**Important Question :**

**🔁 MTV Workflow (How it all works together)**

1. **User makes a request** (e.g., visits /products).
2. Django **URL dispatcher** sends the request to the appropriate **view**.
3. The **view** fetches data from the **model** (if needed).
4. The view passes that data to a **template**.
5. The **template** renders the HTML with the data and returns it as an **HTTP response** to the user.

**Q1. What is Multithreading (In depth):**

### Ans: ****Multithreading**** is a technique where multiple threads run within the same process, allowing for concurrent execution of tasks. Each thread runs independently but shares the same memory space.

### Why Use Multithreading?

* **Concurrency**: Allows tasks to run in parallel, improving performance for I/O-bound tasks.
* **Efficient Resource Utilization**: Uses CPU cycles effectively by switching between threads.
* **Responsive Applications**: Keeps applications smooth (e.g., UI threads in GUI apps).

**Threads vs Processes**

| **Feature** | **Threads** | **Processes** |
| --- | --- | --- |
| Memory | Shared | Separate |
| Communication | Fast (shared memory) | Slow (IPC required) |
| Overhead | Low | High |
| Isolation | Less (can affect others) | More (separate execution) |

**Multithreading in Python**

Python has a **Global Interpreter Lock (GIL)**, which prevents multiple threads from executing Python bytecode **simultaneously** in CPython. Because of this:

* **Multithreading is useful for I/O-bound tasks** (file I/O, network requests, database queries).
* **For CPU-bound tasks**, use **multiprocessing** instead.

Example using Python’s threading module:

**Q2. What is Thread:**

Ans: A **thread** is the smallest unit of execution in a process. It runs independently but shares the same memory and resources with other threads in the same process.

**Key Features of a Thread:**

* **Lightweight:** Uses less memory compared to a process.
* **Shares memory:** Threads within the same process can access shared variables.
* **Faster context switching:** Switching between threads is quicker than between processes.
* **Parallel execution:** Multiple threads can execute tasks concurrently (though Python has GIL limitations).

Python’s **GIL (Global Interpreter Lock)** restricts multiple threads from executing Python bytecode **simultaneously** in CPython.

* This means **Python threads are not true parallel** when running CPU-bound tasks.
* **Workaround:** Use the multiprocessing module for CPU-bound tasks.

### Q3. ****CI/CD (Continuous Integration & Continuous Deployment/Delivery)****

CI/CD is a **DevOps practice** that automates software development, testing, and deployment to deliver applications faster and more reliably. It consists of:

1. **Continuous Integration (CI)** → Automates code integration and testing.
2. **Continuous Deployment (CD)** → Automates deployment to production.
3. **Continuous Delivery (CD)** → Automates the release process but requires manual approval before deployment
4. **Continuous Integration (CI)**

🔹 **Goal:** Merge and test code frequently to detect issues early.  
🔹 **Process:**

* Developers push code changes to a shared repository (e.g., GitHub, GitLab).
* Automated **build and test pipelines** run (unit tests, linting, security checks).
* If successful, the code is merged into the main branch.

🔹 **Example CI Tools:**  
✅ Jenkins  
✅ GitHub Actions  
✅ GitLab CI/CD

✅ **Benefits of CI:**

* Catches bugs early.
* Reduces integration problems.
* Automates testing.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. **Continuous Deployment (CD)**

🔹 **Goal:** Fully automate deployment after passing CI tests.  
🔹 **Process:**

* Code that passes CI tests is automatically deployed to **production**.
* Requires **zero manual intervention**.

🔹 **Example CD Tools:**  
✅ AWS CodeDeploy  
✅ Kubernetes  
✅ ArgoCD  
✅ Terraform

✅ **Best for:** Microservices, cloud-based applications, and frequent releases.

⚠️ **Risk:** A bug can go directly into production if not tested properly!

**3️⃣ Continuous Delivery (CD)**

🔹 **Goal:** Automate deployment **up to staging/pre-production**, requiring **manual approval** for production.  
🔹 **Process:**

* CI pipeline runs.
* Deployment is **automated** to a **staging** environment.
* A manual **approval step** is required before deploying to **production**.

