Azure Cloud Architecture Design Pattern

| **Pattern Name** | **Comment** |
| --- | --- |
| **Ambassador** | A proxy service to handle some generic and common utility task like:   * Circle Breaking * Retry * Monitoring * Security * Logging   Diagram of the Ambassador pattern |
| **Anti-corruption Layer** | Loose coupling modern system with legacy system by providing a façade to isolate them in order to prevent legacy system to “pollute” the modern system  Diagram of the Anti-Corruption Layer pattern |
| **Asynchronous Request-Reply** | Turn time consuming synchronous Api call into polling an Asynchronous Api call using HTTP response status code |
| **Backends for Fronts** | Create separate backend services for different front-end applications due to UI and reactive consideration, e.g., different backend APIs for Desktop and Mobile.  Context-and-problem diagram of the Backends for Frontends pattern |
| **Bulkhead** | Bulkhead is identical to create an **isolation** and **redundant** protection in case of failure, it includes:   * Isolation, divide service to different partitions (connection pools) based on their workloads etc. * Redundant, create on-demand connection pools to handle workload balancing.  |  |  | | --- | --- | | First diagram of the Bulkhead pattern | Diagram showing multiple clients calling a single service. | |
| **Cache-Aside** | Cache non-transient data can improve performance  Using the Cache-Aside pattern to store data in the cache |
| **Choreography** | Each component of a system tries to communicate with each other without a centralized Orchestrator by using **Asynchronous Messaging Pattern** (also called **Publisher-Subscriber Pattern** or **Broadcast** or Azure **Topic**)  Processing a request using a central orchestrator  Processing a request using a choreographer  Publish-subscribe pattern using a message broker |
| **Circuit Breaker** | It works with **Retry Pattern** and manages the following 3 states:   * **Closed**: operation normal, return result * **Open**: operation failed, return failure or exception * **Half-Open**: Retry in a limited scale to see whether to switch to Closed or Open   Circuit Breaker states |
| **Claim Check** | Message holds data, but split large message into a claim check (message id) and the big payload, just send the claim-check to the messaging platform (e.g., Azure Service Bus) and store the payload to an external service (e.g., database) and later claim back the payload using claim check (the message id) to re-construct the original message data.  Diagram of the Claim-Check pattern. |
| **Compensating Transaction** | The difference between **Traditional Transaction** and **Compensting Transaction** is: Compensating Transaction is not Transactional at all, there is no 2 phase commit at all, it is just an eventual consistency model - a typical business operation consists of a series of separate steps. While the operation performs these steps, the overall view of the system state might be inconsistent. But when the operation finishes and all the steps have run, the system should become consistent again.  Diagram that shows the steps for creating an itinerary. The steps of the compensating transaction that cancels the itinerary are also shown.  The compensation logic seems case by case, better to use a local database to lock and store each transaction step and later call an idempotent method to guarantee commit.   * **Traditional Transaction** is if I say I do then I will do for sure. It uses locking mechanism. * **Transaction Compensation** is I will do first and rewind (**undo**) in case of failure. It does not lock, so there will be concurrent conflicts. Similar to **Saga Pattern**. |
| **Competing Consumer** | Count on Azure cloud platform to automatically to create Azure Function App Instances to auto scale out competing message consumers to pull messages simultaneously from Azure Service. The Azure Functions runtime receives a message in PeekLock mode, if the function finishes successfully, it calls Complete on the message, or it may call Abandon if the function fails, and the message will become visible again, allowing another consumer to retrieve it. If the function runs for a period longer than the PeekLock timeout, the lock is automatically renewed as long as the function is running.  Azure Functions can scale out/in based on the depth of the queue, all acting as competing consumers of the queue. If multiple instances of the functions are created, they all compete by independently pulling and processing the messages.  Using a message queue to distribute work to instances of a service |
| **Compute Resource Consolidation** | Group computing intensive resources together and host them in powerful cloud resources. This is more a configuration rather than programming pattern. |
| **CQRS** | **CQRS** means **Command and Query Responsibility Segregation**  That is to separate **Write** (Command) and **Read** (Query) data store in order to reduce contention (transactional locks, dead locks etc.)  CQRS often works with Event Sourcing pattern, where application state is stored as a sequence of events. Each event represents a set of changes to the data. The current state is constructed by replaying the events. In a CQRS context, one benefit of Event Sourcing is that the same events can be used to notify other components — in particular, to notify the read model. The read model uses the events to create a snapshot of the current state, which is more efficient for queries.  When used with the Event Sourcing pattern, the store of events is the write model, and is the official source of information. The read model of a CQRS-based system provides materialized views of the data, typically as highly denormalized views. These views are tailored to the interfaces and display requirements of the application, which helps to maximize both display and query performance.  Using the stream of events as the write store, rather than the actual data at a point in time, avoids update conflicts on a single aggregate and maximizes performance and scalability. The events can be used to asynchronously generate materialized views of the data that are used to populate the read store.  