IE Cheatsheet - MAP1

May 13, 2025

Contents

1 Definitions

1.1 Enterprise Information Systems (EISs)

Can be defined as "software systems for business management, encompassing modules supporting organisational functional areas such as planning, manufacturing, sales, marketing, distribution, accounting, financial, human resources management, project management, inventory management, service and maintenance, transportation and e-business". They are made of computers, software, people, processes and data.

1.2 Customer Relationship Management Systems (CRMs)

There is a need to know the costumers (in large businesses, too many customers and too many ways customers interact with a firm). CRM systems capture and integrate customer data from all over the organization, consolidate and analyze customer data, distribute customer information to various systems and customer touch points across the enterprise, and provide a single enterprise view of customers.

1.3 Supply chain management systems (SCMs)

- Push-based model (build-to-stock) - Schedules based on best guesses of demand
- Pull-based model (demand-driven) - Customer orders trigger events in supply chain
- Sequential supply chains - Information and materials flow sequentially from company to company
- Concurrent supply chains - Information flows in many directions simultaneously among members of a supply chain network

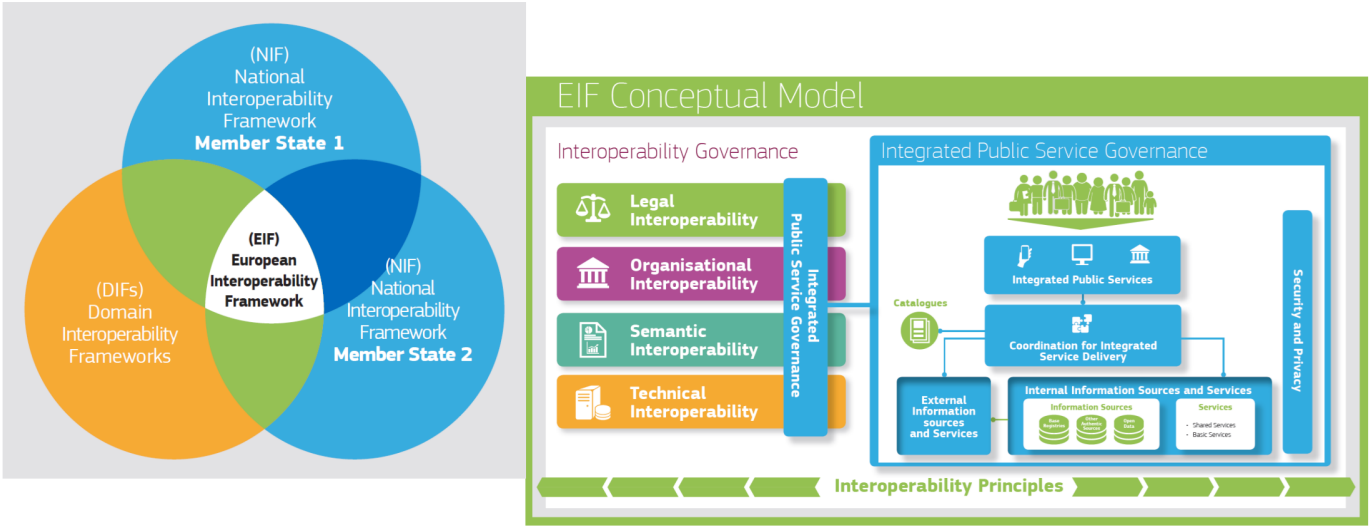
1.4 Enterprise Integration

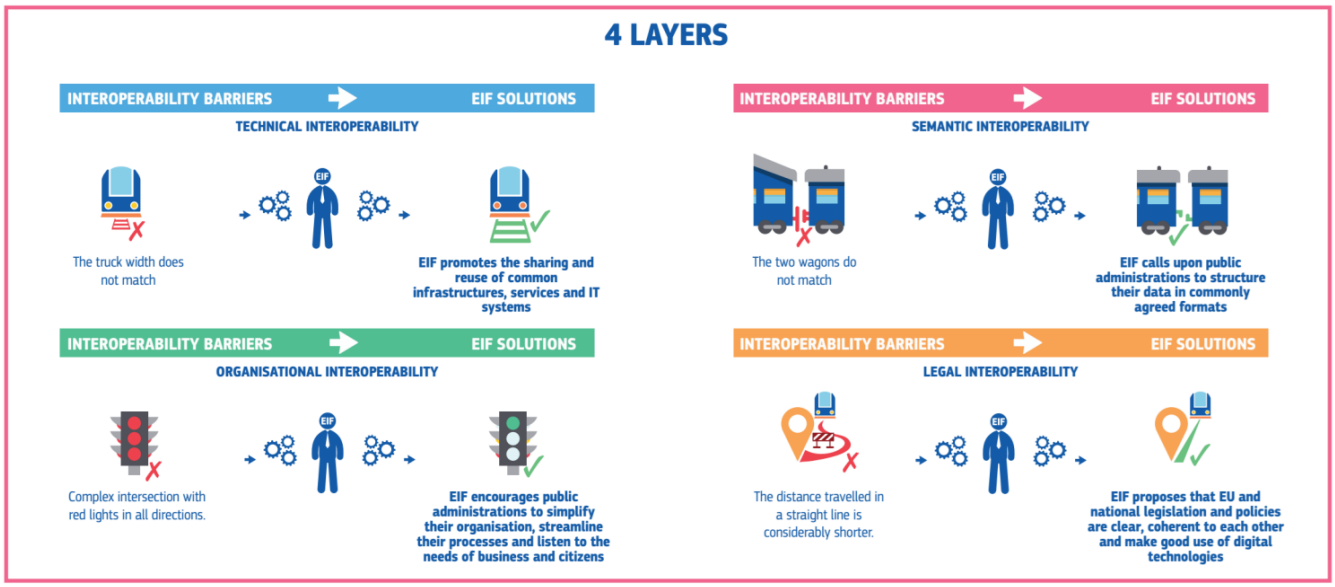
It is the process of ensuring the interaction between enterprise entities necessary to achieve domain objectives. Can be approached in various manners and at various levels, e.g., (i) physical integration,(ii) application integration, (iii) business integration, (iv) through enterprise modeling, and (v) as methodological approach to achieve consistent enterprise-wide decision-making.

