11. Serverless SaaS: Patrones y Estrategias de Arquitectura

Consumir cómputo en un modelo completamente gestionado donde no existen servidores permite a los desarrolladores SaaS y arquitectos desplazar su enfoque, alejándose de la búsqueda de esquivas estrategias de escalado y optimización de costos.

La naturaleza centrada en funciones del cómputo serverless también puede influir en cómo se aborda el diseño e implementación de la arquitectura SaaS multi-tenant.

La compatibilidad entre SaaS y Serverless

Para muchas organizaciones, adoptar un modelo SaaS se trata fundamentalmente de lograr economías de escala que impulsen su crecimiento, eficiencia e innovación. En el centro de esta mentalidad está la necesidad subyacente de construir un entorno SaaS que alinee el perfil de actividad de los tenants con el consumo de recursos de infraestructura.

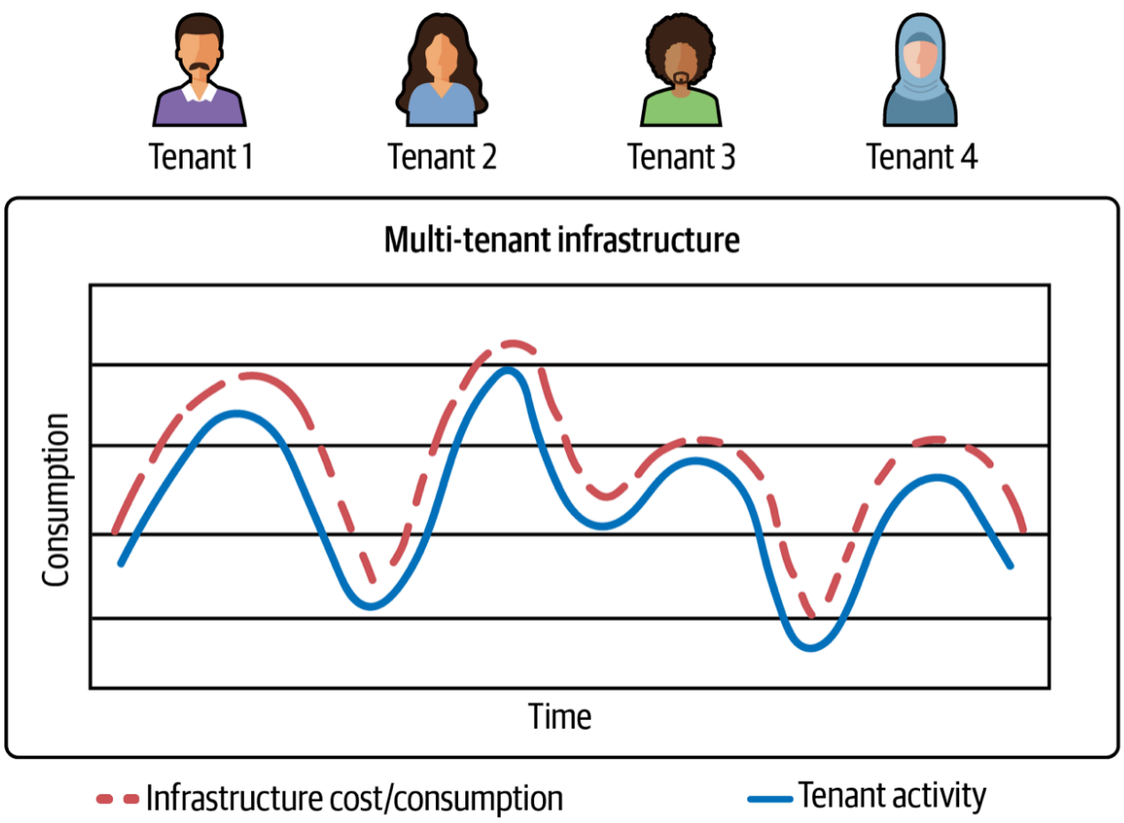

Figure 11-1. Serverless SaaS: aligning activity and consumption

Aunque puede ser difícil anticipar con precisión la actividad de los tenants, el objetivo claro sigue siendo minimizar el aprovisionamiento excesivo de recursos. En este ejemplo, el gráfico de consumo de infraestructura refleja la actividad de los tenants, proporcionando la infraestructura justa para satisfacer sus demandas en cualquier momento.

Se espera que el sistema responda de la manera más dinámica posible, optimizando los costos de infraestructura y habilitando las economías de escala esenciales para construir un negocio SaaS exitoso.

Aquí es donde se podría pensar en la elasticidad de la nube y el escalado horizontal como la respuesta a este problema. Sin embargo, la mayoría de las tecnologías de escalado horizontal se implementan mediante políticas de escalado que determinan cómo y cuándo debe escalar el entorno. Aquí es donde las cosas se complican.

Aunque el cómputo puede escalarse dinámicamente, alguien (usted) todavía tiene que definir cómo y cuándo el sistema necesitará escalar hacia arriba y hacia abajo. Hay que escribir y aplicar estas políticas de escalado y confiar en haber identificado una estrategia que sea tanto eficiente como confiable.

Si el entorno tiene cargas de trabajo relativamente predecibles, este enfoque puede funcionar bien. Sin embargo, en un entorno multi-tenant (como se discutió anteriormente), puede ser bastante difícil construir un conjunto de políticas que aborde universalmente la naturaleza impredecible de las cargas de trabajo de los tenants.

Esto generalmente lleva a escenarios en los que los equipos optan por aprovisionar recursos en exceso y adoptar políticas de escalado más pesimistas para limitar su exposición a problemas de vecino ruidoso (noisy neighbor), rendimiento y resiliencia.

> Con el cómputo serverless, como su nombre lo indica, se elimina por completo cualquier noción de servidores. El código simplemente es ejecutado por un servicio gestionado que asume la responsabilidad de proporcionar los recursos de cómputo que el sistema demanda.

Esto libera a los equipos SaaS para concentrar más tiempo en funcionalidades, eliminando gran parte del trabajo pesado que conlleva construir una estrategia de escalado efectiva y eficiente.

Con serverless, solo se paga por la invocación real de funciones individuales gestionadas. Si una función nunca es llamada, no genera ningún costo.

La naturaleza centrada en funciones del modelo de cómputo serverless también aporta valor adicional potencial que va más allá de la eficiencia. **Generalmente, al ser las funciones la unidad de despliegue, el entorno tendrá un modelo de despliegue mucho más granular.**

Esto permite publicar cambios y actualizaciones con un radio de impacto mucho menor. **Esto puede ser especialmente útil en un entorno multi-tenant donde se pone especial énfasis en lograr cero tiempo de inactividad.**

El modelo de cómputo serverless también puede abrir caminos más simples para atribuir el consumo a tenants individuales. Dado que cada función solo puede ser invocada y consumida por un tenant a la vez, resulta mucho más fácil atribuir el consumo de cómputo a tenants individuales.

Serverless se ha abierto camino en una lista creciente de servicios de infraestructura adicionales. Mensajería, analítica, almacenamiento y una variedad de otros servicios de infraestructura gestionados han comenzado a incorporar capacidades serverless en sus modelos de cómputo. Esto permite llevar la propuesta de valor de serverless a más capas de la arquitectura SaaS.

> Si bien este capítulo se centra en el diseño y construcción de soluciones con cómputo serverless, es importante reconocer que serverless puede no ser adecuado para todas las partes del sistema. Ciertas cargas de trabajo pueden seguir siendo más apropiadas para contenedores u otras tecnologías de cómputo.
> Por ejemplo, si existen partes del sistema que pueden y deben utilizar tareas de larga duración, podría optarse por un modelo de cómputo diferente para esos casos de uso.

Modelos de despliegue

Sin embargo, antes de profundizar, es necesario tener una comprensión común de cómo los microservicios de la aplicación se representarán en un entorno Lambda donde todo el código se escribe y despliega como funciones individuales.

