CPF Deployment Tutorial - Complete Guide

This guide covers deploying the Cybersecurity Psychology Framework (CPF) both locally and in the cloud.

Table of Contents

Quick Start (Cloud Only)

Option 1: Hugging Face Spaces (Recommended for Prototyping)

Step 1: Create HF Space

1. Go to Hugging Face Spaces
2. Click "Create new Space"
3. Choose:
   - **Space name**: `cpf-prototype` (or your preference)
   - **License**: `MIT`
   - **Space SDK**: `Gradio`
   - **Hardware**: `CPU basic` (sufficient for prototype)

Step 2: Upload Files Create these files in your HF Space:

```
cpf-prototype/
├── app.py                  # Main Gradio app
├── requirements.txt        # Dependencies
├── README.md              # Documentation
└── cpf_validation.db      # Will be created automatically
```

app.py (copy the CPF prototype code from previous artifact)

requirements.txt:

```
gradio==4.11.0
transformers==4.36.0
torch==2.1.0
numpy==1.24.3
sqlite3
hashlib
re
json
datetime
```

Step 3: Deploy

- HF Spaces will automatically build and deploy
- Your app will be available at: `https://huggingface.co/spaces/YOUR_USERNAME/cpf-prototype`
- Share link works immediately for demos

Local Development Setup

Option A: Direct Python Installation

Step 1: Environment Setup

```
# Create virtual environment
python -m venv cpf_env

# Activate environment
# On Windows:
cpf_env\Scripts\activate
# On macOS/Linux:
source cpf_env/bin/activate

# Upgrade pip
python -m pip install --upgrade pip
```

Step 2: Install Dependencies

```
pip install gradio==4.11.0
pip install transformers==4.36.0
pip install torch==2.1.0
pip install numpy==1.24.3
pip install streamlit==1.29.0
pip install pandas==2.0.3
pip install plotly==5.17.0
pip install scikit-learn==1.3.0
pip install matplotlib==3.7.2
pip install seaborn==0.12.2
```

Step 3: Create Project Structure

```
mkdir cpf_prototype
cd cpf_prototype

# Create main files
touch app.py
touch validation_dashboard.py
touch requirements.txt
mkdir data
mkdir models
mkdir config
```

Step 4: Run Locally

```
# Run Gradio prototype
python app.py

# Run validation dashboard (separate terminal)
streamlit run validation_dashboard.py --server.port 8502
```

Access Points:

- Gradio app: `http://localhost:7860`
- Validation dashboard: `http://localhost:8502`

Option B: Docker Setup

Step 1: Create Dockerfile

```
FROM python:3.10-slim

WORKDIR /app

# Install system dependencies
RUN apt-get update && apt-get install -y \
    gcc \
    g++ \
    && rm -rf /var/lib/apt/lists/*

# Copy requirements first for better caching
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application files
COPY . .

# Create data directory with proper permissions
```

```
RUN mkdir -p /app/data && chmod 777 /app/data

EXPOSE 7860 8502

# Run both services
CMD ["python", "app.py"]
```

Step 2: Create docker-compose.yml

```
version: '3.8'

services:
  cpf-prototype:
    build: .
    ports:
      - "7860:7860"
      - "8502:8502"
    volumes:
      - ./data:/app/data
      - ./models:/app/models
    environment:
      - GRADIO_SERVER_NAME=0.0.0.0
      - GRADIO_SERVER_PORT=7860
    restart: unless-stopped

  cpf-dashboard:
    build: .
    ports:
      - "8503:8502"
    volumes:
      - ./data:/app/data
    command: ["streamlit", "run", "validation_dashboard.py", "--server.address=0.0.0.0", "--server.port=8502"]
    depends_on:
      - cpf-prototype
    restart: unless-stopped
```

Step 3: Run with Docker

```
# Build and run
docker-compose up --build

# Run in background
docker-compose up -d

# View logs
docker-compose logs -f

# Stop services
docker-compose down
```

Cloud Deployment Options

Option 1: Hugging Face Spaces (Extended)

For Production-Ready Deployment:

Step 1: Upgrade Hardware

- Go to Space settings
- Choose `CPU Upgrade` or `GPU - T4 small` for better performance
- Enable `Persistent Storage` for database retention

Step 2: Environment Variables Add to your Space settings:

```
GRADIO_SHARE=False
GRADIO_AUTH=username:password  # Optional authentication
HF_TOKEN=your_hf_token        # For model downloads
```

Step 3: Add Secrets For production features, add these files:

- `.env` (not committed to git)
- `secrets.toml` for Streamlit secrets

Option 2: Streamlit Cloud

Step 1: Prepare Repository

```
# Push to GitHub
git init
git add .
git commit -m "Initial CPF prototype"
git branch -M main
git remote add origin https://github.com/USERNAME/cpf-prototype.git
git push -u origin main
```

Step 2: Deploy on Streamlit Cloud

1. Go to share.streamlit.io
2. Connect GitHub account
3. Select repository: `USERNAME/cpf-prototype`
4. Main file path: `validation_dashboard.py`
5. Click "Deploy"

Step 3: Configure Secrets In Streamlit Cloud dashboard, add secrets:

```
[database]
path = "cpf_validation.db"

[auth]
username = "admin"
password = "your_secure_password"
```

Option 3: Railway/Render

Railway Deployment:

```
# Install Railway CLI
npm install -g @railway/cli

# Login and deploy
railway login
railway init
railway up
```

Render Deployment:

1. Connect GitHub repository
2. Choose "Web Service"
3. Build command: `pip install -r requirements.txt`
4. Start command: `python app.py`

Option 4: Google Colab (Quick Testing)

Step 1: Create Notebook

```
# Install dependencies
!pip install gradio transformers torch

# Clone your code or copy directly
!git clone https://github.com/YOUR_USERNAME/cpf-prototype.git
%cd cpf-prototype

# Run with public URL
!python app.py --share
```

Step 2: Access

- Colab will display a public URL
- Valid for 72 hours
- Perfect for quick demos

Database Configuration

SQLite (Default - Recommended for Prototyping)

Advantages:

- No setup required
- File-based, portable
- Perfect for development

Configuration:

```
# In your app.py
DATABASE_PATH = "data/cpf_validation.db"
conn = sqlite3.connect(DATABASE_PATH, check_same_thread=False)
```

PostgreSQL (Production)

Step 1: Setup Database

```
# Local PostgreSQL
createdb cpf_validation

# Or use cloud service (Supabase, Neon, etc.)
```

Step 2: Update Connection

```python
import psycopg2
from sqlalchemy import create_engine

DATABASE_URL = "postgresql://user:password@localhost/cpf_validation"
engine = create_engine(DATABASE_URL)
```

Step 3: Environment Variables

```
# .env file
DATABASE_URL=postgresql://user:password@host:port/database
DATABASE_SSL_MODE=require
```

Cloud Database Options

Supabase (Recommended):

1. Create account at supabase.com
2. Create new project
3. Get connection string from Settings > Database
4. Use in your app

Railway PostgreSQL:

```
railway add postgresql
railway variables  # Get DATABASE_URL
```

Production Considerations

Performance Optimization

Model Optimization:

```python
# Use quantized models for faster inference
from transformers import pipeline

# Load with optimization
sentiment_analyzer = pipeline(
    "sentiment-analysis",
    model="cardiffnlp/twitter-roberta-base-sentiment-latest",
    device=0 if torch.cuda.is_available() else -1,
    model_kwargs={"torch_dtype": torch.float16}
)
```

Caching:

```python
import functools
from functools import lru_cache

@lru_cache(maxsize=1000)
def analyze_text_cached(text_hash):
    # Your analysis logic here
    pass
```

Async Processing:

```python
import asyncio
import concurrent.futures

async def process_batch(texts):
    with concurrent.futures.ThreadPoolExecutor() as executor:
        loop = asyncio.get_event_loop()
        tasks = [
            loop.run_in_executor(executor, analyze_text, text)
            for text in texts
        ]
        return await asyncio.gather(*tasks)
```

Security

Authentication:

```python
# Gradio authentication
demo.launch(auth=("username", "password"))

# Streamlit authentication
def check_password():
    def password_entered():
        if st.session_state["password"] == "your_secure_password":
            st.session_state["password_correct"] = True
            del st.session_state["password"]
        else:
            st.session_state["password_correct"] = False

    if "password_correct" not in st.session_state:
        st.text_input("Password", type="password",
                     on_change=password_entered, key="password")
        return False
    elif not st.session_state["password_correct"]:
        st.text_input("Password", type="password",
                     on_change=password_entered, key="password")
        st.error("Password incorrect")
        return False
    else:
        return True
```

Rate Limiting:

```
import time
from collections import defaultdict

class RateLimiter:
    def __init__(self, max_requests=100, window=3600):
        self.max_requests = max_requests
        self.window = window
        self.requests = defaultdict(list)

    def allow_request(self, identifier):
        now = time.time()
        requests = self.requests[identifier]

        # Clean old requests
        requests[:] = [req for req in requests if now - req < self.window]

        if len(requests) >= self.max_requests:
            return False

        requests.append(now)
        return True
```