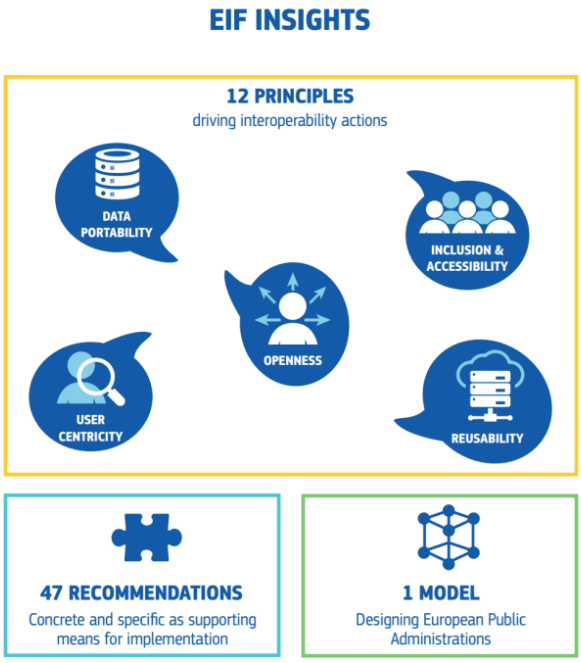
1.5 Enterprise Interoperability

It is the ability for two systems to understand one another and to use functionality of one another. "inter-operate" implies that one system performs an operation for another system. In the context of networked enterprises, interoperability refers to the ability of interactions (Exchange of information and services) between enterprise systems. Interoperability is considered significant if the interactions can take place at least on three different levels: data, services, and processes, with a semantics defined in each business context.

1.6 ISA - Interoperability Solutions for European Public Administrations







1.7 Integration vs. Interoperability

- **Interoperability** has the meaning of coexistence, autonomy, and federated environment
  - "Loosely coupled" means that the components are connected by a communication network and can interact; they can exchange services while continuing locally their own logic of operation
- **Integration** refers more to concepts of coordination, coherence and uniformization
  - "Tightly coupled" indicates that the components are interdependent and cannot be separated
- Two integrated systems are inevitably interoperable, but two interoperable systems are not necessarily integrated

1.8 Integrated information systems classes

Integrated information systems can be divided into three main classes:

1. **Interfaced systems** representing the weakest (but still widely used) form of integration because systems can only exchange data using predefined exchange protocols and data schema (e.g. Comma-Separated Value (CSV) files over FTP (File Transfer Protocol), XML files via TCP/IP and SOAP, SQL schemas over DBlink in the case of Oracle applications, etc.),

2. **Tightly-coupled systems** integrating all data sources by creating logical mappings between them using standardised hard-coded interfaces and predefined global schemata and requiring so-called integrating infrastructures such as Enterprise Application Integration (EAI) platforms (Linthicum, 2000) and, in between,

3. **Loosely-coupled systems** coordinating autonomous component data sources and software applications with a set of federated schemas and open data exchange formats and protocols, preferably XML formats and using, for instance, Enterprise Service Buses (ESB) for message routing (Chappell, 2004). The latter case equates to interoperable enterprise information systems. Of course, there could be many intermediate gradations of IS integration between these two extrema (i.e. interfaced and tightly-coupled).

1.9 Most referred technology for integration

1.9.1 File Transfer

Integration tools have typically a mechanism for transferring and transforming files with various formats:

- **Flat file**, e.g., Comma-separated Values, is an extremely common flat-file format they're easily consumed by Google Spreadsheet, Microsoft Excel, and countless other applications
- **Structured files** - XML and JSON files - file transfer universal solving practically all heterogeneity problems

Stages of communication Encoding – information (object) to file — File transfer — Decoding – file to object
Advantages: All operating systems and programming languages support files. Many applications have ways of exporting or importing files. Support disconnected interaction.
Disadvantages: The complexity of encoding and decoding increases exponentially with complexity of the information to transfer. Performance is limited.

1.9.2 Capturing User Interface – Screen scraping

Extract information directly from the user interface of an application. Integration steps: 1. Define the screens to use. 2. Create a template indicating input and output fields. 3. Replace the terminal with a system that simulates a user, sending and receiving data for each screen.
Advantages: Suitable for integrating applications with no internal information. For example, COBOL programs on legacy mainframes. No changes needed in the application. No direct access to the application data.
Disadvantages: The application interfaces were not designed for integration purposes. It is not trivial for a program to simulate a user. The user interface may be volatile. Performance is low. Can be unstable due to communication problems, server availability, etc.