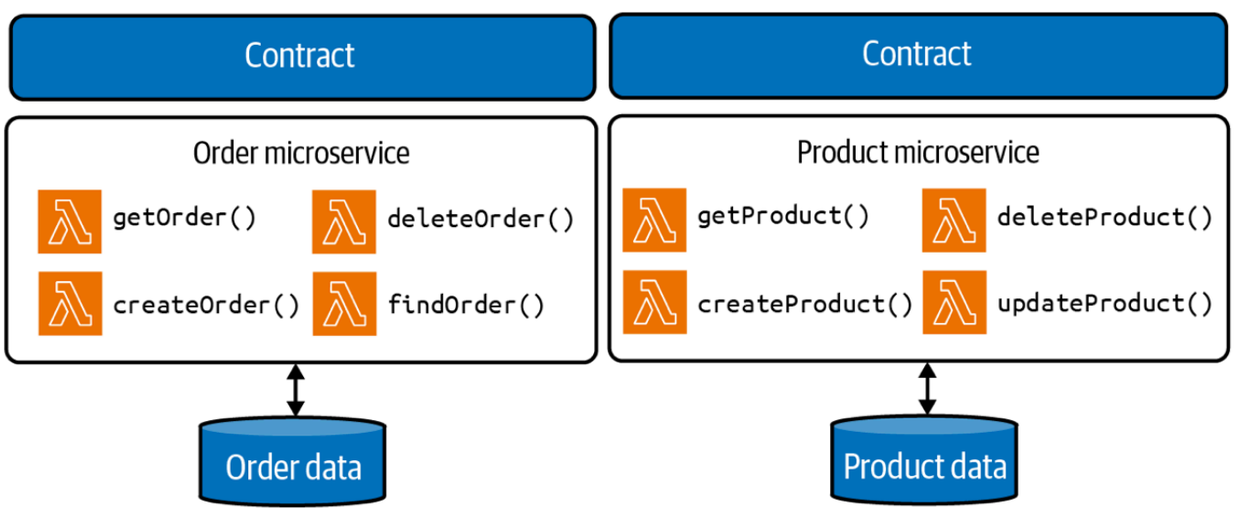

Figure 11-2. Logical microservices

Los servicios exponen un punto de entrada (típicamente una API) que representa su contrato con el sistema. La implementación subyacente de un servicio puede cambiar libremente siempre que no rompa este contrato.

Estos servicios frecuentemente referencian, encapsulan y son propietarios de recursos de almacenamiento. Estos son simplemente los principios básicos de los microservicios, donde se crean servicios autónomos que pueden construirse y desplegarse de forma independiente.

Ahora bien, esto se vuelve más interesante cuando se examina el interior de estos servicios. Cada una de las operaciones dentro de estos microservicios está asociada a una función Lambda separada que es responsable de implementar la funcionalidad para esa operación.

Al mismo tiempo, el servicio gestionado Lambda no tiene conocimiento real de las relaciones entre estas funciones. Por eso se suele hacer referencia a estos servicios como microservicios lógicos. Aunque Lambda no establece ningún vínculo entre estas funciones, los equipos las seguirán viendo como un conjunto agrupado de funciones mapeadas al contrato e implementación del microservicio general.

Los desarrolladores SaaS que trabajan en estos servicios generalmente lo hacen de forma colectiva. Los versionarán, desplegarán y probarán como una unidad. Esencialmente, se está tomando todo el sistema de valores que viene con el modelo de microservicios y ensamblando una visión de las funciones que es consistente con estos principios fundamentales.

Al comenzar a describir la firma de los despliegues, no serán simplemente funciones: se representarán como microservicios lógicos que se despliegan como un grupo de funciones que implementan el contrato/funcionalidad del microservicio.

Despliegues pooled y siloed

La idea de implementar despliegues siloed y pooled se ve un poco diferente cuando se utilizan funciones serverless. Con Lambda, realmente no existen mecanismos que permitan colocar funciones en grupos específicos (aparte de las etiquetas, que no se adaptan bien a la agrupación multi-tenant que se intenta crear).

Esto significa que los modelos de despliegue se implementan realmente desplegando agrupaciones de funciones separadas para los tenants y utilizando mecanismos de enrutamiento para conectar los tenants con sus funciones.

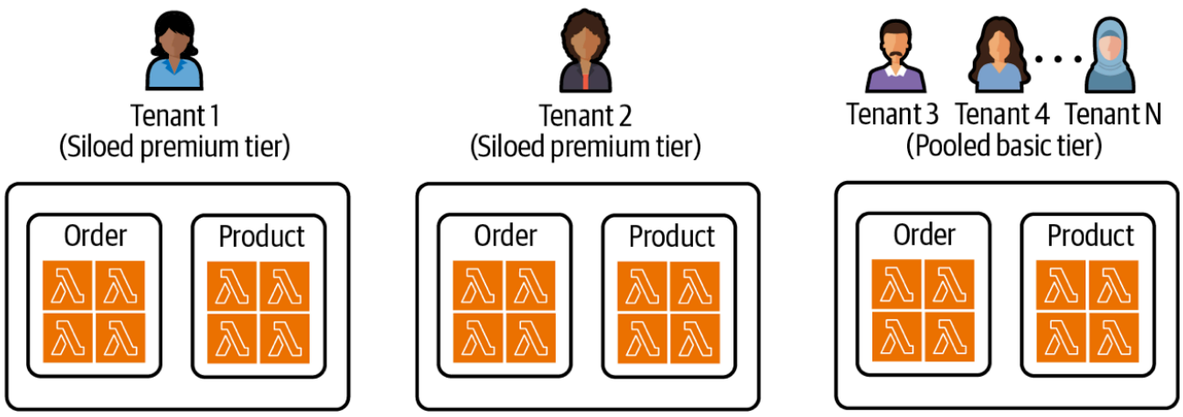

Figure 11-3. Supporting serverless siloed and pooled deployments

A la izquierda y en el centro del diagrama se encuentran dos tenants de nivel premium que tienen recursos de cómputo serverless siloed. Esto significa básicamente que se han aprovisionado y desplegado copias separadas de las funciones Order y Product para cada uno de estos tenants siloed.

Las funciones que se encuentran en estos silos están enteramente dedicadas a estos tenants de nivel premium. En el lado derecho se encuentran los tenants de nivel básico ejecutándose en un modelo pooled.

Al observar estos modelos de despliegue, podría surgir la pregunta de si soportar funciones Lambda siloed realmente aporta valor. Las funciones Lambda, por definición, nunca se comparten. Si un tenant invoca una función, el alcance y la vida de la llamada funcional estarán dedicados a ese único tenant. Si múltiples tenants llaman a esa misma función, Lambda añadirá más instancias de esa función para satisfacer la demanda.

Existen múltiples ventajas que aún pueden derivarse del despliegue de funciones siloed para tenants. El problema del vecino ruidoso (noisy neighbor) es ciertamente una parte importante de esta historia. **Aunque Lambda escala las funciones, aún tiene límites de concurrencia que pueden impactar cuántas ejecuciones simultáneas se permiten para una función.**

Si simplemente se tiene una función desplegada y compartida por todos los tenants, existe el potencial de exceder los límites de concurrencia de Lambda. Esto podría desencadenar throttling y conducir a condiciones de vecino ruidoso. **Al desplegar funciones separadas y dedicadas para los tenants siloed, se puede garantizar que solo un tenant invocará sus funciones.**

El aislamiento de funciones también puede influir en el modelo de aislamiento de tenants del entorno, permitiendo adjuntar políticas de aislamiento en el momento del despliegue. Esto puede simplificar cómo se aplica el aislamiento.

Despliegues en modo mixto

Como se ha visto a lo largo de la discusión sobre despliegues, aislar y agrupar recursos no tiene que ser una proposición de todo o nada. Con serverless, sin duda existen opciones para aislar selectivamente un subconjunto de funciones de tenant (microservicios) para abordar el vecino ruidoso, la estratificación (tiering), el aislamiento y otros requisitos.

Con serverless, esto simplemente significa que se puede adoptar un enfoque más granular para determinar cómo se despliegan las funciones.

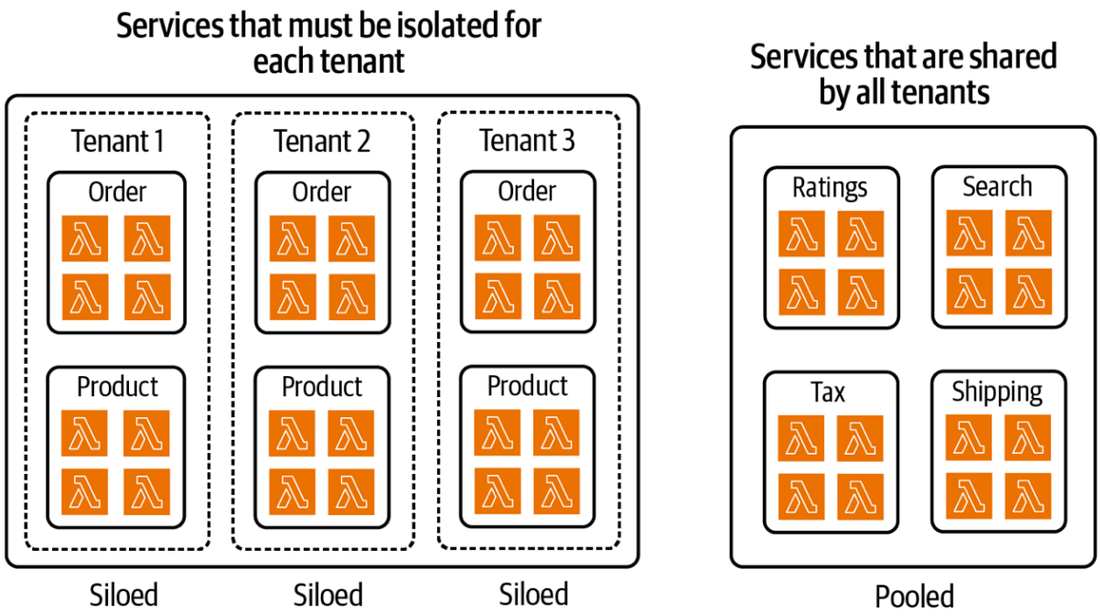

Figure 11-4. Serverless and mixed mode deployment

Sin embargo, serverless introduce un nuevo elemento. Independientemente de lo que esté siloed o pooled, por ejemplo, solo se paga por lo que se consume. También se requiere menos esfuerzo para determinar cómo tendrán que escalar estas funciones siloed y pooled. En su lugar, se puede apoyar más en el servicio Lambda para escalar el cómputo de manera eficiente.

