Design patterns for Containers

Purpose

The purpose of this document is to describe container design patterns in order to illuminate best practices for cloud implementations, which have been compiled by experts.

Scope

Abstract design patterns are in scope for this document; specific applications and implementations are out of scope.

Requirements

Availability patterns

**Geode pattern**

*Features and Advantages*

* Deploy a collection of backend services into a set of **ge**ographical n**ode**s, each of which can service any request for any client in any region.
* This pattern allows serving requests in an *active-active* style, improving latency and increasing availability by distributing request processing around the globe.
* Routes traffic via the shortest path to a nearby geode, which improves latency and performance.
* Each geode is behind a global load balancer.
* Uses a geo-replicated read-write service (such as Azure Cosmos DB) to host the data plane, ensuring cross-geode data consistency.
* Data replication services ensure that data stores are identical across geodes, so *all* requests can be served from *all* geodes.
* Geodes consist of a collection of disparate types of resources, often defined in a template.
* Geodes have no dependencies outside of the geode footprint and are self-contained. No geode is dependent on another to operate, and if one dies, the others continue to operate.
* Geodes are loosely coupled via an edge network and replication backplane. (For example, [Azure Traffic Manager](https://docs.microsoft.com/azure/traffic-manager/traffic-manager-overview) or [Azure Front Door](https://docs.microsoft.com/azure/frontdoor/front-door-overview) could be used for fronting the geodes, while Azure Cosmos DB could act as the replication backplane.) Geodes are not the same as clusters because they share a replication backplane, so the platform takes care of quorum issues.

*Issues and Considerations*

* Use modern DevOps practices and tools to produce and rapidly deploy identical geodes across a large number of regions or instances.
* Use autoscaling to scale out compute and database throughput instances within a geode. Each geode individually scales out, within the common backplane constraints.
* Use a front-end service (like Azure Front Door) that performs dynamic content acceleration, split TCP, and Anycast routing.
* Use a replicating data store (like Azure Cosmos DB) to control data consistency.
* Use serverless technologies where possible, to reduce always-on deployment cost, especially when load is frequently rebalanced around the globe. This strategy allows for many geodes to be deployed with minimal additional investment. Serverless and consumption-based billing technologies reduce waste and cost from duplicate geo-distributed deployments.
* Choose whether to process data locally in each region, or to distribute aggregations in a single geode and replicate the result across the globe. (For example, the [Azure Cosmos DB change feed processor](https://docs.microsoft.com/azure/cosmos-db/change-feed-processor) offers this granular control using its *lease container* concept, and the *leasecollectionprefix* in the corresponding [Azure Functions binding](https://docs.microsoft.com/azure/cosmos-db/change-feed-functions).) Each approach has distinct advantages and drawbacks.
* Geodes can work in tandem, (for example, by using the Azure Cosmos DB change feed and a real-time communication platform like SignalR). Geodes can communicate with remote users via other geodes in a mesh pattern, without knowing or caring where the remote user is located.
* This design pattern implicitly decouples everything, resulting in an ultra-highly distributed and decoupled architecture. Decoupling is a good thing, but consider how to track different components of the same request, as they might execute asynchronously on different instances. Implement a good monitoring strategy.

*When to Use this Pattern*

* To implement a high-scale platform that has users distributed over a wide area.
* For any service that requires extreme availability and resilience characteristics, because services based on the geode pattern can survive the loss of multiple service regions at the same time.