🔹 **Example Continuous Delivery Tools:**  
✅ AWS CodePipeline  
✅ GitLab CI/CD  
✅ Jenkins

✅ **Best for:** Organizations needing control over production releases

**4. CI/CD Workflow**

1.Developer **commits code** to a repository (e.g., GitHub).  
2 **CI pipeline** runs → Builds, tests, and verifies the code.  
3 **CD pipeline** deploys →

* Continuous **Delivery** → Deploys to staging (manual approval for production).
* Continuous **Deployment** → Deploys directly to production.

## ****6️⃣ Benefits of CI/CD****

✅ Faster development cycles.  
✅ Early bug detection.  
✅ Reliable and stable deployments.  
✅ Automates repetitive tasks.  
✅ Reduces human errors.

**✅1. Python**

**Q1. How would you optimize a Python application with high memory usage and slow response times?**

**A:**

* Use memory profiling tools like memory\_profiler to identify memory leaks.
* Optimize data structures, e.g., use set or dict instead of list where possible.
* Use generators and iterators to handle large datasets efficiently.
* Implement caching using functools.lru\_cache or external systems like Redis.
* Minimize global variables and avoid circular references.

**Q2. Explain the Global Interpreter Lock (GIL) in Python and its implications for multithreading.**

**A:**

* GIL is a mutex that protects access to Python objects, preventing multiple native threads from executing Python bytecode simultaneously.
* It leads to inefficiencies in CPU-bound applications.
* Use multiprocessing for CPU-bound tasks and multithreading for I/O-bound tasks to avoid GIL constraints.

**Q3. How would you implement a thread-safe singleton in Python?**

**A:**

* Use threading.Lock to prevent race conditions.

python

import threading

class Singleton:

\_instance = None

\_lock = threading.Lock()

def \_\_new\_\_(cls, \*args, \*\*kwargs):

with cls.\_lock:

if cls.\_instance is None:

cls.\_instance = super(Singleton, cls).\_\_new\_\_(cls)

return cls.\_instance

**Q4. What is the difference between deep copy and shallow copy in Python?**

**A:**

* A **shallow copy** copies references of nested objects, modifying them affects the original.
* A **deep copy** creates a completely independent clone of the object.
* Use copy.deepcopy() for deep copy and copy.copy() for shallow copy.

**Q5. How would you handle circular dependencies in Python modules?**

**A:**

* Use deferred imports to delay the import until it's required.
* Break the code into smaller modules to reduce tight coupling.
* Refactor common functionality into a separate module.

**✅2. Django**

**Q1. How does Django's ORM handle lazy and eager loading? Which approach is more efficient?**

**A:**

* **Lazy Loading:** Retrieves related objects only when accessed. This can lead to the N+1 query problem.
* **Eager Loading:** Uses select\_related() (1-to-1/Many-to-1) or prefetch\_related() (Many-to-Many/Reverse ForeignKey) to reduce the number of queries.
* Eager loading is generally more efficient for reducing database hits.

**Q2. Explain middleware in Django and its order of execution.**

**A:**

* Middleware is a series of hooks that process requests and responses globally.
* Middleware order:
  1. Process request (before view)
  2. Process view (before calling the view)
  3. Process response (after view)
  4. Process exception (on exception)
* Middleware executes from top to bottom in requests and from bottom to top in responses.

**Q3. How would you prevent SQL injection in a Django application?**

**A:**

* Use Django’s ORM to build queries.
* Avoid using raw SQL or sanitize input using django.db.connection.execute() with parameterized queries.
* Implement input validation using Django forms and serializers.

**Q4. What is the difference between select\_related() and prefetch\_related()?**

**A:**

* select\_related(): Performs a single SQL query with joins, used for foreign key relationships.
* prefetch\_related(): Performs multiple queries and joins them in Python, used for many-to-many and reverse relationships.

**Q5. How does Django handle database migrations and schema changes?**

**A:**

* Migrations are stored as .py files in the migrations folder.
* makemigrations generates migration files.
* migrate applies migration changes to the database.
* Use RunPython for custom migration logic.

**✅3. Flask**

**Q1. How would you structure a large Flask application to ensure scalability and maintainability?**

**A:**

* Use a modular blueprint structure.
* Separate business logic, routes, and models.
* Implement a factory pattern to create applications dynamically.
* Utilize app.config for configuration management.

