Task 06 - Discussion Questions

Question 1: In what scenario can your docker-compose file be used and what problem is addressed by this?

Scenarios for Use:

1. **AI Model Deployment Pipeline**
   - Scenario: Deploying trained AI models to production
   - Problem Solved: Ensures consistent environment across development, testing, and production
   - Example: Our compose file separates data, model, and execution logic into isolated containers
2. **Microservices Architecture**
   - Scenario: Breaking monolithic applications into smaller, independent services
   - Problem Solved: Each container handles one responsibility (data storage, model storage, code execution)
   - Benefit: Easy to update one component without affecting others
3. **Reproducible Research**
   - Scenario: Scientific research requiring exact environment replication
   - Problem Solved: Anyone can reproduce our AI model results by running the same containers
   - Benefit: Eliminates "it works on my machine" problems
4. **Testing & CI/CD**
   - Scenario: Automated testing in continuous integration pipelines
   - Problem Solved: Spin up identical test environments on-demand
   - Benefit: Fast, isolated, reproducible tests

5. **Educational/Training Environments**
   - Scenario: Teaching AI/ML concepts to students
   - Problem Solved: Students don't need to install complex dependencies (PyBrain, SciPy, etc.)
   - Benefit: Consistent environment for all students

Problems Addressed:

Problem	How Docker Compose Solves It
Dependency Hell	Each container has isolated dependencies (Python 3.11, PyBrain, SciPy)
Environment Inconsistency	Same containers run identically on any machine (Windows, Mac, Linux)
Complex Setup	Single command (`docker-compose up`) starts entire pipeline
Version Management	Specific versions locked in Dockerfiles (Python 3.11, SciPy 1.10.1)
Resource Isolation	Containers share nothing except explicitly mounted volumes
Scalability	Easy to add more containers or scale existing ones

---

Question 2: What do you need to change in your docker-compose file setting in order to apply the python model constructed?

Changes Required:

Current Setup:

Our current `docker-compose.yml` uses **one specific model** (`currentSolution.pkl`) hardcoded in Container 2.

To Make It Flexible for Different Models:

Option A: Use Environment Variables

```yaml
```

```yaml
version: '3.8'

services:
 knowledgebase:
  image: knowledgebase_app07
  volumes:
   - ./shared_data:/shared_data
   - ${MODEL_PATH}:/knowledgeBase/currentSolution.pkl  # ← Dynamic model path
  environment:
   - MODEL_NAME=${MODEL_NAME}
  command: sh -c "cp /knowledgeBase/currentSolution.pkl /shared_data/${MODEL_NAME}.pkl"
```

Usage:

```bash
MODEL_PATH=../Task04/new_model.pkl MODEL_NAME=model_v2 docker-compose up
```

Option B: Build-Time Arguments

Modify Container 2 Dockerfile:

```dockerfile
FROM busybox:latest

ARG MODEL_FILE=currentSolution.pkl
RUN mkdir -p /knowledgeBase
COPY ${MODEL_FILE} /knowledgeBase/currentSolution.pkl
```

docker-compose.yml:

```yaml
services:
  knowledgebase:
    build:
      context: ./container2_knowledgebase
      args:
        MODEL_FILE: ${MODEL_FILE}
    volumes:
      - ./shared_data:/shared_data
```

Usage:

```bash
MODEL_FILE=alternative_model.pkl docker-compose up --build
```

Option C: Volume Mounting (Most Flexible)

```yaml
```

```yaml
version: '3.8'

services:
 knowledgebase:
  image: busybox:latest
  volumes:
   - ./shared_data:/shared_data
   - ./models:/models  # ← Mount entire models directory
  command: sh -c "cp /models/${MODEL_NAME} /shared_data/currentSolution.pkl"
  environment:
   - MODEL_NAME=${MODEL_NAME:-currentSolution.pkl}
```

Folder structure:

```
Task06/
├── models/
│   ├── currentSolution.pkl
│   ├── model_v2.pkl
│   ├── experimental_model.pkl
```

Usage:

```bash
MODEL_NAME=model_v2.pkl docker-compose up
```

Additional Changes Needed:

1. **Update Container 3** to handle different model types:

```python
```

```python
# UE_07_App5.py
import os

model_path = os.getenv('MODEL_PATH', 'currentSolution.pkl')
with open(model_path, 'rb') as f:
    model = pickle.load(f)
```

2. **Update docker-compose.yml** to pass model info to Container 3:

```yaml
codebase:
  image: codebase_app07
  environment:
    - MODEL_PATH=/shared_data/${MODEL_NAME}
```

3. **Add model validation** to ensure correct model type:

```python
assert isinstance(model, FeedForwardNetwork), "Invalid model type!"
```

Question 3: Imagine an algorithmic approach creating the docker-compose file based on Internet requests. What kinds of variables need to be transferred via an adequate request?

