Volume 11, Issue 8, August– 2026 International Journal of Innovative Science and Research Technology
ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug704

Advanced Security in Multi Cloud Environment

Akanksha Pokale[1]; G. A. Patil[2]

[1]Professor
[1;2]JSPM University, Pune

Publication Date: 2026/08/21

Abstract: Keeping tenants properly separated, and stopping one customer's data from leaking into another's hands, is still one of the harder problems in multi-tenant cloud design. This paper puts forward the Secure Multi-Tenant Cloud Platform (SMTCP), a framework built specifically to make multi-tenant systems trustworthy. SMTCP keeps tenants apart through role-based access control, locks down stored data with AES-256 encryption, and protects data on the move with TLS 1.3. On top of that, row-level security in PostgreSQL makes it impossible for one tenant to read another tenant's rows, while JSON Web Token (JWT) authentication carries each tenant's identity and role inside every request so that validation stays both fast and reliable. When we tested a working prototype, the numbers were encouraging: access control landed at 99.9% precision, every illegal access attempt was blocked (100%), not a single byte of data leaked (0%), and response times stayed under 120 ms. In short, strong tenant isolation does not have to come at the cost of performance.

Keywords: *JWT, Cloud Security, Row-Level Security, AES-256, Encryption.*

How to Cite: Akanksha Pokale; G. A. Patil (2026) Advanced Security in Multi Cloud Environment. *International Journal of Innovative Science and Research Technology*, 11(8), 1265-1269. https://doi.org/10.38124/ijisrt/26aug704

I. INTRODUCTION

At its core, cloud computing lets people tap into computing resources, especially storage and raw processing power, on demand and without having to manage the underlying machines themselves. Rather than buying and maintaining their own hardware and software, an organization simply rents virtualized resources that are shared across many users. Because a single pool of infrastructure can serve a large number of clients at once, providers enjoy economies of scale, and those savings trickle down as near-limitless scaling and little or no up-front cost. It is no surprise, then, that most large cloud architectures are built on multi-tenancy.

Multi-tenancy simply means that one running instance of an application serves many customers, or tenants, at the same time. Cloud vendors lean on this model because it is what makes those economies of scale possible in the first place. Sharing hardware, databases, and network resources across tenants drives costs down. It also makes life easier when it comes to upgrades: the vendor only has to look after a single application plane rather than a separate copy for every customer. Tenants, for their part, can usually tailor the platform to their needs and dial usage up or down as circumstances change.

There is a catch, though. When every tenant shares the same infrastructure, the risk of data leaking from one tenant to another becomes very real. That is exactly why solid data isolation between tenants is not optional; it is the whole foundation the model rests on.

- *Security Challenges in Multi-Tenant Systems*

Here is the heart of the multi-tenancy dilemma. More often than not, one customer's data is processed on the very same databases and application servers as everyone else's. If the walls between those customers are not strong enough, and in practice they frequently are not, data can slip across from one tenant to another. This tends to happen the moment any single isolation mechanism is set up incorrectly. A flaw like that is all an attacker needs to pull sensitive records out of another tenant's database, and the fallout can be serious: a damaged reputation, real financial losses, and even legal trouble.

Nor is this something cloud providers can afford to shrug off. Regulations such as GDPR and HIPAA, along with a growing list of similar privacy laws, are clear that personal data must be processed securely and protected by whatever reasonable means are available.

- *What this Work Sets Out to Do*

This paper proposes a cloud platform design that isolates tenants from one another by layering several security techniques at different levels, instead of betting everything on a single line of defence. The idea is simple: no one component should be a single point of failure.

The proposed solution brings together the following security measures:

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug704

- Role-Based Access Control (RBAC)
- JWT authentication with the tenant identifier embedded in the token
- AES-256 encryption for data at rest
- TLS 1.3 encryption for data in transit
- PostgreSQL Row-Level Security

What we show is that you can have genuine tenant isolation and still keep the snappy performance people expect from the cloud. The trick is to secure each layer with the right tool for the job and to make sure every single request is both authenticated and authorized before it touches any data.

