AI Risk Repository Chatbot: Developer Performance Testing Plan

# Executive Summary

This document provides a comprehensive technical framework for evaluating the performance of the AI Risk Repository Chatbot. It includes automated testing protocols, performance metrics collection, system stress testing, and continuous monitoring strategies.

# 1. Performance Testing Framework

## 1.1 Testing Architecture

Our performance testing framework consists of multiple layers designed to evaluate different aspects of system performance:

* Unit Testing: Individual component performance
* Integration Testing: API endpoint response times
* System Testing: End-to-end query processing
* Load Testing: Concurrent user simulation
* Stress Testing: Breaking point identification

## 1.2 Key Performance Indicators

|  |  |  |  |
| --- | --- | --- | --- |
| Metric | Target | Acceptable | Critical |
| Response Time (P95) | <2s | <3s | >5s |
| Accuracy Rate | >95% | >85% | <85% |
| Error Rate | <2% | <5% | >5% |
| Uptime | >99.9% | >99% | <99% |
| Cache Hit Rate | >80% | >60% | <60% |
| Token Efficiency | <2000/query | <3000/query | >3000/query |

# 2. Ground Truth Dataset Creation

## 2.1 Query Categories

Test queries are organized into five categories to ensure comprehensive coverage:

### Category A: Factual Queries

Queries with verifiable, exact answers from the repository:

test\_queries\_factual = [  
 {"query": "How many risks are in the repository?",   
 "expected": "1612",   
 "type": "exact\_match",  
 "priority": "critical"},  
 {"query": "What percentage of risks are in the discrimination domain?",   
 "expected": "15%",   
 "type": "exact\_match",  
 "priority": "critical"},  
 {"query": "How many domains exist?",   
 "expected": "7",   
 "type": "exact\_match",  
 "priority": "critical"},  
 {"query": "What is the largest risk domain?",   
 "expected": "socioeconomic and environmental harms",   
 "type": "contains",  
 "priority": "high"}  
]

### Category B: Taxonomy Queries

Queries testing understanding of risk categorization:

test\_queries\_taxonomy = [  
 {"query": "What are the 7 domains of AI risk?",   
 "expected\_terms": ["discrimination", "privacy", "misinformation",   
 "malicious use", "security",   
 "human-computer interaction",   
 "socioeconomic and environmental harms"],  
 "type": "contains\_all",  
 "priority": "critical"},  
 {"query": "How are risks categorized by entity?",   
 "expected\_terms": ["human", "AI", "human & AI"],  
 "type": "contains\_any",  
 "priority": "high"}  
]

### Category C: Metadata Queries

Queries testing metadata service functionality:

test\_queries\_metadata = [  
 {"query": "What are the main risk domains in the AI risk repository?",  
 "should\_not\_error": True,  
 "expected\_service": "metadata",  
 "priority": "critical"},  
 {"query": "Show me risks by timing",  
 "expected\_terms": ["pre-deployment", "post-deployment"],  
 "type": "classification",  
 "priority": "high"}  
]

### Category D: Edge Cases

Queries testing system robustness and error handling:

* Empty query: ""
* Single character: "a"
* Special characters: "!@#$%^&\*()"
* SQL injection attempt: "; DROP TABLE risks;--"
* Extremely long query: 5000+ characters
* Non-English characters: "风险是什么？"
* Mixed languages: "What are the risques principales?"

### Category E: Performance Stress Queries

Queries designed to stress specific components:

* Complex multi-domain comparison queries
* Queries requiring extensive document retrieval
* Queries with high token generation requirements
* Rapid sequential queries (rate limiting test)
* Queries requiring multiple service calls

