MULTI-TENANCY: THE ART OF HAPPY NEIGHBORS (AND HEALTHY BUDGETS) IN KAFKA & OPENSEARCH. MASTERING YOUR TENANCY STRATEGY.

My IT journey began a little over 10 years ago as an operations support engineer. Back then, I was taking care of a small cluster of VMs running in a client's data center. Resource planning was fairly easy mathematically, but it was very long-term – a purchase order needed to be put in place for any additional hardware several months in advance. It needed to get necessary approvals; the servers had to be delivered and actually plugged into the rack to be connected to the wider ecosystem. I would frequently over-provision the fleet – just to be safe, as the feedback loop was extremely long. Fast forward 5 years – and magic happened. Now we have AWS and auto-scaling groups. I can get a new machine on demand; I can scale down whenever I need to. Life is simpler.

But now questions pop up, almost greedily – can I do more with this? How much further can I bring down costs? Will my operational load decrease exponentially?

I know that it wasn't just me asking these questions then – and 5+ years from that point, they are still relevant. I know almost every architectural discussion still starts with a question – how much can we make this system multi-tenant?

Throughout my career I was an operations engineer, a software engineer, a DevOps engineer, an infrastructure engineer, and now a Director of Cloud Engineering at Pega. Don't let the title fool you though - I still treat heated architectural discussions as good cardio and prefer to look at code to understand how everything fits together in place of hearing someone else explain it (some people would call this "control freak"; other, nicer people would call it "curiosity"). Having that experience under my belt, in this article I would like to explore the trade-offs in the race for cost optimization and operational efficiency. I want to give you some philosophical considerations for choosing whether your system should be multi-tenant or single-tenant – and to what extent.

My goal is that by the end of this article, you will have a framework at your fingertips – a checklist to go through when considering what to do with your own architecture and which path to take. Let's start by defining what multi-tenant and single-tenant systems mean.

A single-tenant system provides each customer with their own dedicated instance of the software and its supporting infrastructure, including separate databases and application deployments.

A multi-tenant system is trickier to define. In a nutshell, it is a single instance of a software application and its underlying infrastructure that serves multiple distinct customers (tenants).
Yet, multi-tenancy can come in different shapes and forms on various layers. For instance, the runtime might be shared in a multi-tenant system, but databases can be separate for compliance purposes. If the database is shared, the data needs to be isolated logically and encrypted to prevent accidental cross-access.

Cost is higher in single-tenant systems, but factors like blast radius, noisy neighbors, and troubleshooting may be easier to handle.

In this article, I want to explore the choice between single-tenancy and multi-tenancy in terms of technologies we use at Pega – Kafka and OpenSearch. Both present unique challenges in three realms – security, resource contention, and operational complexity. I'll explore each in detail, starting with Kafka.

Kafka in a Multi-Tenant World: You Get a Quota, and YOU Get a Quota!

While Kafka is not a technology designed for persistent, long-term storage, designing a secure, compliant, and performant configuration of Kafka does pose its set of challenges.

Security

I frequently say I consider Kafka as a communication protocol. You have your consumers and producers that can communicate with one another using Kafka topics. So, what are the challenges when it comes to security in Kafka?

Making sure only specific consumers and producers have access to particular topics. This can be done via **ACLs** – e.g., in a microservices setting, you can create a user for every microservice and configure ACLs accordingly for a service to be able to consume/produce from/to specific topics. Similarly, this can be used to separate by environment level – dev, stg, and prod would each get their own user and ACLs to provide data segregation between stages.

Tenant message encryption – while in a single-tenant world, I would likely just encrypt traffic to Kafka in transit and the EBS volume itself, I would need to go a step further and encrypt the actual message payload per each tenant – meaning both the consumer and the producer would need to have access to the key to write or read the message. If latency is a big concern in your system, consider encryption/decryption times as part of your calculations.

Resource Contention

Resource contention is, in my experience, the most common hurdle in multi-tenant Kafka management. Several strategies to manage it include:

Quotas

Quotas are very efficient, albeit a convoluted mechanism in Kafka. The most common ones are **network bandwidth quotas** (limiting producers and consumers in terms of maximum data throughput) and **request rate quotas** (expressed as a percentage of network and I/O threads, prevent clients from overwhelming the brokers with too many requests).