Because the event store is the official source of information, it is possible to delete the materialized views and replay all past events to create a new representation of the current state when the system evolves, or when the read model must change. The materialized views are in effect a durable read-only cache of the data.  When used with the Event Sourcing pattern, the store of events is the write model, and is the official source of information. The read model of a CQRS-based system provides materialized views of the data, typically as highly denormalized views. These views are tailored to the interfaces and display requirements of the application, which helps to maximize both display and query performance.  Using the stream of events as the write store, rather than the actual data at a point in time, avoids update conflicts on a single aggregate and maximizes performance and scalability. The events can be used to asynchronously generate materialized views of the data that are used to populate the read store.  Because the event store is the official source of information, it is possible to delete the materialized views and replay all past events to create a new representation of the current state when the system evolves, or when the read model must change. The materialized views are in effect a durable read-only cache of the data.When using CQRS combined with the Event Sourcing pattern, consider the following:  As with any system where the write and read stores are separate, systems based on **this pattern are only eventually consistent**. There will be some delay between the event being generated and the data store being updated.  The pattern adds complexity because code must be created to initiate and handle events, and assemble or update the appropriate views or objects required by queries or a read model. The complexity of the CQRS pattern when used with the Event Sourcing pattern can make a successful implementation more difficult, and requires a different approach to designing systems. However, event sourcing can make it easier to model the domain, and makes it easier to rebuild views or create new ones because the intent of the changes in the data is preserved.  Generating materialized views for use in the read model or projections of the data by replaying and handling the events for specific entities or collections of entities can require significant processing time and resource usage. This is especially true if it requires summation or analysis of values over long periods, because all the associated events might need to be examined. Resolve this by implementing snapshots of the data at scheduled intervals, such as a total count of the number of a specific action that has occurred, or the current state of an entity.  A CQRS architecture with separate read and write stores |
| **Deployment Stamps** | |  |  | | --- | --- | | It’s about scalability for different tenants of your cloud applications. Consider grouping resources in scale units and provisioning multiple copies of your stamps. Each scale unit will host and serve a subset of your tenants. Stamps operate independently of each other and can be deployed and updated independently. A single geographical region might contain a single stamp, or might contain multiple stamps to allow for horizontal scale-out within the region. Stamps contain a subset of your customers. | An example set of deployment stamps | |
| **Edge Workload Configuration** | Edge computing allows devices in remote locations to process data at the "edge" of the network, either by the device or a local server. And when data needs to be processed in the central datacenter, only the most important data is transmitted, thereby minimizing latency.  This pattern tries to address this challenge of Edge computing. |
| **Event Sourcing** | Instead of storing just the current state of the data in a domain, use an **append-only store** to record the full series of actions taken on that data. The store acts as the system of record and can be used to materialize the domain objects. This can simplify tasks in complex domains, by avoiding the need to synchronize the data model and the business domain, while improving performance, scalability, and responsiveness. It can also provide consistency for transactional data, and maintain full audit trails and history that can enable compensating actions.  Limitation of **CRUD**:   * CRUD systems perform update operations directly against a data store. These operations can slow down performance and responsiveness and can limit scalability, due to the processing overhead it requires. * In a collaborative domain with many concurrent users, data update conflicts are more likely because the update operations take place on a single item of data. * Unless there's another auditing mechanism that records the details of each operation in a separate log, history is lost.   The events are persisted in an event store that acts as the system of record (the authoritative data source) about the current state of the data. The event store typically publishes these events so that consumers can be notified and can handle them if needed. Consumers could, for example, initiate tasks that apply the operations in the events to other systems, or perform any other associated action, that's required to complete the operation. Notice that the application code that generates the events is decoupled from the systems that subscribe to the events. **Events are immutable**. **Event Sourcing is also eventual consistent**.  An overview and example of the Event Sourcing pattern |
| **External Configuration Store** | Don’t deploy applications to cloud with configuration files. Use an external configuration provider like Azure Key Vault.  An overview of the External Configuration Store pattern with optional local cache |