API Request Structure:

Imagine a REST API endpoint:

```
POST /api/v1/compose/generate
```

Required Variables in Request:

```
json
```

```
{
  "compose_metadata": {
    "compose_version": "3.8",
    "project_name": "ai_model_deployment"
  },

  "containers": [
    {
      "name": "activationbase",
      "image": "busybox:latest",
      "data_source": {
        "type": "csv",
        "url": "https://example.com/data.csv",
        "container_path": "/activationBase/currentActivation.csv"
      },
      "volumes": ["./shared_data:/shared_data"],
      "command": "cp /activationBase/currentActivation.csv /shared_data/"
    },

    {
      "name": "knowledgebase",
      "image": "busybox:latest",
      "model_source": {
        "type": "pickle",
        "url": "https://example.com/model.pkl",
        "container_path": "/knowledgeBase/currentSolution.pkl"
      },
      "volumes": ["./shared_data:/shared_data"],
      "depends_on": ["activationbase"]
    },

    {
      "name": "codebase",
```

```
    "image": "python:3.11-slim",
    "code_source": {
      "type": "git",
      "repo": "https://github.com/user/ml-code.git",
      "branch": "main",
      "script_path": "/app/activate.py"
    },
    "dependencies": [
      "pandas",
      "numpy",
      "scipy==1.10.1"
    ],
    "environment": {
      "MODEL_PATH": "/shared_data/currentSolution.pkl",
      "DATA_PATH": "/shared_data/activation_data.csv"
    },
    "volumes": ["./shared_data:/shared_data"],
    "depends_on": ["activationbase", "knowledgebase"]
  }
],

"volumes": {
  "shared_data": {
    "driver": "local"
  }
},

"network": {
  "mode": "bridge",
  "driver": "bridge"
},

"execution": {
```

```
    "restart_policy": "on-failure",
    "timeout": 300,
    "abort_on_container_exit": true
  }
}
```

Key Variables Categorized:

1. Container Configuration

- `container_name` - Unique identifier
- `base_image` - Docker image (e.g., `python:3.11`, `busybox`)
- `image_tag` - Specific version tag
- `build_context` - Path to Dockerfile (if building)

2. Data Sources

- `data_url` - URL to download data
- `data_type` - Format (CSV, JSON, XML, Parquet)
- `data_destination` - Container path
- `data_validation` - Schema or checksum

3. Model Information

- `model_url` - Model download URL
- `model_type` - Format (pickle, ONNX, TensorFlow, PyTorch)
- `model_version` - Semantic version

- `model_framework` - PyBrain, TensorFlow, scikit-learn
- `model_input_shape` - Expected input dimensions
- `model_output_shape` - Expected output dimensions

4. Code Execution

- `script_source` - Git repo or direct URL
- `entry_point` - Main script to execute
- `command_line_args` - Arguments to pass to script
- `working_directory` - Container working directory

5. Dependencies

- `python_version` - Python version (3.9, 3.11, 3.12)
- `pip_packages` - List of PyPI packages with versions
- `system_packages` - OS-level packages (apt, yum)
- `custom_libraries` - Git repos or URLs

6. Resource Constraints

- `cpu_limit` - Max CPU usage (e.g., "2.0")
- `memory_limit` - Max RAM (e.g., "512M")
- `gpu_requirements` - GPU device IDs
- `timeout` - Max execution time

7. Networking

- `exposed_ports` - Ports to expose

- `network_mode` - bridge, host, overlay
- `dns_servers` - Custom DNS
- `hostname` - Container hostname

8. Volume Mounts

- `host_path` - Path on host machine
- `container_path` - Path in container
- `read_only` - Boolean flag
- `volume_driver` - local, nfs, etc.

9. Orchestration Logic

- `depends_on` - Container dependencies
- `execution_order` - Sequential or parallel
- `restart_policy` - always, on-failure, unless-stopped
- `health_check` - Command to check container health

10. Security

- `user` - User ID to run as
- `privileged` - Boolean flag
- `security_opt` - Security options
- `capabilities` - Linux capabilities