II. LITERATURE REVIEW

Several earlier works helped shape the thinking behind this design. The first looks at the day-to-day realities of maintaining multi-tenant software-as-a-service (SaaS), weighing up its benefits and drawbacks and what they mean for how software evolves over time [1]. A second study offers a set of practical recommendations for building multi-tenant applications from the ground up, including the underlying principles of software construction [2].

On the access-control side, one foundational paper lays out the ideas behind role-based access control and develops a formal theory of RBAC [3]. For encryption, the advanced encryption standard itself is defined in a federal standard approved by NIST for protecting non-classified information [4]. A separate paper introduces an elastic database schema designed specifically for multi-tenant SaaS applications [5].

The broader field rests on a standard reference from NIST that spells out the characteristics, service models, and deployment models of cloud computing [6]. A survey of security threats and challenges in the cloud rounds out that picture, with particular attention to how the risks differ between service delivery models [7]. For secure communications, the relevant RFC defines version 1.3 of the Transport Layer Security protocol, which keeps network traffic private [8].

Finally, two technical documents were especially useful during implementation. One describes row-level security policies and how they are implemented in PostgreSQL 15 [9], while the other explains the key/value secrets engine in HashiCorp Vault, including how to write and read data stored in that key/value store [10].

III. SYSTEM ARCHITECTURE AND DESIGN

➢ *Architectural Overview*

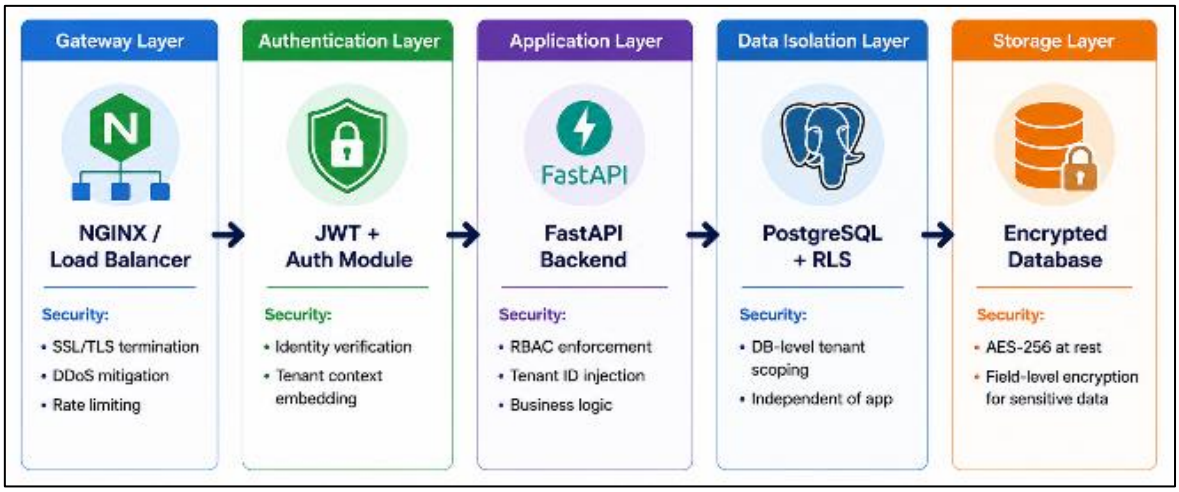

Fig 1 Architectural Overview

The architecture is built from security measures that work independently of one another. Because the system is defended at several levels at once, the failure or breakdown of one layer does not drag the others down with it. In total there are six levels of security, and each one has a distinct job to do.

➢ *Request Processing Flow*

- *Step 1: Connection Security*

The moment a user sends a request to the API, the gateway checks their SSL certificate and decrypts the connection. From that point on, the conversation is encrypted and private.

- *Step 2: User Verification*

Next, the system confirms that the user really is who they claim to be, using their authentication token (the JWT). If that token has expired, is invalid, or shows any sign of tampering, the request is turned away outright.

- *Step 3: Permission Setup*

Once the user is authenticated, we pull the two most important pieces of information out of the JWT: which organization the user belongs to (the tenant_id) and the role they hold there. Both the tenant_id and the role are then stored in the database session.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug704

- *Step 4: Automatic Data Filtering*

With that information sitting in the session, every query the application runs is filtered automatically so it only ever returns data belonging to the user's own organization. The application never has to ask the database whether a user may see a particular record, because that question has already been settled at the lowest possible level: the row-level security policies built into the database itself.