**Q2. What are Flask signals, and how do you use them effectively?**

**A:**

* Flask signals allow sending and listening to application events.
* Use blinker to create signals and connect them using @signal.connect.
* Useful for decoupling components and triggering actions based on specific events.

**Q3. How would you implement request throttling in a Flask API?**

**A:**

* Use Flask-Limiter to restrict API requests per IP.

python

from flask\_limiter import Limiter

app = Flask(\_\_name\_\_)

limiter = Limiter(app, key\_func=get\_remote\_address)

@app.route('/api')

@limiter.limit("5 per minute")

def api():

return "API Response"

**Q4. Explain how to manage Flask sessions securely.**

**A:**

* Use Flask-Session to store session data on the server.
* Configure SECRET\_KEY for encryption.
* Use server-side session storage to mitigate client-side tampering.

**Q5. How would you handle CORS in a Flask application?**

**A:**

* Use Flask-CORS to allow cross-origin requests.

python

from flask\_cors import CORS

app = Flask(\_\_name\_\_)

CORS(app, resources={r"/api/\*": {"origins": "\*"}})

**✅4. RESTful API**

**Q1. How would you design a RESTful API that supports rate limiting, versioning, and authentication?**

**A:**

* Implement rate limiting using Flask-Limiter or Django DRF’s throttle\_classes.
* Versioning through URI (/v1/resource), query parameters, or headers.
* Use JWT for token-based authentication and OAuth2 for delegated access.

**Q2. What are idempotent methods in REST APIs, and why are they important?**

**A:**

* HTTP methods like GET, PUT, and DELETE are idempotent because multiple identical requests have the same effect.
* Ensures consistency and prevents unintended side effects on repeated requests.

**Q3. How do you ensure API backward compatibility when introducing breaking changes?**

**A:**

* Use versioning (URI, header-based, or query param-based).
* Implement feature toggles or API deprecation notices.
* Provide clear migration guidelines.

**Q4. How would you secure a REST API from CSRF and XSS attacks?**

**A:**

* Use CSRF tokens for state-changing requests.
* Sanitize user inputs to prevent XSS.
* Set Content-Type and X-Frame-Options headers correctly.

**Q5. How do you implement pagination and filtering efficiently in a REST API?**

**A:**

* Use limit and offset for pagination.
* Implement filtering through query parameters.
* Consider cursor-based pagination for large datasets.

**✅5. Database**

**Q1. How would you optimize a database with millions of records that frequently experiences slow queries?**

**A:**

* Add appropriate indexes on frequently queried columns.
* Denormalize data where necessary.
* Use partitioning and sharding to split data.
* Analyze query plans to identify bottlenecks.

**Q2. What is the difference between ACID and BASE properties in databases?**

**A:**

* **ACID:** Atomicity, Consistency, Isolation, Durability (traditional relational databases).
* **BASE:** Basically Available, Soft State, Eventually Consistent (NoSQL databases).

**Q3. How would you implement database connection pooling in Python?**

**A:**

* Use SQLAlchemy for ORM and connection pooling.

python

from sqlalchemy import create\_engine

engine = create\_engine('postgresql://user:password@localhost/dbname', pool\_size=10, max\_overflow=20)

**Q4. Explain the difference between optimistic and pessimistic locking in databases.**

**A:**

* **Optimistic Locking:** Assumes no conflicts, checks for conflicts before committing.
* **Pessimistic Locking:** Prevents other transactions from modifying data until the lock is released.

**Q5. What strategies would you use to handle schema migrations in a high-traffic production database?**

**A:**

* Use zero-downtime migration techniques.
* Split schema changes into multiple steps.
* Use feature flags for rolling out schema updates.

**✅6. AWS**

**Q1. How would you design a fault-tolerant architecture for a Flask/Django application on AWS?**

**A:**

* Use EC2 or ECS for hosting with auto-scaling.
* Implement ALB/ELB for load balancing.
* Use RDS with Multi-AZ for the database.
* Configure CloudFront with S3 for static assets.

**Q2. How do you implement blue/green deployment in AWS for a REST API?**

**A:**

* Use AWS Elastic Beanstalk or CodeDeploy.
* Deploy a new version to a separate environment.
* Gradually switch traffic using Route 53.

**Q3. What is the difference between SQS and SNS, and when would you use each?**

**A:**

* **SQS:** Message queuing for decoupling distributed components.
* **SNS:** Publish/subscribe model to fan out messages to multiple subscribers.

