PHASE-5

PERFORMANCE TESTING PHASE DOCUMENT

Streamlining Ticket Assignment for Efficient Support Operations

TEAM ID	**NM2025TMID08979**
Team Leader	Yogesh S
Team Member	Selvin Joshva A
Team Member	Naveen P
Team Member	Ravin Akash S

Introduction:

The **Performance Testing Phase** aims to evaluate the speed, stability, scalability, and responsiveness of the **ticket assignment system** under various workloads. The goal is to ensure the system can handle real-time ticket generation, classification, and assignment efficiently without performance degradation, ensuring smooth operation even during peak usage periods.

Objectives:

- Validate the system's **response time**, **throughput**, and **resource utilization**.
- Ensure the **ML-based ticket classifier** performs efficiently under concurrent loads.
- Identify and remove **bottlenecks** in API, database, or model integration layers.
- Confirm the system's **stability and scalability** to handle increased ticket volumes.
- Maintain acceptable performance levels for **end-user experience** and **agent efficiency**.

Scope of Performance Testing:

In Scope	Out of Scope
Load, Stress, Spike, and Scalability testing	UI design performance
Backend API performance	External 3rd-party system benchmarking
Database query response time	Network infrastructure optimization
Model inference time (ML component)	Real customer ticket feeds

Key Performance Indicators (KPIs):

Metric	Description	Expected Benchmark
Response Time	Time taken for system to respond to a ticket creation or assignment request	≤ 2 seconds
Throughput	Number of requests processed per second	≥ 100 tickets/sec
CPU Utilization	Percentage of CPU used during processing	≤ 70% under normal load
Memory Usage	RAM usage while processing requests	≤ 80% of allocated memory
Model Latency	Time for ticket classification model to predict category/agent	≤ 1.5 seconds
Database Query Time	Time taken to fetch/update ticket records	≤ 0.5 seconds
System Uptime	Availability during testing	≥ 99%

Performance Testing Types and Strategy:

Type	Description	Purpose
Load Testing	Simulate normal and peak ticket traffic	Evaluate system behavior under expected workload
Stress Testing	Push system beyond limits	Identify breaking points and system recovery ability
Spike Testing	Introduce sudden high load for short periods	Test elasticity and resilience
Scalability Testing	Gradually increase number of users/tickets	Ensure linear performance with added resources
Endurance (Soak) Testing	Run system under normal load for long duration	Detect memory leaks and performance degradation

Performance Test Environment:

Component	Configuration / Tool	Description
Hardware	Cloud server (e.g., AWS EC2 / Render)	8 GB RAM, 4 vCPU
Backend	Flask / FastAPI	RESTful ticket assignment API
Database	PostgreSQL	Stores tickets, agents, and performance logs
Frontend	React / HTML Dashboard	Visualizes ticket traffic and system metrics

Testing Tools	JMeter, Locust, Postman	Load and API performance testing
Monitoring Tools	Grafana, Prometheus	Resource usage and latency visualization

Test Scenarios:

Test Case ID	Scenario	Expected Result
PT01	500 tickets submitted within 5 minutes	System maintains ≤ 2 sec response time
PT02	1000 concurrent users creating tickets	No timeouts or crashes
PT03	Database stress test with 10k read/write operations	Queries execute within ≤ 0.5 sec
PT04	Spike load: sudden surge of 2000 tickets	System auto-scales and stabilizes
PT05	Continuous load for 8 hours	No performance degradation or memory leaks
PT06	AI model classifies 100 tickets simultaneously	Model inference ≤ 1.5 sec each
PT07	API throttling test	System gracefully handles excess requests

Performance Testing Process:

1. **Prepare Test Environment** – Deploy backend, model, and database on cloud server.
2. **Define Load Profile** – Determine number of concurrent users and request patterns.