Example Algorithmic Workflow:

```
python
```

```python
def generate_docker_compose(request_data):
    """
    Algorithmically generate docker-compose.yml from API request
    """
    compose = {
        'version': request_data['compose_metadata']['compose_version'],
        'services': {}
    }

    for container in request_data['containers']:
        service = {
            'image': container['image'],
            'container_name': container['name'],
            'volumes': container.get('volumes', []),
            'depends_on': container.get('depends_on', [])
        }

        # Add data source if present
        if 'data_source' in container:
            service['command'] = generate_copy_command(
                container['data_source']
            )

        # Add environment variables
        if 'environment' in container:
            service['environment'] = container['environment']

        compose['services'][container['name']] = service

    return yaml.dump(compose)
```

Security Considerations for Internet Requests:

1. **Authentication**
   - API keys for model/data downloads
   - OAuth tokens for Git repositories
   - Registry credentials for private Docker images
2. **Validation**
   - Whitelist allowed base images
   - Sanitize URLs to prevent SSRF attacks
   - Validate data schemas before processing
3. **Rate Limiting**
   - Limit compose file generation requests
   - Throttle container launches
   - Cap resource usage per user

Question 4: Imagine to have different infrastructure hardware components, such as CPU/GPU processing or microprocessor architectures. What modifications would your docker-compose file setting need to consider them?

Hardware-Specific Modifications:

A. CPU vs GPU Processing

Current Setup (CPU-only):

```
yaml
```

```yaml
services:
  codebase:
    image: codebase_app07
    # No GPU specified
```

Modified for GPU Support:

```yaml
services:
  codebase_gpu:
    image: codebase_app07_gpu
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: 1
              capabilities: [gpu]
    environment:
      - CUDA_VISIBLE_DEVICES=0
```

Dockerfile changes for GPU:

```dockerfile
```

```
FROM nvidia/cuda:11.8.0-cudnn8-runtime-ubuntu22.04

# Install Python
RUN apt-get update && apt-get install -y python3.11

# Install GPU-enabled packages
RUN pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118
RUN pip install tensorflow[and-cuda]

# Rest of setup...
```

B. Multi-Architecture Support (ARM vs x86)

Problem:

- ARM: Apple M1/M2, Raspberry Pi, AWS Graviton
- x86_64: Intel/AMD processors
- Docker images may not be compatible across architectures

Solution 1: Platform-Specific Images

```yaml
services:
  codebase:
    image: ${PLATFORM_IMAGE}  # Set via environment variable
    platform: ${PLATFORM}      # linux/amd64 or linux/arm64
```

Usage:

```bash
# For x86_64
PLATFORM_IMAGE=codebase_app07:amd64 PLATFORM=linux/amd64 docker-compose up

# For ARM
PLATFORM_IMAGE=codebase_app07:arm64 PLATFORM=linux/arm64 docker-compose up
```

Solution 2: Multi-Arch Builds

Build multi-architecture images:

```bash
docker buildx create --use
docker buildx build --platform linux/amd64,linux/arm64 -t codebase_app07:latest .
```

docker-compose.yml:

```yaml
services:
  codebase:
    image: codebase_app07:latest
    # Docker automatically selects correct architecture
```

C. CPU Core Allocation

```yaml
```

```yaml
services:
  codebase:
    image: codebase_app07
    deploy:
      resources:
        limits:
          cpus: '4.0'       # Max 4 CPU cores
          memory: 8G        # Max 8GB RAM
        reservations:
          cpus: '2.0'       # Guaranteed 2 cores
          memory: 4G        # Guaranteed 4GB RAM
```

D. Complete Hardware-Aware docker-compose.yml

```
yaml
```

```yaml
version: '3.8'

services:
  # ==============================================================
  # Data Container (CPU-only, minimal resources)
  # ==============================================================
  activationbase:
    image: activationbase_app07
    platform: ${PLATFORM:-linux/amd64}
    deploy:
      resources:
        limits:
          cpus: '0.5'
          memory: 256M

  # ==============================================================
  # Model Container (CPU-only, minimal resources)
  # ==============================================================
  knowledgebase:
    image: knowledgebase_app07
    platform: ${PLATFORM:-linux/amd64}
    deploy:
      resources:
        limits:
          cpus: '0.5'
          memory: 256M
    depends_on:
      - activationbase

  # ==============================================================
  # Code Container (CPU or GPU, high resources)
  # ==============================================================
  codebase:
```

```yaml
    image: ${CODE_IMAGE:-codebase_app07}  # CPU or GPU image
    platform: ${PLATFORM:-linux/amd64}
    deploy:
      resources:
        limits:
          cpus: ${CPU_LIMIT:-4.0}
          memory: ${MEMORY_LIMIT:-8G}
        reservations:
          cpus: ${CPU_RESERVATION:-2.0}
          memory: ${MEMORY_RESERVATION:-4G}
          devices:
            - driver: nvidia
              count: ${GPU_COUNT:-0}
              capabilities: [gpu]
    environment:
      - CUDA_VISIBLE_DEVICES=${GPU_DEVICES:-}
      - OMP_NUM_THREADS=${CPU_THREADS:-4}
      - MKL_NUM_THREADS=${CPU_THREADS:-4}
    depends_on:
      - knowledgebase
```