| **Federated Identity** | |  |  | | --- | --- | | Delegate authentication to an external identity provider. This can simplify development, minimize the requirement for user administration, and improve the user experience of the application.  Use Identity Provider (the Idp – Identity Provider part of XAML, the other part is Sp – Service Provider or OpenID) | An overview of federated authentication |   SAML / OpenID / Json Web Token |
| **Gatekeeper** | Protect applications and services by using a dedicated host instance to broker requests between clients and the application or service. The broker validates and sanitizes the requests and can provide an additional layer of security and limit the system's attack surface.  It works like Azure API Manager or SOA policy controller to provide an additional layer for security.  An example of the pattern using Cloud Services web and worker roles |
| **Gateway Aggregation** | |  |  | | --- | --- | | Use a gateway to aggregate multiple individual requests into a single request. This pattern is useful when a client must make multiple calls to different backend systems to perform an operation.  Works like Services Orchestration |  | |
| **Gateway Offloading** | Offload shared or specialized service functionality to a gateway proxy. This pattern can simplify application development by moving shared service functionality, such as the use of SSL certificates, from other parts of the application into the gateway. |
| **Gateway Routing** | Route requests to multiple services or multiple service instances using a single endpoint. The pattern is useful when you want to:   * Expose multiple services on a single endpoint and route to the appropriate service based on the request * Expose multiple instances of the same service on a single endpoint for load balancing or availability purposes * Expose differing versions of the same service on a single endpoint and route traffic across the different versions |
| **Geode** | The Geode pattern involves deploying a collection of backend services into a set of geographical nodes, each of which can service any request for any client in any region.  Deploy the service into a number of satellite deployments spread around the globe, each of which is called a geode. The geode pattern harnesses key features of Azure to route traffic via the shortest path to a nearby geode, which improves latency and performance.  Geode overview |
| **Health Endpoint Monitoring** | To verify that applications and services are performing correctly, you can use the Health Endpoint Monitoring pattern. This pattern specifies the use of functional checks in an application. External tools can access these checks at regular intervals through exposed endpoints.  Architecture diagram that shows components that health monitoring checks. Examples include an app, its storage and database, and a content delivery network. |
| **Index Table** | Create indexes over the fields in data stores that are frequently referenced by queries. This pattern can improve query performance by allowing applications to more quickly locate the data to retrieve from a data store.  Similar to create an index table in database. |
| **Leader Election** | |  |  | | --- | --- | | Coordinate the actions performed by a collection of collaborating instances in a distributed application by electing one instance as the leader that assumes responsibility for managing the others. This can help to ensure that instances don't conflict with each other, cause contention for shared resources, or inadvertently interfere with the work that other instances are performing.  Basically we are talking about concurrent or parallel access to resources, similar to Mutex lock in local applications, the elected leader is the **current holder of the Mutex lock**. |  | |
| **Materialized View** | This is more focused on the Query/Read model, it’s similar to relational database view. It may be deformalized for speed and closed to required information. This pattern works closely with **CQRS** and **Event Sourcing** pattern.  Figure 1 shows an example of how the Materialized View pattern might be used |
| **Messaging Bridge** | A bridge component that connects to two or more messaging infrastructures at the same time. The bridge pulls messages from one and pushes them to the other without changing the payload. It’s a message forwarder agent or message trigger service.  Diagram of the Messaging Bridge integrating MSMQ and Service Bus. |
| **Pipes and Filters** | Decompose a task that performs complex processing into a series of separate elements that can be reused. Doing so can improve performance, scalability, and reusability by allowing task elements that perform the processing to be deployed and scaled independently.  Basically, it’s about reusable component design.  Diagram that shows a solution that's implemented with pipes and filters. |
| **Priority Queue** | Prioritize requests sent to services so that requests with a higher priority are received and processed more quickly than those with a lower priority.  Diagram that illustrates a queuing mechanism that supports message prioritization. |
| **Publisher-Subscriber** | Enable an application to announce events to multiple interested consumers asynchronously, without coupling the senders to the receivers.  This pattern is similar to **Azure Service Bus –** **Topics**  Publish-subscribe pattern using a message broker |
| **Queue-Based Load Leveling** | Use a queue that acts as a buffer between a task and a service it invokes in order to smooth intermittent heavy loads that can cause the service to fail or the task to time out. This can help to minimize the impact of peaks in demand on availability and responsiveness for both the task and the service.  This is the Asynchronous pattern providing a buffer. |
| **Rating Limiting** | This pattern is for performing large number of operations at the same time. Solution is to use “Durable Messaging Service Buffer + Throttling Processor” since “Durable Messaging is crash proved and Throttling Processor makes sure that the downstream Throttled Service won’t overwhelm.  A durable messaging flow with three job processors calling into a throttled service. |