Monitoring

Logging:

```
import logging
import sys

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('cpf.log'),
        logging.StreamHandler(sys.stdout)
    ]
)

logger = logging.getLogger(__name__)
```

Health Checks:

```
@app.route('/health')
def health_check():
    return {
        "status": "healthy",
        "timestamp": datetime.now().isoformat(),
        "version": "1.0.0"
    }
```

Troubleshooting

Common Issues

1. Model Download Failures

```
# Solution 1: Set HF token
export HF_TOKEN=your_token

# Solution 2: Manual download
huggingface-cli download microsoft/DialoGPT-medium

# Solution 3: Use local cache
export TRANSFORMERS_CACHE=/path/to/cache
```

2. Memory Issues

```
# Reduce model precision
model_kwargs = {"torch_dtype": torch.float16}

# Use CPU instead of GPU
device = -1  # Force CPU

# Implement model loading on demand
@functools.lru_cache(maxsize=1)
def get_model():
    return pipeline("sentiment-analysis", model="model_name")
```

3. Database Connection Issues

```
# SQLite permission fix
import os
os.chmod("cpf_validation.db", 0o666)

# PostgreSQL connection retry
import time
import psycopg2
```

```
def connect_with_retry(database_url, max_retries=5):
    for attempt in range(max_retries):
        try:
            return psycopg2.connect(database_url)
        except psycopg2.OperationalError:
            if attempt < max_retries - 1:
                time.sleep(2 ** attempt)
            else:
                raise
```

4. Gradio Sharing Issues

```
# Solution 1: Use ngrok
import subprocess
subprocess.run(["ngrok", "http", "7860"])

# Solution 2: Manual sharing
demo.launch(share=True, server_name="0.0.0.0")

# Solution 3: Cloud deployment
# Use HF Spaces or other cloud options
```

Performance Optimization

1. Slow Inference

```
# Batch processing
def analyze_batch(texts, batch_size=32):
    results = []
    for i in range(0, len(texts), batch_size):
        batch = texts[i:i+batch_size]
        batch_results = model(batch)
        results.extend(batch_results)
    return results

# Model quantization
from transformers import AutoModel
import torch

model = AutoModel.from_pretrained(
    "model_name",
    torch_dtype=torch.float16,
    device_map="auto"
)
```

2. Database Performance

```
-- Create indexes for better query performance
CREATE INDEX idx_analyses_timestamp ON analyses(timestamp);
CREATE INDEX idx_analyses_validation_status ON analyses(validation_status);
CREATE INDEX idx_feedback_analysis_id ON feedback(analysis_id);
```

Deployment Debugging

Check Environment:

```
# Python version
python --version

# Package versions
pip list | grep gradio
pip list | grep transformers

# GPU availability
python -c "import torch; print(torch.cuda.is_available())"

# Memory usage
python -c "import psutil; print(f'RAM: {psutil.virtual_memory().percent}%')"
```

Log Analysis:

```
# Gradio logs
tail -f ~/.gradio/logs/gradio.log

# Application logs
tail -f cpf.log

# System logs (Linux)
journalctl -u your-service-name -f
```

Cloud-Specific Issues

Hugging Face Spaces:

```
# Check space logs in HF interface
# Common fix: reduce memory usage
import gc
import torch

def cleanup_memory():
    gc.collect()
    if torch.cuda.is_available():
        torch.cuda.empty_cache()
```

Streamlit Cloud:

```
# Check requirements.txt versions
# Common fix: pin specific versions
streamlit==1.29.0
pandas==2.0.3
```

Quick Reference Commands

Local Development

```
# Start development environment
python -m venv cpf_env && source cpf_env/bin/activate
pip install -r requirements.txt
python app.py

# Run validation dashboard
streamlit run validation_dashboard.py --server.port 8502
```

Docker

```
# Quick start
docker-compose up --build

# Restart services
docker-compose restart

# View logs
docker-compose logs -f cpf-prototype
```

Cloud Deployment

```
# HF Spaces: Upload files via web interface or:
git clone https://huggingface.co/spaces/USERNAME/SPACE_NAME
# Edit files and push

# Streamlit Cloud: Push to GitHub
git add . && git commit -m "Update" && git push

# Railway
railway up
```

Database Management

```
# Backup SQLite
cp cpf_validation.db cpf_validation_backup.db

# View data
sqlite3 cpf_validation.db ".tables"
sqlite3 cpf_validation.db "SELECT COUNT(*) FROM analyses;"
```

This guide covers both quick prototyping and production deployments. Start with local development for testing, then move to cloud deployment for demos and production use.