1.9.3 Web Scraping

Screen scrapers extract information from HTML and other markup languages. Browsers and other web crawlers use many scraping techniques. However, most web pages are intended for human consumption and often mix contents with presentation. Due to widespread scraping, several anti-screen scraping techniques were developed.

1.9.4 Synchronous messages/communication

In synchronous message passing, the components are processes, and processes communicate in atomic, instantaneous actions called rendezvous. If two processes are to communicate, and one reaches the point first at which it is ready to communicate, then it stalls until the other process is ready to communicate. "Atomic" means that the two processes are simultaneously involved in the exchange, and that the exchange is initiated and completed in a single uninterruptable step. In the synchronous mode problems arise when communication links or communicating threads are in an erroneous state (broken links, threads in infinite loops etc.) and thus communicating threads remain blocked, since communication cannot be initiated or completed.

Limitations: Both applications must be running simultaneously – time coupling, Blocking period exist, Fault on communication, Fault on threads, and Different receiver/production capacity rate lead to message loss.

1.9.5 RPC and RMI Remote Procedures and Remote Methods

RPC (Remote Procedure Call) middleware enables one application to trigger a procedure in another application - running on the same computer or on a different computer or network - as if both were part of the same application on the same computer. RPCs are represented on different operating systems, including most Unix and MS Windows systems. "Windows NT, for example, supports lightweight RPCs across processes and, with DCOM, full RPCs."
Advantages: RPC has a good heterogeneity support, because "RPC has bindings for multiple operating systems and programming languages." Marshalling and unmarshalling are automatically generated, thus simplifying the development.

Disadvantages: RPCs don't support group communication. They have no direct support for asynchronous communication, replication and load balancing, therefore leading to a limited scalability. Fault tolerance is worse than by other middleware types, because "many possible faults have to be caught and dealt with in the program."

1.9.6 Web Protocol

Many of the released web protocols were used throughout the years to integrate systems. It consists of well-documented connectivity, yet the complexity depends on the details of each protocol.

• Web services for application integration, e.g.:

• Simple Object Access Protocol (SOAP) - is a standard protocol originally designed to enable communication between applications developed in different languages and platforms.

• Representational State Transfer (REST) - A stateless client/server protocol: each HTTP message contains all the information needed to understand the request. As a result, neither the client nor the server need to record any state of inter-message communications. In practice, many HTTP-based applications use cookies and other mechanisms to maintain session state (some of these practices, such as URL rewriting, are not allowed by the REST rule). A well-defined set of operations that apply to all information resources: HTTP itself defines a small set of operations, the most important of which are POST, GET, PUT and DELETE. Often these operations are combined with CRUD operations for data persistence, where POST does not exactly fit this scheme.

1.9.7 Data Oriented Integration

ODBC, JDBC (Open/Java Database Connectivity) – Independent API for database management systems. Strongly coupled, development effort always involved. An application is independent of the DBMS if it does not use specific aspects such as stored procedures, triggers, or SQL-specific commands. Remote processing imposes a performance penalty due to the way parsing and execution commands are made.

Object request broker (ORB) middleware acts as broker between a request from one application object or component, and the fulfilment of that request by another object or component on the distributed network. ORBs operate with the **Common Object Request Broker Architecture (CORBA)**, which enables one software component to make a request of another without knowing where other is hosted, or what its UI looks like - the "brokering" handles this information during the exchange.

Advantages: Simple both in Microsoft and Java platforms with ODBC and/or JDBC. Relatively low cost because it doesn't require rewriting applications. Most DBMS manufacturers provide drivers.

Disadvantages: A large organization may have hundreds of data bases making it difficult to create an architecture. The Data Schema must be known. Requires to have technical knowledge on database repositories because the operation can have serious consequences on the information. Data types can be different and there is need to transform them. The data is not validated by the application. A strong coupled integration – any changes affect the integration. Replicated data can become inconsistent.

1.9.8 API for Package Applications

API (application programming interface) middleware provides tools developers can use to create, expose and manage APIs for their applications so that other developers can connect to them. Some API middleware includes tools for monetizing APIs - enabling other organizations to use them, at cost. Examples of API middleware include API management platforms, API gateways and API developer portals.

1.9.9 Message-oriented middleware (MOM)

There are two different types of MOM: message queuing and message passing. Message queuing is defined as indirect communication model, where communication happens via a queue. Message from one program is sent to a specific queue, identified by name. After the message is stored in queue, it will be sent to a receiver. In message passing - a direct communication model - the information is sent to the interested parties. One favour of message passing is publish-subscribe (pub/sub) middleware model. In pub/sub clients have the ability to subscribe to the interested subjects. After subscribing, the client will receive any message corresponding to a subscribed topic.

MOM enables application components using different messaging protocols to communicate to exchange messages. In addition to translating - or transforming - messages between applications, MOM manages routing of the messages so they always get to the proper components in the proper order.

The publish/subscribe MOM works slightly differently. This MOM is an event-driven process. If a client wants to participate, it first joins an information bus. Then depending on its function as the publisher, subscriber, or both, it registers an event listener in the bus. The publisher sends a notice of an event to the bus (on the MOM server). The MOM server then sends out an announcement to the registered subscriber(s) that data is available. When the subscriber

requests from a specific publisher some data, the request is wrapped in a message and sent to the bus. The bus then sends an event to the publisher requesting the data.

