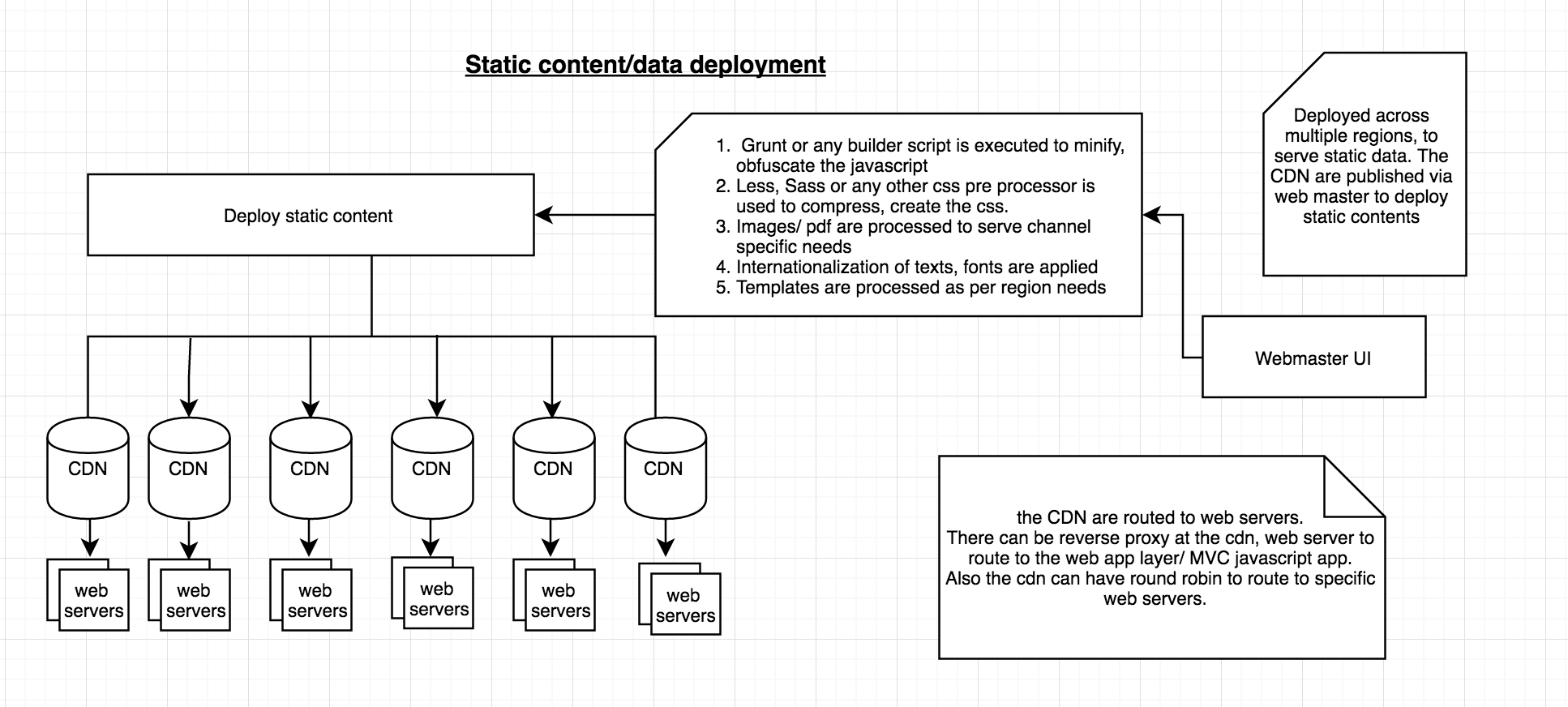
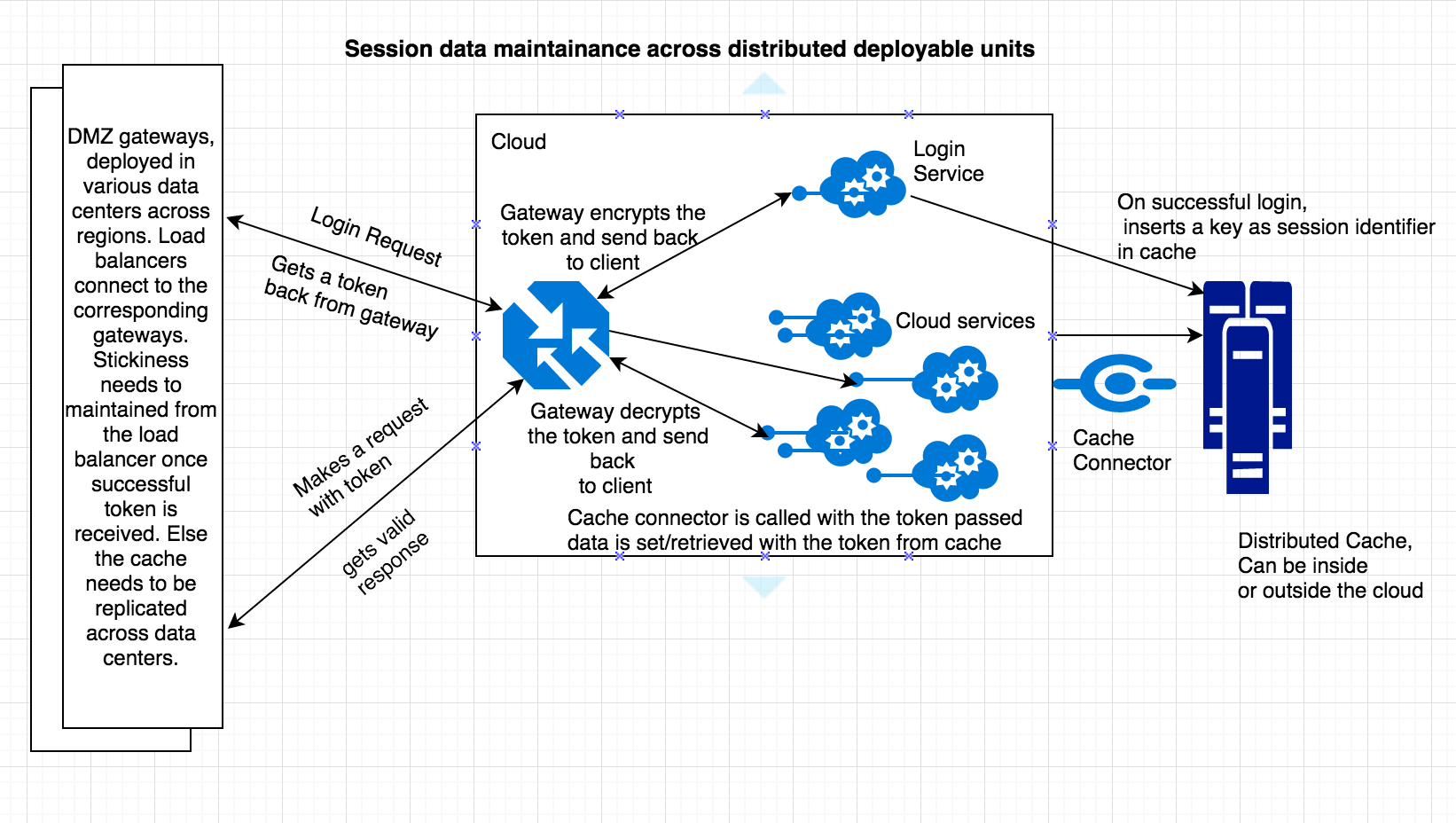
**Data pain areas and how to solve in cloud native architecture.**