# 3. Automated Testing Implementation

## 3.1 Test Suite Architecture

import time  
import json  
import asyncio  
from datetime import datetime  
from collections import defaultdict  
from typing import Dict, List, Any  
  
class PerformanceTestSuite:  
 def \_\_init\_\_(self):  
 self.metrics = {  
 "response\_times": [],  
 "intent\_classification": {"correct": 0, "incorrect": 0},  
 "citation\_validity": [],  
 "error\_types": defaultdict(int),  
 "token\_usage": [],  
 "cache\_metrics": {"hits": 0, "misses": 0},  
 "database\_queries": []  
 }  
 self.test\_results = []  
   
 async def run\_comprehensive\_test(self):  
 """Execute all test categories and collect metrics."""  
 test\_categories = [  
 self.test\_factual\_accuracy,  
 self.test\_taxonomy\_understanding,  
 self.test\_metadata\_service,  
 self.test\_edge\_cases,  
 self.test\_performance\_limits  
 ]  
   
 for test\_func in test\_categories:  
 await test\_func()  
   
 return self.generate\_report()  
   
 def generate\_report(self) -> Dict[str, Any]:  
 """Generate comprehensive performance report."""  
 return {  
 "timestamp": datetime.now().isoformat(),  
 "total\_tests": len(self.test\_results),  
 "passed": sum(1 for t in self.test\_results if t["passed"]),  
 "failed": sum(1 for t in self.test\_results if not t["passed"]),  
 "avg\_response\_time": sum(self.metrics["response\_times"]) / len(self.metrics["response\_times"]),  
 "p95\_response\_time": self.calculate\_percentile(95),  
 "accuracy\_rate": self.calculate\_accuracy\_rate(),  
 "error\_breakdown": dict(self.metrics["error\_types"]),  
 "cache\_hit\_rate": self.calculate\_cache\_hit\_rate()  
 }

## 3.2 Performance Monitoring Decorators

import functools  
import tracemalloc  
import psutil  
  
def performance\_monitor(func):  
 """Decorator to monitor function performance."""  
 @functools.wraps(func)  
 async def wrapper(\*args, \*\*kwargs):  
 # Start monitoring  
 start\_time = time.perf\_counter()  
 tracemalloc.start()  
 start\_memory = psutil.Process().memory\_info().rss / 1024 / 1024 # MB  
   
 try:  
 result = await func(\*args, \*\*kwargs)  
 success = True  
 error = None  
 except Exception as e:  
 result = None  
 success = False  
 error = str(e)  
   
 # End monitoring  
 end\_time = time.perf\_counter()  
 current, peak = tracemalloc.get\_traced\_memory()  
 tracemalloc.stop()  
 end\_memory = psutil.Process().memory\_info().rss / 1024 / 1024 # MB  
   
 # Log metrics  
 metrics = {  
 "function": func.\_\_name\_\_,  
 "execution\_time": end\_time - start\_time,  
 "memory\_used": end\_memory - start\_memory,  
 "peak\_memory": peak / 1024 / 1024, # MB  
 "success": success,  
 "error": error  
 }  
   
 log\_performance\_metrics(metrics)  
 return result  
   
 return wrapper

# 4. Load and Stress Testing

## 4.1 Concurrent User Simulation

async def simulate\_concurrent\_users(num\_users: int, duration: int, queries\_per\_second: float):  
 """Simulate concurrent users accessing the chatbot."""  
   
 async def user\_session(user\_id: int):  
 session\_metrics = []  
 query\_interval = 1.0 / queries\_per\_second  
   
 start\_time = time.time()  
 while time.time() - start\_time < duration:  
 query = random.choice(TEST\_QUERIES)  
   
 query\_start = time.time()  
 try:  
 response = await send\_query(query)  
 latency = time.time() - query\_start  
 session\_metrics.append({  
 "user\_id": user\_id,  
 "timestamp": time.time(),  
 "latency": latency,  
 "success": True  
 })  
 except Exception as e:  
 session\_metrics.append({  
 "user\_id": user\_id,  
 "timestamp": time.time(),  
 "error": str(e),  
 "success": False  
 })  
   
 await asyncio.sleep(query\_interval)  
   
 return session\_metrics  
   
 # Run concurrent sessions  
 tasks = [user\_session(i) for i in range(num\_users)]  
 results = await asyncio.gather(\*tasks)  
   
 return analyze\_load\_test\_results(results)

