International Conference on Science, Engineering & Management Trends

International Journal of Science, Engineering and Technology
ISSN: 2348-4098, P-ISSN: 2395-4752

Design and Implementation of Enterprise-Grade Multi-Tenant Generative AI Platform Using FastAPI and Azure

Indrajeet Trigunayat
M.Tech Scholar, Computer Science & Engineering Department of Computer Science & Engineering
Global Institute of Technology & Management, Gurugram

Under the Supervision of
Dr. Dharmbir Yadav (HoD CSE)
Department of Computer Science & Engineering Global Institute of Technology & Management, Gurugram

Abstract- Generative artificial intelligence has become an important capability for modern digital platforms, but production adoption requires more than model access. Enterprises need secure authentication, tenant isolation, policy-driven governance, auditability, observability, cost control, data protection and resilient operations. This paper presents the design and implementation of an enterprise-grade multi-tenant generative AI platform using FastAPI and Microsoft Azure. The proposed platform introduces a layered architecture consisting of a client access layer, API gateway layer, application orchestration layer, model routing layer, retrieval and knowledge layer, data persistence layer and observability/governance layer. FastAPI is used for high-performance asynchronous API development, dependency injection, request validation and modular service composition. Azure services are used for hosting, identity integration, secret management, monitoring, storage, database services and managed access to large language models. The paper defines a practical multi-tenancy model that separates tenant identity, policy, rate limits, usage budgets, data boundaries and audit events. It also proposes a secure request lifecycle for chat, retrieval-augmented generation and API-based consumption. The contribution of this study is a reference architecture and implementation methodology that can help academic and enterprise teams build scalable, secure and maintainable GenAI systems without tightly coupling application logic to a single model provider. The proposed design supports responsible AI controls, operational telemetry, disaster recovery readiness and future extensibility for additional models, connectors and agentic workloads.

Keywords: Generative AI, Multi-Tenancy, FastAPI, Azure, Large Language Models, Retrieval-Augmented

I. INTRODUCTION

Generative artificial intelligence (GenAI) platforms are rapidly moving from experimental prototypes to production-grade enterprise systems. In early experiments, teams often integrate directly with a single model endpoint and expose a basic chat interface. This approach works for proof-of-concept scenarios, but it is not sufficient when multiple business units, applications and external consumers need governed

International Journal of Science, Engineering and Technology
ISSN: 2348-4098, P-ISSN: 2395-4752

access to AI capabilities. A production platform must manage identity, authorization, tenant boundaries, model access, data privacy, audit trails, cost controls, service-level objectives and operational resilience. The complexity increases further when the platform is multi-tenant. In a multi-tenant GenAI system, different user groups or applications may share common infrastructure while expecting strict isolation of data, configuration, access policy and usage limits. The platform must ensure that a request from one tenant cannot access another tenant's documents, prompts, conversation history, vector indexes, secrets or model quota. At the same time, the platform should reuse common runtime capabilities such as authentication middleware, model routing, observability, caching and deployment automation. This balance between sharing and isolation is the central architectural problem addressed in this paper.

FastAPI is a suitable framework for such a platform because it supports asynchronous request handling, modern Python typing, dependency injection and automatic OpenAPI specification generation. These capabilities make it practical to build modular APIs for chat, completions, embeddings, files, tenant administration, usage reporting and health monitoring. Microsoft Azure provides cloud-native services that can support hosting, secrets, identity, network isolation, observability, API management and managed LLM access. A carefully designed combination of FastAPI and Azure can therefore provide a strong foundation for enterprise-grade AI delivery.

The objective of this research is to propose a design and implementation approach for an enterprise-grade multi-tenant GenAI platform. The paper focuses on architecture, security, request lifecycle, deployment topology, data management and governance controls. The work is intentionally presented in an implementation-neutral way so that it can be adapted by universities, enterprises and research teams without dependence on a specific internal product or proprietary environment.