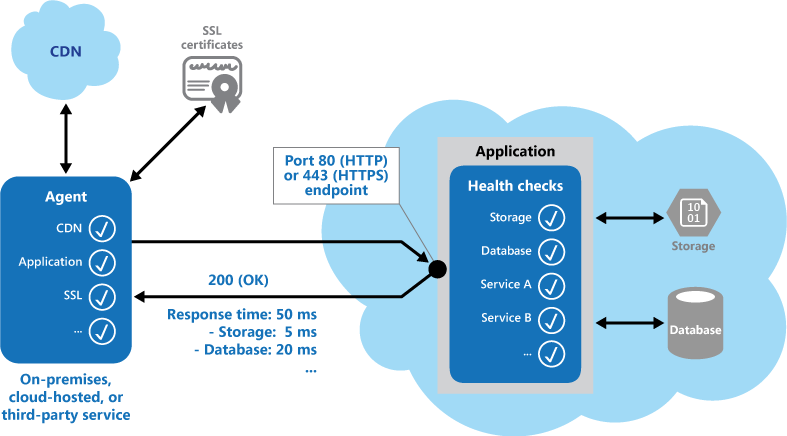
*When Not to Use this Pattern*

* Consider not using this pattern for architectures that have constraints, such that all geodes cannot be equal for data storage. For example, there may be data residency requirements, or there may be an application that needs to maintain temporary state for a particular session, or there may be a heavy weighting of requests towards a single region. In this case, consider using [deployment stamps](https://docs.microsoft.com/en-us/azure/architecture/patterns/deployment-stamp) in combination with a global routing plane that is aware of where a user’s data sits, such as the traffic routing component described within the [deployment stamps pattern](https://docs.microsoft.com/en-us/azure/architecture/patterns/deployment-stamp).
* Do not use this pattern in situations where there's no geographical distribution required. Instead, consider availability zones and paired regions for clustering.
* Do not use this pattern in situations where a legacy platform needs to be retrofitted. This pattern works for cloud-native development only, and can be difficult to retrofit.
* Do not use this pattern to implement simple architectures and requirements, where geo-redundancy and geo-distribution are neither required, nor advantageous.

**Health Endpoint Monitoring pattern**

*Features and Advantages*

* Implement health monitoring by sending requests to an endpoint on the application. The application should perform the necessary checks, and return an indication of its status.



* Health monitoring code in the application could check cloud storage or a database for availability and response time.
* Health monitoring code in the application could check other resources or services located in the application, or located elsewhere but used by the application.
* Checks that could be performed by the monitoring tools include:
  + Validating the response code. For example, an HTTP response of 200 (OK) indicates that the application responded without error. The monitoring system might also check for other response codes to give more comprehensive results.
  + Checking the content of the response to detect errors, even when a 200 (OK) status code is returned. This can detect errors that affect only a section of the returned web page or service response. For example, checking the title of a page or looking for a specific phrase that indicates the correct page was returned.
  + Measuring the response time, which indicates a combination of the network latency and the time that the application took to execute the request. An increasing value can indicate an emerging problem with the application or network.
  + Checking resources or services located outside the application, such as a content delivery network used by the application to deliver content from global caches.
  + Checking for expiration of SSL certificates.
  + Measuring the response time of a DNS lookup for the URL of the application to measure DNS latency and DNS failures.
  + Validating the URL returned by the DNS lookup to ensure correct entries. This can help to avoid malicious request redirection through a successful attack on the DNS server.
* Where possible, run these checks from different on-premises or hosted locations to measure and compare response times. Ideally you should monitor applications from locations that are close to customers to get an accurate view of the performance from each location. In addition to providing a more robust checking mechanism, the results can help you decide on the deployment location for the application—and whether to deploy it in more than one datacenter.
* Tests should also be run against all the service instances that customers use to ensure the application is working correctly for all customers. For example, if customer storage is spread across more than one storage account, the monitoring process should check all of these.