Advantages: The capability of persistently storing messages (Store forward): Buffering messages decouples the rate of production and the rate of consumption, unblocking the sender. Allows to cope with the unavailability of the Service. Tolerates temporary crash faults of the service. Act as a broker to distribute messages accordingly to different routing patterns.

Disadvantages: Asynchronous communication implies a less intuitive programming model (like event programming) than the request-response paradigm. Message queues need to be supported and managed with additional support and investment costs.

1.9.10 Transaction Oriented or Transactional middleware (TM)

TM, or even, transaction processing (TP) monitors were designed in order to support distributed synchronous transactions. The main function of a TP monitor is a coordination of requests between clients and servers that can process these requests. The request is a 'message that asks the system to execute a transaction'. TM provides services to support the execution of data transactions across a distributed network. The best-known transactional middleware are transaction processing monitors (TPMs), which ensure that transactions proceed from one step to the next - executing the data exchange, adding/changing/deleting data where needed, etc. - through to completion.

1.10 Middleware

On the one hand, both a software and a DevOps engineer would describe middleware as the layer that "glues" together software by different system components; on the other hand, a network engineer would state that middleware is the fault-tolerant and error-checking integration of network connections. In other words, they would define middleware as communication management software. A data engineer, meanwhile, would view middleware as the technology responsible for coordinating, triggering, and orchestrating actions to process and publish data from various sources, harnessing big data and the IoT. Given that there is no uniform definition of middleware, it is best to adopt a field-specific approach.

This middleware "is an approach for developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API." **This means that middleware should not serve solely as an object-oriented solution to execute simple request-response commands.** But, middleware can incorporate pull-push events and streams via multiple gateways by combining microservices architectures to develop a **holistic decentralized ecosystem.**

1.11 Service Oriented Architecture (SOA)

The core unit is the service that encapsulates a business function, and services are explicitly defined with interfaces that are independent of implementation (promote the reuse). Services are loosely coupled and invoked through communication protocols that are independent of the location. Each service is instantiated in a single site and invoked remotely on this site by all applications that use it (no replicas with potential independent developments). There is no inheritance or strong dependencies between services. Each service is created (build) once but can be deployed to all systems that require it.

Segunda definição: It is a design approach where multiple services collaborate to provide some end set of capabilities. A service here typically means a completely separate operating system process. Communication between these services occurs via calls across a network rather than method calls within a process boundary. SOA emerged as an approach to combat the challenges of the large monolithic applications. It is an approach that aims to promote the reusability of software; two or more enduser applications, for example, could both use the same services. Goal to promote reusability: It aims to make it easier to maintain or rewrite software, as theoretically we can replace one service with another without anyone knowing, as long as the semantics of the service don't change too much.

Pitfalls: Communication protocols incompatibilities (e.g., SOAP), Lack of guidance about service granularity, Wrong guidance on picking places to split a system, Many specifications.

1.12 Enterprise Service Bus (EBS)

An ESB is a system for interoperating different application systems in a service-oriented architecture (SOA). It represents a distributed computing architecture and is a variant of the client-server model in which an application can assume both a client and a server role. The fundamental concept of an ESB is the fact that a communication "bus" is used and placed between the different applications. In this way, applications are decoupled from each other, operating in such a way that they do not need to be aware of the other systems present on the service bus. The complexity of transport services and

protocols are abstracted by the service bus, offering applications an easy way to interoperate with each other. The ESB concept was born out of the need to escape the point-to-point communication model (applications that interact directly with each other) that proved difficult to manage or monitor when the organization's infrastructure reaches a certain size. They also allow accelerating the integration between heterogeneous applications and thus respond to market needs more quickly.

1.13 Event-driven systems

Event-driven systems are software architectures where the flow of the program is determined by events — things that happen during the execution like user actions, sensor outputs, or messages from other programs. When an event occurs, a corresponding processing and actions are taken.

1.14 Hyperautomation

Hyperautomation is a business-driven approach that involves automating as many processes as possible. The goal is not just to automating tasks, but orchestrating entire workflows and enabling systems to make decisions, adapt, and evolve.

2 Messaging systems

2.1 Publish/Subscribe Messaging

Publish/subscribe (pub/sub) messaging is a pattern that is characterized by the sender (publisher) of a piece of data (message) not specifically directing it to a receiver. Instead, the publisher classifies the message somehow, and that receiver (subscriber) subscribes to receive certain classes of messages. Pub/sub systems often have a broker, a central point where messages are published, to facilitate this pattern.

2.2 Asynchronous communication

An asynchronous call performed through queues allows the caller to continue working while the request is processed. In this model, either the sender or the receiver can have the queue, or both can have one.

2.2.1 Time (de)coupling vs. asynchronous

Time decoupling and asynchronous communications are different concepts. Time coupling implies that both the sender and receiver of the information on an integration must be running simultaneously, or the communication fails, where the service invocation may be synchronous or asynchronous. In time decoupling there is no requirement that both are up and running, tolerating transient crash faults of the server. Obviously, in time decoupling, only an asynchronous communication pattern makes sense. Therefore, to have time decoupling, we need some form to persist messages.

2.3 MOM key concepts

2.3.1 Channel

Corresponds to message queues with unique name. Guarantees persistence. Point-to-point – dedicated channel to deliver unique message. pub/sub – a copy of a message for each client.

2.3.2 Message

Package of data that is passed through a channel. Could be: document, event, image, video, etc. To allow interoperability the same schema is required, otherwise message need to be converted.