## 4.2 Stress Test Scenarios

|  |  |  |  |
| --- | --- | --- | --- |
| Scenario | Users | Duration | Expected Outcome |
| Baseline Load | 10 | 5 min | All requests succeed, <2s latency |
| Normal Load | 50 | 10 min | 95% succeed, <3s P95 latency |
| High Load | 100 | 10 min | 90% succeed, <5s P95 latency |
| Stress Test | 200 | 5 min | Identify breaking point |
| Spike Test | 10→100→10 | 15 min | Graceful scaling |
| Endurance Test | 50 | 60 min | No memory leaks |

# 5. Database and Cache Performance

## 5.1 Query Optimization Testing

def test\_database\_performance():  
 """Test ChromaDB query performance."""  
   
 test\_cases = [  
 {  
 "name": "Simple keyword search",  
 "query": "privacy risks",  
 "expected\_time": 0.5  
 },  
 {  
 "name": "Complex semantic search",  
 "query": "What are the unintended consequences of AI in healthcare?",  
 "expected\_time": 1.0  
 },  
 {  
 "name": "Metadata filtering",  
 "query": "risks where domain='discrimination' AND timing='post-deployment'",  
 "expected\_time": 0.3  
 }  
 ]  
   
 for test in test\_cases:  
 start = time.time()  
 results = vectorstore.similarity\_search(  
 test["query"],  
 k=10,  
 filter=test.get("filter", None)  
 )  
 elapsed = time.time() - start  
   
 assert elapsed < test["expected\_time"], f"{test['name']} too slow: {elapsed}s"  
 assert len(results) > 0, f"{test['name']} returned no results"

## 5.2 Cache Effectiveness Metrics

* Cache hit rate per query type
* Cache size growth over time
* Cache eviction rate
* Time saved by cache hits
* Memory usage by cache
* Cache warm-up time after restart

# 6. Token Usage Optimization

## 6.1 Token Tracking Implementation

class TokenUsageTracker:  
 def \_\_init\_\_(self):  
 self.usage\_history = []  
 self.cost\_per\_1k\_input = 0.01 # Example rate  
 self.cost\_per\_1k\_output = 0.03 # Example rate  
   
 def track\_query(self, query: str, response: str, model: str):  
 usage = {  
 "timestamp": datetime.now(),  
 "model": model,  
 "input\_tokens": self.count\_tokens(query),  
 "output\_tokens": self.count\_tokens(response),  
 "total\_tokens": None,  
 "estimated\_cost": None  
 }  
   
 usage["total\_tokens"] = usage["input\_tokens"] + usage["output\_tokens"]  
 usage["estimated\_cost"] = (  
 (usage["input\_tokens"] / 1000) \* self.cost\_per\_1k\_input +  
 (usage["output\_tokens"] / 1000) \* self.cost\_per\_1k\_output  
 )  
   
 self.usage\_history.append(usage)  
 return usage  
   
 def get\_optimization\_recommendations(self):  
 avg\_input = sum(u["input\_tokens"] for u in self.usage\_history) / len(self.usage\_history)  
 avg\_output = sum(u["output\_tokens"] for u in self.usage\_history) / len(self.usage\_history)  
   
 recommendations = []  
 if avg\_input > 500:  
 recommendations.append("Consider prompt optimization to reduce input tokens")  
 if avg\_output > 1000:  
 recommendations.append("Consider response length limits")  
   
 return recommendations

# 7. Citation Validation Framework

## 7.1 Automated Citation Testing

def validate\_citations(response: str, documents: List[Document]) -> Dict:  
 """Validate all citations in a response."""  
   
 validation\_results = {  
 "total\_citations": 0,  
 "valid\_citations": 0,  
 "broken\_links": [],  
 "unsupported\_claims": [],  
 "citation\_coverage": 0.0  
 }  
   
 # Extract all RID citations  
 rid\_pattern = r'RID-\d{5}'  
 citations = re.findall(rid\_pattern, response)  
 validation\_results["total\_citations"] = len(citations)  
   
 # Validate each citation  
 for rid in citations:  
 # Check if RID exists in documents  
 if any(doc.metadata.get("rid") == rid for doc in documents):  
 validation\_results["valid\_citations"] += 1  
   