1. **Managing static contents and content caching and reload.**



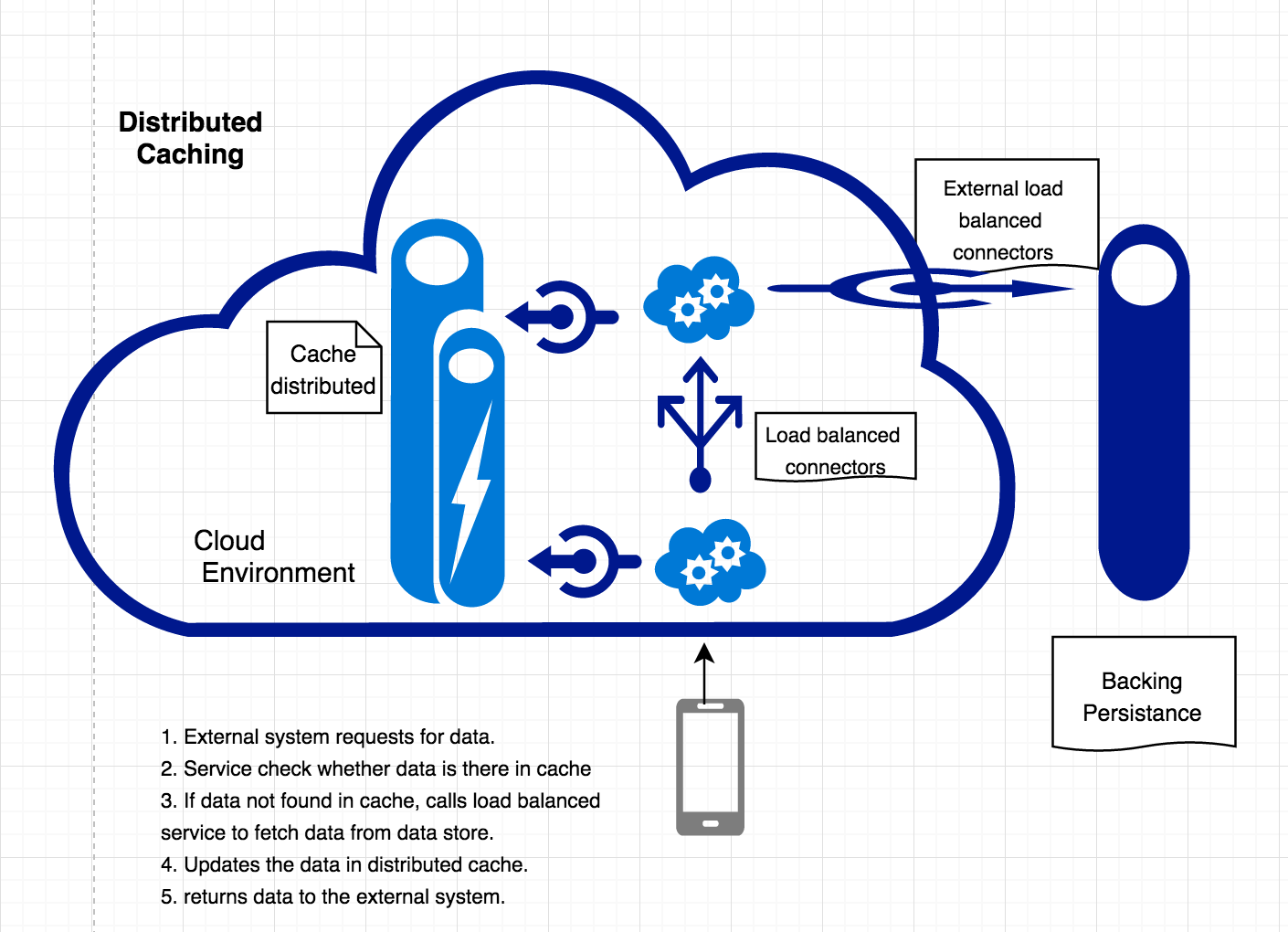
Any digital space has lot of static contents (css, js, images, pdf, etc.) and these needs to be deployed and managed across multiple regions. These items are needed to be taken care while working with static data.