2.3.3 Router

It is component in the messaging system that decides the destination of a message. Different messages may be handled by different applications. The routing decision is handled in the messaging system.

There are several types of routers: Message-type filter (Routes messages based on their type), broadcast routing (Sends the message to all connected receivers, regardless of content or type), content-based routing (Inspects message content to decide where to route it), and dynamic routing (receivers configuring, at runtime, rules for routing messages).

2.3.4 Translator

Translator provides the ability to convert message content from one structure into another. A transformation map defines the translations required. For example, counting the number of items is one transformation.

2.3.5 Endpoint

Endpoint is the software component that abstracts the sender and receiver application from the messaging system. For sending messages: invoke the send method with the desired mode (e.g., fireand-forget, synchronous or asynchronous). For receiving messages, wither **Pooling**, where a receiver is checking in the messaging system if there are any new message available or **Callback**, where a receiver is notified by the messaging system when any new message is available.

2.4 Advanced Message Queuing Protocol (AMQP)

Historically, there was a lack of standards governing the use of message-oriented middleware. Most of the major vendors have their own implementations, each with its own application programming interface (API) and management tools. The Advanced Message Queuing Protocol (AMQP) is an approved OASIS and ISO standard that defines the protocol and formats used between participating application components. AMQP specifies flexible routing schemes, including point-to-point, fan-out, publish/subscribe, and request-response, transaction management, queuing, distribution, security, management, clustering, federation and heterogeneous multi-platform support.

2.5 Kafka

2.5.1 Kafka and Zookeeper

ZooKeeper is a reliable storage running on a cluster of nodes. Kafka uses Apache ZooKeeper to maintain the list of the brokers and all the information related to partitions, offsets, etc. ZooKeeper can guarantee that the local replicas never diverge (maintain consistency). Zookeeper is like a huge index, it knows which broker has which message in a cluster.

2.5.2 MOM versus Kafka

MOMs provide Reliable communication in presence of transient faults on consumers, Buffering between producers and consumers, Publish/subscription and point to point channels. Kafka adds with High ingestion rate of messages, Distributes architecture tolerating faults from brokers, Disk retention policies.

MOMs normally provide Message transformation, Transactional messages, Automatic indexes for reading, Dynamic routing of messages. Kafka does not provide any of these features.

2.5.3 Commit log

A commit log is designed to provide a durable record of all transactions so that they can be replayed to consistently build the state of a system. Similarly, data within Kafka is stored durably, in order, and can be read deterministically. In addition, the data can be distributed within the system to provide additional protections against failures, as well as significant opportunities for scaling performance.

When in a Kafka cluster, the events are immutable, and the commit log they got to is append only.

2.5.4 Kafka Message

A message is simply an array of bytes as far as Kafka is concerned, so the data contained within it does not have a specific format or meaning to Kafka. If you are approaching Kafka from a database background, you can think of this as similar to a row or a record. A message can have an optional piece of metadata, which is referred to as a key. The key is also a byte array and, as with the message, has no specific meaning to Kafka. Keys are used when messages are to be written to partitions in a more controlled manner. For efficiency, messages are written into Kafka in batches. A batch is just a collection of messages, all of which are being produced to the same topic and partition. An individual round trip across the network for each message would result in excessive overhead, and collecting messages together into a batch reduces this. Of course, this is a trade-off between latency and throughput: the larger the batches, the more messages that can be handled per unit of time, but the longer it takes an individual message to propagate. Batches are also typically compressed, providing more efficient data transfer and storage at the cost of some processing power.

2.5.5 Kafka topic and partitions

Messages in Kafka are categorized into topics. The closest analogies for a topic are a database table or a folder in a filesystem. Topics are additionally broken down into a number of partitions. Going back to the "commit log" description, a partition is a single log. Messages are written to it in an append-only fashion and are read in order from beginning to end. Note that as a topic typically has multiple partitions, there is no guarantee of message ordering across the entire topic, just within a single partition.
From a performance point of view, it is the number of partitions that matters. But since each topic in Kafka has at least one partition, if you have N topics, you inevitably have at least N partitions. However, keep in mind that partitions have

costs (latency, more memory, more file descriptors, etc.).
Producers are typically much faster than consumers, then always calculate for consumers' expectations. What is the maximum throughput expected to achieve when consuming from a single partition? (Considering, always one consumer reading from a partition, so if you know that your slower consumer writes the data to a database that never handles more than 50 MB per second from each thread writing to it, then you know you are limited to 50MB throughput when consuming from a partition). If you are sending messages to partitions based on keys, adding partitions later can be very challenging, so calculate throughput based on your expected future usage, not the current usage. Avoid overestimating, as each partition uses memory and other resources on the broker and will increase the time for leader elections.

2.5.6 Use of Keys

Partition Number = hash(key) % #Partitions(topic) Keys are used when messages are to be written to partitions in a more controlled manner. The simplest such scheme is to generate a consistent hash of the key and then select the partition number for that message by taking the result of the hash module to the total number of partitions in the topic. This ensures that messages with the same key are always written to the same partition (provided that the partition count does not change).

2.5.7 Message offset

The offset—an integer value that continually increases—is another piece of metadata that Kafka adds to each message as it is produced. Each message in a given partition has a unique offset, and the following message has a greater offset (though not necessarily monotonically greater).