 # Verify snippet is accessible  
 snippet\_path = f"/snippet/{rid}"  
 if not check\_snippet\_exists(rid):  
 validation\_results["broken\_links"].append(rid)  
 else:  
 validation\_results["unsupported\_claims"].append(rid)  
   
 # Calculate citation coverage  
 if validation\_results["total\_citations"] > 0:  
 validation\_results["citation\_coverage"] = (  
 validation\_results["valid\_citations"] /   
 validation\_results["total\_citations"]  
 )  
   
 return validation\_results

# 8. Regression Testing Strategy

## 8.1 Continuous Integration Pipeline

# .github/workflows/performance-tests.yml  
name: Performance Regression Tests  
  
on:  
 push:  
 branches: [main, develop]  
 pull\_request:  
 branches: [main]  
  
jobs:  
 performance-test:  
 runs-on: ubuntu-latest  
   
 steps:  
 - uses: actions/checkout@v2  
   
 - name: Set up Python  
 uses: actions/setup-python@v2  
 with:  
 python-version: '3.11'  
   
 - name: Install dependencies  
 run: |  
 pip install -r requirements.txt  
 pip install pytest-benchmark  
   
 - name: Run performance tests  
 run: |  
 python -m pytest tests/performance/ \  
 --benchmark-only \  
 --benchmark-compare \  
 --benchmark-autosave  
   
 - name: Check performance regression  
 run: |  
 python scripts/check\_regression.py  
   
 - name: Upload results  
 uses: actions/upload-artifact@v2  
 with:  
 name: performance-results  
 path: .benchmarks/

## 8.2 Performance Baseline Management

Baseline metrics to track across releases:

* Average response time for each query category
* P50, P95, P99 latency percentiles
* Memory usage under standard load
* Database query execution times
* Token usage per query type
* Error rates by category
* Cache effectiveness metrics

# 9. Performance Monitoring Dashboard

## 9.1 Real-time Metrics Collection

from prometheus\_client import Counter, Histogram, Gauge  
import time  
  
# Define metrics  
query\_counter = Counter('chatbot\_queries\_total', 'Total number of queries')  
query\_duration = Histogram('chatbot\_query\_duration\_seconds', 'Query duration')  
active\_sessions = Gauge('chatbot\_active\_sessions', 'Number of active sessions')  
error\_counter = Counter('chatbot\_errors\_total', 'Total errors', ['error\_type'])  
  
@query\_duration.time()  
def process\_query\_with\_metrics(query: str):  
 """Process query with automatic metrics collection."""  
 query\_counter.inc()  
   
 try:  
 with active\_sessions.track\_inprogress():  
 response = process\_query(query)  
 return response  
 except Exception as e:  
 error\_counter.labels(error\_type=type(e).\_\_name\_\_).inc()  
 raise

## 9.2 Alert Thresholds

|  |  |  |
| --- | --- | --- |
| Metric | Warning Threshold | Critical Threshold |
| Response Time P95 | >3 seconds | >5 seconds |
| Error Rate | >5% | >10% |
| Memory Usage | >80% | >90% |
| Cache Hit Rate | <60% | <40% |
| Active Sessions | >150 | >200 |
| Token Usage/Query | >3000 | >5000 |

# 10. Performance Optimization Checklist

## 10.1 Query Processing Optimizations

* Implement query result caching with Redis
* Use connection pooling for database connections
* Optimize vector similarity search with FAISS indices
* Implement query batching for bulk operations
* Add request deduplication for identical concurrent queries

## 10.2 Response Generation Optimizations

* Implement streaming responses for long outputs
* Use response templates for common queries
* Cache frequently used document snippets
* Optimize prompt engineering to reduce token usage
* Implement progressive disclosure for complex responses

# 11. Testing Deliverables

**Week 1:** Automated test suite implementation

**Week 1:** Performance baseline report

**Week 2:** Load test results and analysis

**Week 2:** Database optimization recommendations

**Week 3:** Token usage optimization report

**Week 3:** CI/CD pipeline configuration

**Week 4:** Performance monitoring dashboard

**Week 4:** Final optimization recommendations

*MIT FutureTech - AI Risk Repository | Developer Testing Documentation*