La historia más simple de costos y escalado del cómputo serverless puede hacer esto más atractivo para algunos equipos. Como mínimo, serverless reduce algo de la fricción y los desafíos que pueden asociarse con el soporte de un modelo de despliegue en modo mixto.

Consideraciones adicionales de despliegue

Existen algunos matices de los modelos de despliegue serverless que podrían influir en el enfoque para seleccionar qué recursos de cómputo se aíslan (siloed) o agrupan (pooled). Para comprender las opciones disponibles, es necesario comenzar examinando el ciclo de vida de las funciones gestionadas por el servicio Lambda.

Cada vez que se invoca una función, Lambda tiene dos posibles caminos. Si se invoca una función por primera vez, Lambda necesitará crear la primera instancia de esa función. Luego, una vez completada la solicitud, una solicitud posterior puede reutilizar esa función. La idea es que Lambda obtiene eficiencia reutilizando las instancias que han sido ejecutadas recientemente.

Existen dos dimensiones específicas de este ciclo de vida en las que conviene enfocarse. La primera es el arranque en frío (cold start). Un arranque en frío describe la invocación de una función que no ha sido ejecutada recientemente. En estos casos, puede observarse algo de latencia adicional asociada al procesamiento de la solicitud. El impacto de esta latencia variará según la pila tecnológica utilizada, la naturaleza del código y las dependencias de la función, y otros factores.

- En un entorno pooled, el impacto de los arranques en frío probablemente sea insignificante, ya que habrá muchos tenants ejercitando el sistema, lo que debería limitar la frecuencia de los arranques en frío.
- Sin embargo, en un entorno siloed que solo es ejercitado por un único tenant, podrían presentarse más instancias en las que el arranque en frío impacte la experiencia del tenant.

El otro problema del ciclo de vida tiene que ver con los residuos de estado (state residue). Cada vez que Lambda procesa una invocación de función para un tenant, esa función se ejecutará exclusivamente para ese tenant. Lambda escalará para satisfacer las necesidades de múltiples tenants creando más instancias de una función dada. Aunque puedan estar ejecutándose múltiples copias de una función, cada invocación sigue mapeada a un único tenant.

Una vez que una función ha completado el procesamiento de una solicitud de tenant, el sistema puede reutilizar esa instancia para procesar una solicitud de otro tenant. En general, todo esto funciona bien y la reutilización de una función previamente ejecutada no debería causar problemas. **Sin embargo, si la implementación de la función de alguna manera retiene o referencia información de estado que no se libera al completarse, ese estado podría ser accedido por una solicitud de tenant posterior.**

Idealmente, el código no debería emplear ningún elemento que permita que el estado se transfiera de una solicitud a la siguiente. Sin embargo, dada la exposición potencial aquí, las funciones deben aprovechar políticas/bibliotecas que garanticen que el estado se limpie cuando terminen de ejecutarse.

Despliegue del control plane

En un entorno Lambda, las opciones están principalmente limitadas a los mecanismos de alto nivel y grano grueso que se utilizan para agrupar y aislar cualquier recurso en la nube.

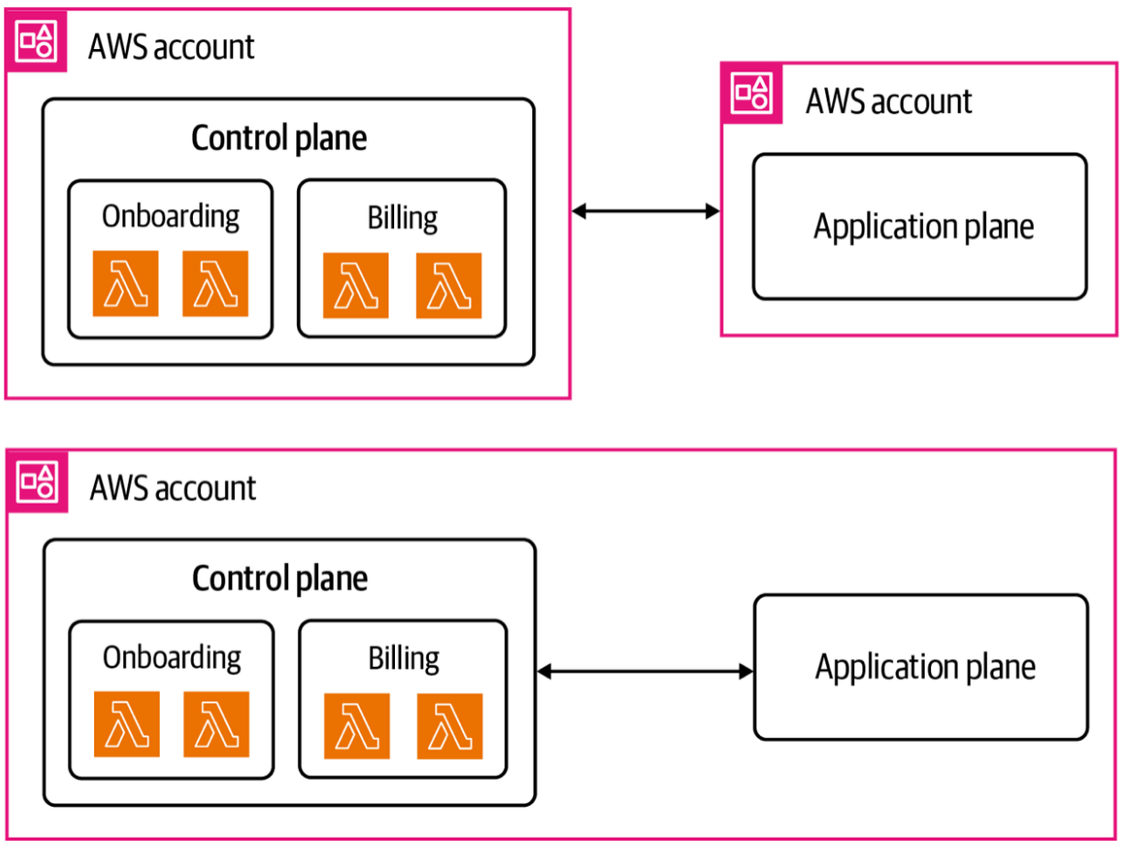

Figure 11-5. Deploying a serverless control plane

En general, las opciones se reducen a determinar qué cuenta de AWS alojará el control plane.

Implicaciones operativas

La idea de tener múltiples copias de las funciones desplegadas en estas configuraciones siloed y pooled puede ciertamente plantear preguntas sobre cómo esto impacta la huella de recursos operativa de la solución.

Con serverless, se pueden tener unidades de despliegue y gestión mucho más granulares. Por ejemplo, en un modelo de cómputo tradicional, la unidad de gestión y visibilidad operativa tiende a estar más a nivel del microservicio, donde el microservicio representa el compuesto de todas las operaciones que soporta ese servicio.

Con serverless, cada una de esas operaciones podría corresponder a funciones individuales. Si a esto se le añade la necesidad de soportar múltiples entornos de tenant, se puede imaginar cómo esto podría hacer crecer rápidamente la complejidad operativa del entorno.

Estos factores no sugieren que serverless sea una mala idea. Sin embargo, sugieren que podría ser necesario invertir más energía para llegar a una experiencia operativa que contemple esta visión más granular. **Se querrá que el operational telemetry permita enfocarse en las funciones individuales del sistema. Poder identificar problemas de salud, disponibilidad y escalabilidad significa tener perspectivas más ricas sobre cómo se están desempeñando estas funciones individuales (no solo los servicios).** Los mecanismos y herramientas están disponibles para que esto funcione, pero es algo que debe estar en el radar al diseñar el sistema.

Estrategias de enrutamiento

La mecánica para habilitar el modelo de enrutamiento serverless es relativamente sencilla. Sin embargo, existen diferentes patrones de enrutamiento que pueden adoptarse según las necesidades de la solución.