E. Hardware Detection Script

Create `detect_hardware.py`:

```python
```

```python
import platform
import subprocess
import os

def detect_hardware():
    """Detect hardware and set environment variables"""

    # Detect architecture
    arch = platform.machine()
    if arch in ['x86_64', 'AMD64']:
        os.environ['PLATFORM'] = 'linux/amd64'
    elif arch in ['aarch64', 'arm64']:
        os.environ['PLATFORM'] = 'linux/arm64'

    # Detect GPU
    try:
        result = subprocess.run(['nvidia-smi'], capture_output=True)
        if result.returncode == 0:
            os.environ['GPU_COUNT'] = '1'
            os.environ['CODE_IMAGE'] = 'codebase_app07_gpu'
        else:
            os.environ['GPU_COUNT'] = '0'
            os.environ['CODE_IMAGE'] = 'codebase_app07'
    except FileNotFoundError:
        os.environ['GPU_COUNT'] = '0'
        os.environ['CODE_IMAGE'] = 'codebase_app07'

    # Detect CPU cores
    import multiprocessing
    cores = multiprocessing.cpu_count()
    os.environ['CPU_LIMIT'] = str(cores)
    os.environ['CPU_THREADS'] = str(cores // 2)
```

```
    print(f"Hardware detected:")
    print(f"  Platform: {os.environ['PLATFORM']}")
    print(f"  GPU: {'Yes' if os.environ['GPU_COUNT'] > '0' else 'No'}")
    print(f"  CPU Cores: {cores}")

if __name__ == '__main__':
    detect_hardware()
```

Usage:

```bash
python detect_hardware.py && docker-compose up
```

F. Infrastructure-Specific Configurations

Cloud Providers:

AWS EC2 (x86_64 with GPU):

```yaml
```

```yaml
services:
  codebase:
    image: codebase_app07_gpu
    platform: linux/amd64
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: 1
              capabilities: [gpu]
```

AWS Graviton (ARM64):

```yaml
services:
  codebase:
    image: codebase_app07:arm64
    platform: linux/arm64
    deploy:
      resources:
        limits:
          cpus: '8.0'  # Graviton3 has many cores
```

Google Cloud TPU:

```yaml
```

```yaml
services:
  codebase:
    image: gcr.io/tpu-pytorch/xla:latest
    environment:
      - TPU_NAME=my-tpu
      - XRT_TPU_CONFIG="..."
```

Summary Table:

Hardware Component	docker-compose Modification	Additional Requirements
CPU Cores	`deploy.resources.limits.cpus`	None
RAM	`deploy.resources.limits.memory`	None
NVIDIA GPU	`deploy.resources.reservations.devices`	nvidia-docker2 installed
AMD GPU	`devices: [/dev/dri]`	ROCm support
x86_64 Architecture	`platform: linux/amd64`	None
ARM64 Architecture	`platform: linux/arm64`	Multi-arch images
TPU	Custom image + environment variables	GCP account
FPGA	Device mapping + custom drivers	FPGA drivers

Question 5: Imagine the course tutors to test your puzzle piece setup realized up to now. What files need to be submitted so that course tutors are able to evaluate your work?

Complete Submission Package:

Required Files Structure:

```
Task06-Docker-Compose/
│
├── README.md                    # ← Comprehensive documentation
├── discussion.txt               # ← Answers to 5 questions
│
├── container1_activationbase/
│   ├── Dockerfile               # ← Container 1 build instructions
│   └── currentActivation.csv    # ← Sample data entry
│
├── container2_knowledgebase/
│   ├── Dockerfile               # ← Container 2 build instructions
│   └── currentSolution.pkl      # ← AI model file
│
├── container3_codebase/
│   ├── Dockerfile               # ← Container 3 build instructions
│   ├── UE_07_App5.py            # ← Model activation script
│   └── pybrain/                 # ← PyBrain library (entire folder)
│
├── docker-compose.yml           # ← Orchestration file
│
├── App07.py                     # ← Main execution script
├── prepare_files.py             # ← Stage 1 preparation script
├── test_containers.py           # ← Individual container tests
├── run_compose_with_output.py   # ← Compose with output capture
│
└── output/                      # ← All execution results
    ├── stage1_prepare_files/
    │   ├── currentActivation.csv
    │   ├── currentSolution.pkl
    │   └── UE_07_App5.py
    ├── stage2_container_tests/
    │   ├── container1_output.txt
    │   ├── container2_output.txt
```

```
│   ├── container3_pybrain_test.txt
│   └── test_results.json
└── stage3_compose_execution/
    ├── docker_compose_output.log
    ├── activation_data.csv
    ├── currentSolution.pkl
    └── execution_summary.txt
```

