Multi-Tenancy in a Microservice Architecture

AMIT GUD

Amit has worked for multiple companies in the storage and systems domain, from early-stage startups to multi-billion dollar companies. Currently focused on making Uber's infrastructure robust, Amit has a track record of tackling impactful issues relating to large-scale systems, performance, and scalability. Amit has a master's degree from Kansas State University. He has worked on multiple research papers and has authored multiple (pending) patents. amitgud@gmail.com

Microservice architecture is increasingly common for a scalable system with high developer velocity and short time-to-market. It allows the flexibility for teams to operate on independent schedules while meeting the externally committed service level agreements (SLAs). As architectural complexity evolves along with a business, some aspects of the microservice architecture become critical for the developer and business velocity.

One such aspect is to be able to safely and reliably roll out new changes to the architecture in the areas of actual code, service configuration, data semantic, and data schema. With diverse teams working on interoperating services, it becomes critical to be able to roll out a change to a service only after ascertaining the change's impact on dependent services. As multiple teams churn out features for their services, they often have to validate whether the new changes meet the SLAs. Being able to do this easily has direct and positive impact on developer velocity.

Another aspect critical for business continuation and growth is being able to reuse parts of the architecture in a modular way to add new product lines. With the right layers of abstraction and modularity this can not only be cost effective but can also speed up time to market.

One of the most effective ways of addressing both these aspects is by allowing multiple tenants to co-exist in a microservice architecture. A tenant could be a test, canary, shadow, a different service tier, or a different product line altogether. Being able to guarantee isolation and make routing decisions based on the tenancy of the traffic would provide us the infrastructure agility needed for developer velocity and effectively new product innovations.

Having the ability to be able to attach a notion of tenancy to both data-in-flight (e.g., requests, messages in the messaging queue) as well as data-at-rest (e.g., storage, persistent caches) allows for isolation guarantees, fairness guarantees, and tenancy-based routing opportunities. This helps us achieve a variety of things, including better integration testing framework, shadow traffic routing, recording and replaying traffic, hermetic replay of live traffic for experimentation, capacity planning, realistic performance and stress testing, and even things like canary deployments and being able to run multiple business-critical product lines on the same microservice stack.

Stateless services, which are typically containerized applications that do not keep state locally, are more widely deployed than stateful applications and short-lived "serverless" or lambda services. Architecture discussed here is more suited to stateless services.

Microservices Landscape

In this section we will explore microservice landscape and various use-cases for multi-tenancy within microservice architecture.

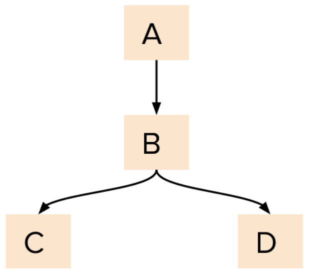

Figure 1: Request flow in a microservice architecture

Integration Testing

One of the most appealing aspects of a microservice architecture is developer velocity. It allows teams to roll out new features and bug fixes for their services independent of others. A team may typically own a handful of services. These services could be interacting with multiple other services as part of its business logic and would have agreed upon SLAs.

For example, consider Figure 1. Here we have a simple scenario of four microservices A, B, C, and D. Service A gets a request from the outside world. It processes the request by connecting to B, which in turn connects to C and D to process the request.

In this example, if we make a change to service B, we will have to make sure it still interoperates well with A, C, and D. Services A, C, and D may belong to different teams, and we may not have control over their deployment schedules. This can be considered an integration testing scenario where we want to test a service's interaction with other services in the system. In this example, and in any microservice architecture in general, there are two fundamental ways of doing integration testing.

Parallel Testing Stack

One approach would be to create a parallel stack, sometimes referred to as a staging environment, which looks and feels like a production stack, but will be used only for handling test traffic. This stack always exists and is always running production code although it is completely isolated from the production stack and is smaller in scale. In this approach, the team making a change would deploy the service with the new code in the test stack. This approach allows us to safely test any service without affecting the production stack. Any bugs or issues would be contained in the test stack only.

In this approach we will need the ability to ascertain that test traffic never leaks to the production stack. This can be achieved by physically isolating the two stacks into separate networks and also by making sure test tools only operate on the test stack.

Although this approach sounds logical, there are a number of downsides.