The major research questions are: How can FastAPI be used to implement a modular GenAI orchestration layer? How can Azure services be mapped to enterprise platform requirements such as identity, hosting, observability and secret management? How can a practical multi-tenancy model be implemented for users, applications, documents and model usage? What governance controls are necessary for secure and reliable GenAI operations?

II. BACKGROUND AND RELATED WORK

Enterprise GenAI platforms combine multiple research and engineering areas: cloud-native application architecture, API security, large language model orchestration, retrieval-augmented generation, multi-tenant software design, observability and responsible AI governance. Unlike traditional web applications, GenAI platforms must handle probabilistic model outputs, high-latency requests, token-based cost models and sensitive user-provided context. As a result, the platform architecture must address both conventional software risks and AI-specific risks.

FastAPI provides a Python-based API framework that is widely used for building typed web services. Its dependency injection system allows shared services such as authentication, database sessions, policy enforcement and tenant context resolution to be injected into route handlers in a consistent manner [1]. This is useful in a GenAI platform because each request must be evaluated with tenant, user, model, quota and data-access context before the platform invokes model or retrieval services.

Azure provides multiple services relevant to production GenAI systems. Azure OpenAI in Foundry Models enables access to enterprise-ready generative AI capabilities and integrates with the Azure ecosystem

International Journal of Science, Engineering and Technology
ISSN: 2348-4098, P-ISSN: 2395-4752

[2]. Azure API Management can enforce rate limits and quotas using policy configuration, which is important for protecting backend resources and controlling usage spikes [3], [4]. Microsoft guidance for production GenAI emphasizes security, privacy, lifecycle management and operational maturity for distributed AI applications [5].

Responsible AI and risk management frameworks are also relevant. The NIST AI Risk Management Framework: Generative AI Profile describes GenAI-specific risk management actions and positions them as a companion to the broader AI RMF [6]. These concepts map directly to platform controls such as audit logs, user transparency, misuse monitoring, data handling rules and continuous evaluation of model behavior.

The related work indicates that a successful enterprise GenAI platform cannot be reduced to model invocation. It requires a layered control plane and data plane. The control plane defines tenants, users, policies, quotas, model entitlements, keys and compliance rules. The data plane handles runtime requests such as chat, retrieval, embeddings, document processing and response streaming. This paper uses that distinction to propose a reference architecture.

III. PROBLEM STATEMENT AND OBJECTIVES

Problem statement: Many GenAI implementations begin as single-tenant applications that directly call an LLM endpoint. Such systems lack reusable tenancy controls, standardized APIs, centralized observability, model abstraction, auditability and cost governance. When adoption increases, these gaps create security risks, operational instability and maintainability issues. The problem is to design a scalable platform that provides shared AI capabilities while preserving tenant isolation and enterprise governance.

The objectives of the proposed research are listed below.

1. Design a layered reference architecture for an enterprise-grade multi-tenant GenAI platform using FastAPI and Azure.
2. Define a tenancy model that covers users, applications, API keys, documents, prompts, vector indexes, usage budgets and audit events.
3. Propose a secure request lifecycle for chat, retrieval-augmented generation and API-based model access.
4. Identify implementation components such as API gateway policies, FastAPI middleware, model routing, data persistence, secret management and observability.
5. Discuss evaluation criteria for performance, scalability, security, reliability and operational governance.

The scope of this work is limited to platform design and implementation methodology. It does not claim to train a new foundation model. Instead, it focuses on how enterprise applications can safely consume existing large language models through a governed platform layer.

IV. PROPOSED SYSTEM ARCHITECTURE

The proposed architecture follows a layered model. Each layer has a clear responsibility and communicates through well-defined interfaces. This separation improves maintainability, simplifies compliance review and allows the platform to evolve when new models, storage engines, identity providers or user interfaces are introduced.