2.5.8 Kafka Producer

Producers create new messages. A message will be produced to a specific topic. By default, the producer will balance messages over all partitions of a topic evenly. In some cases, the producer will direct messages to specific partitions. This is typically done using the message key and a partitioner that will generate a hash of the key and map it to a specific partition. This ensures that all messages produced with a given key will get written to the same partition. The producer could also use a custom partitioner that follows other business rules for mapping messages to partitions. The reads from a single partition maintain the order in which the messages were produced but within a multiple partition Topic global order is not guaranteed.

Sending policies:

- **Fire-and-forget**
  - Sending a message to the server and do not really care if it arrives successfully or not.
  - Most of the time, it will arrive successfully, since Kafka is highly available, and the producer will retry automatically.
  - However, some messages will get lost using this method.
- **Synchronous send**
  - Sending a message and testing the acknowledge: the `send()` method returns a future object.
  - The sender can call `get()` to wait on the future and see if send was successful.
- **Asynchronous send**
  - Calling the `send()` method with a callback function, which gets triggered when it receives a response from the Kafka broker.
  - **Whenever the synchronous response take too long, or no error processing need to be done.**

2.5.9 Kafka Consumer

Consumers read messages. The consumer subscribes to one or more topics and reads the messages in the order in which they were produced to each partition. The consumer keeps track of which messages it has already consumed by keeping track of the offset of messages. By storing the next possible offset for each partition, a consumer can stop and restart without losing its place.

2.5.10 Kafka Consumer Group

Consumers work as part of a consumer group, which is one or more consumers that work together to consume a topic. The group ensures that each partition is only consumed by one member. In this way, consumers can horizontally scale to consume topics with a large number of messages. Additionally, if a single consumer fails, the remaining members of the group will reassign the partitions being consumed to take over for the missing member: Rebalance. The mapping of a consumer to a partition is often called ownership of the partition by the consumer. On rebalancing, Kafka needs to know the offset where the actual consumers were reading. Each consumer group have its own offset. Consumer groups remember the offsets where they left off. Create a new consumer group for each application that needs all the messages from one or more topics. Add consumers to an existing consumer group to scale reading and processing messages from topics. Obviously, it is bounded by the number of partitions in the topic.

2.5.11 Kafka broker

A single Kafka server is called a broker. The broker receives messages from producers, assigns offsets to them, and writes the messages to storage on disk. It also services consumers, responding to fetch requests for partitions and responding with the messages that have been published. Depending on the specific hardware and its performance characteristics, a single broker can easily handle thousands of partitions and millions of messages per second. A broker can define a retention period for data which can be configured by each Topic: a given period and a given size.

2.5.12 Kafka Cluster

Kafka brokers are designed to operate as part of a cluster. Within a cluster of brokers, one broker will also function as the cluster controller (elected automatically from the live members of the cluster). The controller is responsible for administrative operations, including assigning partitions to brokers and monitoring for broker failures. A partition is owned by a single broker in the cluster, and that broker is called the leader of the partition. A replicated partition is assigned to additional brokers, called followers of the partition. Replication provides redundancy of messages in the partition, such that one of the followers can take over leadership if there is a broker failure. All producers must connect to the leader in order to publish messages, but consumers may fetch from either the leader or one of the followers. In kafka cluster, the zookeepers need to know the address of each other and kafka needs to know the address of all zookeepers. Best size of the kafka cluster depends on the data storage available on each kafka brokes versus overall storage needed and the capacity to handle requests,

2.6 Kafka Streams

First and foremost, a data stream is an abstraction representing an unbounded dataset. Unbounded means infinite and ever growing. The dataset is unbounded because over time, new records keep arriving. Note that this simple model (a stream of events) can be used to represent just about every business activity we care to analyze.

Kafka streams are unbounded, ordered, have immutable data records and are replayable (to correct errors, try new methods of analysis, or perform audits).

Continuous and nonblocking processing: Stream processing fills the gap between the requestresponse world, where we wait for events that take two milliseconds to process, and the batch processing world, where data is processed once a day and takes eight hours to complete. Most business processes don't require an immediate response within milliseconds but can't wait for the next day either. Most business processes happen continuously, and as long as the business reports are updated continuously and the line of business apps can continuously respond, the processing can proceed without anyone waiting for a specific response within milliseconds. Business processes such as alerting on suspicious credit transactions or network activity, adjusting prices in real-time based on supply and demand, or tracking deliveries of packages are all a natural fit for continuous but nonblocking processing.

2.6.1 Topology

A stream processing application includes one or more processing topologies. A processing topology starts with one or more source streams that are passed through a graph of stream processors connected through event streams, until results are written to one or more sink streams. Each stream processor is a computational step applied to the stream of events in order to transform the events.

2.6.2 Time

- **Event time** - This is the time the events we are tracking occurred and the record was created.
- **Log append time** - This is the time the event arrived at the Kafka broker and was stored there, also called ingestion time.

- **Processing time** - This is the time at which a stream processing application received the event in order to perform some calculation.

2.6.3 State

- **To keep track of more information** (how many events of each type did we see this hour, etc.).

- **Local or internal state** - State that is accessible only by a specific instance of the stream processing application. This state is usually maintained and managed with an embedded, in-memory database running within the application. Extremely fast but limited memory.

- **External state** - State that is maintained in an external data store, often a NoSQL system like Cassandra. Virtually unlimited size and the fact that it can be accessed from multiple instances of the application or even from different applications. The downside is the extra latency and complexity introduced with an additional system, as well as availability—the application needs to handle the possibility that the external system is not available.