| **Retry** | Services calls failure might be caused by transient faults that can be retried.   * Cancel. If the fault indicates that the failure isn't transient or is unlikely to be successful if repeated, the application should cancel the operation and report an exception. For example, an authentication failure caused by providing invalid credentials is not likely to succeed no matter how many times it's attempted. * Retry. If the specific fault reported is unusual or rare, it might have been caused by unusual circumstances such as a network packet becoming corrupted while it was being transmitted. In this case, the application could retry the failing request again immediately because the same failure is unlikely to be repeated and the request will probably be successful. * Retry after delay. If the fault is caused by one of the more commonplace connectivity or busy failures, the network or service might need a short period while the connectivity issues are corrected or the backlog of work is cleared. The application should wait for a suitable time before retrying the request. |
| **Saga** | A saga is a sequence of local transactions that updates each service and publishes a message or event to trigger the next transaction step. If a step fails, the saga executes compensating transactions that counteract the preceding transactions.  It works like commit one transactional step at one time and continue to next step until it completes or rollback in a First In Last Out (stack) pattern.    Because Saga has a Local Transaction in Database so it’s safer than **Compensating Transaction**.  In Saga patterns:   * *Compensable transactions* are transactions that can potentially be reversed by processing another transaction with the opposite effect. * A *pivot transaction* is the go/no-go point in a saga. If the pivot transaction commits, the saga runs until completion. A pivot transaction can be a transaction that is neither compensable nor retryable, or it can be the last compensable transaction or the first retryable transaction in the saga. * *Retryable transactions* are transactions that follow the pivot transaction and are guaranteed to succeed.   There are two common saga implementation approaches, choreography and orchestration.  **Choreography** is a way to coordinate sagas where participants exchange events without a centralized point of control. With choreography, each local transaction publishes domain events that trigger local transactions in other services.  Here is an example of Bank Transfer (Credit and Debit) and Sending Receipt Saga Transaction workflow. Notice that the Saga Orchestrator is a durable Function App which means it saves the state into a durable storage like database.    **Orchestration** (like a Transaction Manager to notify each Resource Manager, or like a Distributed Transaction Coordinator) is a way to coordinate sagas where a centralized controller tells the saga participants what local transactions to execute. The saga orchestrator handles all the transactions and tells the participants which operation to perform based on events. The orchestrator executes saga requests, stores and interprets the states of each task, and handles failure recovery with compensating transactions. |
| **Scheduler Agent Supervisor** | This pattern is similar to Saga Orchestration pattern where it has a coordinator which is the Supervisor, the difference is Saga handles a sequential ACID transaction where all the steps involved either successful or failed, if one of them failed will trigger backward sequential rollback called Compensation. Scheduler Agent Supervisor pattern handles a workflow (linear steps) ACID transaction with different kind of paths and steps where makes it even more complicated.  Scheduler: is the workflow  Agent: is one of the steps in the workflow  Supervisor: is the Orchestrator or Coordinator   * The **Scheduler** arranges for the steps that make up the task to be executed and orchestrates their operation. These steps can be combined into a pipeline or workflow. The Scheduler is responsible for ensuring that the steps in this workflow are performed in the right order. As each step is performed, the Scheduler records the state of the workflow, such as "step not yet started," "step running," or "step completed." The state information should also include an upper limit of the time allowed for the step to finish, called the complete-by time. If a step requires access to a remote service or resource, the Scheduler invokes the appropriate Agent, passing it the details of the work to be performed. The Scheduler typically communicates with an Agent using [asynchronous request/response messaging](https://learn.microsoft.com/en-us/azure/architecture/patterns/async-request-reply). This can be implemented using queues, although other distributed messaging technologies could be used instead.   The Scheduler performs a similar function to the Process Manager in the [Process Manager pattern](https://www.enterpriseintegrationpatterns.com/patterns/messaging/ProcessManager.html). The actual workflow is typically defined and implemented by a workflow engine that's controlled by the Scheduler. This approach decouples the business logic in the workflow from the Scheduler.   * The **Agent** contains logic that encapsulates a call to a remote service, or access to a remote resource referenced by a step in a task. Each Agent typically wraps calls to a single service or resource, implementing the appropriate error handling and retry logic (subject to a timeout constraint, described later). When implementing retry logic, pass a stable identifier across all retry attempts so that the remote service can use it for any deduplication logic it may have. If the steps in the workflow being run by the Scheduler use several services and resources across different steps, each step might reference a different Agent (this is an implementation detail of the pattern). * The **Supervisor** monitors the status of the steps in the task being performed by the Scheduler. It runs periodically (the frequency will be system-specific), and examines the status of steps maintained by the Scheduler. If it detects any that have timed out or failed, it arranges for the appropriate Agent to recover the step or execute the appropriate remedial action (this might involve modifying the status of a step). Note that the recovery or remedial actions are implemented by the Scheduler and Agents. The Supervisor should simply request that these actions be performed.   