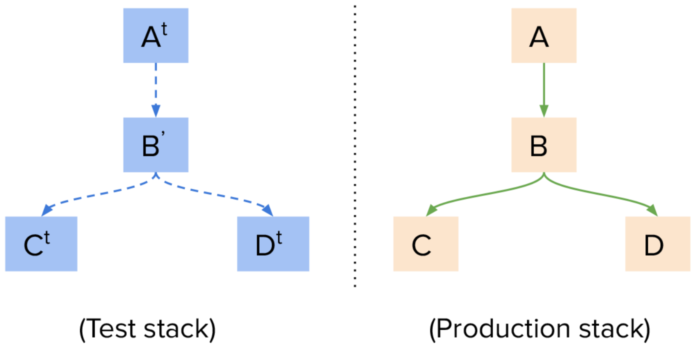

Figure 2: Parallel testing stack architecture

Operational Cost

Having to provision an entire stack along with all its data stores, message queues, and other infrastructure components means additional hardware and maintenance cost.

Synchronization Issues

The test stack is only useful if it is identical to the production stack. As the two stacks deviate from each other, the testing becomes far less effective. There is an additional burden on the infrastructure components to keep the stacks in sync. A lag is possible while the two stacks are being brought in sync, and this lag may degrade over a period of time.

Unreliable Testing

Since teams are going to deploy their experimental and potentially buggy code to the test stack, services may or may not be able to handle the traffic correctly, leading to frequently failing tests. For example, the team owning service A would trigger a test of their new code that fails due to a bug in service B. This would be hard to diagnose, and we couldn't ascertain changes to service A were safe until the test passes, which means we would be blocked until the team owning service B deployed clean code back to the test stack. This particular downside can be mitigated by having a routing framework to route traffic to yet another sandbox environment where the service-under-test is instantiated. This also requires the ability to tag traffic with additional information (e.g., the service-under-test, where it can be located, etc.).

Inaccurate Capacity Planning

To be able to assess the capacity of an entire stack or sub stack, we would have to push the test load on the test stack. If we want to test for a particular capacity that we want to achieve, we would have to increase the capacity of the test stack before we could apply the delta load (target capacity minus current production load) on to the test stack. This delta load may not be able to saturate the test stack, thus making it unclear as to how much more capacity we should add to the production stack to achieve the target capacity.

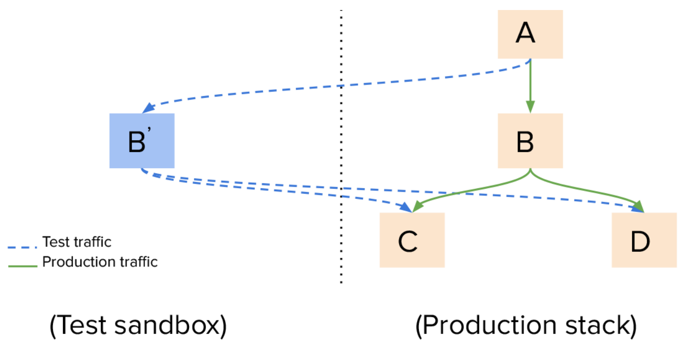

Figure 3: Testing in production

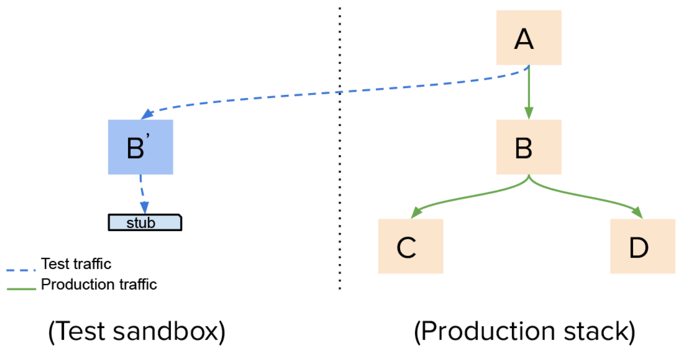

Figure 4: Shadow traffic routing to sandbox environment

Testing in Production

Another approach to integration testing in a microservice architecture would be to make the current production stack multi-tenant and allow both test as well as production traffic to flow through it. Figure 3 shows one such example. This rather ambitious approach does mean making sure every service in the stack is able to handle production requests alongside test requests.