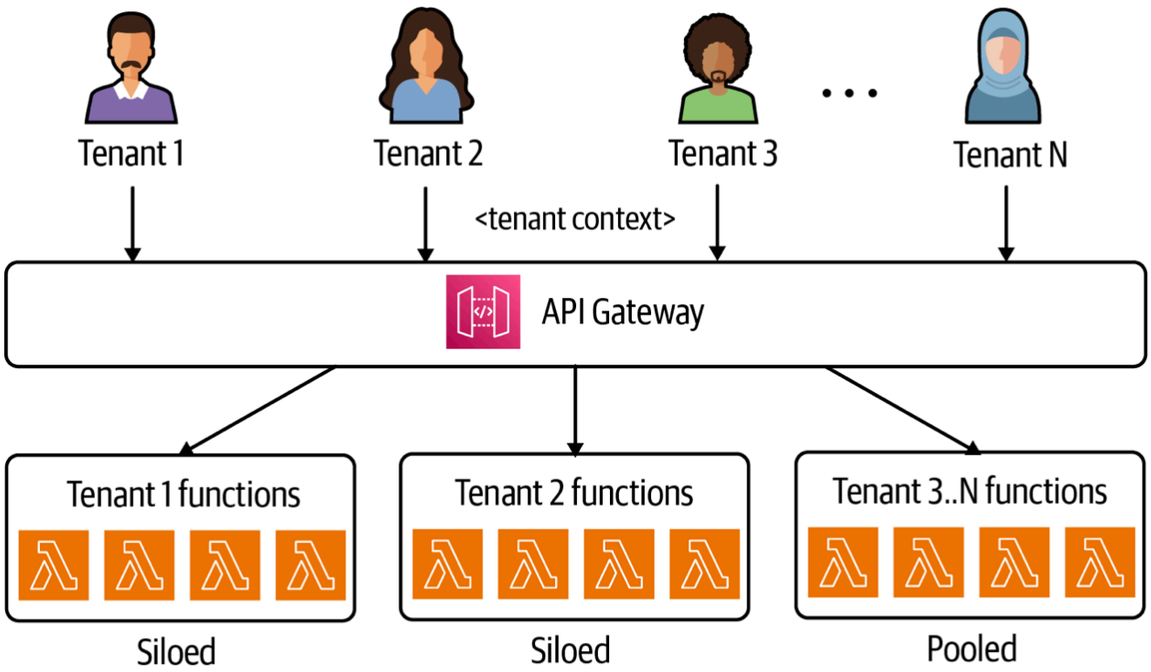

Figure 11-6. Routing to tenant deployments

Aquí, fue necesario crear tres copias separadas de estas funciones para soportar los requisitos de despliegue de los tenants. Ahora, a medida que las solicitudes ingresan al sistema, es necesario poder utilizar el contexto del tenant para enrutar estas solicitudes a la función Lambda apropiada. **En este ejemplo, se puede ver que este mapeo de funciones está definido por el API Gateway.**

Aunque es conveniente tener todo este enrutamiento resuelto a través de una única instancia del API Gateway, podría llegar un punto en que esto se vuelva inmanejable. El número de tenants que se necesita soportar, el número de rutas que se mapean: hay múltiples factores que podrían sugerir la necesidad de otro enfoque.

Una forma de evitar esto sería considerar el soporte de instancias separadas del API Gateway para cada uno de los despliegues de tenant.

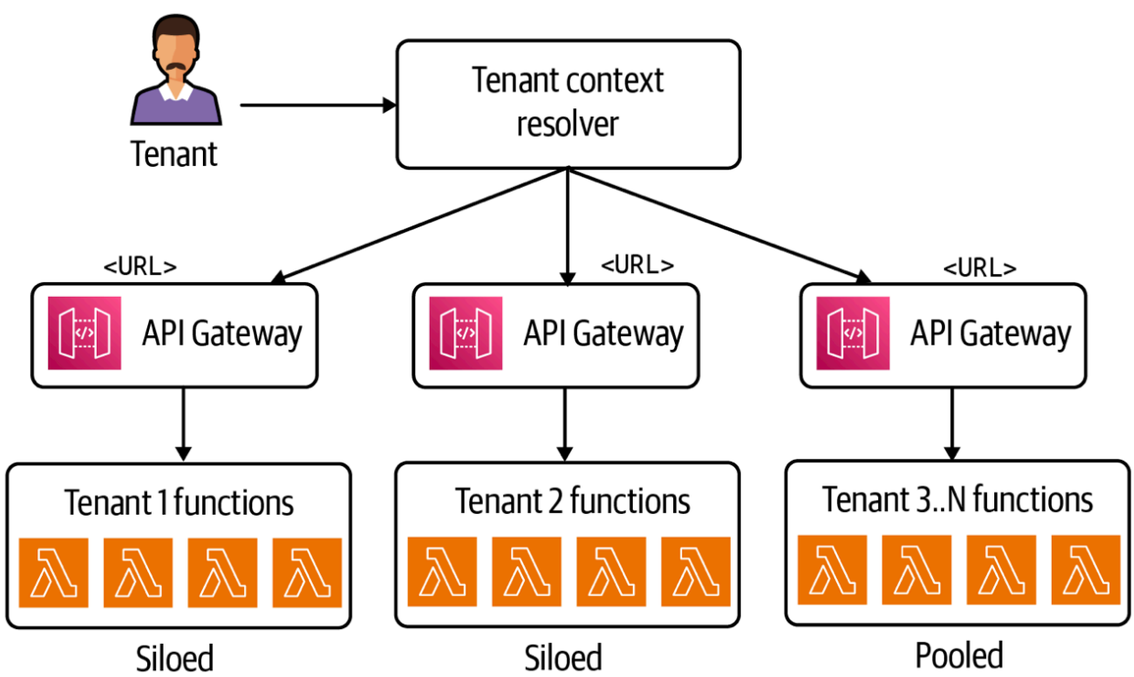

Figure 11-7. Separate API Gateway instances for each tenant

Esto significa que cada API Gateway solo será responsable de procesar y enrutar solicitudes a un único conjunto de funciones para un silo de tenant determinado o para los tenants pooled. Esto crea un vínculo lógico algo más estrecho entre cada API Gateway y las funciones individuales que pertenecen a cada despliegue. También habilita un control más granular sobre las políticas que podrían implementarse a nivel del API Gateway para cada despliegue.

Si bien este modelo tiene sus ventajas, requiere crear algún tipo de mapeo entre los tenants y su URL de API Gateway correspondiente.

El costo también debe ser un factor al elegir la estrategia de enrutamiento. Es cierto, por ejemplo, que podrían añadirse API Gateways separados para cada tenant. Esta puede ser una estrategia perfectamente razonable si se tiene un número pequeño

de tenants siloed con su propio API Gateway. **Sin embargo, si se intenta escalar esto a cientos o miles de tenants, podría impactar el costo, las operaciones, el despliegue y una serie de otras dimensiones del entorno SaaS.**

Onboarding y automatización del despliegue

La estrategia de onboarding, aprovisionamiento y despliegue para entornos serverless típicamente se apoya en las herramientas tradicionales que aprovisionan y configuran la infraestructura por tenant.

Si se trabaja con AWS, que es donde se centrará este análisis, esto generalmente se logra mediante una combinación de herramientas DevOps, incluyendo CDK, CloudFormation, Terraform, entre otras. Además de estas herramientas, también existe el Serverless Application Model (SAM), orientado específicamente a la experiencia de configuración y despliegue serverless.

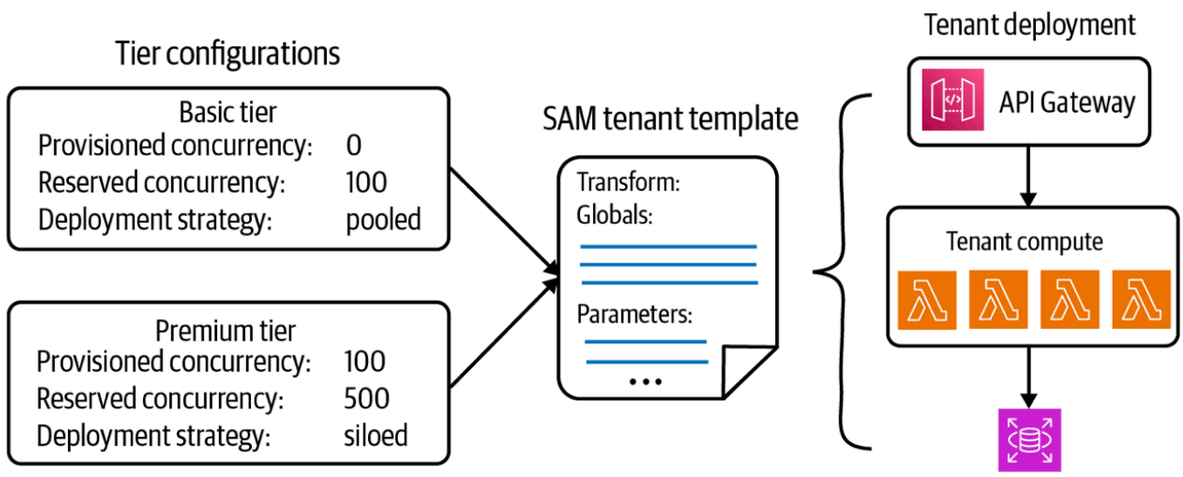

Figure 11-8. Defining serverless tiered environments

A la derecha del diagrama se puede ver un despliegue de tenant. Este es un marcador conceptual que representa la plantilla universal para la infraestructura y los recursos necesarios para soportar los despliegues de tenant (como parte del application plane).