3. **Configure Testing Tools** – Set up JMeter or Locust scripts for ticket generation and API calls.
4. **Execute Tests** – Perform load, stress, and scalability tests as per schedule.
5. **Monitor Metrics** – Track CPU, memory, and latency using Grafana or built-in monitors.
6. **Analyze Results** – Compare KPIs against benchmarks and identify bottlenecks.
7. **Optimize System** – Tune code, database indexes, or ML model as required.
8. **Re-Test for Validation** – Confirm improvements after optimization.

Performance Optimization Strategies:

- Use **asynchronous request handling** in Flask/FastAPI to improve response time.
- Implement **caching mechanisms** for repetitive ticket queries.
- Optimize **database indexing** and queries for faster retrieval.
- Compress and batch ticket data transfers using **JSON streams**.
- Use **pre-trained lightweight ML models** for faster inference.
- Scale horizontally using **load balancers** when handling large traffic.

Risk Assessment and Mitigation:

Risk	Impact	Likelihood	Mitigation
Server overload during peak load	High	Medium	Auto-scaling or cloud-based load balancer
High model latency	High	Medium	Optimize model and use GPU inference

Database deadlocks	Medium	Low	Use connection pooling and query optimization
Memory leaks	Medium	Low	Regular monitoring and garbage collection
Network latency	Medium	Low	Deploy services closer to data region

Reporting and Documentation:

Each test cycle will generate:

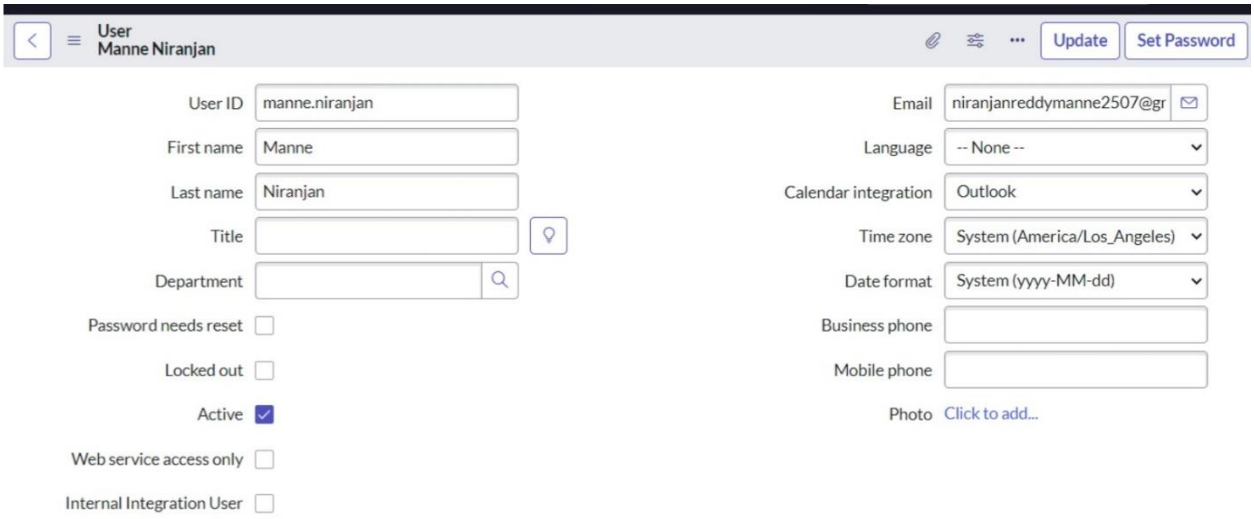
- **Performance Test Report** (metrics, graphs, summaries)
- **Error Logs** and **Resource Usage Charts**
- **Optimization Summary** with identified improvements