In this approach, since service B is to be tested, the test build will be instantiated in an isolated sandbox area which is allowed to access production services C and D. The test traffic will be routed to B. Production traffic will flow as usual through the production instances.

Although this is a simplified view, it helps explain that multi-tenancy can help solve integration testing use cases. There are two basic requirements that emerge from testing in a production use case, which also form the basis of multi-tenant architecture:

- Traffic routing: being able to route traffic based on the kind of traffic flowing through the stack.
- Isolation: being able to reliably isolate resources between testing and production thereby ascertaining no side effect.

The isolation requirement here is particularly broad since we want all the possible data-at-rest to be isolated, including configuration, logs, metrics, storage (private or public), and message queues. This isolation requirement is not only for the service that is under test but for the entire stack. We will look at the details in the next section.

Multi-tenancy paves the way for other use cases beyond integration testing. We discuss some such use cases below.

Canary Deployments

When a developer makes a change to their service, even though the change is well reviewed and tested, we may not want to deploy the change to all the running instances of the service at once. This is to make sure the entire user base is not vulnerable should there be an issue or bug with the change being made. The idea is to roll out the change first to a smaller set of instances, with limited blast radius, called "canaries," monitor the canaries with a feedback loop, and then gradually roll them out widely.

A canary can be treated as yet another tenant in our multi-tenant architecture where the canary is a property of a request that can be used for making routing decisions and where resources are isolated for canary deployments. At any given time a service might have a canary deployed to which all the canary traffic will be routed. The decision to sample requests as canary can be made closer to the edge of the architecture based on attributes of the request itself: user type, product type, user location, etc.

Capture/Replay and Shadow Traffic

Being able to see how a change to a service would fare while serving actual production traffic is a great way of getting a strong signal on the safety of the change being made. Replaying already captured live traffic or replaying a shadow copy of live production traffic in a hermetically safe environment is another use case of multi-tenancy. Figure 4 shows an example of routing shadow traffic to a sandbox environment. In this we stub responses for any outbound calls made by the instance being tested. This can be treated as a subcategory of integration testing since these use cases are within the realm of testing and experimentation.

Replay traffic is technically test traffic and can be part of a test tenancy allowing for isolation from other tenancies. We do have the flexibility to assign a separate tenancy to allow further isolation from other test traffic. We discuss in later sections the implications of increasing the cardinality of tenancies and mitigation strategies.

Another important use case for a multi-tenant architecture is to protect and isolate multiple business-critical product lines or different tiers of the user base.

Tenancy-Oriented Architecture

In a tenancy-oriented microservice architecture, tenancy is a first-class citizen. The notion of tenancy is attached to both data-in-flight (e.g., requests, messages in the messaging queue) as well as data-at-rest (e.g., storage, persistent caches, configuration data, logs, metrics). In this section, we will look in a bit more detail at the aspect of making a microservice architecture multi-tenancy.

Tenancy Context

Since microservice architecture is a group of disparate services running on an interconnected network, we need the ability to attach a tenancy context to an execution sequence. As the request enters the system through an edge gateway, we would want to learn more about the tenancy of the request by attaching tenancy context to it. We want this context to stay with the request for the life of the request and get propagated to any new requests that are generated in the same business logic context.

Here is a simple tenancy context format and some examples:

```
{ "request-tenancy" : <product-code>/<tenancy-id>/<tenancy
-tags>... }

Examples:

"request-tenancy" : "product-foo/production"
"request-tenancy" : "product-bar/production/canary"
"request-tenancy" : "product-bar/production/health-probe"
"request-tenancy" : "product-foo/testing/TID1234"
"request-tenancy" : "product-bar/testing/shadow/SID5678"
```

Context Propagation

In general, when any service in the call chain receives a request, we want tenancy context to be available with it. The service may or may not make decisions based on the tenancy context as part of its business logic. However, it is required that the service propagates the context as it makes further requests as part of processing the same original incoming request. Most services may not need to look at the tenancy context, but some may optionally look into the request context to bypass some business logic. For example, an audit service verifying users' phone numbers may want to bypass the check for test traffic since the users involved in a test request would be test users. In the example of transaction processing services talking to a bank gateway to transfer funds for users, for test traffic, we would want to stub out the bank gateway or alternatively talk to the bank's staging gateway, if one is available for testing, to prevent any real transfer of money.

