Key requirements:

1. API for viewing product catalog
2. API for submitting orders
3. API for viewing order status
4. Data should be saved to a local data store
5. At the same time, an order fulfillment request is to be sent to a Redis server
6. Create nodejs service to monitor fulfillment events

Key quality attributes (and assumptions):

1. Performance – Under normal operation, the system is able to handle writes at 500 orders/day and reads at 20 rps.
2. Fault tolerance – The system should be able to function in case of application failure or database failure.

Evaluation of database (SQL or NoSQL):

SQL:

Pros:

1. Schema is pre-determined and restricted: doesn't matter in this application, schema is to be decided from scratch by developer
2. Good for complex queries where aggregation operations like max, min, etc. are required to be performed

Cons:

1. Harder to scale
2. Sharding/replication is a pain
3. Enabling fault tolerance is hard and requires a lot of manual steps

NoSQL:

Pros:

1. Schema is dynamic
2. Easy to scale and perform sharding or replication
3. In-built mechanisms for fault tolerance

Cons:

1. Cannot perform complex query operations such as joins and aggregation functions
2. Data type checks are not as strict as SQL

Decision:

There are going to be more reads in the system than writes so replication/sharding should be used to increase performance. The data can be stored either in an unstructured or structured form so in this case either SQL or NoSql can be used. If we are using replication/sharding, then there has to be some form of consistency for propagating writes into the database. Since we dealing with product orders, strict consistency should be maintained because users want to see whether their order was successful immediately after placing an order. Keeping these things in mind a NoSQL datastore is the better choice and since we might need to query our database on a number of attributes like product name, price, order status, etc. a document based data store such as **Mongodb** appears to be the best choice. Mongodb also stores documents in a JSON format which is the data format that is most common in APIs so it will spare us the need to perform format conversions.

MongoDB collection design:

Though a number of attributes can be created for products and orders, to keep the collection design and API implementation simple for demonstration, only a handful of attributes are chosen.

products:

```
[
  {
    "_id": "566a129cc269d66b56000006",
    "price": 100,
    "name": "product1",
    "__v": 0
  }
]
```

orders:

```
[
  {
    "_id": "566a1cddcf5cedb068000001",
    "prodsPurchased": [
      {
        "name": "product1",
        "price": 100,
        "_id": "566a129cc269d66b56000006"
      },
      {
        "name": "product2",
        "price": 200,
        "_id": "566a12a3c269d66b56000007"
      }
    ],
    "status": "Received",
    "__v": 0
  }
]
```

In the design for Order collection, data for products ordered is duplicated from products collection. This has a trade-off between integrity and performance. For performance reasons this design was chosen. Another reason for this is to keep track of the price at which the products were purchased. For e.g. If we just keep a reference of productIds purchased in an order and over time if the price of that product increases, the order's price will also increase but we don't want that.

API Design:

PRODUCT

Endpoint	HTTP method	Parameters	Description
/api/product	GET	-	Returns all products (or

			product catalog)
/api/product	POST	name price (for e.g. name=product1 and price=100)	Creates a new product with 'name' product name and given 'price'

ORDER

Endpoint	HTTP method	Parameters	Description
/api/order	GET	-	Returns all orders
/api/order/:orderId	GET	-	Returns the order with _id : orderId
/api/order	POST	productIds (for e.g. productIds=566a12a3c269d66b56000007,566a12b1c269d66b56000009)	Creates a new order containing details of products purchased with product ID in 'productIds'

<u>Handling errors/failure scenarios:</u>
Errors/failures handled:

- Incomplete or no parameters specified in Product POST
- Incomplete or no parameters specified in Order POST
- Error while saving a product or order
- Error while fetching product details during Order POST
- No products being found on fetching product details during Order POST
- No order found on Order GET for a specific Order ID
- Not being able to connect to Kinvey
- Error while saving order in Kinvey datastore

Errors/failures to be handled:

- MongoDB or Redis connection error (What should be done in that case?)
- App goes down: Can be solved by running multiple instances of app behind an ELB
- Local datastore can't be reached: Store data in a log file or a queue and try to save again when the datastore comes online

<u>Testing strategy:</u>
The first step in testing will be unit testing wherein individual API URLs are tested to find if they function normally in isolation and also if they behave as expected in edge or boundary cases. We can then proceed with functional testing in which we can verify real usage scenarios to observe the expected output. After this we can perform integration testing for the entire end to end system. These tests can be automated using tools such as SoapUI or frameworks such as Mocha. After we have tested the API, we can run some load and stress tests to achieve our required benchmarks or find and improve upon the problems identified. If security is also to be tested, we can perform penetration testing and/or fuzz testing as well.

Performance implications for a high volume of requests:
Our local datastore i.e. mongoDB can be easily scaled horizontally using replica sets to handle a high volume of requests. Our application can also be scaled by deploying it on multiple machines and having a load balancer redirect requests to these nodes. To use this setup, we will need to separate the application and database into two separate tiers so that each application node can interact with a mongos (mongoDB sharding) service to distribute load at storage tier.

How to make the app fault tolerant:
A critical part of the app where the fault can occur is when redis publishes a order fulfillment message but the app which listens to those messages is dead. In this case, we will be losing a huge amount of data if the app is down for long and no data will be saved in kinvey data store during this time. A simple solution to tackle this is to have a secondary fulfillment event listener which monitors the primary event listener app periodically and when it detects failure of primary, it acts as primary. Once the old primary is online again, it switches its role to secondary and starts monitoring the new primary. The number of secondaries can be increased to provide a higher degree of fault tolerance.

Possible improvements in future:

- In future product catalog might become very huge and retrieving all products in a single GET will be time-consuming and resource heavy, so pagination with offset can be implemented in GET query.
- Users might need to filter a query based on attributes, this can be added later.
- Modification of order status is currently not supported.