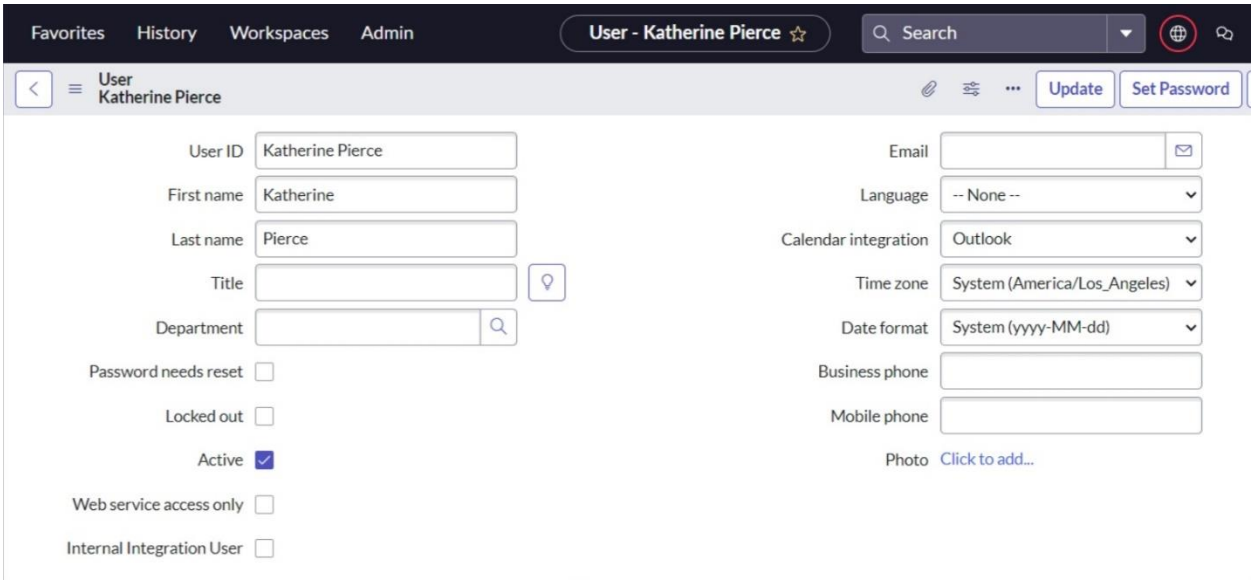
Reports will be reviewed weekly to track performance consistency and verify optimizations.

Expected Outcomes:

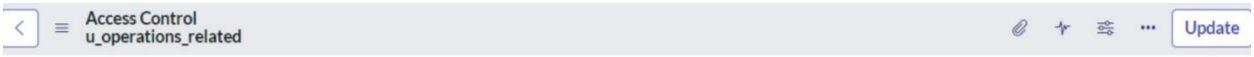
- Verified system that maintains optimal performance under load.
- Stable backend API and responsive dashboard.
- Scalable infrastructure that supports real-time ticket assignment.
- Improved user and agent satisfaction due to reduced response time.

Screenshots:





	Column label	Type	Reference	Max length	Default value
	Created by	String	(empty)	40	
	Created	Date/Time	(empty)	40	
	Sys ID	Sys ID (GUID)	(empty)	32	
	Updates	Integer	(empty)	40	
	Updated by	String	(empty)	40	
	Updated	Date/Time	(empty)	40	
×	Assigned to group	Reference	Group	40	
×	Assigned to user	Reference	User	32	
×	Comment	String	(empty)	40	
×	Issue	String	(empty)	40	
×	Name	String	(empty)	40	
×	Priority	String	(empty)	40	
×	Service request No	String	(empty)	40	javascript:getNextObjNumberPadded();
×	Ticket raised Date	Date/Time	(empty)	40	
+	*Insert a new row...*				



Definition

Access Control Rules allow access to the specified resource if *all three* of these checks evaluate to true:

1. The user has one of the roles specified in the **Role** list, or the list is empty.
2. Conditions in the **Condition** field evaluate to true, or conditions are empty.
3. The script in the **Script** field (advanced) evaluates to true, or sets the variable "answer" to true, or is empty.