Both are set up per user per broker.

Network bandwidth quotas will be limited, e.g., by AWS instance types. I.e., by EBS and Network bandwidth. EBS bandwidth is crucial here, since Kafka writes to the disk very greedily. E.g., for rg6.large and rg6.2xlarge, the sum of all producer and consumer byte rate quotas for all users connected to the Kafka cluster cannot exceed 4750Mi.

Request rate quotas are trickier and will depend on the amount of network and I/O threads configured on your cluster. Request rate quotas limit the percentage of time a client can use the request handler and network threads of each broker during a specific time period. It's calculated via the following formula: ((num.io.threads + num.network.threads) * 100)%. For instance, if you have 3 I/O threads and 8 network

threads, this value in your cluster will equal 1100% ((3+8)*100%). So, 1100% will be the value you can distribute among your users per broker.

The quota every user will receive does not dictate their usage – it is the upper bound they will not be able to cross, thus allowing us to have governance and prevent noisy neighbors from disturbing those around them.

There are different strategies to implement quotas, and there is no one approach. It depends on the predicted usage and criticality of every client. In a microservices world where each of the services has equal rights to Kafka, the best approach would be to distribute quotas equally among all services (that is, unless you deem a particular service more critical than the rest). Challenges will arise when new services are onboarded, since quotas will potentially need to be recalculated for everyone.

If the system is multi-environment instead of multi-tenant (i.e. one end client uses the system, with all their environment types like dev, stg, and production using the same infrastructure), consider not binding production with any quotas, opting to only restrict the lower environments. This way, if a client's production needs more resources, it will be visible in Kafka server metrics instead of in request throttling.

Separate Clusters to Handle Dev & Stg vs Prod Workloads

Another approach is to separate clusters by workload type – i.e., give production its own cluster while keeping lower environment types on a different cluster. The cons of this approach include increased cost and slightly different looking lower environments than production, so if you want to keep the testing as close to the real deal as possible, this might not be the way to go.

Operational Complexity

All of the above play into the operational complexity. There are a few others.

Upgrades

In Kafka, upgrades can be fairly painless. Clients can use an older Kafka client version than Kafka server version. Kafka is insanely backward compatible – when first starting to work with Kafka, I did a test with my team to check upgrades post-API/protocol breaking change which occurred in v. 0.10. We tested upgrade from Kafka 0.9.0.1 to 2.7.0 while

producers were still writing to topics and consumers were still reading from them. The system was entirely stable, and everyone could still write and read messages.

That being said, while everything is working, there is no guarantee everything would work quickly. During one of the incidents, for instance, we found a couple of consumers using an older Kafka client with our newer Kafka server – messages were being processed, yet we were seeing an odd lag adding sometimes even a second to total processing time. If you are upgrading your multi-tenant Kafka cluster to a newer version, consider how every tenant will have to be coordinated for the upgrade of their client.

Observability

One additional factor that makes multi-tenancy in Kafka more challenging is observability. On top of regular client and server metrics monitoring (e.g., broker and producer/consumer metrics), consider building out tenant-specific dashboards and quota monitoring dashboards. Ideally, the tenant-specific dashboard would include data from the client side (i.e., to see how long message processing takes from a client perspective) as well as from the server side (how long the server actually takes to process the message), along with throttling metrics to see if quotas are configured in accordance with usage.

Capacity Planning & Blast Radius

The question of how the Kafka cluster is going to look like broker-wise is a common point. Choosing to scale vertically or horizontally is a great consideration. Luckily, unlike some other technologies, scaling down in Kafka with proper replication and partition reassignment is fairly easy, so one does not have to choose one or the other.

So, how does one scale Kafka? As with quotas, there is no one right approach.

Generally, Kafka is designed to scale out horizontally – with Zookeeper, a 500-broker cluster can function normally; with Kraft, theoretically, we can scale up to above 1000 nodes. Observability for that amount of brokers will be challenging, so I would personally not go that far without explicit need. A common **soft limit for partitions per broker is 4000**, so an appropriate amount of brokers needs to be allocated to accommodate that, keeping in mind that the replication factor will increase the total number of partitions.