International Journal of Science, Engineering and Technology
ISSN: 2348-4098, P-ISSN: 2395-4752

Table 1. Proposed layered architecture for enterprise GenAI platform

Layer	**Primary Responsibility**	**Representative Azure/FastAPI Capability**
Client Access Layer	Web UI, mobile UI, external API clients and internal applications	HTTPS clients, SDKs, browser applications
API Gateway Layer	Routing, throttling, quota, subscription validation and perimeter policy	Azure API Management, gateway policies
Identity and Tenant Context Layer	Authentication, authorization and tenant resolution	OIDC/OAuth2 integration, FastAPI dependencies
Application Orchestration Layer	Chat orchestration, prompt processing, model request lifecycle	FastAPI routers, services, middleware
Model Routing Layer	Model abstraction, provider selection, failover and policy enforcement	Azure OpenAI, provider adapters
Knowledge and Retrieval Layer	Document ingestion, chunking, embeddings and vector search	Storage, database, AI Search or vector DB
Data and Audit Layer	Conversation state, configuration, audit events and usage records	PostgreSQL/Cosmos DB, storage, logs
Observability and Governance Layer	Metrics, traces, logs, alerts, policy evidence and SLO reporting	Azure Monitor, Application Insights, dashboards

The client access layer should not contain sensitive business logic. It should call the platform through authenticated APIs. The API gateway layer provides a central policy enforcement point for external calls. It can apply throttling, rate limits, request validation and routing. For enterprise workloads, gateway policies reduce the probability that a single tenant or consumer can overload backend services.

The identity and tenant context layer is responsible for determining who the caller is, which tenant the caller belongs to, which entitlements apply and whether the request is allowed. In FastAPI, this can be implemented using middleware for correlation identifiers and dependencies for authentication, authorization and tenant resolution. The same tenant context is then passed to downstream services so that model selection, retrieval scope and logging are consistent.

The orchestration layer is the core application layer. It receives requests such as chat completion, summarization, document question answering, embedding generation or agent task execution. It validates the request, applies policy, retrieves relevant context if required, constructs a model request,

International Journal of Science, Engineering and Technology
ISSN: 2348-4098, P-ISSN: 2395-4752

invokes the model router and streams or returns the response. This layer should remain independent of any one model vendor so that the platform can support multiple model providers.

The knowledge and retrieval layer supports retrieval-augmented generation. Documents are uploaded, scanned, extracted, chunked, embedded and stored with tenant-scoped metadata. At query time, the platform retrieves only documents permitted for the current tenant and user. This prevents cross-tenant data leakage and ensures that generated responses are grounded in authorized content.

The observability and governance layer collects operational and compliance evidence. Logs should include correlation IDs, tenant IDs, user IDs, model identifiers, token usage, latency, policy decisions and error categories. Sensitive prompts and responses should be handled according to data classification policy. In high-sensitivity environments, raw content may be masked, encrypted or excluded from logs while preserving useful metadata.

V. MULTI-TENANCY DESIGN

Multi-tenancy is a design pattern in which multiple tenants share the same application platform while maintaining logical isolation. In GenAI platforms, the meaning of tenant isolation must be broader than database separation. It must cover authentication context, configuration, model access, prompt templates, document collections, vector indexes, conversation history, API keys, usage budgets and audit records.

The proposed design uses tenant context as a first-class object. Each inbound request is converted into an internal TenantContext structure containing tenant_id, user_id, application_id, roles, data_scope, allowed_models, policy_version, request_id and budget metadata. All service functions accept this context explicitly or obtain it through request-scoped dependency injection. This approach reduces accidental bypass of tenant controls.