Los tenants de nivel básico y premium tendrán cada uno despliegues que coincidan con esta arquitectura. Pueden estar configurados de manera diferente, pero comparten una huella de recursos común. La conclusión clave es que, al hacer el onboarding de tenants, será necesario aprovisionar un nuevo despliegue (nivel premium) o configurar tenants para agregarlos a un despliegue existente (nivel básico).

El aprovisionamiento y la configuración de estos trabajos serán gestionados por la plantilla SAM que se ve en el centro del diagrama. Esta plantilla base describe toda la infraestructura que se incluirá en cada despliegue de tenant.

A la izquierda, se puede ver dónde se han creado archivos de configuración de nivel separados que suministran todos los parámetros utilizados para definir las variaciones asociadas a cada nivel.

El ajuste de concurrencia aprovisionada (provisioned concurrency) se utiliza para controlar el número de entornos de ejecución preinicializados que se desean en el entorno Lambda (nivel).

- Para los tenants de nivel básico, se ha establecido en cero asumiendo que la actividad concurrente de múltiples tenants mantendrá la mayoría de las funciones activas (warm), reduciendo la necesidad de preinicializar funciones de nivel básico.
- Mientras tanto, para los tenants de nivel premium, se ha optado por usar algún nivel de concurrencia aprovisionada para superar los arranques en frío que podrían presentarse con mayor frecuencia en un entorno siloed.

En la Figura 11-9 se proporciona un ejemplo de cómo podrían incorporarse estos conceptos en el flujo de onboarding.

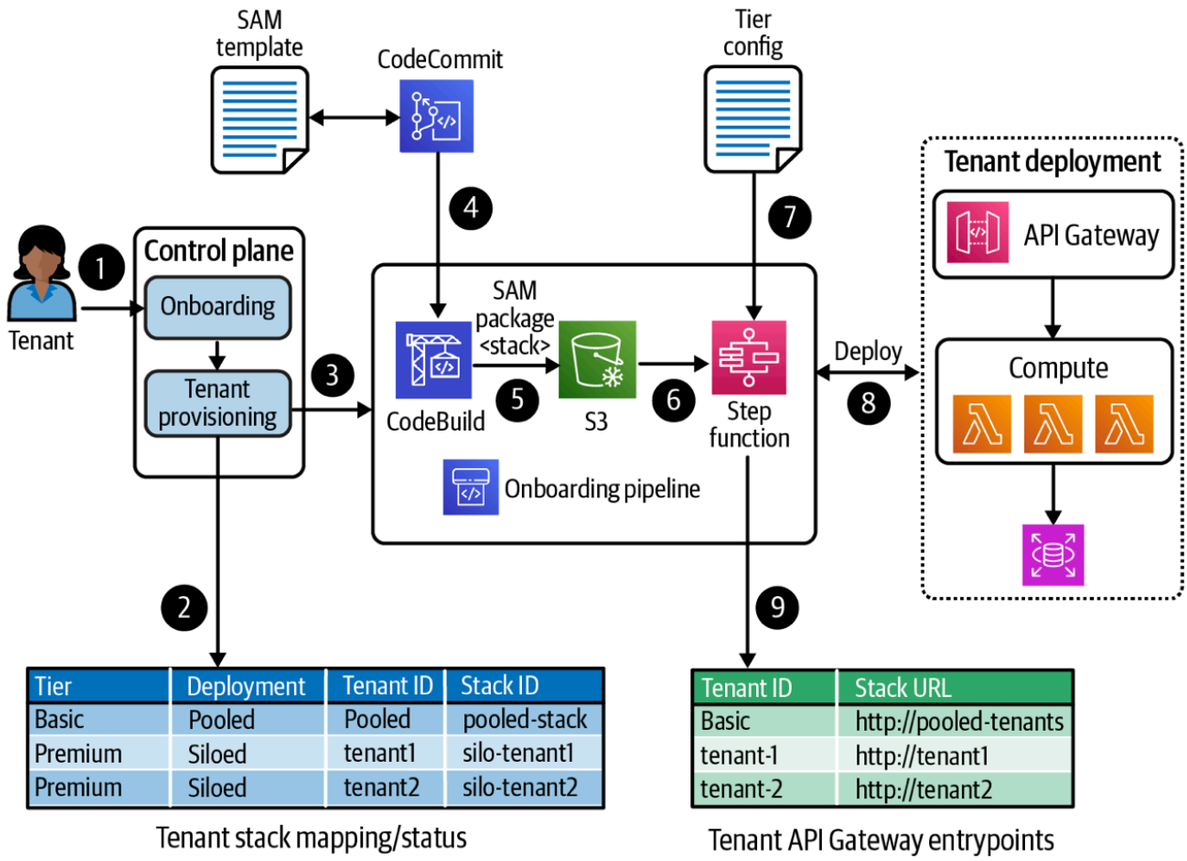

Figure 11-9. Onboarding orchestration

En la parte inferior izquierda del diagrama, se ha introducido una nueva tabla para soportar la experiencia de onboarding. Esta tabla es esencial para este proceso específico de onboarding serverless. Se utiliza para llevar registro de los diferentes despliegues de tenant que forman parte del entorno serverless.

1. El aprovisionamiento de tenants consulta esta tabla durante el onboarding (paso 2). Si el tenant que se está incorporando es un tenant de nivel básico y es el primero en ser añadido al entorno, el servicio de aprovisionamiento insertará una nueva fila en esta tabla (mostrada aquí como la primera fila).
2. Una vez creada esta entrada, el servicio de Tenant Provisioning invocará el pipeline de onboarding, que utiliza AWS CodePipeline para automatizar el flujo de onboarding (paso 3).
3. Este pipeline de código usa AWS CodeBuild para recuperar y procesar la plantilla SAM universal que describe los entornos de tenant. En este ejemplo, la plantilla se recupera de un repositorio de AWS CodeCommit (paso 4).
4. El proceso de construcción empaquetará la plantilla y la desplegará en un bucket de S3 para que pueda ser referenciada desde una ubicación estándar y accesible en adelante (paso 5).
5. El último paso de este proceso es ejecutar la plantilla SAM empaquetada. Esto se logra invocando una step function de Lambda (paso 6).
6. Esta step function recupera los ajustes de configuración de nivel que se discutieron anteriormente, enviándolos como parámetros en una solicitud de despliegue SAM que referencia la plantilla empaquetada en S3 (paso 7).

Para esta solución, se ha optado por usar un API Gateway separado para cada despliegue. Para que esto funcione, será necesario llevar registro de qué URL de API Gateway mapea a cada tenant o nivel. Estos datos se utilizarán para enrutar las solicitudes de tenant a la función de cada tenant.

Cuando el proceso se ejecute para el siguiente tenant de nivel básico, el servicio de Tenant Provisioning verá que ya existe una entrada de nivel básico en la tabla de mapeo del stack de tenants. Por lo tanto, en lugar de redesplegar la infraestructura nuevamente, solo introducirá las entradas de configuración incrementales necesarias para este nuevo tenant.

Onboarding de tenant premium siloed

La mayor parte del proceso de extremo a extremo es prácticamente igual. La diferencia clave aquí es que un tenant siloed terminará con su propia entrada única en la tabla de mapeo del stack de tenants.

La otra pieza del rompecabezas es el despliegue de actualizaciones. Una vez que estos entornos estén en funcionamiento, aún se necesitará alguna forma de publicar cambios en la arquitectura serverless que tenga conciencia de los diferentes niveles y modelos de despliegue.