The Scheduler, Agent, and Supervisor are logical components and their physical implementation depends on the technology being used. For example, several logical agents might be implemented as part of a single web service.  The Scheduler maintains information about the progress of the task and the state of each step in a durable data store, called the state store. The Supervisor can use this information to help determine whether a step has failed. The figure illustrates the relationship between the Scheduler, the Agents, the Supervisor, and the state store.  Scheduler Agent Supervisor pattern:    Process Manager Enterprise Pattern: |
| Sequential Convoy | Messages in a queue is asynchronous and multiplexing of interleaved with each other based on the Competing Consumers pattern. Sequential Convoy pattern uses features like Azure Service Bus Message Sessions (attached a Session ID in each message) to peek-lock messages belonging to a particular session or other categories definition.  De-multiplexing of interleaved messages like group by and order:    Sequential Convoy pattern: |
| **Sharding** | Divide a data store into a set of horizontal partitions or shards. This can improve scalability when storing and accessing large volumes of data. Similar to a Hash table lookup, use the location as the Key and its hash as the Shard address to find the data stored in it. |
| **Sidecar** | For re-usable utility services such as such as monitoring, logging, configuration, and networking. These peripheral tasks can be implemented as separate components or services, loose coupling and isolated with the parent application so that each service can be built using different languages and technologies, having its own dependencies and required language-specific libraries.    A sidecar service is not necessarily part of the application, but is connected to it. It goes wherever the parent application goes. Sidecars are supporting processes or services that are deployed with the primary application. On a motorcycle, the sidecar is attached to one motorcycle, and each motorcycle can have its own sidecar. In the same way, a sidecar service shares the fate of its parent application. For each instance of the application, an instance of the sidecar is deployed and hosted alongside it.  Advantages of using a sidecar pattern include:   * A sidecar is independent from its primary application in terms of runtime environment and programming language, so you don't need to develop one sidecar per language. * The sidecar can access the same resources as the primary application. For example, a sidecar can monitor system resources used by both the sidecar and the primary application. * Because of its proximity to the primary application, there's no significant latency when communicating between them. * Even for applications that don't provide an extensibility mechanism, you can use a sidecar to extend functionality by attaching it as its own process in the same host or sub-container as the primary application.   The sidecar pattern is often used with containers and referred to as a sidecar container or sidekick container. |
| Static Content Hosting | Store static content (front end html, images, style sheets and JavaScript etc.) into cloud-based storage service, separate frontend static with back end dynamic services. |
| **Strangler Fig** | Incrementally migrate a legacy system by gradually replacing specific pieces of functionality with new applications and services. As features from the legacy system are replaced, the new system eventually replaces all of the old system's features, strangling the old system and allowing you to decommission it.  Like a zombie parasite gradually to fully control the host.    This pattern helps to minimize risk from the migration, and spread the development effort over time. With the façade safely routing users to the correct application, you can add functionality to the new system at whatever pace you like, while ensuring the legacy application continues to function. Over time, as features are migrated to the new system, the legacy system is eventually "strangled" and is no longer necessary. Once this process is complete, the legacy system can safely be retired. |
| **Throttling** | allow applications to use resources only up to a limit, and then throttle them when this limit is reached. Throttling doesn’t mean just reject extra requests, it:   * Rejects frequent requests from the same user * Disables or degrades the functionality of selected nonessential services. E.g., streams video output using a lower resolution * Use durable buffer to accelerate request reception but translate them later asynchronously, like using Queue-based Load Leveling pattern |
| **Valet Key** | Works like JWT (Json Web Token for authentication and authorization)  This key or token is usually referred to as a valet key. It provides time-limited access to specific resources and allows only predefined operations such as reading and writing to storage or queues, or uploading and downloading in a web browser. Applications can create and issue valet keys to client devices and web browsers quickly and easily, allowing clients to perform the required operations without requiring the application to directly handle the data transfer. This removes the processing overhead, and the impact on performance and scalability, from the application and the server.  The client uses this token to access a specific resource in the data store for only a specific period, and with specific restrictions on access permissions, as shown in the figure. After the specified period, the key becomes invalid and won't allow access to the resource. |