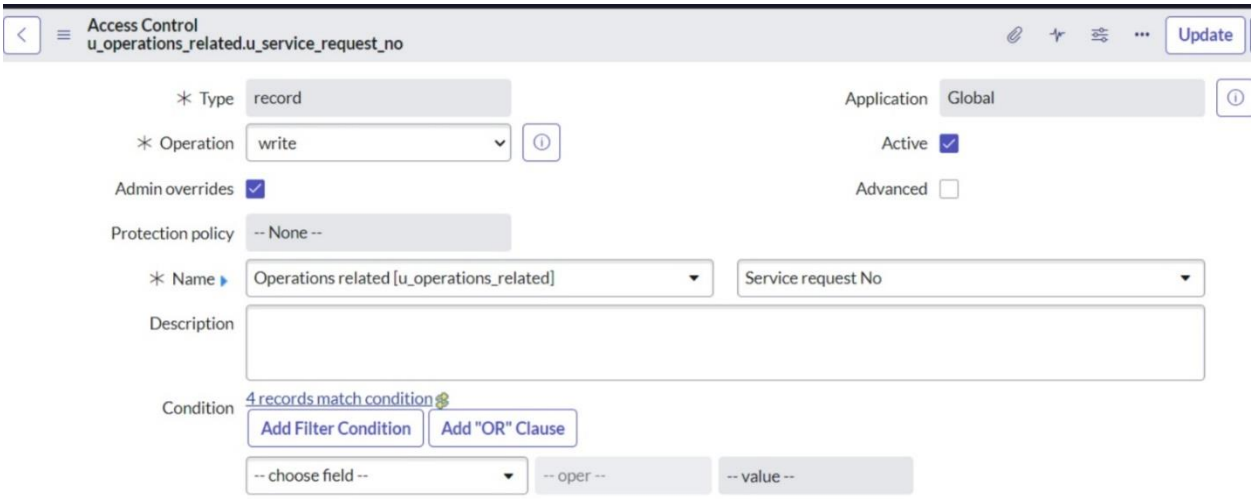
The three checks are evaluated independently in the order displayed above.