*Issues and Considerations*

* Consider how to validate the response. For example, is just a single 200 (OK) status code sufficient to verify the application is working correctly? While this provides the most basic measure of application availability, and is the minimum implementation of this pattern, it provides little information about the operations, trends, and possible upcoming issues in the application.
* Make sure that the application correctly returns a 200 (OK) only when the target resource is found and processed. In some scenarios, such as when using a master page to host the target web page, the server sends back a 200 (OK) status code instead of a 404 (Not Found) code, even when the target content page was not found.
* Consider the number of endpoints to expose for an application. One approach is to expose at least one endpoint for the core services that the application uses and another for lower priority services, allowing different levels of importance to be assigned to each monitoring result. Also consider exposing more endpoints, such as one for each core service, for additional monitoring granularity. For example, a health verification check might check the database, storage, and an external geocoding service that an application uses, with each requiring a different level of uptime and response time. The application could still be healthy if the geocoding service, or some other background task, is unavailable for a few minutes.
* Consider whether to use the same endpoint for monitoring as is used for general access, but to a specific path designed for health verification checks, for example, /HealthCheck/{GUID}/ on the general access endpoint. This allows some functional tests in the application to be run by the monitoring tools, such as adding a new user registration, signing in, and placing a test order, while also verifying that the general access endpoint is available.
* Consider the type of information to collect in the service in response to monitoring requests, and how to return this information. Most existing tools and frameworks look only at the HTTP status code that the endpoint returns. To return and validate additional information, you might have to create a custom monitoring utility or service.
* Consider how much information to collect. Performing excessive processing during the check can overload the application and impact other users. The time it takes might exceed the timeout of the monitoring system so it marks the application as unavailable. Most applications include instrumentation such as error handlers and performance counters that log performance and detailed error information, this might be sufficient instead of returning additional information from a health verification check.
* Consider caching the endpoint status. It could be expensive to run the health check too frequently. If the health status is reported through a dashboard, for example, you don't want every request from the dashboard to trigger a health check. Instead, periodically check the system health and cache the status. Expose an endpoint that returns the cached status.
* Consider how to configure security for the monitoring endpoints to protect them from public access, which might expose the application to malicious attacks, risk the exposure of sensitive information, or attract denial of service (DoS) attacks. Typically this should be done in the application configuration so that it can be updated easily without restarting the application. Consider using one or more of the following techniques:
* Consider securing the endpoint by requiring authentication. You can do this by using an authentication security key in the request header or by passing credentials with the request, provided that the monitoring service or tool supports authentication.
  + Use an obscure or hidden endpoint. For example, expose the endpoint on a different IP address to that used by the default application URL, configure the endpoint on a nonstandard HTTP port, and/or use a complex path to the test page. You can usually specify additional endpoint addresses and ports in the application configuration, and add entries for these endpoints to the DNS server if required to avoid having to specify the IP address directly.
  + Expose a method on an endpoint that accepts a parameter such as a key value or an operation mode value. Depending on the value supplied for this parameter, when a request is received the code can perform a specific test or set of tests, or return a 404 (Not Found) error if the parameter value is not recognized. The recognized parameter values could be set in the application configuration.
  + DoS attacks are likely to have less impact on a separate endpoint that performs basic functional tests without compromising the operation of the application. Ideally, avoid using a test that might expose sensitive information. If you must return information that might be useful to an attacker, consider how you will protect the endpoint and the data from unauthorized access. In this case just relying on obscurity is not enough. You should also consider using an HTTPS connection and encrypting any sensitive data, although this will increase the load on the server.
* Consider how to access an endpoint that's secured using authentication. Not all tools and frameworks can be configured to include credentials with the health verification request. For example, Microsoft Azure built-in health verification features cannot provide authentication credentials. Some third-party alternatives are [Pingdom](https://www.pingdom.com/" \t "_blank), [Panopta](https://www.panopta.com/" \t "_blank), [NewRelic](https://newrelic.com/" \t "_blank), and [Statuscake](https://www.statuscake.com/" \t "_blank).
* Consider how to ensure that the monitoring agent is performing correctly. One approach is to expose an endpoint that simply returns a value from the application configuration or a random value that can be used to test the agent.
* Consider ensuring that the monitoring system performs checks on itself, such as a self-test and built-in test, to avoid it issuing false positive results.

*When to Use this Pattern*

* Monitoring websites and web applications to verify availability.
* Monitoring websites and web applications to check for correct operation.
* Monitoring middle-tier or shared services to detect and isolate a failure that could disrupt other applications.
* Complementing existing instrumentation in the application, such as performance counters and error handlers. Health verification checking does not replace the requirement for logging and auditing in the application. Instrumentation can provide valuable information for an existing framework that monitors counters and error logs to detect failures or other issues. However, it cannot provide information if the application is unavailable.

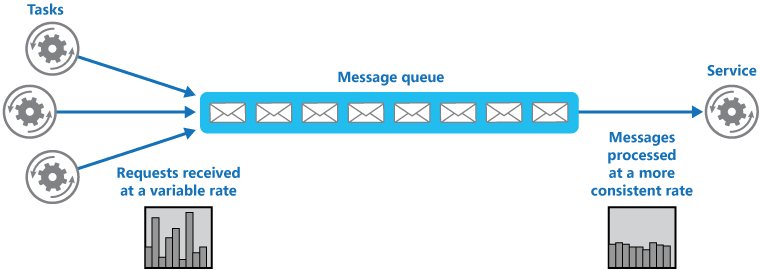
*When Not to Use this Pattern*

* This pattern cannot be used, if required security credentials cannot be provided to access the resources being monitored.
* This pattern will fail, if the application being monitored becomes unavailable.

