Multi-tenancy guidance for ISVs running Amazon Neptune databases

AWS Prescriptive Guidance

AWS Prescriptive Guidance: Multi-tenancy guidance for ISVs running Amazon Neptune databases

Table of Contents

Multi-tenancy guidance for ISVs running Amazon Neptune databases

Brian O'Keefe, Veeresham Gande, Dana Owens, and Nima Seifi, Amazon Web Services

Multi-tenancy is a computer systems architecture where a single instance of an application serves multiple customers. Each customer is referred to as a *tenant*. In a multi-tenant architecture, these instances of the application operate in a shared environment where each tenant is physically co-located on the same infrastructure but is logically separated.

As an independent software vendor (ISV), you can use Amazon Neptune to power applications that require navigation across highly connected data. You might be managing a cloud-based software as a service (SaaS) application in your account and providing tenants with subscriptions. Tenants can then access the service over the internet or privately over AWS PrivateLink. The economics of this model work for both parties, because the tenant gets access to software that's less expensive than it would be for them to purchase, build, and maintain. As the ISV, you can charge more for the subscription than it costs you to create and maintain the software. The question is how do you scale your business to multiple tenants.

Multi-tenancy provides ISVs with important economical and operational benefits. The multi-tenant architecture gives your organization a better return on investment (ROI). Multi-tenancy also simplifies operational requirements so that your organization can move more quickly and reduce the cost of delivering the software to your tenants.

This document provides guidance on effectively running a multi-tenant ISV application using Amazon Neptune. This guidance is based on best practices gained over years of supporting ISVs' successful delivery of SaaS solutions to their customers. Evaluating this guidance in the context of your organization's goals and architectural principles will help you find ways to optimize your solution.

> **Note**
>
> This document does not provide an exhaustive list of best practices. It supplements the document Applying the AWS Well-Architected Framework for Amazon Neptune by providing additional specific guidance for multi-tenancy ISV workloads. We recommend reviewing the considerations in both documents when designing your solution.

SaaS data-partitioning models

One of the challenges for SaaS developers is to design architectural patterns for representing and organizing data in a multi-tenant environment. These multi-tenant storage mechanisms and patterns are typically referred to as data partitioning.

In a multi-tenant SaaS environment, it's important to distinguish between data partitioning and tenant isolation. These concepts, while related, are not synonymous. Data partitioning refers to the method of storing data for each tenant. However, partitioning alone does not guarantee tenant isolation. Additional measures are necessary to ensure that the data of one tenant remains inaccessible to another.

The three common data-partitioning models in multi-tenant SaaS systems are silo, pool, and hybrid. Your choice of any model depends on factors such as the following:

- Compliance
- Noisy neighbors
- Tiering strategy
- Operational requirements
- Tenant-isolation needs

Additionally, each database type available on AWS typically offers a unique collection of data partitioning and tenant-isolation models. When looking at how tenant graphs can be organized to support the various needs of your solution, consider the models that Amazon Neptune provides.

Many ISVs start their design on Neptune with one of the following assertions:

- The ISV solution requires physical separation of customers across separate clusters.
- The ISV solution requires constructs such as named databases or schemas found in traditional relational database management systems.

After consideration, ISVs realize that these assertions aren't true because, under almost all workloads, each of their customers has a disconnected graph in their database. Implementing the data modeling and access guidance discussed in this document prevents those data boundaries from being crossed and maintains customer data privacy.

This guide describes both the silo model and the pool model, but most ISVs choose the pool model for cost and operational efficiency. The guide briefly discusses a hybrid model that combines aspects of both silo and pool models. Some ISVs use a hybrid model for their largest customers to accommodate regulatory or compliance requirements of the size of graph.

