**PERFORMANCE TESTING**

**Objective of Performance Testing:**

The purpose of this phase is to evaluate the efficiency, scalability, and responsiveness of the AI-powered application.

Key goals:

Measure response time of AI model for different prompts.

Assess resource utilization (CPU, GPU, memory).

Validate the scalability of Gradio web interface under multiple user requests.

Identify bottlenecks that affect performance.

**Test Environment:**

Programming Language: Python 3.10+

Frameworks:

gradio (UI + API simulation)

torch (GPU/CPU execution)

transformers (model loading & inference)

Model: ibm-granite/granite-3.2-2b-instruct (2B parameters, instruction-tuned)

Hardware Configurations Tested:

CPU Mode: Intel i5, 8GB RAM (no GPU)

GPU Mode: NVIDIA RTX 3060, 12GB VRAM

**Performance Testing Methodology:**

Metrics Considered:

1. Response Latency → Time from user input to model output.

2. Throughput → Number of queries handled per second.

3. Resource Usage → CPU %, GPU memory consumption, RAM usage.

4. Scalability → How the system behaves with multiple users.

5. Accuracy & Relevance → Coherence of generated answers.

Test Scenarios:

Scenario 1: Single user query (baseline test).

Scenario 2: Multiple sequential queries (stress test).

Scenario 3: Long prompts (edge case handling).

Scenario 4: Parallel requests using Gradio’s API.

**Results & Observations:**

Test Case CPU Mode (Avg Response) GPU Mode (Avg Response) Observations

Simple City Analysis (short input) ~8-12s ~2-4s GPU significantly faster

Citizen Query (medium length) ~12-18s ~4-6s Stable results

Long prompt (edge case, ~500 tokens) ~25s+ ~10-12s Higher latency, but output generated

Concurrent requests (5 users) High lag, occasional crash Moderate slowdown, still responsive Needs queue management

**Bottlenecks Identified:**

CPU-only execution → Very slow, not suitable for production.

Memory usage → Granite 2B model requires >8GB RAM for smooth execution.

Prompt length sensitivity → Longer inputs increase response latency.

Scalability limits → Gradio app struggles under multiple parallel requests without GPU.

**Optimization Suggestions:**

1. Model Optimization:

Use quantized versions (e.g., INT8, 4-bit models) to reduce memory load.

Apply smaller Granite models (1B or distilled versions) for faster inference.

2. Infrastructure Improvements:

Deploy on GPU-based servers (AWS, GCP, Azure).

Add caching mechanism for repeated queries.

Use asynchronous request handling in Gradio.

3. Scalability Enhancements:

Integrate with FastAPI backend for load balancing.

Use Docker + Kubernetes for distributed deployment.

**Conclusion of Testing:**

The system works well for individual users and medium-length queries on GPU.

Performance degrades significantly on CPU-only setups.

For scalable deployment, GPU infrastructure and optimization are mandatory.

Further testing with real-world data load is required before production rollout.