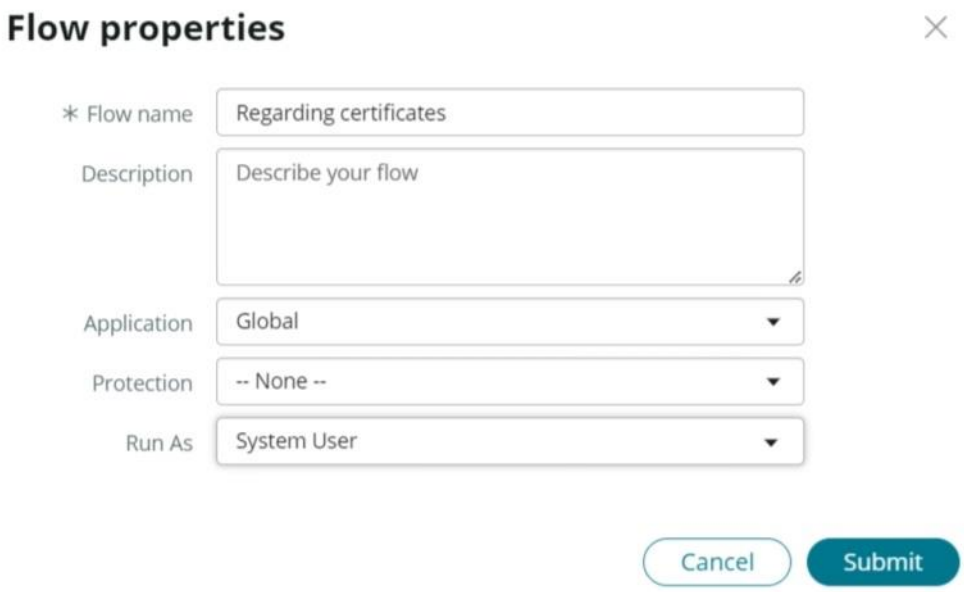
More Info

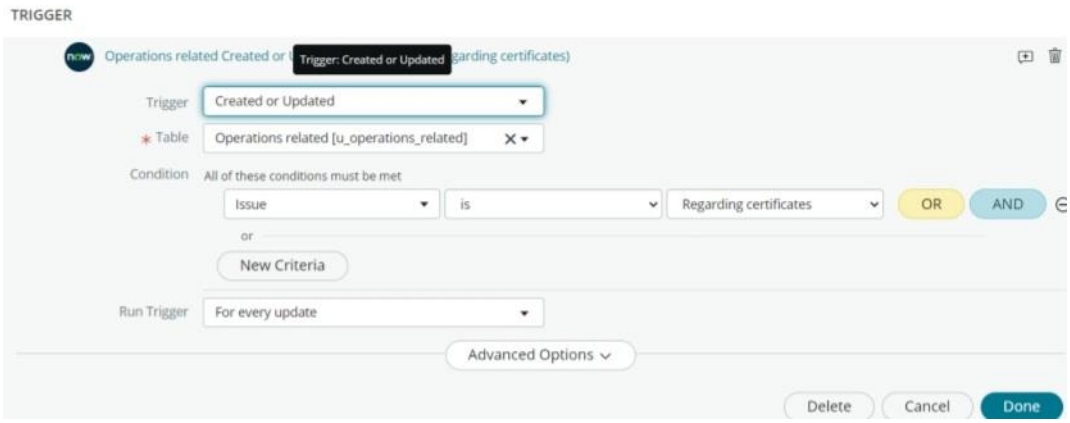
Requires role

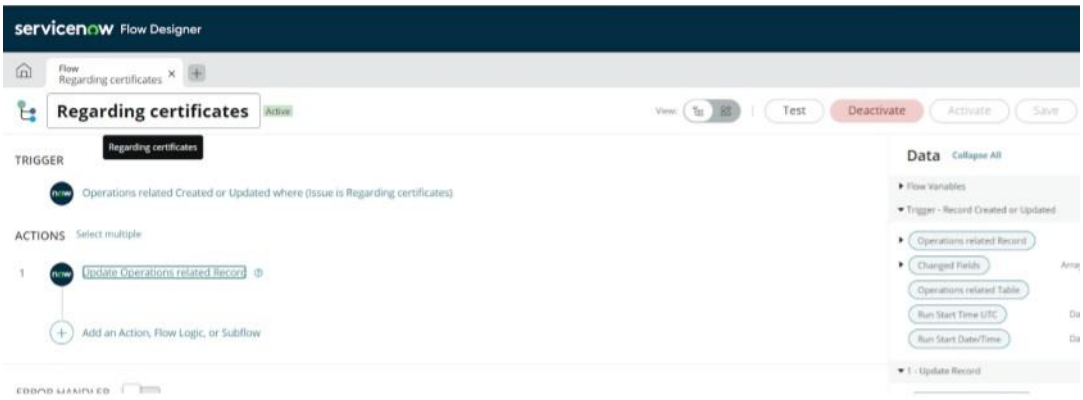
1 to 3 of

	Role
×	u_operations_related_user
×	Platform_role
×	Certification_role
+	*Insert a new row...*


Access Control
u_operations_related.u_service_request_no
Update
* Type
record
Application
Global
* Operation
write
Active
Admin overrides
Advanced
Protection policy
-- None --
* Name
Operations related [u_operations_related]
Service request No
Description
Condition
4 records match condition
Add Filter Condition
Add "OR" Clause
-- choose field --
-- oper --
-- value --


Flow properties
* Flow name
Regarding certificates
Description
Describe your flow
Application
Global
Protection
-- None --
Run As
System User
Cancel
Submit





Conclusion:

The performance testing phase ensures the **ticket assignment system** is not only functionally correct but also **efficient, scalable, and resilient**. Through systematic testing and optimization, the system is prepared to handle real-world ticket volumes, ensuring high availability and reliability in support operations.