Tenancy context propagation can be achieved with open source tools like OpenTracing [1] and Jaeger [2]. These tools allow distributed context propagation in a language- and transport-agnostic way.

Tenancy context should also be propagated to other data-in-flight objects, like messages in a messaging queue like Kafka. Newer versions of Kafka support adding headers, and OpenTracing tools can be used to add context to messages flowing through Kafka. We will touch upon how we can achieve isolation for messaging systems like Kafka in a subsequent section.

Another set of objects that we would want tenancy context to be propagated to is data-at-rest. This includes all the data storage systems that are used by the services for storing their persistent data, like MySQL, Cassandra, AWS, etc. Distributed caches like Redis and Memcached can also be classified under data-at-rest. All the storage systems and caches that get used in the architecture need to be able to support the ability to store context along with the data at a reasonable granularity to allow retrieval and storage of data based on the tenancy context. At a high level the only requirement from the data-at-rest component is the ability to isolate data and traffic based on the tenancy.

Exactly how the data is isolated and how the tenancy context is stored along with the data is an implementation detail that is specific to the storage system. We will take another look at tenancy-based isolation in storage in the next section.

Tenancy-Based Routing

Once we have the ability to tag a request with tenancy, we can route requests based on its tenancy. Such routing is crucial for the testing use cases: testing in production, record/replay, and shadow traffic. Also, canary deployment requires the ability to route the canary requests to particular service instances running in the isolated canary environments.

It is important to consider the deployment and services tech stack for coming up with a routing solution that works seamlessly without overhead. Languages in which services are written as well as the transports and encoding they use to communicate with each other might need to be considered for providing a fleet-wide routing solution. Open source service mesh tools like Envoy [3] or Istio [4] are highly suited for providing tenancy-based routing that works agnostic to service language and the transport or encoding used.

Generically, the tenancy-based routing can be implemented either at ingress or at the egress of the service. At egress, the service discovery layer can help determine what service instance to talk to depending on the request's tenancy. Alternatively, the routing decision can be made at the ingress with the request rerouted to the correct instance, as shown in Figure 5.

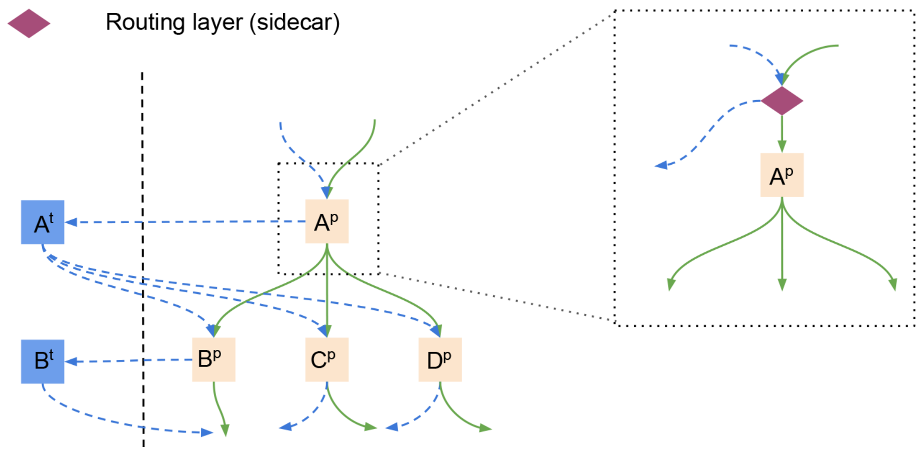

Figure 5: Tenancy-based router routing between test and production traffic

In this example, a *sidecar* can be used to forward the request to a test instance if the request tenancy is test. A sidecar can be a process acting as a proxy to all the traffic to the service and is co-located with the service. The traffic first is received by the service's sidecar where we are able to inspect the request's tenancy context and make a routing decision based on that context.

We do need additional metadata in the tenancy context depending on the use case we want to address. For example, for testing-in-production, we want to redirect test traffic to test instance of a service if the service is under test. We can add additional information in the context that will allow this behavior.

```
{
  "request-tenancy" : <product-code>/<tenancy-id>/
                      <tenancy-tags>...
  "services_under_test" : [
    "foo" : {
      "redirect" : <test instance Id>,
    },
    ...
  ]
}
```

When we are making routing decisions, we can check if the `request-tenancy` is test traffic and the request recipient is one of the `services_under_test`. If these conditions are satisfied, we route the request to the `<test instance Id>`.