**Q4. How would you secure sensitive data stored in AWS S3?**

**A:**

* Enable encryption at rest (SSE-S3, SSE-KMS).
* Use S3 bucket policies and IAM roles to restrict access.
* Enable versioning and logging.

**Q5. How do you monitor and log API requests effectively in AWS?**

**A:**

* Use AWS CloudWatch for logging and monitoring.
* Enable AWS X-Ray for request tracing.
* Integrate with Amazon OpenSearch for log analytics.

**📚1. Basic File Read and Write in Python**

**Program:**

python

# Write data to a file

with open('data.txt', 'w') as file:

file.write('Hello, this is a test file.\n')

file.write('Python file operations example.')

# Read data from a file

with open('data.txt', 'r') as file:

content = file.read()

print('File Content:\n', content)

✅**Key Concepts:**

* File handling (open, read, write)
* Context manager (with statement)

**🚀2. Read and Write JSON File in Python**

**Program:**

python

import json

# Data to write

data = {'name': 'John', 'age': 30, 'city': 'New York'}

# Write JSON data to a file

with open('data.json', 'w') as json\_file:

json.dump(data, json\_file, indent=4)

# Read JSON data from a file

with open('data.json', 'r') as json\_file:

loaded\_data = json.load(json\_file)

print('Loaded JSON data:', loaded\_data)

✅**Key Concepts:**

* json.dump and json.load
* Working with structured data

**🗂️4. Read and Write CSV File in Python**

**Program:**

python

import csv

# Writing data to CSV

with open('data.csv', mode='w', newline='') as file:

writer = csv.writer(file)

writer.writerow(['Name', 'Age', 'City'])

writer.writerow(['John', 30, 'New York'])

writer.writerow(['Alice', 25, 'Los Angeles'])

# Reading data from CSV

with open('data.csv', mode='r') as file:

csv\_reader = csv.reader(file)

for row in csv\_reader:

print(row)

✅**Key Concepts:**

* csv.writer and csv.reader
* Reading and writing tabular data

**📝6. Read and Write YAML File in Python**

**Program:**

python

import yaml

# Data to write

data = {'name': 'John', 'age': 30, 'city': 'New York'}

# Write YAML data to a file

with open('data.yaml', 'w') as yaml\_file:

yaml.dump(data, yaml\_file)

# Read YAML data from a file

with open('data.yaml', 'r') as yaml\_file:

loaded\_data = yaml.safe\_load(yaml\_file)

print('Loaded YAML data:', loaded\_data)

✅**Key Concepts:**

* yaml.dump and yaml.safe\_load
* Parsing structured data formats

**📢9. Append Data to File and Read Using Python**

**Program:**

python

# Append data to a file

with open('append\_data.txt', 'a') as file:

file.write('Appending new data.\n')

# Read appended data

with open('append\_data.txt', 'r') as file:

content = file.read()

print('Appended File Content:\n', content)

✅**Key Concepts:**

* Appending mode ('a')
* Reading appended data

**Difference between SQL and NO SQL Databases.**

| **Feature** | **SQL (Relational DB)** | **NoSQL (Non-relational DB)** |
| --- | --- | --- |
| **Data Model** | Structured, tabular (rows and columns) | Flexible (JSON, key-value, document, graph, etc.) |
| **Schema** | Fixed schema (predefined tables and types) | Dynamic schema (you can store different formats) |
| **Query Language** | SQL (Structured Query Language) | Varies: MongoDB uses JSON-like queries, etc. |
| **Scalability** | Vertical (scale-up: more CPU, RAM) | Horizontal (scale-out: add more servers) |
| **Transactions** | Strong ACID compliance (Atomicity, Consistency, Isolation, Durability) | Often BASE (Basically Available, Soft state, Eventually consistent) |
| **Examples** | MySQL, PostgreSQL, Oracle, SQL Server | MongoDB, Cassandra, Redis, CouchDB, DynamoDB |
| **Use Case** | Complex queries, structured data, strong consistency | High-speed, large-scale, semi-structured or unstructured data |
| **Joins** | Supported (foreign key relationships) | Typically not supported (denormalized data instead) |
| **Best For** | Financial systems, ERPs, legacy applications | Real-time apps, IoT, big data, content management |