2.6.4 Table

To convert a table to a stream, we need to capture the changes that modify the table. Take all those insert, update, and delete events and store them in a stream. Most databases offer change data capture (CDC) solutions for capturing these changes, and there are many Kafka connectors that can pipe those changes into Kafka where they will be available for stream processing.

2.6.5 Time Windows

Most operations on streams are windowed operations, operating on slices of time. When calculating moving averages, We want to know the size of the window, how often the window moves (advance interval), how long the window remains updatable (grace period). There is a tumbling window (5-minute window every 5 minutes) or a Hopping Window (5-minute window every 1 minute. They overlap, so events belong to multiple windows.)

2.6.6 Stream Processing Design Patterns

Single-Event Processing, handled with a simple producer and consumer. Processing with Local State, done using local state (rather than a shared state), where it is stored in-memory using embedded DB, which also persists the data to disk for quick recovery after restarts. Multiphase Processing/Repartitioning (First, we calculate the daily gain/loss for each stock symbol. We can do this on each instance with a local state. Then we write the results to a new topic with a single partition. This partition will be read by a single application instance that can then find the top 10 stocks for the day. The second topic, which contains just the daily summary for each stock symbol, is obviously much smaller with significantly less traffic than the topics that contain the trades themselves, and therefore it can be processed by a single instance of the application).

3 Microservices

3.1 ArchiMate definitions

A business service represents explicitly defined behavior that a business role, business actor, or business collaboration exposes to its environment.
An application service represents an explicitly defined exposed application behavior.
A technology service represents an explicitly defined exposed technology behavior.

3.2 Microservices definitions

3.2.1 1

A microservice is a tiny (o store não gosta da palavra) and independent software process that runs on its own deployment schedule and can be updated independently.

A microservice is an independently deployable component of bounded scope that supports interoperability through message-based communication. Microservice architecture is a style of engineering highly automated, evolvable software systems made up of capabilityaligned microservices.

3.2.2 2

An approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms built around business capabilities and independently deployable by fully automated deployment machinery

3.2.3 3

Microservices are independently releasable services that are modelled around a business domain. A service encapsulates functionality and makes it accessible to other services via networks—you construct a more complex system from these building blocks. One microservice might represent inventory, another order management, and yet another shipping, but together they might constitute an entire ecommerce system. Microservices are an architecture choice that is focused on giving you many options for solving the problems you might face.

3.2.4 Key concepts

Run independently, Independent Deployability, Distributed data management (Owning their own state, Bindings to other a bounded context), Modeled around a Business Domain (Size is variable, could small or large. Depends on the business context), Flexibility, Alignment with the organization architecture, Command Query Responsibility Segregation (CQRS) (Read/Write your own domain data, Read-only representation of other domains data, Private data representation (might be different format).)

Drawbacks: Developers must deal with the additional complexity of creating a distributed system. Implementing use cases that span multiple services requires careful coordination between the teams. Developers must implement the interservice communication mechanism.

3.3 Asynchronous Event-driven Microservices

- **Granularity** - services map neatly to bounded contexts and can be easily rewritten when business requirements change.
- **Scalability** - individual services can be scaled up and down as needed.
- **Technological flexibility** - services use the most appropriate languages and technologies. This also allows for easy prototyping using pioneering technology.
- **Business requirements flexibility** - ownership of granular microservices is easy to reorganize. There are fewer crossteam dependencies compared to large services, and the organization can react more quickly to changes in business requirements that would otherwise be impeded by barriers to data access.
- **Loosely coupling** - event-driven microservices are coupled on domain data and not on specific implementation API. Data schemas can be used to greatly improve how data changes are managed.
- **Continuous delivery support** - it's easy to ship a small, modular microservice, and roll it back if needed.
- **High testability** - microservices tend to have fewer dependencies than large monoliths, making it easier to mock out the required testing endpoints and ensure proper code coverage.

3.4 Types of Coupling

Implementation coupling - This occurs when components or services rely on the specific internal details of each other's implementation. Changes in one component's code, data structures, or algorithms can directly force changes in the coupled components, even if the functionality remains the same. This tight coupling makes the system brittle and difficult to evolve independently.
Temporal coupling - Temporal coupling happens when components or services must execute in a specific order or at roughly the same time. If one component is unavailable or slow, it can directly impact the functionality of the others. This can lead to complex dependencies and challenges in scaling or deploying services independently.
Deployment coupling - Deployment coupling arises when components or services must be deployed together. Changes to one component necessitate redeploying others, even if they haven't changed. This hinders independent releases and can increase the risk and complexity of deployments.
Domain coupling - the desired coupling for microservices. Focuses on the relationships between services based on the business domain. Services collaborate because they operate on related concepts or participate in the same business processes. However, the key is to minimize the other forms of coupling (implementation, temporal, and deployment) between these domain-related services. This allows each microservice to evolve, scale, and be deployed independently while still working together to fulfill business needs

3.5 Reactive systems

- **Responsive** - able to handle requests in a timely fashion.
- **Resilient** - able to manage failures gracefully.
- **Elastic** - able to scale up and down according to the load and resources.
- **Message driven** - using asynchronous message-based communication among the components forming the system.
- **Commands** - actions that a user wishes to perform. In general, commands are sent to a specific receipient, and a result is sent back to the client.
- **Events** - Events are actions that have successfully completed. Represents a fact, something that happened. Events are immutable. To refute a previously sent fact, you need to fire another event invalidating the fact.
- **Message** - A message is a self-contained data structure describing the event and any relevant details about the event, such as who emitted it, at what time it was emittes, and potentially its unique ID.
- **Time decoupling** - Asynchronous message passing also enables time decoupling. Events are not lost if there are no consumers. The events are stored and delivered later. Time decoupling increases the independence of components, and keeps coupling to a minimum