1. Multiple CDN deployed across regions.
2. The CDN are load balanced to perform reverse proxy/route to specific web servers.
3. Webmaster UI to push the static contents across the CDNs and web servers.
4. Grunt or any builder is used to build the static contents.
5. Necessary care needs to be taken to package the scripts (obfuscate, compress, minify)
6. CSS compilers to be used to process the less, sass scripts to compiled css.
7. Fluid images and sprite to be used for faster processing of images.
8. Re deployment needs to be performed by webmaster such that all the scripts are replicated across multiple CDNs.
9. Web templates needs to be cached in browser/ device for fast rendering.
10. Care should be taken from the browser/device to check resource modified at the server end. Techniques like heartbeat, auto reload, versioning, adding date-stamp, http-modified needs to be implemented for latest scripts.
11. Versioning, blue green deployment of scripts should be in place for faster deployment life cycle and quick reversal of scripts.
12. Load balanced urls should be used from the app to query the static content, such that contents are processed at quicker rate.
13. **Session data management in distributed environment.**



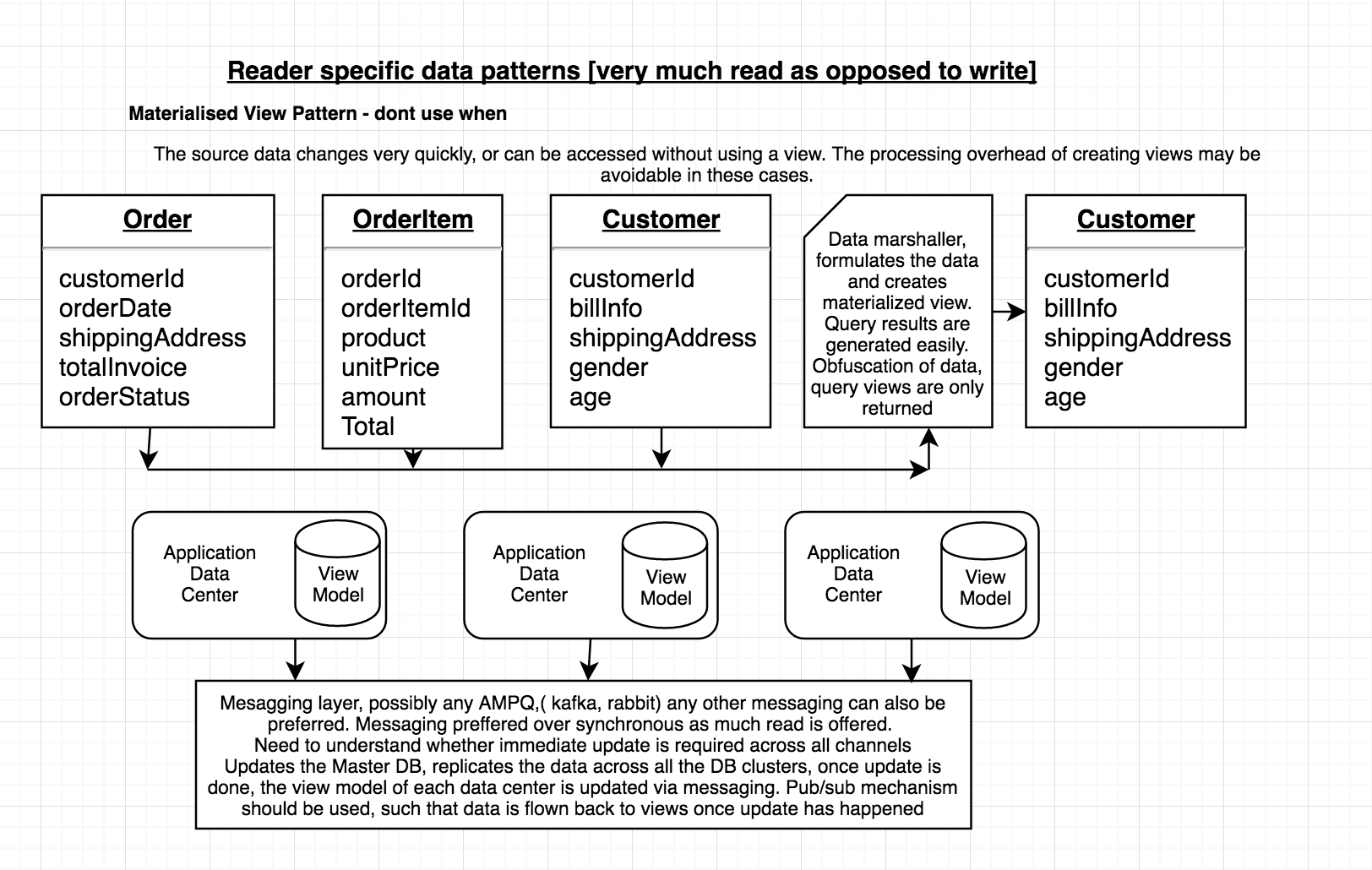
In distributed environment while working with cloud native apps, the deployable apps needs to created keeping in mind the 12 factor (<https://12factor.net/>) principles. The app needs to be as stateless as possible. But while working in an enterprise environment, where there are lot of moving parts and lot of data residing in legacy systems, we need an efficient session data management such that we decouple apis as much as possible and also we keep in mind of recurring information used across the session which can be easily used. General guidelines are prescribed below.