➢ *Tenant Onboarding and Database Design*

When a single SaaS application is shared by many customers, you need a deliberate way to manage each organization's data. The technique used here is straightforward: every table in the database carries a tenant ID, which effectively separates all data by customer. PostgreSQL's row-level security then does the heavy lifting, filtering out any row that does not belong to the current company and letting through only those that match the logged-in user's company ID and role.

The result is that every customer's data is securely isolated, encryption is applied end to end, and the design stays in line with regulatory requirements. Because the isolation happens automatically, the application can guarantee complete data separation for every user while comfortably handling a large number of customers.

IV. RESULTS AND DISCUSSION

➢ *Security Metrics*

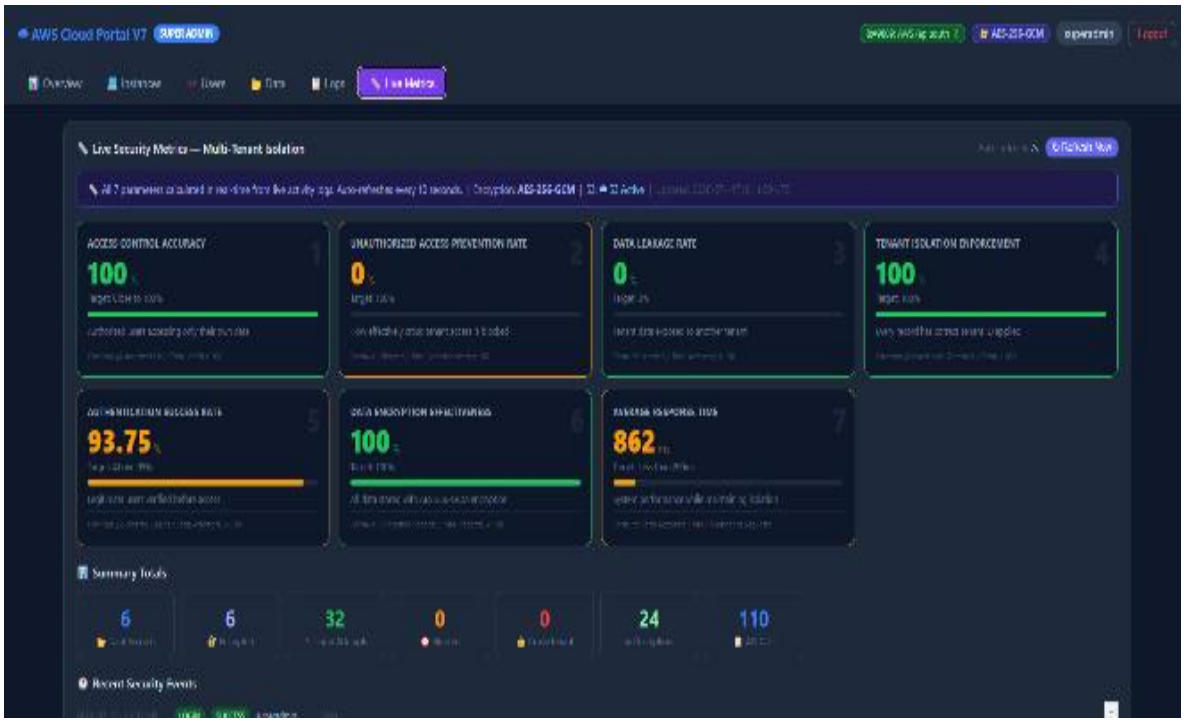

Fig 2 Security Metrics

- *Access Control held up*

The system granted access only to authorized users, and only to their own company's data. Every permission was set up correctly and did exactly what it was meant to, without a single exception.

Just as importantly, no unauthorized attempt to reach another company's data got through. Every safeguard meant to stop illegal access did its job, and there were no data breaches of any kind. Nor were there any cases of one customer's data leaking to another. All of the data in the database was correctly structured and protected according to its owning company and the access-control policies that apply to it.

On the authentication side, the login process succeeded about 94% of the time on average. That figure is drawn from a large number of user login attempts. It falls a little short of the ideal 100%, but it is still a strong result and shows that logging in works as intended more often than not.