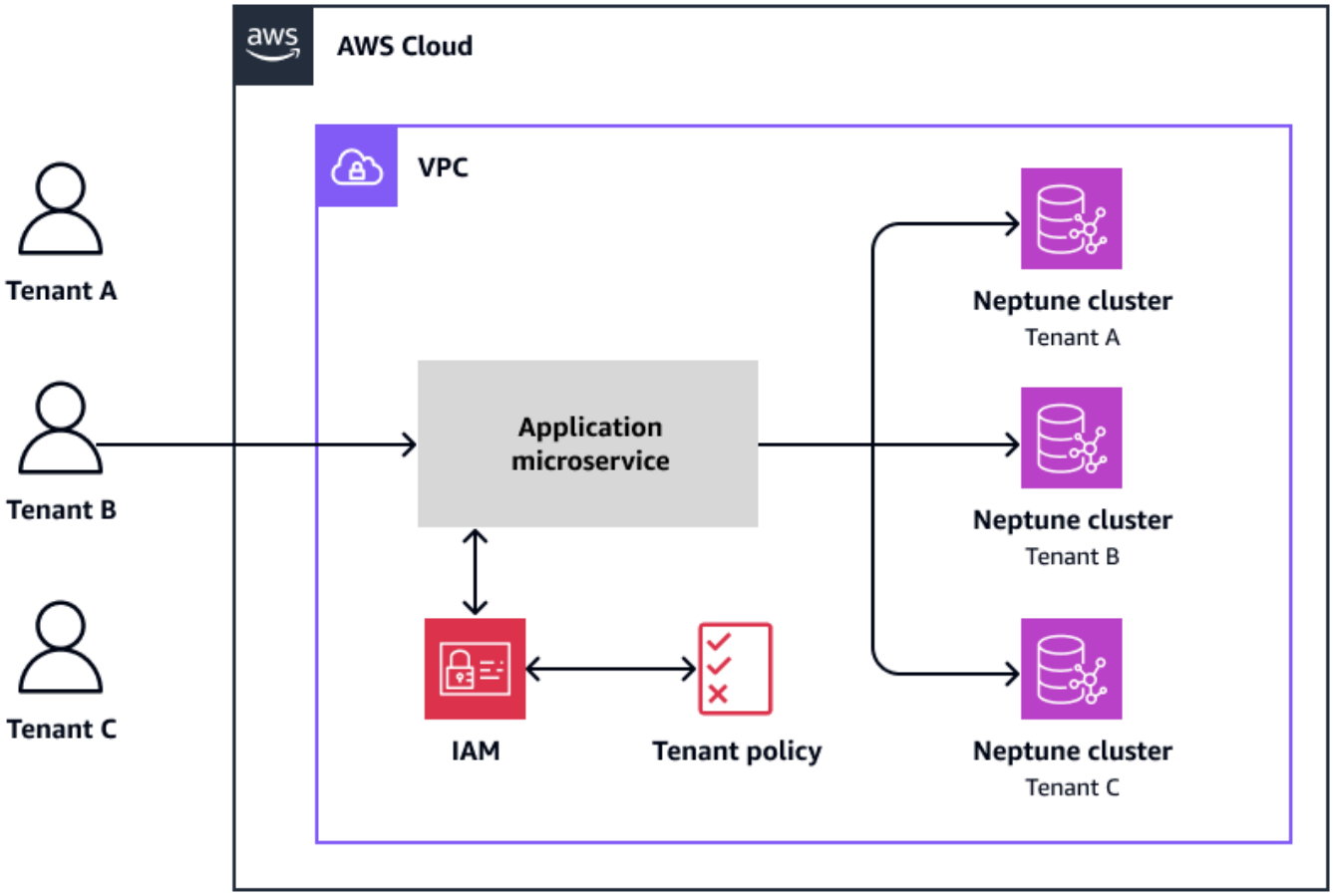
Silo-model multi-tenancy

Some multi-tenant SaaS environments might require tenants' data to be deployed on fully separated resources because of compliance and regulatory requirements. In some cases, large customers require dedicated clusters to reduce noisy neighbor impact. In those situations, you can apply the silo model.

In the silo model, storage of tenant data is fully isolated from any other tenant data. All constructs that are used to represent the tenant's data are considered physically unique to that client, meaning that each tenant will generally have distinct storage, monitoring, and management. Each tenant will also have a separate AWS Key Management Service (AWS KMS) key for encryption. In Amazon Neptune, a silo is one cluster per tenant.

Cluster per tenant

You can implement a silo model with Neptune by having one tenant per cluster. The following diagram shows three tenants accessing an application microservice in a virtual private cloud (VPC), with a separate cluster for each tenant.



Each cluster has its individual endpoint to help ensure distinct access points for efficient data interaction and management. By placing each tenant in its own cluster, you create the well-defined boundary between tenants ensuring customers that their data is successfully isolated from other tenants' data. This isolation is also appealing for SaaS solutions that have strict regulatory and security constraints. Additionally, when each tenant having its own cluster you don't have to worry about noisy neighbor, where one tenant imposes a load that could adversely affect the experience of other tenants.

While the cluster-per-tenant silo model has advantages, it also introduces management and agility challenges. The distributed nature of this model makes it harder to aggregate and assess tenant activity and the operational health across all tenants. Deployment also becomes more challenging because setting up a new tenant now requires the provisioning of a separate cluster. Upgrading becomes more challenging in environments with a shared client layer when client upgrades and versions are tightly coupled to the database upgrade.

Neptune supports both serverless and provisioned clusters. Assess whether your application workload is better handled by serverless or provisioned instances. In general, if your workload has a

constant level of demand, provisioned instances will be more cost effective. Serverless is optimized for demanding, highly variable workloads with heavy database usage for short periods of time followed by long periods of light activity or no activity.

When using a Neptune provisioned cluster per tenant, you must select an instance size that approximates the maximum load of your tenant's demand. This dependence on a server also has a cascading impact on the scaling efficiency and cost of your SaaS environment. While a goal of SaaS is to size dynamically based on actual tenant load, a Neptune provisioned cluster requires you to over-provision to account for heavier periods of usage and spikes in loads. Over-provisioning increases the cost per tenant. Additionally, as tenant usage changes over time, scaling up or scaling down the cluster must be applied separately for each tenant.

The Neptune team generally advises against a silo model because of the higher cost incurred by idle resources and the additional operational complexities. However, for highly regulated or sensitive workloads require this additional isolation, customers might be willing to pay the additional cost.

Implementation guidance for the silo model

To implement a cluster-per-tenant silo-isolation model, create AWS Identity and Access Management (IAM) data-access policies. These policies control access to tenants' Neptune clusters by ensuring that tenants can access only the Neptune cluster containing their own data. Attach the IAM policy for each tenant to an IAM role. The application microservice then uses the IAM role to generate fine-grained temporary credentials using the `AssumeRole` method of AWS Security Token Service (AWS STS). These credentials, which have access only to the Neptune cluster for that tenant, are used to connect to the tenant's Neptune cluster.

The following code snippet shows a sample data-based IAM policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "neptune-db:ReadDataViaQuery",
        "neptune-db:WriteDataViaQuery"
      ],
      "Resource": "arn:aws:neptune-db:us-east-1:123456789012:tenant-1-cluster/*",
      "Condition": {
```