1. As a part of login response, when user is successfully authenticated and authorized, the client needs to be issued an authorization token. This token can be based on oAUTH, oAUTH2, JWT or any other form of authorization token.
2. On successful login, a session token needs to be inserted to a distributed cache which can reside inside or outside the cloud PAAS, depending on the nature of the client.
3. A gateway service, salts and encrypts the session token and passes back to the client along with authorization token for the apis.
4. On Subsequent requests with the authorization token, this session token needs to be passed, which the gateway normalizes and decrypts and passes to the apis.
5. The apis puts data as map associated with session token and puts it in distributed cache via cache connectors.
6. The cache connectors can be used to set/update data in distributed cache.
7. The cache should be partitioned or replicated depending on the nature of the app.
8. The data in cache associated with session token should also be partitioned within the bounded context such that easy lookup can happen for cached data.
9. Logout apis should shred all the data associated with the cache.
10. Analytics tool can be associated with the cache which can work in entirely different thread, such that useful information can be processed out to drive business requirement.
11. **Distributed caching and usages**



Applications use a distributed cache to get high availability of repeated access to information held in a data persistence. However, it is usually impractical to expect that cached data will always be completely consistent with the data in the data store. Applications should implement a strategy that helps to ensure that the data in the cache is up to date as far as possible, but can also detect and handle situations that arise when the data in the cache has become stale. Several issues need to be taken care while implementing the pattern, - Lifetime and scope of cached data, shredding data, consistency, in-memory caching.