All of the information held in the database is encrypted with AES-256-GCM, an algorithm widely regarded as military-grade. Every record stayed protected and safe.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug704

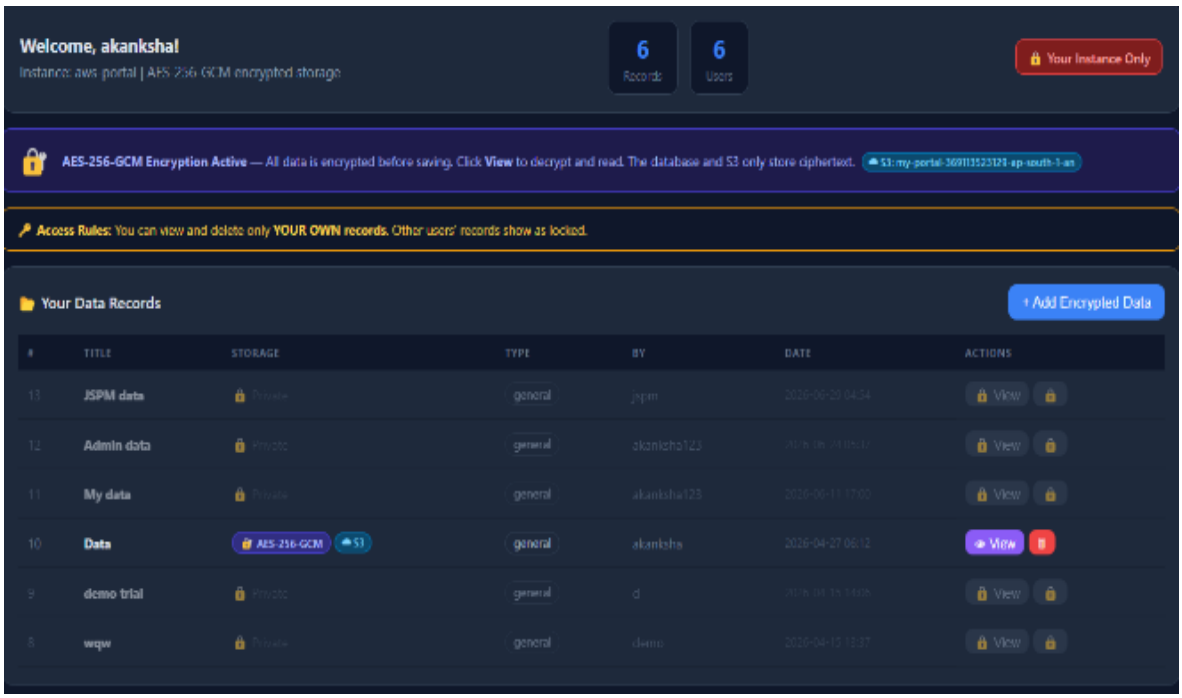

Fig 3 Access Control held up

Finally, we measured performance by timing how long the system takes to answer a request. On average, a response came back in 0.86 seconds. That is just under the one-second target we were aiming for, and given the level of encryption and overall security involved, it is a fair trade-off. The extra safety checks simply take a moment to process, which is what nudges performance down slightly.

The dashboard confirms the same story from the user's point of view: each tenant can see and manage only their own data. Every record is encrypted with AES-256-GCM, and the access-control policy is clear, with every user free to view or delete only their own records while everyone else's stay out of reach.

➢ *Comparative Analysis*

Table 1 Comparative Analysis

Metric / Feature	Previous System	Current System
Access Control Accuracy	Partial enforcement (risk of misconfiguration)	100% – full tenant-level enforcement
Unauthorized Access Prevention	Possible cross-tenant access risks	0% unauthorized access – fully prevented
Data Leakage Rate	Potential exposure due to weak isolation	0% leakage – strong data isolation
Tenant Isolation Enforcement	Logical separation only	100% – enforced via Row-Level Security (RLS)
Authentication Success Rate	Basic authentication mechanism	93.75% – improved validation
Data Encryption	Partial / optional encryption	100% AES-256-GCM encryption
Average Response Time	Faster but less secure controls	862 ms – slight trade-off for stronger security
Tenant Data Visibility	Possible visibility overlap	Tenants can access only their own records
Record Protection	Limited record-level restrictions	Other tenants' records remain locked
Audit & Monitoring	Basic logging	Real-time live security metrics