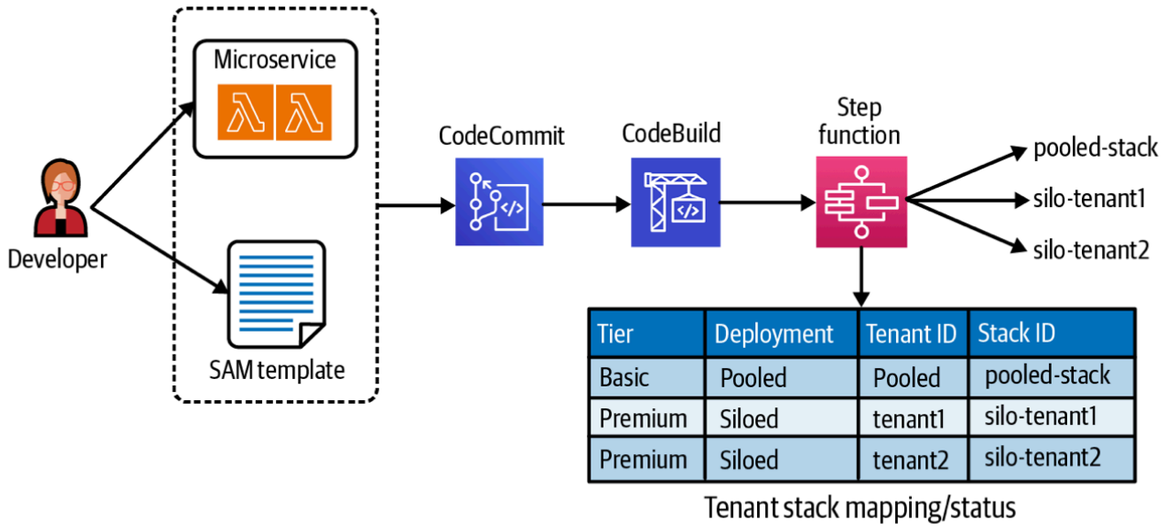

Figure 11-10. Applying tier-aware updates

A la izquierda se puede ver un desarrollador que está introduciendo un nuevo microservicio en un entorno donde ya se han desplegado varios tenants con diferentes niveles.

También será necesario actualizar la plantilla SAM para reflejar la presencia del nuevo microservicio. Ambos se muestran siendo incorporados a un repositorio de CodeCommit.

A partir de aquí, se usará CodeBuild para empaquetar la plantilla actualizada. Luego, la step function iterará sobre todas las entradas en la tabla de mapeo del stack de tenants para aplicar esta plantilla actualizada a cada uno de los entornos de tenant.

No existen elementos integrados ni herramientas que puedan soportar directamente esta necesidad de llevar registro de los stacks de tenant y aplicar las actualizaciones a todos los diferentes entornos. ¿Es una step function y una tabla la forma correcta de implementar esto? Quizás.

También vale la pena señalar que este mismo mecanismo podría utilizarse para escalonar el despliegue de correcciones o nuevas funcionalidades. Podría ampliarse esta tabla de mapeo del stack de tenants, añadiendo indicadores adicionales que señalen cómo y cuándo se aplicarían las actualizaciones a los tenants. **Esto podría convertirse en parte de una estrategia de despliegue canary o por oleadas (wave).**

Aislamiento de tenant

En entornos serverless, las estrategias de aislamiento pueden aplicarse en múltiples capas de la arquitectura multi-tenant. Por ejemplo,

- se pueden introducir políticas de aislamiento en la capa del API Gateway, observando las solicitudes entrantes de tenants y controlando las funciones y operaciones que puede invocar cada tenant.
- También existen oportunidades para adjuntar políticas de aislamiento directamente a las funciones.

Aislamiento pooled con inyección dinámica

Aislar los recursos de tenant en un entorno pooled es siempre más desafiante. Generalmente, con cualquier modelo pooled, será necesario aprovechar alguna forma de políticas aplicadas en tiempo de ejecución como parte del modelo de aislamiento.

Con políticas en tiempo de ejecución, los desarrolladores SaaS necesitarán introducir fragmentos de código que apliquen las políticas de aislamiento a cada solicitud de tenant.

Una forma de abordar este problema es mediante la inyección de credenciales de aislamiento. Esta estrategia mueve la mayor parte de los componentes clave de la implementación de aislamiento pooled hacia el API Gateway como un paso de preprocesamiento que se aplica a cada solicitud entrante.

Con la inyección, el objetivo es generar las credenciales de aislamiento antes de que la solicitud llegue al microservicio Order. El microservicio simplemente recibiría las credenciales y las aplicaría a sus interacciones con la tabla Order.

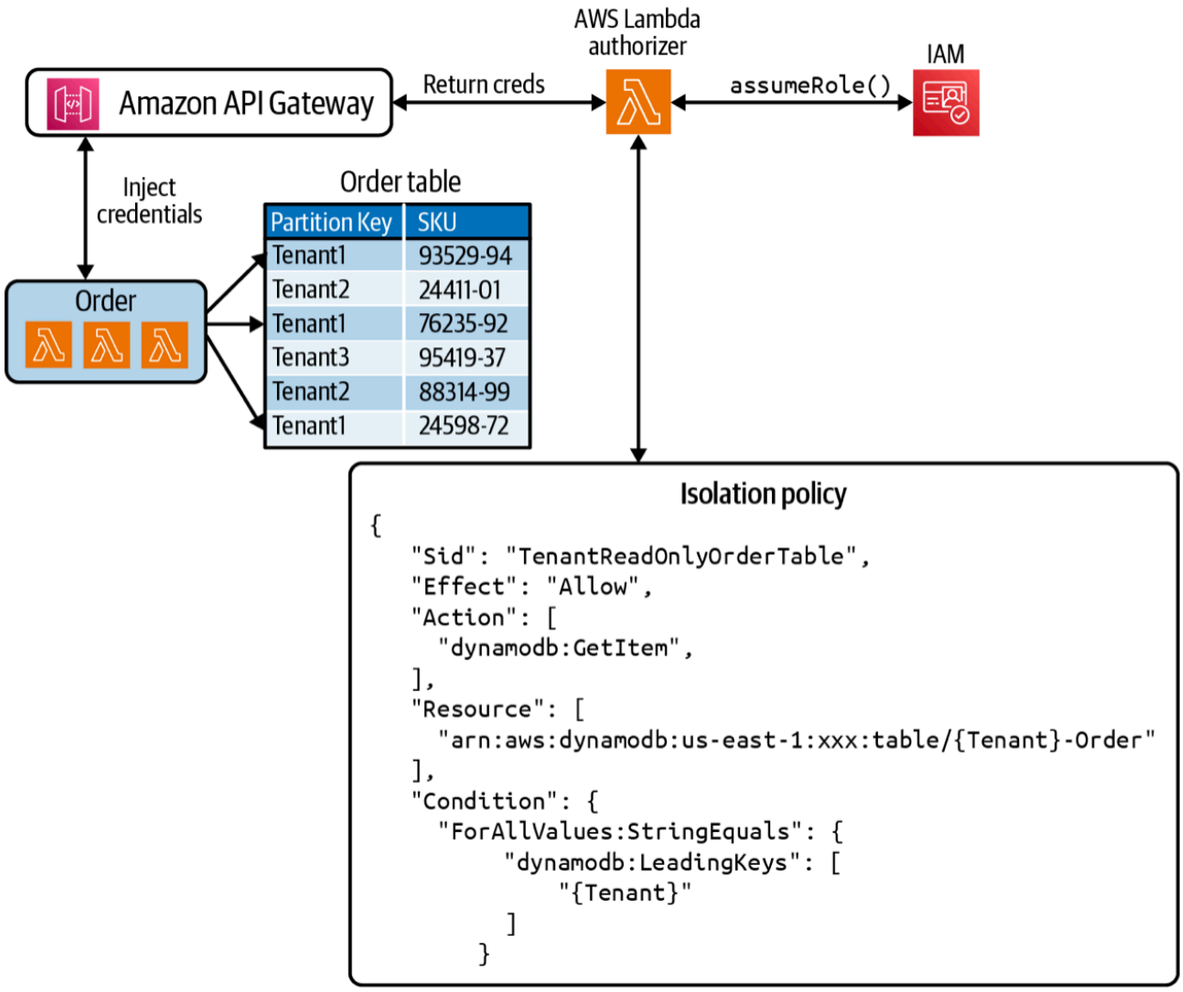

Figure 11-11. Serverless isolation with credential injection

Todas las partes móviles de este mecanismo de inyección se implementan en la capa del API Gateway de la arquitectura.

Esta función autorizadora extrae el contexto del tenant de la solicitud, determina la naturaleza de la operación que se está realizando e identifica la política de aislamiento que se utilizará para delimitar el acceso.

Las credenciales devueltas por este proceso se inyectan en el encabezado de la solicitud que se envía al microservicio. El servicio aún tendrá cierta responsabilidad en la aplicación de estas credenciales. **En este ejemplo, las credenciales se utilizarían al inicializar el cliente de base de datos (DynamoDB) y se aplicarían a cada solicitud que intente acceder a los datos en la tabla Order.**

Este enfoque también crea la oportunidad de almacenar en caché las credenciales en el API Gateway, ayudando a los equipos a superar algo de la latencia y la sobrecarga que conlleva adquirir credenciales para cada solicitud de tenant.