1. **Materialized view pattern**



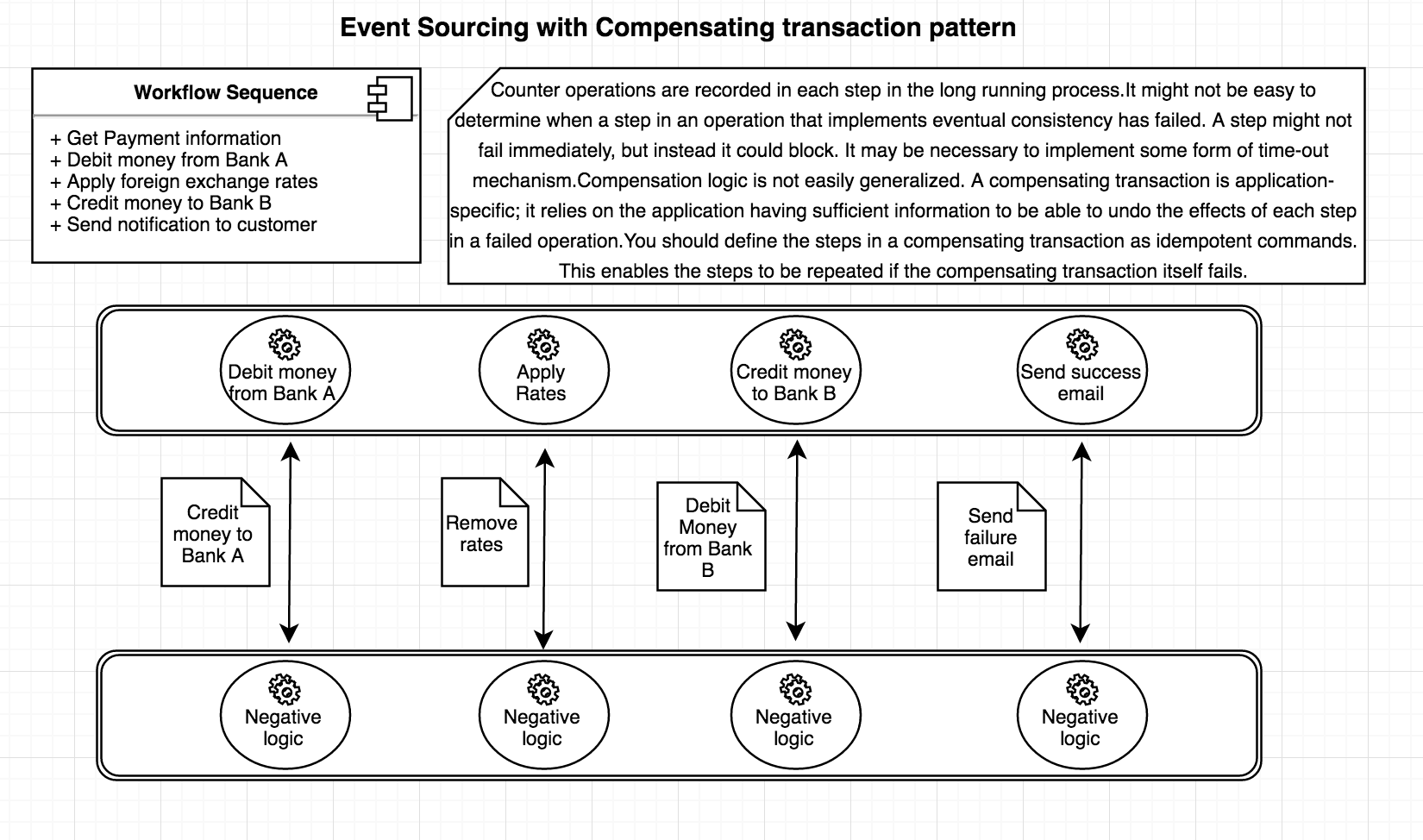
As application development, we have a tendency to model the data based on how they are stored, rather than how data is required. We tend to normalize our tables, put data within bounded context, create relations on data storage, or create series of aggregates. The chosen storage format is usually closely related to the format of the data, requirements for managing data size and data integrity, and the kind of store in use. But problem arises, when there are huge reader queries for a subset of data, involves unnecessary extraction which may require huge complex processing of query.

1. Create materialized views from the source data to serve the client.
2. The views can be created based on UI/UX experience.
3. Consistency must be essential while using this pattern, on any update of the data, the view must be updated back to reflect the current state of data.
4. The views can also be cached in distributed cache, when data changeability is less frequent.
5. The views can be based on screen, UI/UX design for faster lookup. Bridging the disjoint when using different data stores based on their individual capabilities.
6. Index Table pattern can be used to create secondary indexes over data sets for data stores that do not support native secondary indexes, this helps to query data examining current fields.

Not prescribed, when model/source data is relatively easy.

Not prescribed, when model/source data changes very frequently, as there is additional overhead to create and maintain the views.

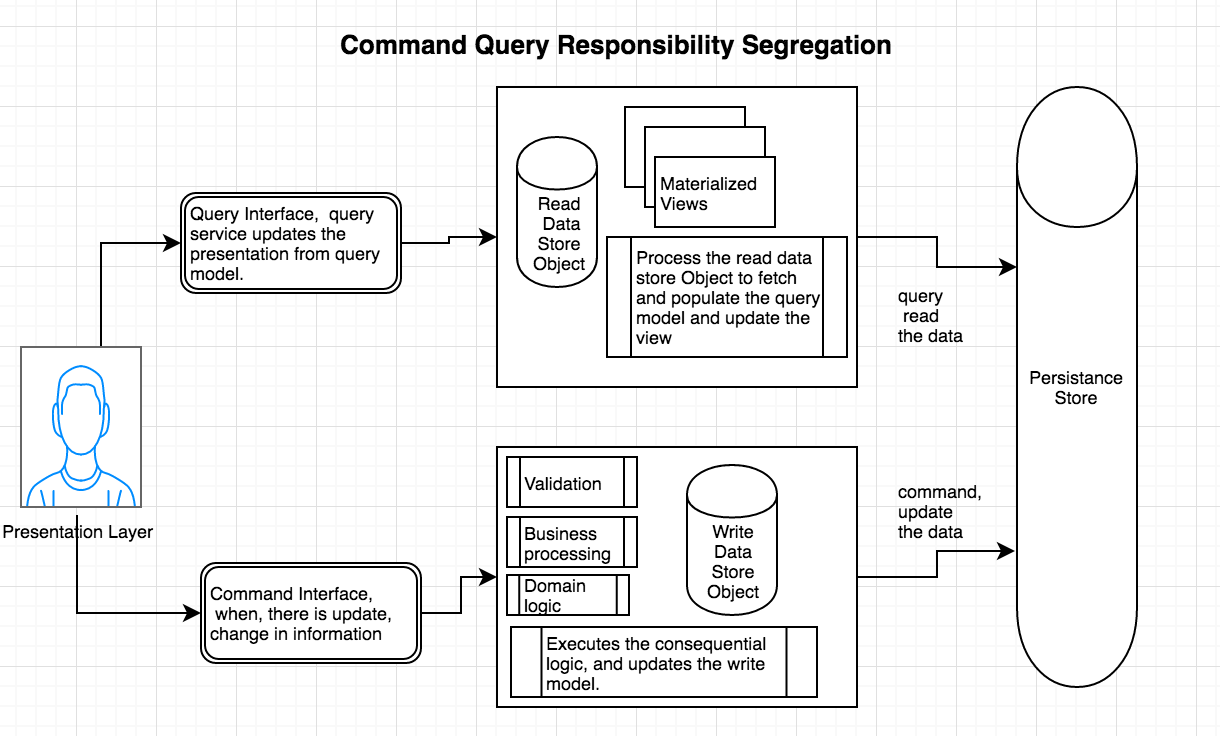
1. **Event Sourcing with compensating transaction pattern**