**Queue-Based Load Leveling pattern**

*Features and Advantages*

* Tasks post messages containing the data required by the service to a queue. The queue acts as a buffer, storing messages until they are retrieved by the services. The services retrieve the messages from the queue and process them. Requests from a number of tasks, which can be generated at a highly variable rate, can be passed to a service through the same message queue. The figure below shows using a queue that is acting to level the load on a service.



* The queue decouples the tasks from the service, and the service can handle the messages at its own pace regardless of the volume of requests from concurrent tasks.
* There is no delay to a task if the service is not available at the time it posts a message to the queue.
* This pattern can help to maximize availability, because delays arising in services will not have an immediate and direct impact on the application, which can continue to post messages to the queue, even when the service is not available or is not currently processing messages.
* This pattern can help to maximize scalability, because both the number of queues and the number of services can be varied to meet demand.
* This pattern can help to control costs, because the number of service instances deployed only have to be adequate to meet average load, rather than the peak load.
* This pattern offers an advantage over service throttling, since queues can prevent the service from failing, due to exceeding the throttling threshold.

*Issues and Considerations*

* It is necessary to implement application logic that controls the rate at which services handle messages to avoid overwhelming the target resource. Avoid passing spikes in demand to the next stage of the system. Test the system under load to ensure that it provides the required leveling, and adjust the number of queues and the number of service instances that handle messages to achieve this.
* Message queues are a one-way communication mechanism. If a task expects a reply from a service, it might be necessary to implement a mechanism that the service can use to send a response. For more information, see the [Asynchronous Messaging Primer](https://msdn.microsoft.com/library/dn589781.aspx).
* Be careful, if you apply autoscaling to services that are listening for requests on the queue. This can result in increased contention for any resources that these services share and thereby diminish the effectiveness of using the queue to level the load.

*When to Use this Pattern*

* This pattern is useful for any application that uses services, which are subject to overloading.

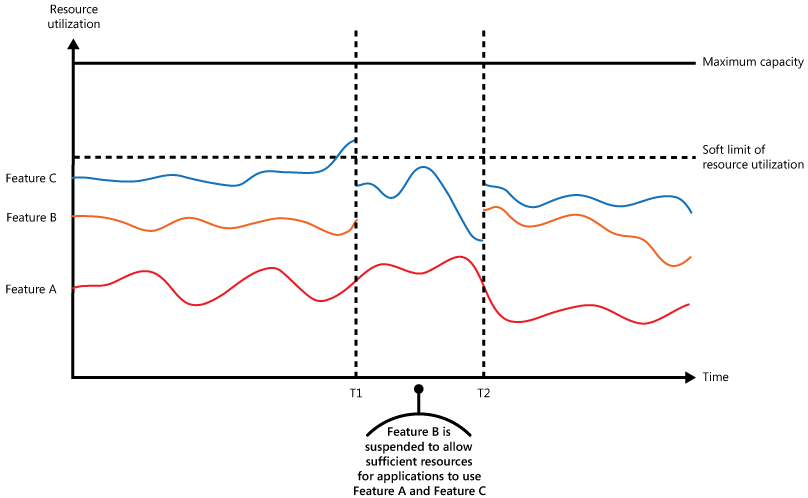
*When Not to Use this Pattern*

* This pattern is not useful, if the application expects a response from the service with minimal latency.