```
        "ArnEquals": {
          "aws:PrincipalArn": "arn:aws:iam::123456789012:role/tenant-role-1"
        }
      }
    }
  ]
}
```

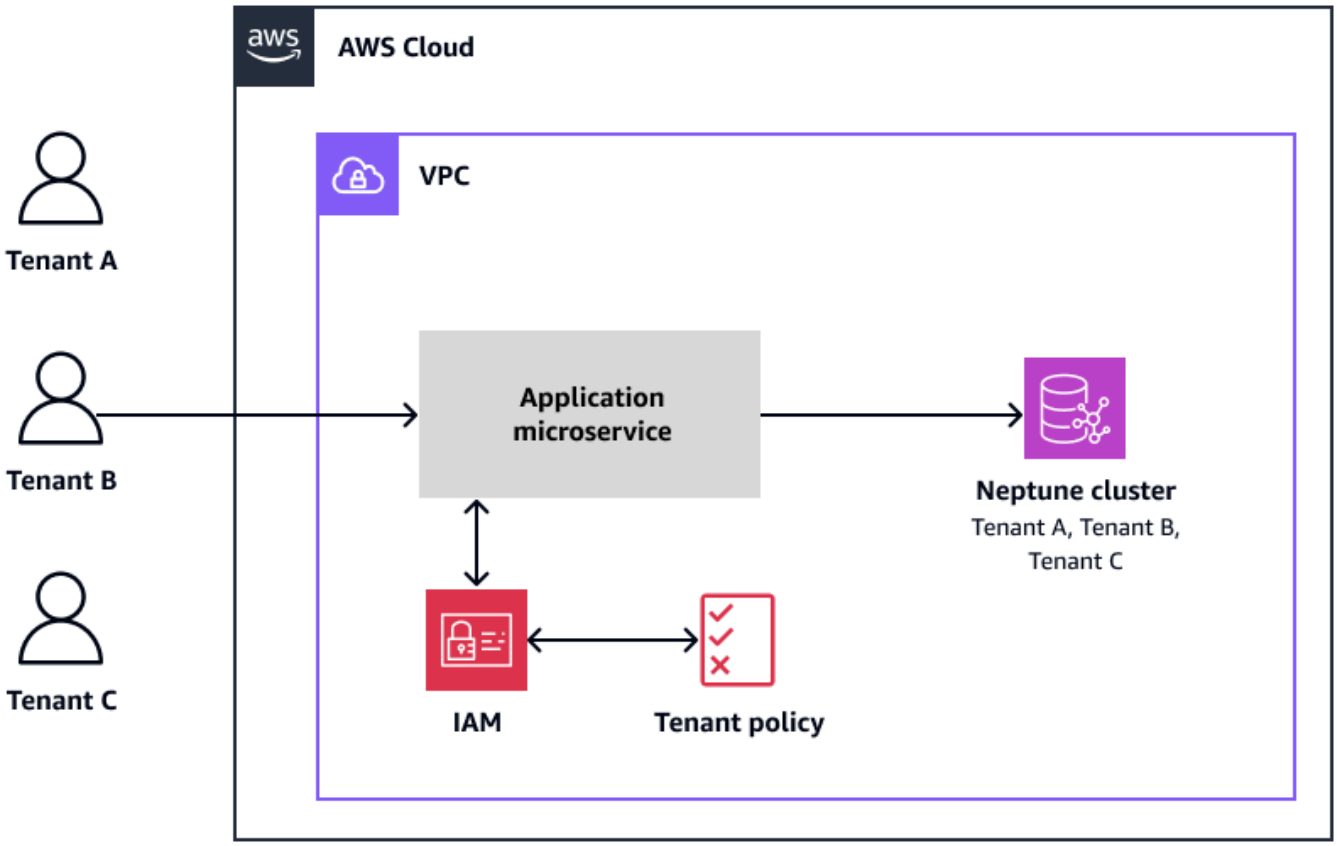
The code provides a sample tenant, `tenant-1`, with read and write query access to their respective Neptune cluster. The `Condition` element ensures that only the calling entity (the principal), which has assumed the `tentant-1` IAM role (`tenant-role-1`), is allowed to access `tenant-1`'s Neptune cluster.

Pool-model multi-tenancy

Sometimes it isn't necessary or feasible to implement the silo model because of cost or operational overhead:

- You might not have the resources to maintain an individual cluster per tenant.
- It might not be necessary to physically separate each tenant's data, and a logical separation is enough to meet their needs and compliance requirements.

The following diagram shows the pool model, with tenant data is placed in a single Amazon Neptune cluster, and all tenants share a common database.



This pool-isolation model reduces the management overhead and can improve the operational efficiency because there are fewer clusters to manage. Also, compute resources can be shared across multiple customers instead of remaining idle during customer inactive periods.

When you use the pool model, there are two ways to model data. Your approach depends on whether you're building a labeled property graph (LPG) or a graph with the Resource Description Framework (RDF).

Pool model for labeled property graphs

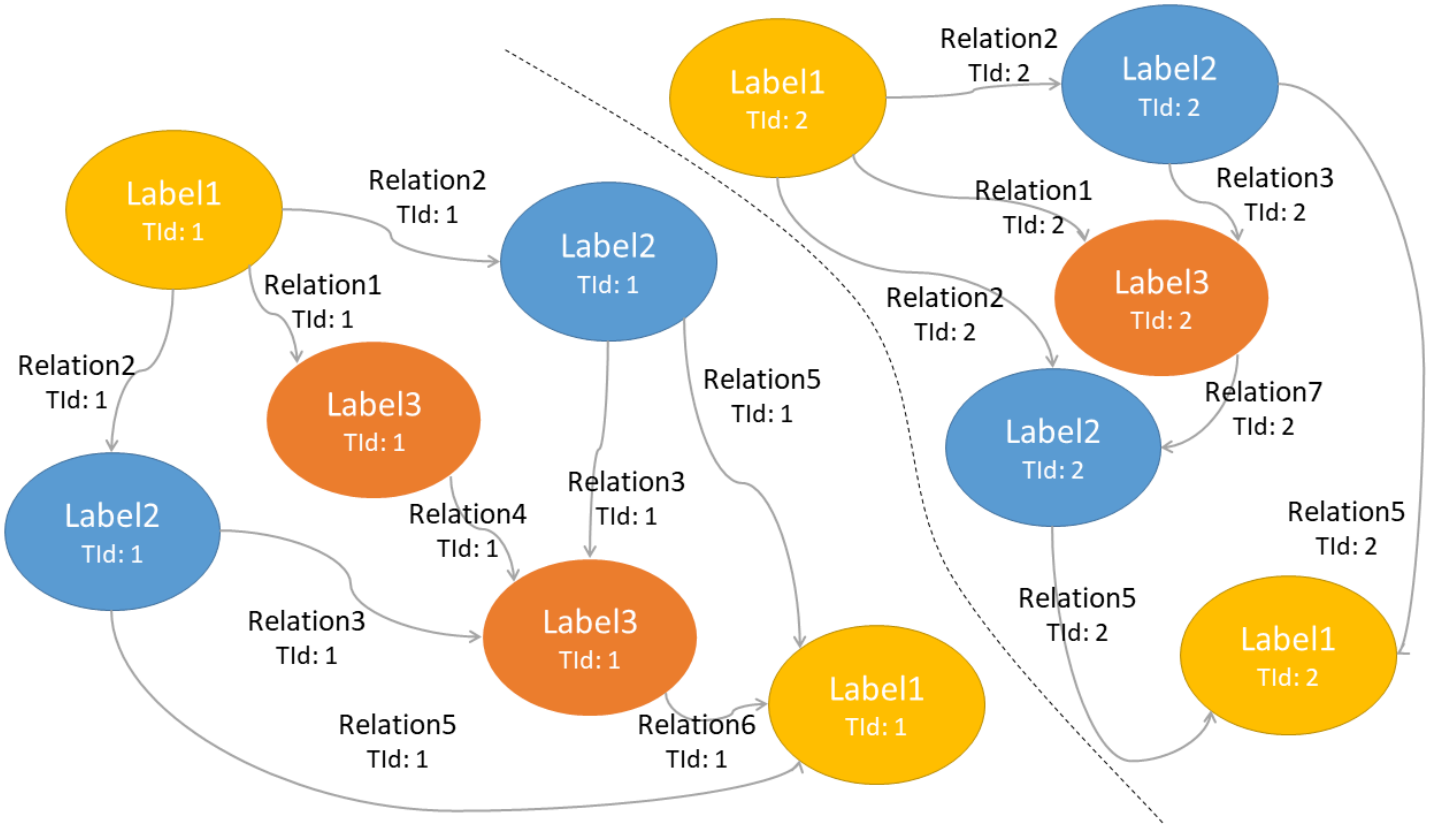
There are three different approaches to the pool model for LPGs on Amazon Neptune:

- **Property strategy** – Choose the property strategy when you need to prioritize use of established library constructs such as the Apache TinkerPop Gremlin language's PartitionStrategy over performance.
- **Prefix-label strategy** – We recommend the prefix-label strategy for most scenarios based on performance and limiting noisy neighbor effects.
- **Multiple-label strategy** – The multiple-label strategy has the improved performance of the prefix-label strategy. It also supports running queries that span all of the tenants on a cluster (for example, ISV queries for reporting or monitoring across all tenants).

Property strategy

With LPGs, users can add key-value pair properties to nodes, or vertices, and edges. To achieve logical separation, most customers intuitively model this as a unique property on every node and edge with a common *tenant property key*. The tenant property key represents the all the tenants that own the node. The *tenant identifier* is a unique value that identifies an individual tenant.

The following diagram shows this model. The two disconnected subgraphs have various labeled nodes and edges, with the tenant property key represented by `TId`. Every node and edge from one subgraph has a `TId` value of 1. In the other subgraph, every node and edge has a `TId` value of 2.



Within labeled property graphs, there are two ways to manage this. The Gremlin query language offers the PartitionStrategy traversal library to help manage data partitioning of the data. The code in the following example expects every node and edge to have a property called `TId`:

```
strategy1 = new PartitionStrategy(partitionKey: "TId", writePartition: "1",
 readPartitions: ["1"])
strategy2 = new PartitionStrategy(partitionKey: "TId", writePartition: "2",
 readPartitions: ["2"])
```

When new nodes or edges are written, the property `"TId"` is added with a value of `"1"` or `"2"`, depending on whether `strategy1` or `strategy2` was selected. For the customer with `"TId"` of `"1"`, you use `strategy1`. The following example shows writing data for that customer:

```
g.withStrategies(strategy1).addV("Label1").property("Value", "123456").property(id,
 "Item_1")
```

For read queries, a filter for `"TId == '1'"` or `"TId == '2'"` is added to every node or edge traversal by using `strategy1` or `strategy2`, respectively. These partition strategies simplify your code, but they aren't necessary. The benefit of using the strategy is that it can be injected at an

authorization level and passed to the lower-level code that forms the query. This separates the code that determines the customer identifier (`TId`) from the logic of the query.

The following example code shows a Gremlin query to read data:

```
g.withStrategies(strategy1).V().hasLabel("Label1")
```

The preceding code is equivalent to the following example:

```
g.V().hasLabel("Label1").has("TId", "1")
```

Likewise, when writing data by using Gremlin, you can use the following query:

```
g.withStrategies(strategy1).addV("Label1").property("Value").property(id, "Item_1")
```

The preceding code is equivalent to the following example, which does not use the partition strategy and therefore requires the `"TId"` property to be explicitly written:

```
g.addV("Label1").property("TId", "1").property("Value").property(id, "Item_1")
```

In openCypher, these libraries do not exist. You are responsible for writing and modifying your queries to add the tenant identifier as a property on nodes and edges. For example:

```
CREATE (n:Item {`~id`: 'Item_1', Value: '123456', TId: '1'})
CREATE (n:Item {`~id`: 'Item_2', Value: '123456', TId: '2'})
```

Note the similarity between the Gremlin code without the partition strategy. You can then read the node written from the first `CREATE` statement by using the following code:

```
MATCH (n:Item {TId: '1'})
RETURN n
--or
MATCH (n:Item)
WHERE n.TId == '1'
RETURN n
```

You might choose the property strategy when you want to use native TinkerPop Gremlin constructs such as PartitionStrategy. However, this model has performance drawbacks on Amazon Neptune

compared with the prefix-label strategy. For a discussion of these performance drawbacks, see the Performance implications for the LPG models section.

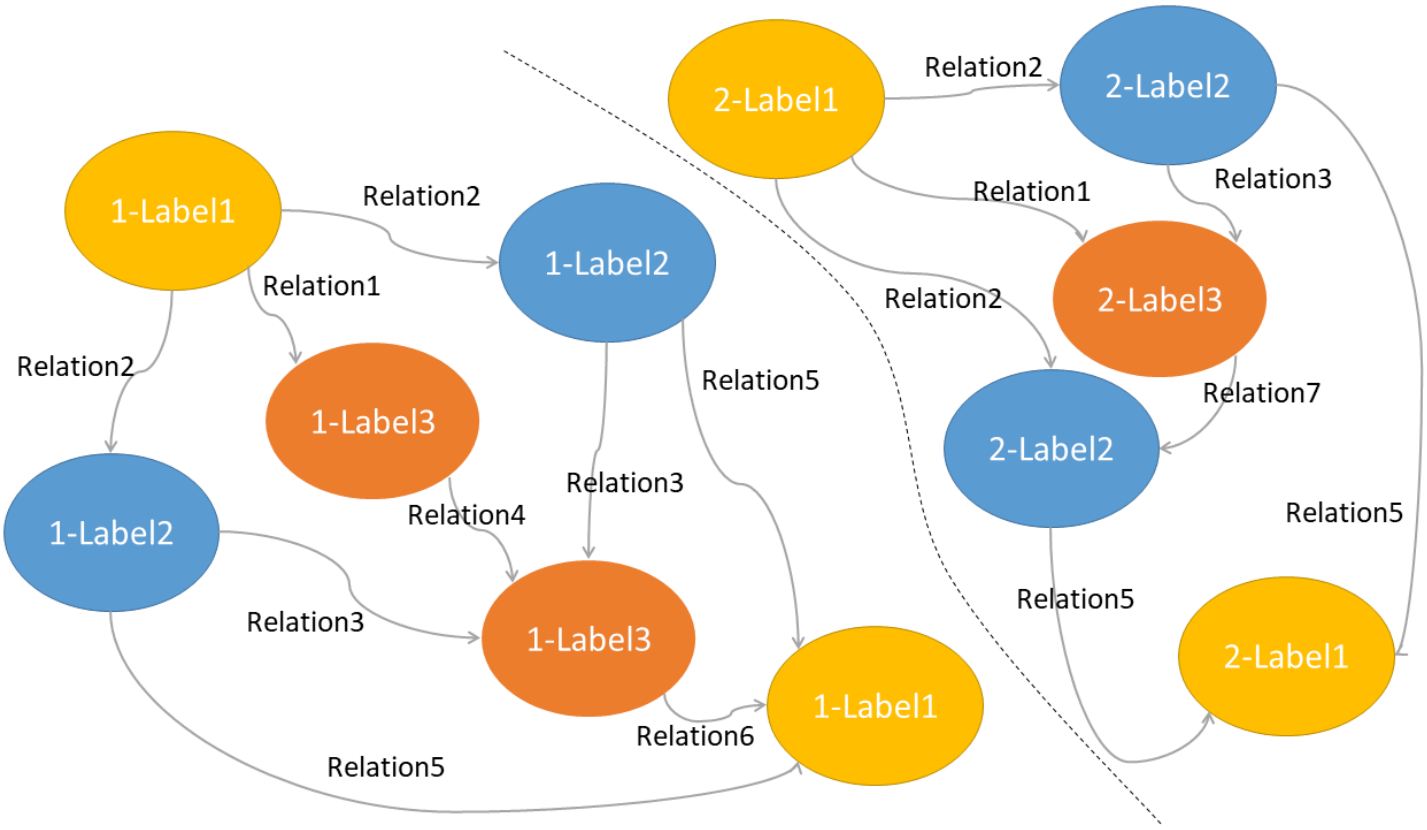
If the following conditions apply, consider modeling the property strategy only on nodes, not on edges:

- Your graph has significantly more edges than labels.
- Each tenant is a disconnected graph.
- You access the graph only by using nodes as a starting point, not labels.

Prefix-label strategy

If performance is a top concern, we highly recommend considering the prefix-label strategy over the property strategy.

In the prefix-label strategy, you label each node with a combination of tenant identifier and node label. For example, if the tenant has an identifier of `"1"` and the node label is `"Label1"`, you specify the node label as `"1-Label1"`. The following diagram shows two disconnected subgraphs that use this model.



When writing data in Gremlin, you can add an identifying number to any node's label:

```
g.addV("1-Label1")
g.addV("2-Label6")
```

When querying this graph, you can check for the existence of this prefix on a node:

```
g.V().hasLabel("1-Label1")
```

In openCypher you can write data by using a CREATE statement:

```
CREATE (n:`1-Label1` {`~id`: 'Item_1', Value: 'XYZ123456'})
```

To query the data that you wrote in openCypher, use the following code:

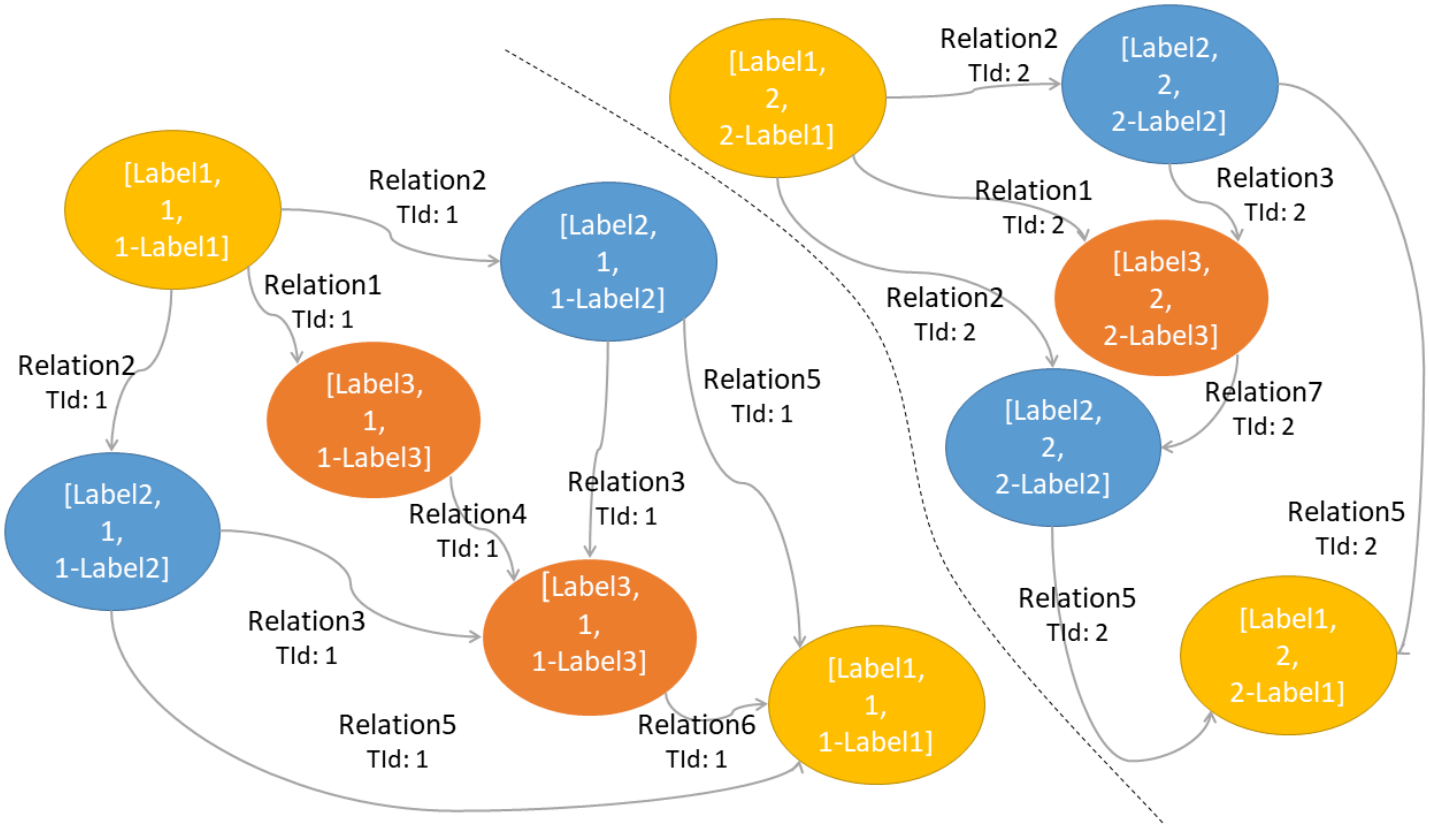
```
MATCH n= (:`1-Label1`)
RETURN n
```

The prefix-label strategy assumes that all nodes are assigned to one or more tenants and that permissions are not assigned at the edge scope. Avoid using this strategy on edge labels, because that will cause a large number of predicates and will negatively impact Neptune performance.

There are two primary drawbacks to the prefix label approach. First, it's difficult to run any queries that span across tenants. An example is a query that counts all nodes of a given label for reporting or monitoring. If this is your use case, consider combining this strategy with the multiple-label strategy. For more information about combining strategies, see the Hybrid model section. Second, the prefix-label strategy requires controls that enforce proper application of the appropriate prefix to every query to prevent data leakage. However, this strategy is the most efficient option for workloads that require low latency queries, and we highly recommend it. The Performance implications for LPG models section provides examples of why this is the most efficient strategy.

Multiple-label strategy

The third option is to use a multiple-label strategy. For this approach, you add extra labels to every node on the graph. For example, if you need to filter across all of the data for a given tenant, add the tenant ID label. If you need to filter across all data for a given label regardless of tenant, add that label. The following diagram shows the multiple-label strategy applied by using three labels for each node.



You can now access the graph by using three different patterns:

- Filter on `Label1` to return all nodes with `Label1` across all tenants.
- Filter on `1` to return all nodes for tenant 1.
- Filter on `1-Label1` to return all nodes for only tenant 1 with label `Label1`.

For LPGs, there are two ways to implement this.

In Gremlin, you can use the traversal strategy called SubgraphStrategy to limit the scope of all queries to only vertices with a specific label, such as `"Label1"`:

```
g.withStrategies(
    new SubgraphStrategy(
        vertices=hasLabel("Label1")
    )
)
```

Unlike PartitionStrategy, SubgraphStrategy impacts reading data only, not writing data. To write the data, manually assign the labels in each query:

```
g.addV("Label1").property("Value","XYZ123456")
.addV("Label2").property("Value","XYZ123456")
```

When reading the data, you can use SubgraphStrategy to query all nodes with `"Label1"`:

```
g.withStrategies(
new SubgraphStrategy(vertices=.hasLabel("Label1"))
).
V().has("Value","XYZ123456")
```

Neptune returns only the first record, which has `"Label1"` and a value of `"XYZ123456"`. It's equivalent to the following query, which doesn't use SubgraphStrategy:

```
g.V().hasLabel("Label1").hasValue("XYZ123456")
```

In this basic query, it appears that SubgraphStrategy is more complex to use. Keep in mind that your libraries can provide an instance of g with the strategy already defined. Developers don't have to ensure that the proper filters are applied:

```
def getGraphTraversal():
return g.withStrategies(new SubgraphStrategy(vertices=.hasLabel("Label1"))

getGraphTraversal().has("Value","XYZ123456")
```

The openCypher libraries don't have these constructs, so you must create multiple labels for each node:

```
CREATE (n:`1`:`Label1`:`1-Label1` {`~id`: 'Item_1', Value: '12345'})
```

When you use these labels to filter for a subgraph, you can return nodes that have the customer label you are looking for or that share a relationship with another node that has that label:

```
MATCH n=(:Label1:`1`)
// or
MATCH n=(:`1-Label1`)
```

The multiple-label strategy gives you the most flexibility to query nodes by type (`Label1`) or tenant (1), or to use the more efficient prefix-label strategy when performance is of most importance (`1-Label1`).

The major drawback to this strategy is that each label is an extra object stored in your graph. An object is a node, edge, or a property on a node or edge in LPGs. Ingestion speed is measured and bound by objects per second, and storage costs depend on the number of gigabytes consumed. This means that extra objects might have a measurable impact at large scale.

Performance implications for the LPG models

The AWS Skill Builder course Data Modeling for Amazon Neptune describes in depth the Neptune data model internals and modeling implications, but we will summarize the important considerations for these designs here. Consider having three tenants (T1, T2, T3) on a single Neptune cluster. These tenants have the following attributes:

- Tenant 1 (T1) has 100 million nodes total, and 10 million are of type Item.
- Tenant 2 (T2) has 10 million nodes total, and 1 million are of type Item.
- Tenant 3 (T3) has 100 million nodes total, and 1 million are of type Item.

Run a query that will retrieve the items for Tenant 3 by using the property strategy. Neptune inspects the statistics for two index calls:

- Where `tenant property key=T3` has 100 million results
- Where `label = Item` has 12 million results (10 million from T1 + 1 million from T2 + 1 million from T3)

The Neptune query optimizer determines that the latter query is best applied first (12 million results) and then inspects each item for `tenant property key=T3`. You retrieve 12 million items to find the 1 million results.

Notice the noisy neighbor impact of this query. If you had 100 million Item nodes per tenant, the first query would have 300 million results instead of 12 million (This is overly simplified for illustrative purposes. The Neptune optimizer might have applied a different order of operations).

Next, consider the prefix-label strategy. Make a single index call where `label=T3-Item`, which returns 1 million results. This accomplishes the same result as the property strategy, but it retrieves 11 million fewer records. In addition, you no longer have noisy neighbor concerns because the label doesn't overlap in the index.

The multiple-label strategy doesn't provide query performance improvement over the property strategy directly. Filtering by property value is comparable to filtering by label value when the

search space is also comparable. Instead, the multiple-label strategy supports more flexibility. The multiple-label strategy provides performance equivalent to the prefix-label strategy for `label=T3` or the label `T3-Item`. The multiple-label strategy provides performance equivalent to the property strategy for `label=Item`. The benefit is to support a variety of access patterns.

Pool model for RDF

The Resource Description Framework (RDF) has a concept of named graphs, which provides a logical way of separating data. In Amazon Neptune, you have a default named graph and user-defined named graphs. You can create as many named graphs as you want. Collectively, they are called the RDF dataset. All named graphs, default or user-defined, are defined by an Internationalized Resource Identifier(IRI) within the RDF dataset. In Neptune, unless a user declared a named graph when writing data, all triples are considered part of the default named graph.

There are multiple use cases for named graphs:

- Data partitioning and data isolation
- Data provenance
- Versioning
- Inference

This guide focuses on the data-partitioning use case. We recommend creating one user-defined named graph for each tenant.

SPARQL query options using Graph Store HTTP Protocol

The following example queries use SPARQL Protocol and RDF Query Language (SPARQL) and Graph Store HTTP Protocol to query or create a named graph for a tenant.

- `HTTP GET` – To retrieve a specific graph of a tenant:

```
curl --request GET 'https://your-neptune-endpoint:port/sparql/gsp/?graph=http%3A//
www.example.com/named/tenant1'
```

- `HTTP PUT` – To create or replace a specific named graph with a payload specified in the request:

```
curl --request PUT -H "Content-Type: text/turtle" \  --data-raw "@prefix ex: http://
example.com/ . ex:subject ex:predicate ex:object ." \
```

```
'https://your-neptune-endpoint:port/sparql/gsp/?graph=http%3A//www.example.com/named/
tenant1'
```

In RDF, an object is a triple.

- HTTP POST – To create a new named graph if one doesn't exist, or merge with an existing graph:

```
curl --request POST -H "Content-Type: text/turtle" \
--data-raw "@prefix ex: http://example.com/ . ex:subject ex:predicate ex:object ." \