In an enterprise application, we have steps where we need to process a large workflow, where there are various moving items and touch points. This data may be spread across a span of data sources held in a variety of geographic locations. To avoid contention and improve performance in a distributed environment such as this, an application should not attempt to provide strong transactional consistency. Rather, the application should implement ***eventual consistency***. A significant challenge in the eventual consistency model is how to handle a step that has failed irrecoverably. In this cases it may be necessary to undo all of the work completed by the previous steps in the operation.

* There must be negation/ antagonistic operations for each of the atomic operations being performed to implement eventual consistency.
* The system that handles the step in original operation, the compensating operation must be resilient. It should not lose information.
* Event sourcing needs to be implemented and events once written can never be edited/updated to support data consistency.
* Time out locks must be placed across to complete the operation, and ready available of the resource to be taken care to complete seamless completion.
* Consider using retry logic that is more forgiving than usual to minimize failures that trigger a compensating transaction.
* Complete Rebuild, Temporal Query, Event replay needs to be achieved to guarantee proper event sourcing. Snapshots may be taken when large workflow needs to be taken care.

1. **Command Query Responsibility Segregation (CQRS)**



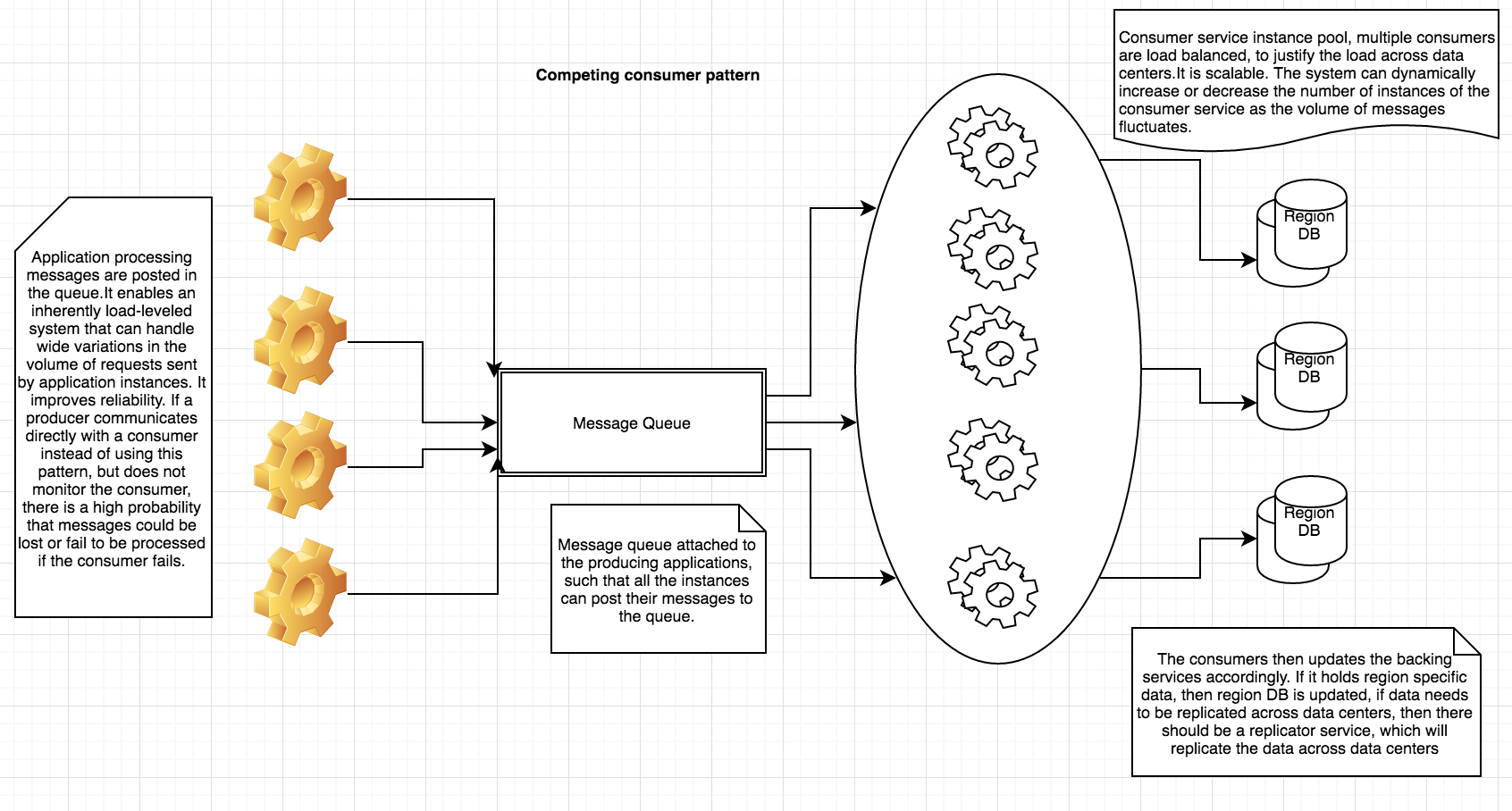
In traditional data management systems, both updates to a data(command) and request for data (query)are executed against the same set of data model in a single data repository. Typically, we create CRUD (create, read, update, delete) operations applied to same data representation of entity. But we frequently land up into multiple problems, historical data for delete/update operations, mismatch between read and write representation of data, data conventions in collaborative design, managing security and locks as same data set can be independently subject to read write operations. To solve,