Table 2. Tenancy isolation model

Tenancy Area	Isolation Rule	Implementation Recommendation
Identity	Users and applications must map to one or more authorized tenants	OIDC claims, group mapping, tenant registry
Configuration	Tenant settings must not be shared without explicit inheritance	Tenant configuration table with versioning
Documents	Documents must be queryable only within authorized tenant scope	Tenant metadata filters and storage prefixes
Vector Data	Embeddings must include tenant and document access metadata	Filtered vector search or per-tenant index
API Keys	Keys must be tenant-bound and revocable	Hashed key storage and key rotation
Usage	Token and request consumption must be tracked per tenant	Usage ledger and budget checks
Audit	Security-relevant events must include tenant identifiers	Append-only audit event table

There are three common data isolation models: shared database with tenant_id filters, shared database with separate schemas and physically separate databases. The best choice depends on regulatory requirements, scale and cost. A shared database with strict tenant_id filters is simple and cost-effective, but it requires rigorous testing and query review. Separate schemas provide stronger logical boundaries.

International Journal of Science, Engineering and Technology
ISSN: 2348-4098, P-ISSN: 2395-4752

Physically separate databases provide the strongest isolation but increase operational complexity. The proposed architecture supports all three options by keeping tenant context separate from business logic.

For vector search, per-tenant indexes provide clear isolation but may create management overhead when the number of tenants is large. A shared index with mandatory tenant metadata filtering can be efficient, but it must be implemented carefully. Every embedding record should contain tenant_id, document_id, access_group, sensitivity_level and retention metadata. Query filters must be applied before results are used for prompt construction.

Tenant-aware usage management is essential because LLM cost is often token-based. The platform should track prompt tokens, completion tokens, embedding tokens, document processing units and failed requests. Budget enforcement may be soft or hard. A soft budget generates alerts when consumption approaches a threshold. A hard budget rejects or downgrades requests when limits are exceeded.

VI. IMPLEMENTATION METHODOLOGY USING FASTAPI

FastAPI allows the application to be organized into routers, services, repositories and middleware. The implementation should start with a domain model that separates platform administration from runtime inference. Core modules may include auth, tenants, policies, models, prompts, chat, files, retrieval, usage, audit and health.

A typical FastAPI request lifecycle begins at middleware. Middleware attaches a correlation ID, records request start time, validates headers and initializes request-scoped logging. Route-level dependencies then authenticate the caller, resolve tenant context and enforce permissions. The router delegates to a service class, which applies domain rules and calls model or retrieval adapters. After execution, telemetry is emitted and usage records are stored.

Table 3. FastAPI components mapped to platform responsibilities

FastAPI Component	Purpose in Proposed Platform
Routers	Expose APIs for chat, files, embeddings, tenants and administration
Dependencies	Inject authenticated user, tenant context, database session and policy services
Middleware	Correlation ID, request timing, security headers and structured logging
Pydantic Models	Validate request and response contracts
Background Tasks	Handle lightweight asynchronous post-processing
Service Classes	Encapsulate business logic and orchestration
Provider Adapters	Abstract Azure OpenAI and other model providers

The platform should avoid placing business logic directly inside route handlers. Route handlers should be thin and should mainly coordinate validation, dependency resolution and service invocation. This improves unit testing and makes it easier to reuse services across web, API and background processing workloads.

International Journal of Science, Engineering and Technology
ISSN: 2348-4098, P-ISSN: 2395-4752

The model router can be implemented as a provider abstraction. A ModelRequest object contains normalized fields such as tenant_id, model_alias, prompt_messages, temperature, max_tokens, tools, safety_settings and streaming preference. The router maps model_alias to an actual provider deployment based on tenant entitlement and availability. This allows the platform to introduce new model versions without breaking client contracts.

Streaming responses require careful design. For chat experiences, users expect partial tokens to arrive as the model generates output. FastAPI can return streaming responses, but the platform must also handle cancellation, timeout, error propagation and usage accounting. Usage may be finalized after the stream completes. If the client disconnects, the platform should still record the partial request status for audit and troubleshooting.

The file ingestion pipeline should be decoupled from interactive chat requests. Large documents may require extraction, malware scanning, chunking, embedding and indexing. These operations can exceed normal HTTP timeouts. Therefore, the upload API should create a job record and return a job identifier. Background workers can process the file asynchronously and update the job status. This design improves user experience and prevents web workers from being blocked by long-running processing.