**Throttling pattern**

*Features and Advantages*

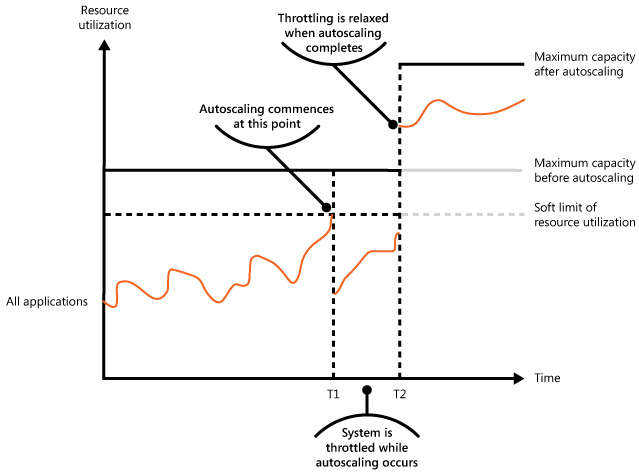
* An alternative strategy to autoscaling is to allow applications to use resources only up to a limit, and then throttle them when this limit is reached. The system should monitor how it's using resources, so that, when usage exceeds the threshold, it can throttle requests from one or more users. This will enable the system to continue functioning and meet any service level agreements (SLAs) that are in place.
* Several different throttling strategies could be implemented, including the following:
  + Rejecting requests from an individual user who has already accessed system APIs more than *n* times per second over a given period of time. This requires the system to meter the use of resources for each tenant or user running an application. For more information, see the [Service Metering Guidance](https://msdn.microsoft.com/library/dn589796.aspx).
  + Disabling or degrading the functionality of selected nonessential services so that essential services can run unimpeded with sufficient resources. For example, if the application is streaming video output, it could switch to a lower resolution.
  + Using load leveling to smooth the volume of activity (this approach is covered in more detail by the [Queue-based Load Leveling pattern](https://docs.microsoft.com/en-us/azure/architecture/patterns/queue-based-load-leveling)). In a multi-tenant environment, this approach will reduce the performance for every tenant. If the system must support a mix of tenants with different SLAs, the work for high-value tenants might be performed immediately. Requests for other tenants can be held back, and handled when the backlog has eased. The [Priority Queue pattern](https://docs.microsoft.com/en-us/azure/architecture/patterns/priority-queue) could be used to help implement this approach.
  + Deferring operations being performed on behalf of lower priority applications or tenants. These operations can be suspended or limited, with an exception generated to inform the tenant that the system is busy and that the operation should be retried later.
  + The figure below shows an area graph for resource use (a combination of memory, CPU, bandwidth, and other factors) against time for applications that are making use of three features. A feature is an area of functionality, such as a component that performs a specific set of tasks, a piece of code that performs a complex calculation, or an element that provides a service such as an in-memory cache. These features are labeled A, B, and C.  The area immediately below the line for a feature indicates the resources that are used by applications when they invoke this feature. For example, the area below the line for Feature A shows the resources used by applications that are making use of Feature A, and the area between the lines for Feature A and Feature B indicates the resources used by applications invoking Feature B. Aggregating the areas for each feature shows the total resource use of the system.



The previous figure illustrates the effects of deferring operations. Just prior to time T1, the total resources allocated to all applications using these features reach a threshold (the limit of resource use). At this point, the applications are in danger of exhausting the resources available. In this system, Feature B is less critical than Feature A or Feature C, so it's temporarily disabled and the resources that it was using are released. Between times T1 and T2, the applications using Feature A and Feature C continue running as normal. Eventually, the resource use of these two features diminishes to the point when, at time T2, there is sufficient capacity to enable Feature B again.

* The autoscaling and throttling approaches can also be combined to help keep the applications responsive and within SLAs. If the demand is expected to remain high, throttling provides a temporary solution while the system scales out. At this point, the full functionality of the system can be restored.

The next figure shows an area graph of the overall resource use by all applications running in a system against time, and illustrates how throttling can be combined with autoscaling.

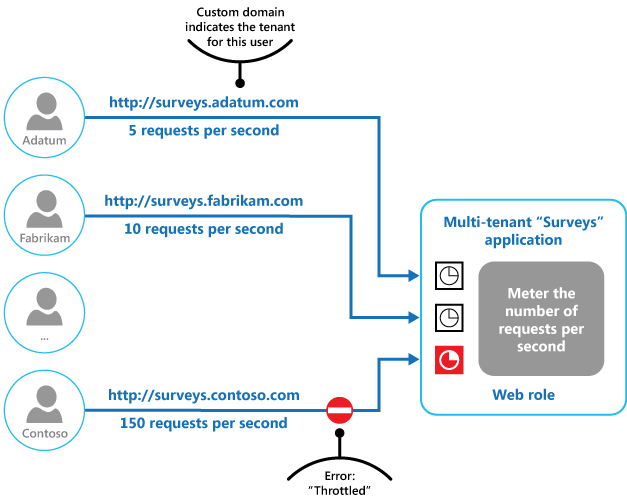


At time T1, the threshold specifying the soft limit of resource use is reached. At this point, the system can start to scale out. However, if the new resources don't become available quickly enough, then the existing resources might be exhausted and the system could fail. To prevent this from occurring, the system is temporarily throttled, as described earlier. When autoscaling has completed and the additional resources are available, throttling can be relaxed.