1. Command and Query Responsibility Segregation (CQRS) is a pattern that segregates the operations that read data (Queries) from the operations that update data (Commands) by using separate interfaces.
2. The read store should be used to perform queries. Materialized view can be used for presentation formatted data entity and maximize performance.
3. Read store can be replicated and can be placed geographically closer to the application.
4. Read or write stores can scaled independently depending on the load.
5. Write operation can perform validation, business processing, domain logic before saving in persistence store. Event sourcing can be placed to track the commands usage. We can also replay the commands to understand client behavior and usage of system, where the temporal failure of one subsystem should not affect the availability of the others.
6. horizontal scaling can be easily done, in scenarios where performance of data reads must be fine-tuned separately from performance of data writes, especially when the read/write ratio is very high.
7. Very helpful pattern where one team of developers can focus on the complex domain model that is part of the write model, and another less experienced team can focus on the read model and the user interfaces and presentation.

* Not recommended, when the domain or the business rules are simple,
* Where a simple CRUD-style user interface and the related data access operations are sufficient.

1. **Competing consumer pattern**

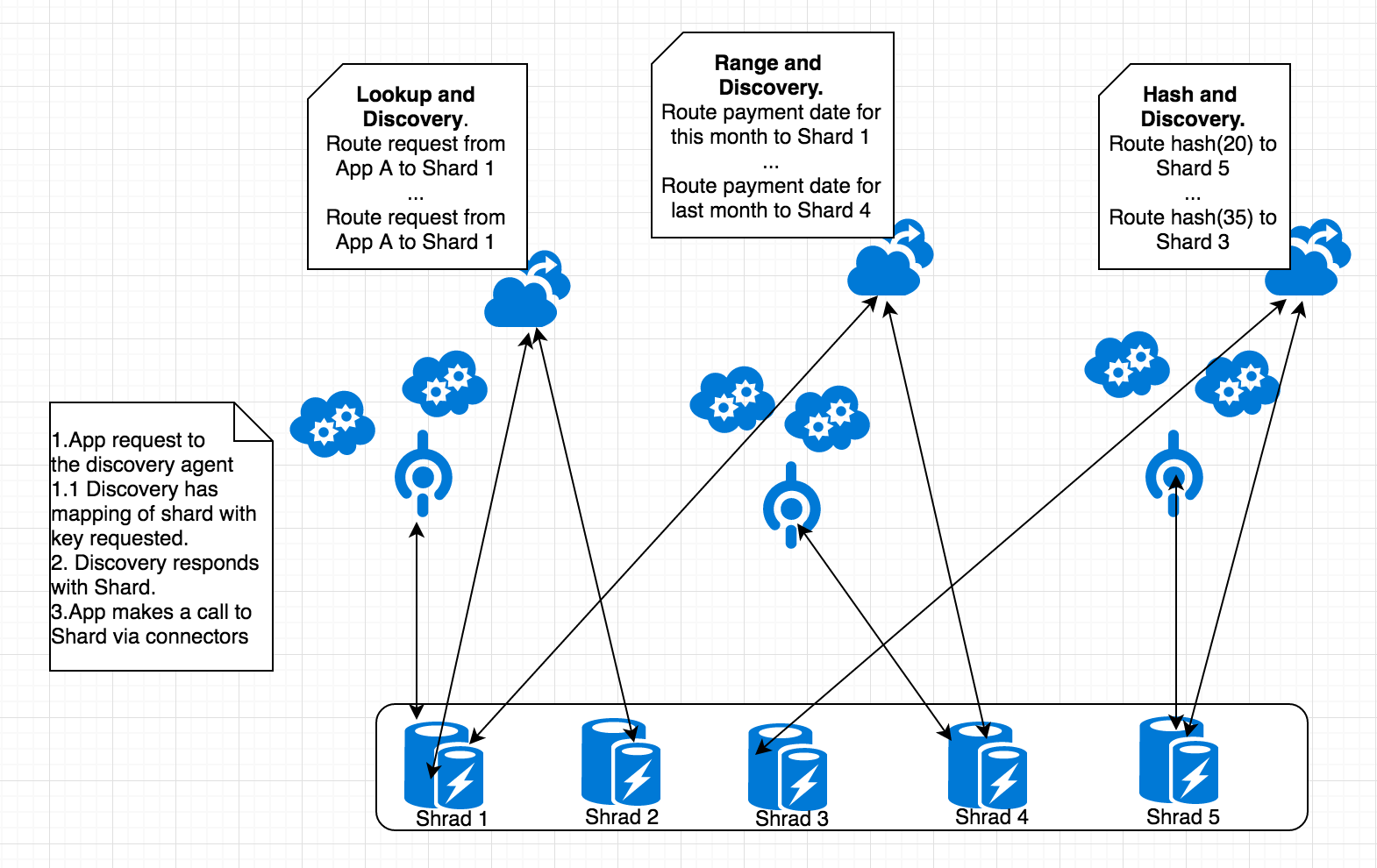
Typical messaging pattern when load balanced consumers are used to work on messages.