In a multi-tenant world, everything you do will have a larger blast radius. When adding additional brokers, it is important to consider the load the cluster is under – during high load, adding brokers is not great because the new brokers aren't yet servicing requests, yet they're asking existing brokers to send data to them. Scaling vertically might actually be better in this scenario when we need a quick boost of performance. That being said, during broker resize, that broker will become unavailable for some time, which means less capacity until the resize happens.

My rule of thumb is normally to provision more small to medium size nodes initially and upscale vertically during runtime if I need to. Since quotas are calculated per broker, this also gives me a higher absolute value to distribute between Kafka users.

OpenSearch in a Multi-Tenant World: Indexing Without Secrets

Search is used to store persistent information, for indexing data. The key problem with search is data access and encryption. How to ensure data is encrypted but still searchable?

Security

How do you search through fully encrypted data? You can encrypt the phrase and look up the hash, sure. But what if you want to search for a part of a phrase? You are unlikely to find that hash. And what about semantic search, where context and conceptual relevance also play a role?

Up until recently, encryption was not possible for OpenSearch in the context of searching encrypted data. Therefore, most solutions that cared about data compliance had to opt to be single-tenant.

This changed in the past year or so with the release of the **Portal26 plugin**. The plugin provides "encryption-in-use," meaning data remains encrypted even when actively being indexed and searched. This goes beyond traditional data-at-rest and data-in-transit encryption. It enables the construction of encrypted search indexes and intercepts and transforms queries against protected data. The Portal26 plugin **allows for index-level encryption keys**. This enables storing multiple customers' data on the same node while ensuring complete data isolation between tenants.

Of course, this involves a performance overhead, but it is not significant.

Resource Contention

In OpenSearch, similar to Kafka, resource contention is a key challenge in multi-tenant environments. Noisy neighbors can significantly impact the performance of the entire cluster.

Key metrics to monitor include: **CPU, memory, disk I/O, and network throughput** at the node and index level. Strategies to minimize contention include:

- **Shard Allocation Awareness:** Strategically distribute shards (the basic units of data storage in OpenSearch) across different nodes, availability zones, or even racks. This helps avoid "hot spots" and provides better fault tolerance.
- **Index Templates with Routing:** Allow for automatically directing data for a specific tenant to certain node groups or even dedicated nodes. This approach enables prioritizing critical tenants or isolating those that generate the highest load.
- **Circuit Breakers:** Built-in mechanisms in OpenSearch that prevent single, overly expensive queries from consuming all cluster resources, which could lead to instability.
- **Client-side Throttling/Rate Limiting:** Enforcing limits on the client application to prevent a single tenant from sending too many queries in a short period. This is a proactive approach that protects the cluster from overload.
- **Dedicated Nodes:** For the most critical or resource-intensive tenants, consider allocating dedicated data nodes, or even entire, smaller clusters. While this partially blurs the line between multi- and single-tenant, it is an effective isolation strategy.

For cluster stability, it's typically observed that an OpenSearch cluster remains stable with **up to 30,000 indices and 75,000 shards**. In a multi-tenant setting, if we are expecting clients' data to grow, I would recommend provisioning a new multi-tenant cluster for subsequent clients once we reach about **40-50% of cluster capacity** (index and shard-wise). This proactive approach accommodates for clients' growth and prevents operational complexity later, especially if you want to keep all particular client's data within one cluster, eliminating the need to move data between clusters.

Operational Complexity

Operational complexity in multi-tenant OpenSearch is as high as in Kafka, and perhaps even higher due to the nature of indexing and searching.

Upgrades

Upgrades in OpenSearch can be performed in a rolling fashion (zero downtime), but they require careful planning, especially in a multi-tenant environment. Changes in API, indexing behavior, or search logic can affect all tenants. Testing client compatibility and potential regressions is crucial. Coordinating client-side updates with each tenant can be a logistical nightmare.