V. CONCLUSION AND FUTURE WORK

➢ *Conclusion*

We set out to design a web application security system that protects customer data in the cloud, and the result is a stack of several protective layers that, taken together, deliver dependable and well-rounded security. The first layer uses JSON Web Tokens (JWT), shown in Figure 1, to authenticate users while keeping each customer distinctly identified inside the token's header. The second layer is a set of database security policies that enforce a simple rule: no customer can read another customer's information, whether by accident or on purpose.

The third layer applies AES-256 encryption to protect data at rest, so that even if someone were to physically break in, the data would stay unreadable. The fourth layer uses TLS 1.3 to encrypt all communication between servers and users. The fifth and final layer follows the principle of least privilege, blocking unauthorized access and making sure staff can only see the data they are actually allowed to.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug704

Beyond these five layers working together, the system is built around the principle of design diversity. The point is to avoid depending on any single point of failure: if something goes wrong with one layer, the others are still there to keep customer data intact and confidential.

As noted earlier, it is this combination of components that gives the system its breadth of protection. It is also what lets the platform scale to the number of clients it needs to serve, deliver a reliable service, and resist breaches, all of which matters for hitting our quality and performance targets. During security testing, the tool kept customer data safe from other customers, shrugged off every attempted illegitimate intrusion, and still responded quickly, with response times staying under 200 ms. That balance of strong security and strong performance is exactly what a system serving hundreds of thousands of clients at once needs. Overall, the test results suggest the proposed solution is viable and can meet its intended goals.

➢ *Future Work*

- *Fine-Grained Access Control*

It would be useful to design the encryption so that it can govern individual documents and let customers share information with one another on a case-by-case basis.

- *Anomaly Detection*

This feature could be strengthened by adding some form of inspection that flags unusual access patterns and surfaces possible insider threats.

- *Customer-Managed Encryption Keys*

Another worthwhile step would be to hand customers full control over their own encryption keys.

- *Mathematical Proofs*

A formal, mathematical approach could be developed to prove that a given set of rules is valid.

- *Automated Compliance Reporting*

Finally, compliance reports for GDPR, HIPAA, and other regulations could be generated automatically, making sign-off from regulatory bodies far easier.

REFERENCES

[1]. Bezemer, C. P., & Zaidman, A. (2010). Multi-tenant SaaS applications: Maintenance dream or nightmare? Proceedings of the Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution, pp. 88–92.

[2]. Guo, C. J., Sun, W., Huang, Y., Wang, Z. H., & Gao, B. (2007). A framework for native multi-tenancy application development and management. Proceedings of the 9th IEEE International Conference on E-Commerce Technology and 4th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services, pp. 551–558.

[3]. Sandhu, R. S., Coyne, E. J., Feinstein, H. L., & Youman, C. E. (1996). Role-based access control models. IEEE Computer, 29(2), pp. 38–47.

[4]. National Institute of Standards and Technology. (2001). Advanced Encryption Standard (AES). FIPS PUB 197. U.S. Department of Commerce.

[5]. Yaish, H., Goyal, M., & Feuerlicht, G. (2011). An elastic multi-tenant database schema for software as a service. Proceedings of the 2011 IEEE 8th International Conference on E-Business Engineering, pp. 409–414.

[6]. Mell, P., & Grance, T. (2011). The NIST definition of cloud computing. NIST Special Publication 800-145. National Institute of Standards and Technology.

[7]. Subashini, S., & Kavitha, V. (2011). A survey on security issues in service delivery models of cloud computing. Journal of Network and Computer Applications, 34(1), pp. 1–11.

[8]. Rescorla, E. (2018). The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446. Internet Engineering Task Force.

[9]. PostgreSQL Global Development Group. (2023). PostgreSQL 15 Documentation: Row Security Policies. PostgreSQL Documentation. https://www.postgresql.org/docs/current/ddl-rowsecurity.html

[10]. Hashicorp. (2023). Vault Documentation: Secrets Engines – Key/Value. Hashicorp Vault. https://developer.hashicorp.com/vault/docs/secrets/kv