* Throttling an application, and the strategy to use, is an architectural decision that impacts the entire design of a system. Throttling should be considered early in the application design process because it isn't easy to add once a system has been implemented.
* If a service needs to temporarily deny a user request, it should return a specific error code so the client application understands that the reason for the refusal to perform an operation is due to throttling. The client application can wait for a period before retrying the request.
* Throttling must be performed quickly. The system must be capable of detecting an increase in activity and react accordingly. The system must also be able to revert to its original state quickly after the load has eased. This requires that the appropriate performance data is continually captured and monitored.
* Throttling can be used as a temporary measure while a system autoscales. In some cases it is better to simply throttle, rather than to scale, if a burst in activity is sudden and is not expected to be long lived because scaling can add considerably to running costs.
* If throttling is being used as a temporary measure while a system autoscales, and if resource demands grow very quickly, the system might not be able to continue functioning—even when operating in a throttled mode. If this is not acceptable, consider maintaining larger capacity reserves and configuring more aggressive autoscaling.

*When to Use this Pattern*

* To ensure that a system continues to meet service level agreements.
* To prevent a single tenant from monopolizing the resources provided by an application.
* To handle bursts in activity.
* To help cost-optimize a system by limiting the maximum resource levels needed to keep it functioning.
* Use this pattern in a multi-tenant system in order to prevent the users from one tenant affecting the responsiveness and availability of the application for all other users.  The following figure illustrates how throttling can be implemented in a multi-tenant system. Users from each of the tenant organizations access a cloud-hosted application where they fill out and submit surveys. The application contains instrumentation that monitors the rate at which these users are submitting requests to the application.



In order to prevent the users from one tenant affecting the responsiveness and availability of the application for all other users, a limit is applied to the number of requests per second the users from any one tenant can submit. The application blocks requests that exceed this limit.

*When Not to Use this Pattern*

* Consider alternatives to using this pattern, if none of the aforementioned use cases apply, or if implementing the Throttling pattern for the application would also require use of the following related patterns, but there is some reason that implementation of that necessary related pattern cannot be effected:
  + [Instrumentation and Telemetry Guidance](https://msdn.microsoft.com/library/dn589775.aspx). Throttling depends on gathering information about how heavily a service is being used. This related pattern describes how to generate and capture custom monitoring information.
  + [Service Metering Guidance](https://msdn.microsoft.com/library/dn589796.aspx). Describes how to meter the use of services in order to gain an understanding of how they are used. This information can be useful in determining how to throttle a service.
  + [Autoscaling Guidance](https://msdn.microsoft.com/library/dn589774.aspx). Throttling can be used as an interim measure while a system autoscales, or to remove the need for a system to autoscale. This related pattern contains information on autoscaling strategies.
  + [Queue-based Load Leveling pattern](https://docs.microsoft.com/en-us/azure/architecture/patterns/queue-based-load-leveling). Queue-based load leveling is a commonly used mechanism for implementing throttling. A queue can act as a buffer that helps to even out the rate at which requests sent by an application are delivered to a service.
  + [Priority Queue pattern](https://docs.microsoft.com/en-us/azure/architecture/patterns/priority-queue). A system can use priority queuing as part of its throttling strategy to maintain performance for critical or higher value applications, while reducing the performance of less important applications.

Data Management patterns

**Cache-Aside pattern**

*Features and Advantages*



Management and Monitoring patterns

**Ambassador pattern**

*Features and Advantages*

* In the [Ambassador](https://docs.microsoft.com/en-us/azure/architecture/patterns/ambassador) design pattern, client frameworks and libraries are placed into an external process, which acts as a proxy between your application and external services.
* Deploy the proxy on the same host environment as your application to allow control over routing, resiliency, security features, and to avoid any host-related access restrictions.
* Standardize and extend instrumentation. The proxy can monitor performance metrics such as latency or resource usage, and this monitoring happens in the same host environment as the application.
* Features that are offloaded to the ambassador can be managed independently of the application. So, the ambassador can be updated and modified without disturbing the application's legacy functionality.
* Separate, specialized teams can implement and maintain security, networking, or authentication features that have been moved to the ambassador.