Este enfoque particular de aislamiento aleja las políticas del microservicio. Ahora se gestionan y procesan de forma centralizada a nivel del API Gateway.

Algunos equipos prefieren que estas políticas de aislamiento sean propiedad, versionadas y gestionadas por cada microservicio, especialmente porque las políticas suelen estar estrechamente conectadas a los microservicios individuales.

Ciertamente podría argumentarse que las políticas deberían estar encapsuladas por el servicio y verse como parte de su implementación subyacente. Con este enfoque, la implementación haría fluir el contexto del tenant hacia las funciones serverless, y cada función incluiría el código necesario para adquirir las credenciales con alcance de tenant.

Aislamiento en tiempo de despliegue

Aplicar aislamiento a funciones siloed es una historia mucho más sencilla. Si se ha optado por un modelo siloed, esto significa que estos tenants siloed tendrán funciones dedicadas que solo pueden ser ejecutadas por un tenant.

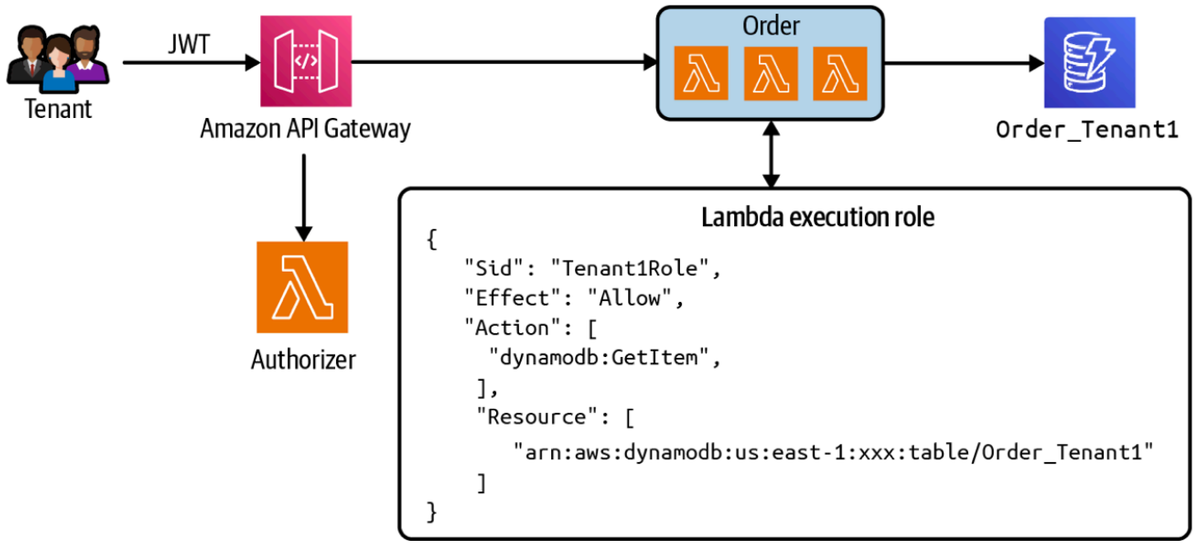

Figure 11-12. Deployment-time siloed isolation with Lambda functions

Cuando el proceso de aprovisionamiento crea este microservicio, adjunta un rol de ejecución Lambda a cada una de las funciones en ese microservicio.

Esta política restringe el acceso de esta función a la tabla Order_Tenant1. Cualquier intento de acceder a la tabla de otro tenant será denegado.

Ahora todo sucede durante el despliegue y, durante la vida de estas funciones desplegadas, el microservicio está restringido a la tabla de órdenes del Tenant 1. Esto es más sencillo de construir y tiene menor sobrecarga en tiempo de ejecución. Otra ventaja es que las políticas de aislamiento pueden delimitarse a nivel de función, lo que significa que pueden ser más granulares y enfocarse exclusivamente en implementar las necesidades de aislamiento de funciones individuales.

Soporte simultáneo de aislamiento silo y pool

El desafío es que cada nivel podría emplear diferentes esquemas de aislamiento. Esto significa que el código común en las funciones necesitaría soportar contextualmente diferentes enfoques para acceder a los datos y aplicar políticas de aislamiento.

```
def __get_dynamodb_table(event, dynamodb):
  if (is_pooled_deploy=='true'):
    accesskey = event['requestContext']['authorizer']['accesske
    secretkey = event['requestContext']['authorizer']['secretke
    sessiontoken =
      event['requestContext']['authorizer']['sessiontoken']
    dynamodb = boto3.resource('dynamodb',
      aws_access_key_id=accesskey,
      aws_secret_access_key=secretkey,
      aws_session_token=sessiontoken
    )
  else:
    if not dynamodb:
      dynamodb = boto3.resource('dynamodb')
    return dynamodb.Table(table_name)
```

Si se trata de un tenant pooled, el cliente de base de datos (DynamoDB) deberá inicializarse con las credenciales inyectadas por el API Gateway. Sin embargo, si se trata de un tenant de nivel premium (siloed), no es necesario utilizar estas credenciales inyectadas.

Aislamiento basado en rutas

Se pueden y deben utilizarse los modelos de aislamiento en tiempo de despliegue y en tiempo de ejecución descritos anteriormente. Al mismo tiempo, también pueden introducirse protecciones más tradicionales a nivel del API Gateway de la arquitectura SaaS serverless.

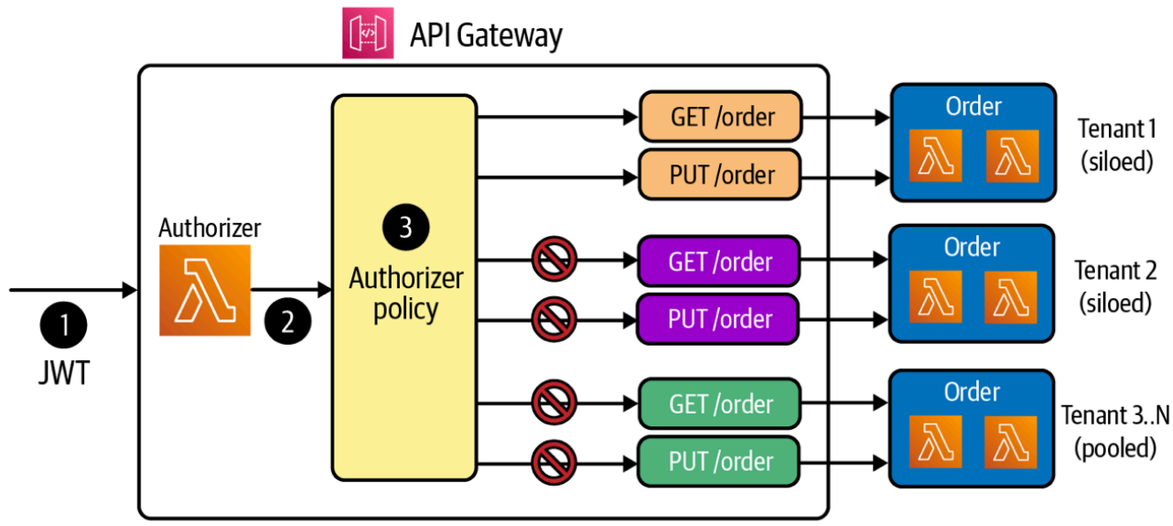

Figure 11-13. Controlling access at the API Gateway level

En este ejemplo, se muestran algunos de los diferentes mecanismos que pueden utilizarse a nivel del API Gateway de la arquitectura. Si se trabaja el diagrama de izquierda a derecha,

- se puede ver que esto comienza con la solicitud entrante que incluye un JWT con contexto de tenant (paso 1).
- Este JWT entra al API Gateway y es procesado por un autorizador. Este autorizador extraerá el contexto del tenant del JWT y utilizará este contexto para configurar una política de autorizador (paso 2).
- Esta política puede configurar el comportamiento y habilitar las rutas del API Gateway.
- Cuando llega una solicitud del Tenant 1, se desea garantizar que esta solicitud solo se enrute a funciones válidas del Tenant 1. Aquí es donde la política del autorizador está configurada para bloquear el acceso a las rutas que acceden a los paths que pertenecen a otros tenants (paso 3).

También podría considerarse aplicar otra variación de este modelo a entornos serverless donde se tienen instancias separadas del API Gateway para cada uno de los grupos de funciones (despliegues siloed y pooled).