Data Isolation

We want to get to an architecture where every infrastructure component understands tenancy and is able to isolate traffic based on tenancy. Typical infrastructure components that are used in a microservice architecture are: logging, metrics, storage, message queues, caches, and configuration. Isolating data based on tenancy requires dealing with the infrastructure components individually. For example, we might want to start emitting tenancy context as part of all the logs and metrics generated by a service. This helps developers to filter based on the tenancy, which might help avoid erroneous alerts or prevent heuristics or training data getting skewed.

Similarly for storage, underlying storage architecture needs to be taken into account to efficiently create isolation between tenants. Some storage architectures might lean more readily towards multi-tenancy than others. Two high-level approaches are either to embed the notion of tenancy explicitly alongside the data and co-locate data with different tenancies or to explicitly separate out data based on the tenancy. The latter approach provides better isolation guarantees, while the former might offer less operational overhead. For messaging queue systems like Kafka, we can either transparently roll out a new topic for the tenancy or dedicate a separate Kafka cluster altogether for that tenancy.

For data isolation, context needs to be propagated up to the infrastructure components. It is important to make sure services have minimal overhead with respect to data isolation. We would ideally want services to not deal with tenancy explicitly. We would also ideally want to place the isolation logic at a central choke point from which all the data flows through. The Edge Gateway is one such choke point where the isolation logic can be implemented and is the preferred approach. Client libraries can be another alternative to implement tenancy-based isolation, although coding language diversity makes it a bit harder to keep the logic in sync among all the language-specific client libraries.

Similarly for config isolation, we want the configuration data for a service to be tenancy-specific, making sure configuration change for one tenancy does not affect another.

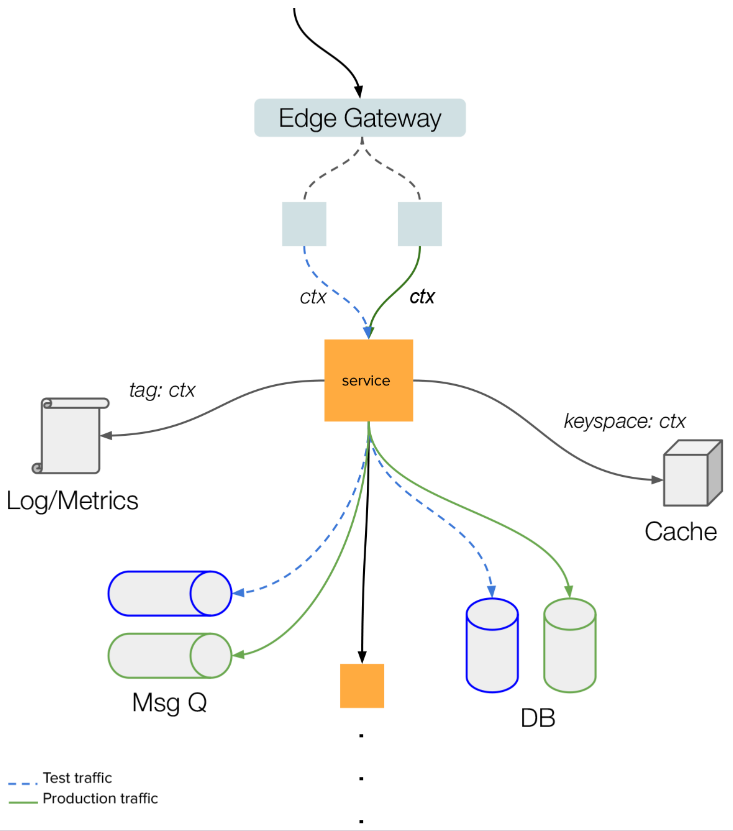

Figure 6: Data isolation for logs, metrics, storage, cache, and message queues

Conclusion

Microservice-based architectures are still evolving and are becoming instrumental in providing the agility that businesses and developers need. A carefully planned multi-tenant architecture can help realize ROI in terms of increased developer productivity and ability to support evolving lines of business.

References

[1] Open-Tracing: https://opentracing.io/.

[2] Jaeger: https://www.jaegertracing.io.

[3] Envoy: https://www.envoyproxy.io/.

[4] Istio: https://istio.io/.