*Issues and Considerations*

* If the consuming service is containerized, the ambassador should be created as a separate container on the same host, with the appropriate links configured for communication.
* The proxy adds some latency overhead. Consider whether a client library, invoked directly by the application, is a better approach.
* Consider the possible impact of including generalized features in the proxy. For example, the ambassador could handle retries, but that might not be safe unless all operations are idempotent.
* Consider a mechanism to allow the client to pass some context to the proxy, as well as back to the client. For example, include HTTP request headers to opt out of retry or specify the maximum number of times to retry.
* Consider how you will package and deploy the proxy.
* Consider whether to use a single shared instance for all clients or an instance for each client.

*When to Use this Pattern*

* When the need is to build a common set of client connectivity features for multiple languages or frameworks.
* When the need is to offload cross-cutting client connectivity concerns to infrastructure developers or other more specialized teams.
* When the need is to support cloud or cluster connectivity requirements in a legacy application or an application that is difficult to modify.

*When Not to Use this Pattern*

* When network request latency is critical. A proxy will introduce some overhead, although minimal, and in some cases this may affect the application.
* When client connectivity features are consumed by a single language. In that case, a better option might be a client library that is distributed to the development teams as a package.
* When connectivity features cannot be generalized and require deeper integration with the client application.

**Sidecar pattern**

*Features and Advantages*

* In the [Sidecar](https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar) design pattern, an auxiliary service is logically associated with the primary application, supplying peripheral tasks, such as logging, configuration, and proxies to remote services.
* The sidecar resides in a container, usually separate from the main application container.  So, sidecars can be written in different languages from each other and different from the main application, if desired.  Because both sidecar and the main application reside in the same node, latency in communications between them exists, but is minimal.
* If it is desired for the sidecar to provide extensibility functionality to the main application, then it is possible to co-locate the sidecar, as its own process, within the same host or sub-container, as the primary application.
* The sidecar's lifecycle coincides with the lifecycle of the main application.
* The sidecar can access the same resources as the primary application.  For example, a sidecar could monitor system resources used by both the sidecar and the primary application.

*Issues and Considerations*

* Consider the deployment and packaging format you will use to deploy services, processes, or containers. Containers are particularly well suited to the sidecar pattern.
* When designing a sidecar service, carefully decide on the inter-process communication mechanism. Try to use language- or framework-agnostic technologies unless performance requirements make that impractical.
* Before putting functionality into a sidecar, consider whether it would work better as a separate service or a more traditional daemon.
* Also consider whether the functionality could be implemented as a library or using a traditional extension mechanism. Language-specific libraries may have a deeper level of integration and less network overhead.

*When to Use this Pattern*

* Your primary application uses a heterogeneous set of languages and frameworks. A component located in a sidecar service can be consumed by applications written in different languages using different frameworks.
* A component is owned by a remote team or a different organization.
* A component or feature must be co-located on the same host as the application
* You need a service that shares the overall lifecycle of your main application, but can be independently updated.
* You need fine-grained control over resource limits for a particular resource or component. For example, you may want to restrict the amount of memory a specific component uses. You can deploy the component as a sidecar and manage memory usage independently of the main application.

*When Not to Use this Pattern*

* When inter-process communication needs to be optimized. Communication between a parent application and sidecar services includes some overhead, notably latency in the calls. This may not be an acceptable trade-off for chatty interfaces.
* For small applications, where the resource cost of deploying a sidecar service for each instance is not worth the advantage of isolation.
* When the service needs to scale differently than or independently from the main applications. If so, it may be better to deploy the feature as a separate service.

Technology Options

These design patterns can be implemented by any cloud provider, which supports robust communication among containers on single node, or across multiple nodes.

Feature Comparison

|  |  |  |  |
| --- | --- | --- | --- |
|  | <Option 1> | <Option 2> | … |
| <Feature 1> |  |  |  |
| … |  |  |  |