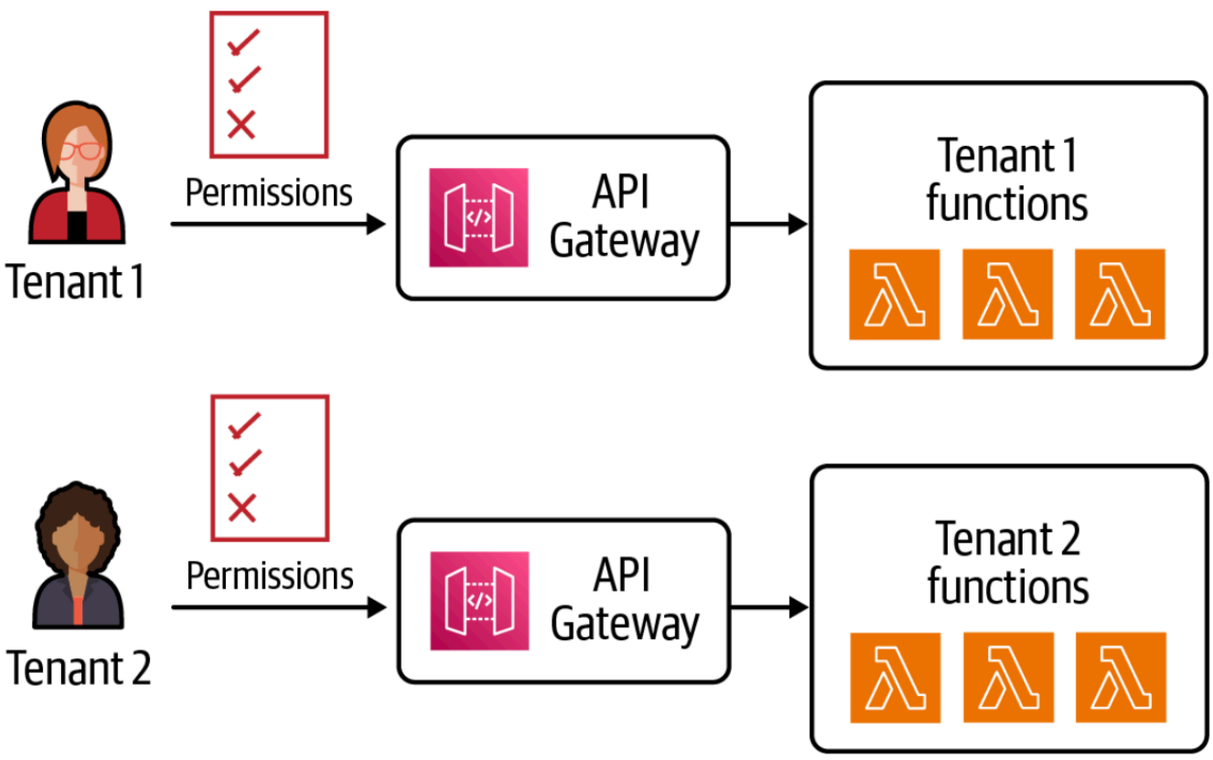

Figure 11-14. Limiting access via separate API Gateways

Con este modelo, pueden aplicarse políticas de aislamiento directamente al API Gateway, adjuntando políticas específicas por tenant para prevenir el acceso entre tenants. Es importante señalar que querrá aplicar el API Gateway por tenant con cierta precaución.

Concurrencia y vecino ruidoso

Podría ser tentador asumir que la naturaleza gestionada de las funciones Lambda significa que no hay necesidad de preocuparse por que los tenants saturen las funciones o creen condiciones de vecino ruidoso.

Para entender mejor cómo Lambda aborda este tema, se debe comenzar por observar cómo Lambda escala sus funciones.

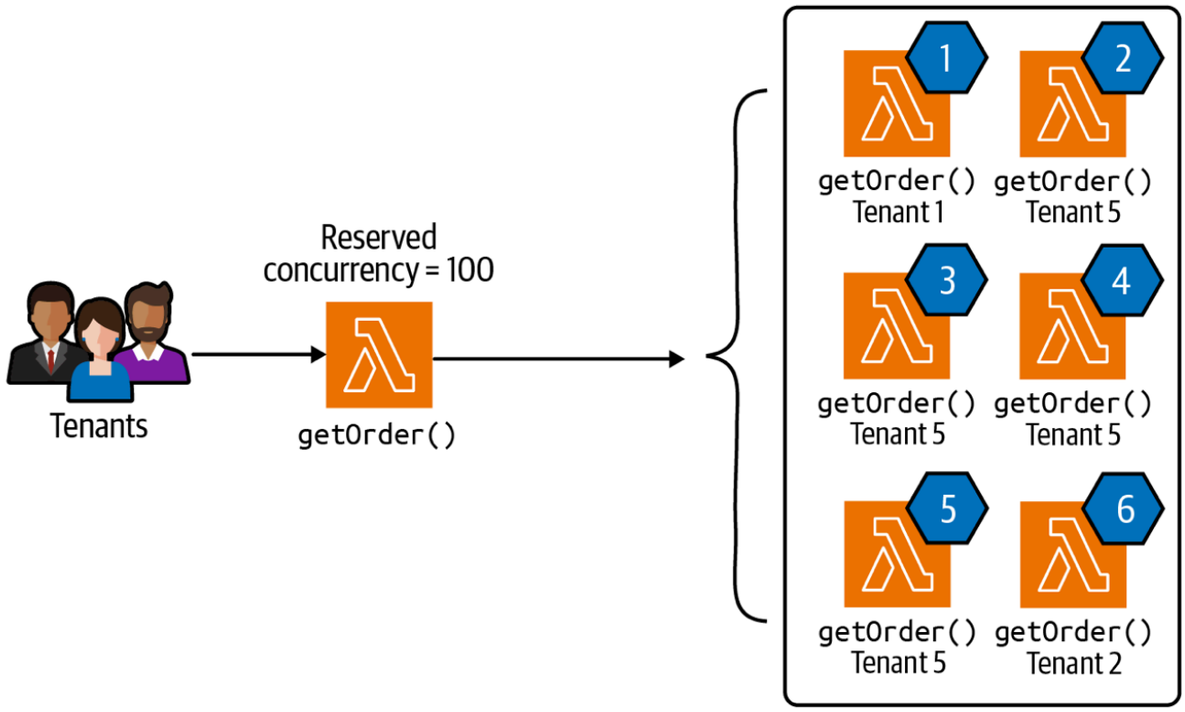

Figure 11-15. Managing the concurrency and scale of serverless functions

A la izquierda del diagrama, se puede ver un grupo de tenants que están consumiendo la función getOrder(). Cada vez que se realiza una solicitud a esta función, Lambda ejecutará una instancia única de esa función. Esto significa que, en cualquier momento dado, podría haber múltiples instancias de esta función en ejecución.

Dado que Lambda no puede escalar infinitamente el número de instancias de forma concurrente, es necesario pensar en cómo limitar el número de instancias concurrentes que se permitirían para esta función.

Aquí es donde entra en juego el concepto de concurrencia reservada (reserved concurrency) de Lambda. En este ejemplo, esa concurrencia reservada se ha establecido en 100, indicando que no puede haber más de 100 instancias concurrentes de esa función ejecutándose en un momento dado.

Este mismo mecanismo puede utilizarse para dar forma a la estrategia de estratificación (tiering) de la aplicación. Por ejemplo, podrían asignarse diferentes configuraciones de concurrencia reservada a cada uno de los despliegues de funciones por nivel en el entorno multi-tenant.

Al diseñar la arquitectura SaaS serverless, se debe desarrollar una estrategia general de concurrencia para determinar cómo asignar mejor la concurrencia entre las funciones que forman parte del sistema.

Más allá del cómputo serverless

En realidad, el alcance del tema serverless es mucho más amplio que Lambda, extendiéndose a una amplia gama de servicios que forman parte del stack de AWS. Almacenamiento, mensajería, analítica y una variedad de otros servicios del stack de AWS han estado añadiendo activamente soporte para funcionalidades serverless.

Tradicionalmente, muchos de los servicios que se ejecutan en AWS han requerido que los desarrolladores SaaS seleccionen y dimensionen los recursos de cómputo para la instancia particular de ese servicio.

Algunas bases de datos, por ejemplo, requieren predeterminar la huella de recursos de cómputo de la base de datos. Esto típicamente llevaría a los equipos a aprovisionar en exceso su base de datos para garantizar que pueda satisfacer los patrones cambiantes de consumo de base de datos de sus tenants.

Con una opción serverless, estos mismos servicios pueden reducir la necesidad de vincularse a un tamaño o perfil de cómputo específico. En su lugar, el cómputo se convierte en una capa gestionada del servicio, escalando y dimensionando según las cargas de trabajo reales que se colocan sobre el servicio.

El objetivo es llevar el valor del modelo serverless a una gama más amplia de servicios, permitiendo llevar las ventajas de serverless a más dimensiones de la arquitectura SaaS.