VII. AZURE-BASED DEPLOYMENT ARCHITECTURE

Azure can host the proposed platform using a combination of App Service or container platforms, API Management, Key Vault, managed databases, storage, monitoring and managed identity. The exact selection depends on institutional policy and workload size. The important principle is to avoid embedding secrets in code or configuration files and to use managed identity wherever possible.

Azure API Management can be placed in front of the FastAPI backend to provide subscription management, request throttling, quotas and consistent public API policy. Azure documentation defines rate-limit policies for limiting call rates and quota policies for enforcing renewable or lifetime call volume and bandwidth limits [3], [4]. These capabilities are directly useful for tenant and application-level governance.

Azure Key Vault should be used to store secrets such as database connection strings, signing keys, encryption keys and provider credentials. Application components should retrieve secrets using managed identity. This reduces the risk of secret leakage and supports centralized rotation. For higher-security deployments, private endpoints and network restrictions should be considered for Key Vault, databases and storage accounts.

The compute layer can be implemented on Azure App Service, Azure Container Apps or Azure Kubernetes Service. App Service is simpler for small to medium workloads and supports deployment slots for blue-green release strategies. Container Apps provides flexible scale-to-zero and event-driven workloads. Kubernetes offers advanced orchestration but increases operational overhead. For a research implementation, App Service or Container Apps is often sufficient.

The data layer may use PostgreSQL for relational metadata such as tenants, users, policies, conversations, files, audit events and usage records. Object storage can hold uploaded documents and generated artifacts. A vector search engine can store embeddings. The design should ensure that each storage layer enforces tenant metadata, retention rules and access policies.

International Journal of Science, Engineering and Technology
ISSN: 2348-4098, P-ISSN: 2395-4752

Observability should include metrics, logs and traces. Key metrics include request latency, token usage, error rates, model provider failures, retrieval latency, ingestion job duration, queue depth and tenant-level consumption. Distributed tracing helps identify whether time is spent in API gateway, application code, retrieval, database calls or model invocation. Alerts should be defined for SLO breaches, quota anomalies, repeated authorization failures and provider outage patterns.

VIII. SECURITY, GOVERNANCE AND RESPONSIBLE AI CONTROLS

Security must be designed into the platform rather than added after implementation. The proposed platform uses layered controls: authentication at the perimeter, authorization in the application layer, tenant scoping in data access, encryption in transit and at rest, audit logging for sensitive operations and monitoring for anomalous behavior.

The authentication mechanism should use enterprise identity standards such as OAuth2 or OpenID Connect. After authentication, the platform should map identity claims to tenant permissions. Authorization should be based on roles and policy. Example roles include tenant_admin, application_owner, developer, auditor and end_user. Sensitive operations such as creating API keys, changing model entitlements or deleting documents should require elevated privileges and must be audited.

Data protection is a key concern in GenAI platforms because prompts may contain confidential information. The platform should classify data, minimize logging of raw content, encrypt sensitive records and apply retention policies. Where content logging is required for evaluation or troubleshooting, masking and approval workflows should be considered. Users should be informed about what data is stored and how it is used.

Responsible AI controls should include prompt filtering, abuse detection, model response safety checks, citation or grounding for retrieval scenarios, feedback capture and evaluation datasets. The NIST GenAI profile highlights the need to manage unique GenAI risks and align risk management activities with organizational goals and priorities [6]. In platform terms, this means the control plane should preserve evidence of policy decisions, model versions, evaluation results and incident handling.

API governance is also required. Each client application should have a distinct identity or API key. Keys should be stored as hashes, rotated periodically and revocable. Rate limits and quotas should be applied per tenant and per application. Administrators should be able to view consumption, configure budgets and disable abusive clients without affecting other tenants.

- Least privilege access for users, applications and managed identities.
- Tenant-scoped data access enforced in repositories and retrieval filters.
- Encryption for stored documents, conversations, secrets and audit-sensitive metadata.
- Immutable or append-only audit records for administrative and security events.
- Human review workflow for high-risk model behavior and policy exceptions.
- Monitoring for anomalous token consumption, repeated failures and prompt abuse patterns.