1. **Create read update delete pattern**

|  |  |
| --- | --- |
| ../Screen%20Shot%202017-02-01%20at%203.40.19%20PM.png | This is a typical pattern which is used widely but **strongly advised not to use unless**   1. Your data entity is relatively simple. 2. Your queries are very straight forward. 3. Event loss does not matter to you. 4. Data history retention is of no concern. 5. Data is very well defined in its own context |

1. **Sharding strategies, lookup, range, hash.**



A data store may be subject to many limitations, storage space, geography, bandwidth, latency, infrastructure and resources. **Sharding** is a type of database partitioning that separates very large databases the into smaller, faster, more easily managed parts called data shards. Each shard has the same schema, but holds its own distinct subset of the data. A shard is a data store in its own right (it can contain the data for many entities of different types), running on a server acting as a storage node. Sharding can also be considered based on application pattern in multi-tenant system, - based on workload, transactions per second, geographic regions, data isolation and privacy and regulatory requirements.

We can choose from couple of strategies which using the pattern.

Lookup discovery:

1. A shard key is being associated with each shard.
2. Apps request the lookup discovery to return the shard associated with the key.
3. Apps connect the shard via cloud connectors.
4. Multiple tenants might share the same shard, but the data for a single tenant will not be spread across multiple shards. The shards can be replicated to prevent data loss.
5. The mapping between the shard key and the physical storage may be based on physical shards where each shard key maps to a physical partition. Alternatively, a technique that provides more flexibility when rebalancing shards is to use a virtual partitioning approach where shard keys map to the same number of virtual shards, which in turn map to fewer physical partitions.

Range and discovery:

1. Related items are mapped in specific shards.
2. Approach of bounded model can be used here. Some data stores support two-part shard keys comprising a partition key element that identifies the shard and a row key that uniquely identifies an item within the shard.
3. Items subjected to range query, must be essential grouped together with same value of partition key but a unique row key.

Hash and discovery:

1. Aims at evenly distribute data across shards and also balances data across shards.
2. Item is stored based on hash on one or more attributes of data.
3. The discovery is aware of the hash map to efficiently route the request based on hash.
4. The Sharding logic computes the shard in which to store an item based on a hash of one or more attributes of the data.
5. The chosen hash function should distribute data evenly across shards. Some random element can add spice to the function.

**Interesting questions:**

While creating this document, I have listed down some interesting questions, which will help CCoE to shape and evolve and prescribe solution to common day to day problems. Couple of these questions can be answered with the design pattern as specified above. Am in the process of updating the document, so that we can have a place to support enterprise problems on data access.

1. Are we ready for a public cloud?
2. Are we ready for a single international deployment continuously?
3. Is my org structure ready for a devops culture?
4. Data location and stores. Distributed data processing?

Sample Problem question: I am cloud ready and looking to move to public cloud, I have my data centers in three locations. How do I move forward?

1. Data access.

Sample Problem question: I have very secure PII customer data. They have been warehoused in data centers with high firewall and security? Can I move to public cloud keeping all the above in mind, and what’s is my roadmap to achieve the same?

1. Data marshaling and processing.

Sample Problem question 1: I have a very huge normalized data in large volumes. Where do we place our processor? Should I do it in cloud, or produce formatted results?

Sample Problem question 2: We are a trading firm and we need to process huge volumes with minimal latency?

1. Event driven stores

Sample problem question: I want to maintain the entire workflow of a sequence right from origination to end. How can I move to event driven stores and what is my road map to cloud?

1. Event sourcing, CQRS and CRUD patterns.
2. Data messaging.

Sample problem question: How do we simplify messaging and evolve with maximum throughput?

1. Data domain modelling and decomposition

Sample problem question: I have a huge big monolith database. How can I align my modelling to my decomposed structure of micro services? Am I expecting huge TPS if I keep my data store intact?

1. Data latency and corruption.

Sample problem question: I need to serve clients all over the world? Do I have to buy datacenters throughout the place to minimize latency? How do I scale up processing keeping integrity in mind?

1. Domain driven modelling and accelerate.

Sample problem question: I am starting up a new venture in agile and my road map is MVP. How can cloud help me to drive a domain driven model?

1. Data mining from logs and create value propositions and customer centric ideas.

Sample problem question: I log the entire customer journey. How can I generate customer centric ideas?

1. Data regulations.

Sample problem question: I have lot of regulatory data. Can I move to public cloud?