Submission Checklist:

Category 1: Docker Build Files (REQUIRED)

- [ ] `container1_activationbase/Dockerfile`
- [ ] `container2_knowledgebase/Dockerfile`
- [ ] `container3_codebase/Dockerfile`

Category 2: Data Files (REQUIRED)

- [ ] `container1_activationbase/currentActivation.csv`
- [ ] `container2_knowledgebase/currentSolution.pkl`
- [ ] `container3_codebase/pybrain/` (entire folder)

Category 3: Code Files (REQUIRED)

- [ ] `container3_codebase/UE_07_App5.py`
- [ ] `App07.py`
- [ ] `prepare_files.py`

Category 4: Orchestration (REQUIRED)

- [ ] `docker-compose.yml`

Category 5: Documentation (REQUIRED)

- [ ] `README.md`
- [ ] `discussion.txt`

Category 6: Test Results (RECOMMENDED)

- [ ] `output/stage2_container_tests/test_results.json`
- [ ] `output/stage3_compose_execution/execution_summary.txt`
- [ ] `output/stage3_compose_execution/docker_compose_output.log`

Category 7: Helper Scripts (OPTIONAL)

- [ ] `test_containers.py`
- [ ] `run_compose_with_output.py`

How Tutors Will Evaluate:

Step 1: Build Containers

```bash
cd container1_activationbase
docker build -t activationbase_app07 .

cd ../container2_knowledgebase
docker build -t knowledgebase_app07 .

cd ../container3_codebase
docker build -t codebase_app07 .
```

Success Criteria:

- All 3 containers build without errors
- Correct tags applied (`activationbase_app07`, etc.)

Step 2: Test Individual Containers

```bash
docker run --rm activationbase_app07
docker run --rm knowledgebase_app07
docker run --rm codebase_app07 python3 -c "import sys; sys.path.append('/opt/pybrain'); from pybrain.structure impo
```

Success Criteria:

- Container 1 outputs CSV data
- Container 2 lists model file
- Container 3 successfully imports PyBrain

Step 3: Run Docker Compose

```bash
docker-compose up --abort-on-container-exit
```

Success Criteria:

- All 3 containers start in correct order
- Data copied to shared volume
- Model loaded successfully

- Prediction output displayed

Step 4: Verify Outputs

```bash
dir shared_data
```

Success Criteria:

- `activation_data.csv` exists
- `currentSolution.pkl` exists
- Both files have correct content

Step 5: Review Documentation

- Read `README.md` for completeness
- Check `discussion.txt` for thoughtful answers
- Verify all 5 questions answered

Grading Rubric (Example):

Component	Points	Evaluation Criteria
Container 1 Build	10	Dockerfile correct, builds successfully, correct tag
Container 2 Build	10	Dockerfile correct, builds successfully, correct tag
Container 3 Build	15	Dockerfile correct, PyBrain installed, builds successfully

Component	Points	Evaluation Criteria
docker-compose.yml	20	Correct syntax, proper orchestration, dependencies set
File Copying	10	Data and model copied correctly to shared volume
Model Activation	15	Model loads and produces prediction output
Documentation	10	README complete, clear instructions
Discussion Answers	10	All 5 questions answered thoroughly
Total	**100**	

Optional: Create Submission Archive

bash

```bash
# Create ZIP file for submission
python create_submission.py
```

create_submission.py:

python

```python
import shutil
import os

files_to_include = [
    'README.md',
    'discussion.txt',
    'docker-compose.yml',
    'App07.py',
    'prepare_files.py',
    'container1_activationbase/',
    'container2_knowledgebase/',
    'container3_codebase/',
    'output/'
]

shutil.make_archive('Task06_Submission', 'zip', '.', files_to_include)
print("✓ Created Task06_Submission.zip")
```

Final Submission:

Single file: `Task06_Submission.zip`

OR

Git repository with all files committed and tagged:

```bash
git add .
git commit -m "Task 06 Complete - Docker Compose Orchestration"
git tag task06-submission
git push origin main --tags
```

Summary

These discussion answers demonstrate:

1. Understanding of Docker Compose use cases
2. Flexibility in model deployment
3. API design for dynamic compose generation
4. Hardware-aware containerization
5. Complete submission requirements for evaluation