'https://your-neptune-endpoint:port/sparql/gsp/?graph=http%3A//www.example.com/named/
tenant1'
```

Tenant isolation for RDF

For logical isolation of data with necessary guardrails in place at the application layer, create a mapping between the tenant and user-defined named graphs. When you design multi-tenancy for an RDF dataset, be aware of the following aspects of RDF and SPARQL:

- In Neptune, when you query without specifying a named graph, it retrieves all triples that match the pattern across all named graphs in the database.
- In RDF, there are no constraints around connections between nodes of different named graphs. For instance, in the previous diagram, a node in `:G1` can be connected to a node in :G2 through an edge.

For example, if an end user of a particular tenant submits a query to the API, the API should validate the following requirements before it submits the query to the Neptune database:

- Any query scoped at a single tenant must specify a named graph. Otherwise, you risk leaking data across tenants.
- Update or Delete queries should always specify a named graph.
- Nodes on either side of an edge or relationship should always belong to the correct named graph.

For additional information about best practices, see the Neptune documentation.

Prepare for growth

When you use the pool model successfully, you eventually outgrow the size of a single Neptune cluster. Tenants grow, or the number of tenants grows, and the ingestion rate of data needed across all of your customers exceeds the cluster's capability. When this occurs, you will need to split your customers across multiple clusters. Design for this configuration upfront instead of trying to retrofit for it later. Even if your initial scale is to use only a single cluster, mock up the components you that will need to route tenants across multiple clusters in the future when you reach that scale.

If your solution requires more resources based on your tenant's size, prepare for their growth as well. If several of the customers on a single cluster grow significantly, that cluster might no longer support your requirements. Design a strategy to either move tenants to another cluster or split an existing cluster into two by using the Amazon Neptune DB cloning feature.

Be familiar with the Neptune Copy-on-Write Protocol, which can save you money when you implement DB cloning., If you split a cluster because of ingestion bottlenecks, it might be more efficient not to delete data from the clusters, providing your policies allow that. The two clusters will share a data page if it is unchanged but not if the data page was modified (because some data on it was deleted).

> **Note**
>
> This guidance applies to the most recent Neptune version at the time of this writing, which is Neptune version 1.3.1. This guidance might change in future versions as the Neptune storage layer evolves.

Limitations for multi-tenancy scenarios

Be aware that some Neptune features are not built for multi-tenancy scenarios. Tenants should not be given direct access to Neptune endpoints in a pool model because these multi-tenancy strategies aren't enforced at the database level. Always keep some kind of proxy between your customers and the Neptune endpoint that enforces the designs described in this document. Examples of such a proxy include the following:

- Appending the label filters in your client layer
- Having an API that maps the authentication token to a tenant ID and injects this filter into the query

This guidance also applies to giving customers direct access to features such as Neptune graph notebooks, Neptune graph-explorer, or Neptune Streams.

Hybrid model multi-tenancy

SaaS solutions often use a blend of silo and pool models. Various factors influence the decision of when and how to employ both silo and pool models within the same environment.

One such factor is tiering, where a SaaS solution offers unique experiences to each tier of tenants. For example, if your tiers are Free, Standard, and Premium, your Free tier tenant data could be stored in a shared Neptune cluster using a pool model. For your Standard and Premium tier tenants, you could use a cluster-per-tenant silo model.

Additionally, some SaaS providers have the capability to build their pool solution on a shared Amazon Neptune cluster as their foundation. Subsequently, they can create a separate Neptune cluster for tenants that require siloed storage, often because of compliance and regulatory mandates.

Although this can add a level of complexity to your data-access layer and management profile, it can also offer your business a way to tier your offering to fulfill customer requirements.

Operational best practices for ISVs

Many of the guidelines in this section are best practices for all customers, but they have added significance for ISVs.

Update your Neptune cluster with the newest versions

In the Amazon Neptune release notes, you can see that every version brings a number of bug fixes, performance improvements, and new features. Keep your Neptune clusters on the latest version as much as possible. If you find a previously undiscovered bug in your workload and your cluster is on the latest version, Neptune engineers can create a private patch for your cluster (if warranted and you want it). The patch can bridge until the next release when that fix will be generally available. To help with updating your clusters to the newest version, use the Neptune Blue/Green solution.

Use deltas instead of delete and replace for data ingestion

You can use several techniques to ingest, or write, data to Neptune. Many customers try to simplify their data ingestion by deleting and reinserting their graph every time a change is received in the feed. They might add a `last-modified` property to each node and periodically scan for nodes that haven't been modified since some specified date and delete them. While these techniques simplify the data ingestion process, they have long-term health and scalability implications for your Neptune cluster.

First, Neptune uses dictionary encoding of strings. Unless you explicitly specify the IDs of nodes and edges, Neptune generates a GUID represented as a string for the ID and stores that string in the dictionary. If you are constantly deleting and adding objects, the automatically generated IDs will cause bloat in the dictionary.

Second, Neptune scales up to ingest about 120 K objects per second at the maximum. If you continuously delete and add objects, you consume a lot of that bandwidth on objects that essentially are not changing. This limits the number of tenants that you can host on a cluster, requires larger writer instances in the clusters, and requires more I/O operations. All of these factors increase your costs.

We highly recommend that you develop a way to calculate the true delta of what has changed instead of using the delete and add methods. However, some data sources aren't conducive to this (for example, API calls that return the current state, or events that do not track exactly

what changed). If your raw data source is not conducive to identifying changes, use your extract, transform, and load (ETL) processes to calculate the delta. For example, you can maintain snapshots from each previous data capture in Parquet format, use AWS Glue to calculate the differences between those snapshots, and push only the differences to Neptune.

Model how Neptune costs will evolve with your tenants

Whether you use a silo, pool, or hybrid model, your cloud costs will scale with your tenants' size. Tenants that require more concurrent connections need larger instances or more read replicas than those with fewer concurrent connections. The same applies to tenants that require more rapid data ingestion. The three components of Neptune cluster cost are instance size (and number), data size (GB-months), and I/O operations (per million). While these costs are generally workload specific, they scale with size and data volume, they can be measured by using AWS tools. Track and understand the economies of scale against key indicators of your tenants' sizes, including how their sizes vary over time. If the unpredictability of your I/O charges impacts your margins, consider choosing Neptune I/O-Optimized storage for a more predictable cost.

Scale your clusters for customer demand

There is no tried or true formula for right-sizing your Neptune instance size. The Neptune documentation provides guidance, but there are too many variables to recommend a direct mapping. These variables include but aren't limited to the following:

- Data model
- Data shape
- Query concurrency
- Query complexity.

Plan testing to determine the optimal size for your workloads and tenant profiles. In general, we recommend using provisioned instances for cost efficiency and predictability. If your customer-experience goals prioritize optimal scaling over costs, consider using Neptune Serverless instances to ensure a more consistent experience regardless of workload fluctuations.

If your tenant read workloads have significant variability in their peaks and troughs, combine Neptune Serverless instances with Neptune auto-scaling. It usually takes 10-15 minutes for a new read replica to come online after it's initialized. This means that auto-scaling alone can

handle prolonged changes in traffic, but it isn't sufficient for rapidly changing spikes in activity. By combining Neptune Serverless and Neptune auto-scaling, you can both scale instances up or down and scale the number of read replicas in and out.

If your tenants have significantly different workload profiles or service level agreements (SLAs), consider using custom endpoints and dedicated read replicas to direct traffic to instances that are optimized for that traffic. Optimization can include a different sizing of the instance, specific query patterns, or pre-warming the buffer cache.

Next steps

If you are just starting on your journey implementing Amazon Neptune for your multi-tenancy ISV application, put extra thought into your desired model. Changing the model will be more costly later in your journey.

If you are early in your journey, verify that you are using the best model for your needs and that you are following the guidance for that model.

Plan ahead. When you are early in your journey, it's tempting to defer work on sharding customers across clusters or optimizing your ETL processes to provide the delta of changes instead of deleting and re-adding vertices and edges. As you scale, those decisions might negatively impact performance and cost.

Finally, if you are already well into your journey, this guidance might reassure you that your architecture is optimal, or it might provide changes to improve your architecture.

If you have questions on this guidance or need further assistance, please reach out to your AWS account team and ask for a session with a Neptune specialist.

Resources

- Amazon Neptune documentation
- Data Modeling for Amazon Neptune (course)
- Applying the AWS Well-Architected Framework for Amazon Neptune
- SaaS Lens Well-Architected Framework
- Guidance for Multi-Tenant Architectures on AWS
- SaaS Tenant Isolation Strategies: Isolating Resources in a Multi-Tenant Environment
- Apache TinkerPop documentation
- SPARQL

Contributors

Contributors to this guide include:

- Brian O'Keefe, Principal WW SSA Neptune, AWS
- Veeresham Gande, Senior Technical Account Manager, AWS
- Dana Owens, Startup Solutions Architect, AWS
- Nima Seifi, Startup Solutions Architect, AWS

Document history

The following table describes significant changes to this guide.

Change	Description	Date
Initial publication	—	September 3, 2024