Observability

Similar to Kafka, comprehensive observability is absolutely essential. In addition to standard cluster metrics (CPU, memory, disk, network) and indexing/search metrics, you need:

- **Tenant-Specific Dashboards:** Visualize metrics specific to each tenant, such as:
    - **Search Latency:** Response time for queries for a given tenant.
    - **Indexing Rates:** How quickly data is being indexed for a given tenant.
    - **Query Success/Failure Rates:** Percentage of successful and failed queries.
    - **Resource Consumption per Index/Tenant:** Identify which tenant consumes the most resources.
- **Query Monitoring:** Track expensive queries that can burden the cluster and attribute them to specific tenants to identify "noisy neighbors."

Capacity Planning

Capacity planning in OpenSearch focuses on the number and size of shards and nodes. OpenSearch scales horizontally by adding more data nodes.

- **Shard Management:** An incorrect number of shards (too many small, too few large) can lead to performance issues and resource consumption. Each shard (and every copy of a shard) is a separate Lucene instance or process. Meaning that creation of many smaller shards will end up spinning off many Lucene instances that will consume CPU and memory. Too many shards can lead to unnecessary resource consumption and performance issues.
  Too many large shards are also not ideal. OpenSearch clusters frequently rebalance shards across nodes to maintain an even distribution of data and load.

Moving very large shards during these rebalancing operations consumes more network bandwidth and CPU resources, making the process slower and potentially impacting cluster performance. When shards become excessively large, queries take longer to complete because each query has to process a larger volume of data within that single shard. Each search operation on a shard uses a single CPU thread, meaning that a larger data set on that shard will take longer to scan and process. Finding the right balance is key.

- **Impact of Schema Changes (Mapping):** In a multi-tenant environment, where each tenant might have slightly different data schema requirements, managing and deploying changes to index mappings becomes complex and can affect many tenants simultaneously.

Conclusion: Choosing Your Path to Success

The journey through the labyrinth of multi-tenant and single-tenant architectures, especially in the context of technologies like Kafka and OpenSearch, reveals that there is no single, universal answer. As we've seen, security, resource contention, and operational complexity all present unique challenges that demand in-depth analysis and a strategic approach.

Kafka, with its role as a communication protocol and durable event log, requires meticulous management of ACLs and consideration of message-level encryption for individual tenants. Resource contention can be mitigated using extensive quotas and intelligent capacity planning, but all of this adds to operational complexity, especially in the context of upgrades and observability.

OpenSearch, as a powerful indexing and search tool, puts the challenge of data encryption in use at the forefront. Here, plugins like Portal26 become a game-changer, enabling secure searching of encrypted data. However, resource contention and operational complexity, stemming from managing shards, queries, and deploying updates, remain key points to consider.

Time for Action

Now is the time to take this knowledge and apply it in practice. Instead of blindly following the trend of multi-tenancy or sticking to the safe haven of single-tenancy, ask yourself the following questions:

1. **What is the true cost of my current architecture**? Do the financial benefits of multi-tenancy outweigh the potential operational costs and security risks?
2. **What are my security and compliance requirements?** Do I have the tools (like Portal26) and processes that can meet these requirements in a shared environment?
3. **What is the expected load pattern and its variability?** Will my tenants be "noisy neighbors," or can I effectively manage resource contention?
4. **What is the maturity level of my operational team?** Are we ready for the added complexity of managing, monitoring, and troubleshooting in a multi-tenant environment?
5. **Do I have the right observability and automation tools** to effectively scale and maintain such a complex system?

Benefits Within Reach

By consciously approaching these questions and assessing the security, resource contention, and operational complexity of your architecture, you will build systems that are:

- **More Resilient:** Better isolation and resource management minimize the risk of failures and disruptions.
- **More Cost-Effective:** Optimal resource utilization translates into lower infrastructure bills.
- **More Secure:** Well-thought-out security strategies protect your data and your customers' data.
- **Easier to Manage:** Understanding complexity allows for proactive problem-solving and efficient planning.

The choice between multi-tenancy and single-tenancy is not an all-or-nothing decision but a spectrum of possibilities. Remember that even in a multi-tenant system, you can implement elements of single-tenant isolation in critical areas. The key is to understand the trade-offs and make conscious decisions that best serve your product and your customers.