International Journal of Science, Engineering and Technology
ISSN: 2348-4098, P-ISSN: 2395-4752

IX. PROPOSED REQUEST LIFECYCLE

The runtime request lifecycle is designed to be deterministic before the model call and observable after the model call. A chat request begins when a client sends a message to the API gateway. The gateway applies request size limits, subscription validation, rate limits and basic header checks. The request then reaches the FastAPI backend.

1. The middleware assigns a correlation ID and initializes structured logging.
2. Authentication dependencies validate the caller token or API key.
3. Tenant resolution converts caller identity into a TenantContext object.
4. Authorization policies verify that the caller can use the requested operation and model.
5. Budget checks confirm that the tenant has remaining usage allowance.
6. If retrieval is requested, the retrieval service searches only authorized tenant-scoped documents.
7. The prompt builder combines system instructions, user input, retrieved context and safety constraints.
8. The model router selects a provider deployment based on tenant policy and availability.
9. The response is returned or streamed to the client.
10. Usage, audit events, latency metrics and error details are recorded.

This lifecycle ensures that security and governance decisions occur before any model invocation. It also ensures that operational evidence is collected after the request. By maintaining the same lifecycle for web and API consumers, the platform reduces inconsistencies and simplifies compliance review.

X. EVALUATION AND DISCUSSION

The proposed architecture can be evaluated through qualitative and quantitative criteria. Quantitative evaluation includes latency, throughput, token consumption, ingestion duration, retrieval accuracy, error rate and resource utilization. Qualitative evaluation includes maintainability, auditability, security posture, tenant isolation quality and ease of onboarding new models or tenants.

Table 4. Suggested evaluation criteria

Evaluation Area	Metric	Expected Platform Behavior
Performance	P95 request latency	Stable under normal concurrency and graceful under provider latency
Scalability	Requests per minute per tenant	Horizontal scaling without cross-tenant impact
Security	Unauthorized access attempts	Denied and audited consistently
Tenancy	Cross-tenant retrieval tests	No document or vector leakage
Reliability	Provider failure handling	Fallback, retry or user-friendly failure response
Cost Governance	Token usage by tenant	Accurate ledger and budget enforcement
Maintainability	Time to add a model adapter	Minimal changes outside provider abstraction

International Journal of Science, Engineering and Technology
ISSN: 2348-4098, P-ISSN: 2395-4752

A prototype implementation can be tested using simulated tenants and workloads. For example, Tenant A and Tenant B can upload separate documents and execute retrieval queries. The expected result is that Tenant A never receives Tenant B content, even when keywords overlap. Similar tests can be written for API keys, model entitlements and budget limits.

Performance testing should separate model latency from application latency. LLM calls are often the dominant latency component, but poor application design can add avoidable overhead through unnecessary database calls, synchronous file processing or inefficient retrieval. Traces should therefore measure each stage independently: gateway, authentication, retrieval, prompt construction, model invocation and response streaming.

Reliability testing should include provider timeout, quota exhaustion, database latency and storage failures. The platform should fail safely. For example, if retrieval fails, the system may either reject the request or continue without retrieval depending on policy. If tenant context cannot be resolved, the request must be rejected. If audit logging fails for a high-risk administrative action, the platform may block the action rather than proceed without evidence.

The proposed architecture improves maintainability because policy, orchestration, retrieval and model routing are separated. A new model provider can be added by implementing a provider adapter. A new tenant can be onboarded by creating configuration, entitlements, limits and data scopes. A new application can be onboarded by issuing an application identity or key and assigning policy. This modularity is necessary for long-term platform evolution.

XI. LIMITATIONS

This paper proposes a reference architecture and implementation methodology rather than reporting results from a large-scale production benchmark. Actual performance will depend on model provider latency, selected Azure compute services, database configuration, vector search implementation and network topology. The design also assumes that tenant identity and policy metadata are trustworthy and kept up to date.

