers.
- **Evaluation** needs ML thinking: precision/recall for retrieval and guardrails, train/test splits, overfitting, leakage, statistical significance in A/B tests.

- Production issues (drift, imbalance, bias, monitoring) are classic MLOps topics.

It shows the candidate understands *why* models behave as they do, not just how to call an API.

Quick definitions: classification predicts a **category** (fraud / not fraud, uses cross-entropy); regression predicts a **number** (transaction amount, uses MSE/MAE).

Q16. What is data drift vs model (concept) drift vs training–serving skew? Batch vs online inference? ★ · Asked: Google

Answer

- **Data drift:** the **input distribution changes** over time (new product names, new customer segment, users start asking about a new policy) while the relationship may stay the same.
- **Concept/model drift:** the **relationship between inputs and the target changes** (fraudsters change tactics, so old patterns no longer indicate fraud) → model accuracy decays.
- **Training–serving skew:** features/preprocessing at serving time **differ from training** (different tokenization, missing feature, different scaler) – a bug, not a change in the world.

Detect/handle: monitor input statistics (PSI, KS test, embedding-distribution shift), prediction distributions and live metrics; retrain or update the knowledge base; share one feature/preprocessing pipeline between training and serving.

- **Batch inference:** predictions for many records **on a schedule** (nightly scoring, bulk document summarization) – cheap, high throughput, not real-time.
- **Online (real-time) inference:** prediction **per request with low latency** (chatbot, transaction fraud check at payment time) – needs always-on serving and scaling.