Another limitation is that responsible AI quality cannot be guaranteed only through architecture. Model evaluation, red-teaming, user education and continuous monitoring are required. The platform can provide controls and evidence, but governance teams must define the risk appetite and review processes suitable for their institution or enterprise.

Finally, multi-tenancy introduces trade-offs. Strong physical isolation may increase cost and operational effort. Shared infrastructure improves efficiency but requires stronger testing and controls. The final implementation should choose the isolation model based on risk classification, compliance obligations and expected scale.

XII. CONCLUSION AND FUTURE SCOPE

This paper presented the design and implementation approach for an enterprise-grade multi-tenant generative AI platform using FastAPI and Azure. The proposed architecture separates client access, API gateway governance, identity and tenant context, application orchestration, model routing, retrieval, data

International Journal of Science, Engineering and Technology
ISSN: 2348-4098, P-ISSN: 2395-4752

persistence and observability. The paper also described a tenancy model that covers identity, configuration, documents, vector data, API keys, usage and audit events.

FastAPI provides a strong application-layer foundation through asynchronous APIs, typed request models and dependency injection. Azure services provide the surrounding cloud capabilities required for hosting, policy enforcement, identity integration, secret management, monitoring and managed model access. Together, these technologies can support a secure and scalable GenAI platform when implemented with clear boundaries and governance controls.

Future work may include building a full prototype, conducting load tests, comparing vector isolation strategies, implementing automated policy tests, measuring retrieval quality, evaluating cost optimization strategies and extending the platform for agentic workflows. Additional research may also explore standardized OpenAI-compatible APIs, cross-region failover, confidential computing and automated AI risk assessment pipelines.

REFERENCES

1. FastAPI Documentation, "Dependencies," FastAPI. Available: https://fastapi.tiangolo.com/tutorial/dependencies/
2. Microsoft Azure, "Azure OpenAI in Foundry Models," Microsoft Azure. Available: https://azure.microsoft.com/en-us/products/ai-foundry/models/openai
3. Microsoft Learn, "Azure API Management policy reference - rate-limit," Microsoft. Available: https://learn.microsoft.com/en-us/azure/api-management/rate-limit-policy
4. Microsoft Learn, "Azure API Management policy reference - quota," Microsoft. Available: https://learn.microsoft.com/en-us/azure/api-management/quota-policy
5. Microsoft Learn, "Generative AI: technology guidance," Microsoft AI Playbook. Available: https://learn.microsoft.com/en-us/ai/playbook/technology-guidance/generative-ai/
6. National Institute of Standards and Technology, "Artificial Intelligence Risk Management Framework: Generative Artificial Intelligence Profile," NIST AI 600-1, 2024. Available: https://www.nist.gov/publications/artificial-intelligence-risk-management-framework-generative-artificial-intelligence
7. Microsoft Learn, "Azure documentation," Microsoft. Available: https://learn.microsoft.com/en-us/azure/
8. Microsoft Learn, "Azure API Management policy reference - rate-limit-by-key," Microsoft. Available: https://learn.microsoft.com/en-us/azure/api-management/rate-limit-by-key-policy
9. Microsoft Learn, "Azure API Management policy reference - quota-by-key," Microsoft. Available: https://learn.microsoft.com/en-us/azure/api-management/quota-by-key-policy
10. National Institute of Standards and Technology, "AI Risk Management Framework," NIST. Available: https://www.nist.gov/itl/ai-risk-management-framework

Author Information

Indrajeet Trigunayat is an M.Tech scholar in Computer Science & Engineering at Global Institute of Technology & Management, Gurugram. His research interests include generative AI platforms, cloud-native architecture, API governance, reliability engineering and enterprise software design.

Dr. Dharmbir Yadav is HoD CSE, Department of Computer Science & Engineering, Global Institute of Technology & Management, Gurugram. This work was prepared under his supervision.