3.6 Nonblocking I/O (Reactor, proactor pattern)

- **Blocking network I/O** - synchronous communication where a client and the server connect before interaction starts. Communication is blocked until the operation completes.
- **Multithread blocking network I/O** - Execute concurrent requests having multiple threads. Resources expended waiting for the clients requests. Concurrency limited by the number of threads available.
- **Nonblocking network I/O** - the system enqueues I/O operations and returns immediately, so the caller is not blocked. When the response comes back, the system stores the result in a structure. When the caller needs the result, it interrogates the system to see whether the operation completed.
  - **Continuation-passing style (CPS)** - style of programming in which control is passed explicitly in the form of a continuation (usually a callback).

Gives the possibility to handle multiple concurrent requests or messages with a single thread. The **reactor pattern** allows associating I/O events with event handlers. Invokes the event handlers when the expected event is received. Avoiding the creation of a thread for each message, request and connection.
The **Proactor pattern** can be seen as an asynchronous version of the reactor. Useful when long-running event handlers invoke a continuation when they complete. Such mechanisms allow mixing nonblocking and blocking I/O. (Receives event, sends ok to client, later callback the client when it has forwarded to the event handler).

3.7 Common architecture

• Bottom layer, handles client connections, outbound requests, and response writing.
• Middle layer, provides easier and high-level APIs such as HTTP requests, responses, Kafka messages.
• Top layer, the code developed by you, that is just a collection of event handlers. Uses the features provided by the reactive framework to interact with other services or middleware.
• However, your code cannot block the event loop thread, otherwise the architeture will be blocked!
• Design and develop non-blocking code, always!

3.8 Quarkus Framework

3.8.1 Imperative Model

The order of the commands cannot be changed or the result will be different. Sometimes, is the only possible way! However, in between sending a request to the database and receiving the response, what is the I/O thread doing? It is just being idle. Many inefficiencies.

3.8.2 Reactive Model

Instead of an I/O thread waiting, it begins processing another incoming request. It continues to do so until it's been notified that a database response is ready for processing. How? A continuation (a callback) is provided to process the database response Basically, changes context when it starts to become idle and does other things.

3.8.3 Quarkus supported models

• A developer's choice of imperative or reactive is an API choice, and not a framework one
• Quarkus reactive model is always non-blocking, relying on Eclipse Vert.x
• Imperative model requires the execution by a worker thread and not I/O thread. Offload from Quarkus is done using context switch. However is cost time and resources • Yet, Third option!
• Non-blocking an blocking handlers can coexist, as long as we offload blocking execution onto worker threads and invoke continuations when a blocking handler completes.
• Using I/O thread for as much work as possible
• Using the @Blocking and @NonBlocking annotations

3.8.4 RESTEasy

With RESTEasy Reactive the annotations @Blocking and @NonBlocking can be used to indicate on which threads the request should be handled. By default, @NonBlocking uses an I/O Thread. A method returning Uni or Multi is executed on an I/O thread except if annotated with @Blocking that uses a worker Thread. Methods returning any other object is executed on a worker thread, except if the @NonBlocking annotation is used.

	Annotation required to the method returning Uni or Multi	Annotation required to the method returning any other datatype
I/O Thread	@NonBlocking (it's by default)	@NonBlocking
Worker Thread	@Blocking	Default: @Blocking

3.8.5 Mutiny = Multi and Uni

SmallRye Mutiny is the reactive programming library of Quarkus. Mutiny is built around three key aspects:
• Event-driven: listening to events from stream and handling them appropriately
• Easily navigable API: Navigating the API is driven by an event type and the available options for that event
• Only two types: Multi (data streams) and Uni (single result) can handle any desired asynchronous actions

3.8.6 Reactive Object-relational mapping (ORM)

• Three distinct models available:
• Blocking database client
• Non-blocking database client
• Non-blocking database with pipelining – database shared connection (Several requests can be done while waiting for one to finish in the same thread) (for compatible databases)
Hibernate ORM enables developers to more easily write applications whose data outlives the application process. As an ORM framework, Hibernate is concerned with data persistence as it applies to relational databases. Hibernate provides both imperative and reactive APIs. These APIs support two facets:
• non-blocking database clients, and
• reactive programming as a means of interacting with relational databases.

3.8.7 Reactive Messaging: Quarkus with Kafka

• Quarkus offers a message-driven development model that is simple but powerful to consume, process, and create messages
• Old JAVA JMS is not fully compatible with Kafka and is a blocking API which do not allow the implementation of reactive principles
• Reactive Messaging can send, consume, and process messages in a protocol-agnostic way
• Messages transit on channels

4 Perguntas

4.1 What is it and explain in detail how it works

Terraform state persistency: It's how Terraform tracks infrastructure. The state file stores resource mappings and metadata. It allows Terraform to compare real infrastructure with config files to determine changes to apply.

REST API: A REST API (Representational State Transfer) is a web service architecture that uses HTTP methods (GET, POST, etc.) and stateless communication, returning resources typically in JSON.
Processing rate between Kafka Producer and Kafka Consumer, which one has more computational consumption: Generally, Kafka Consumers have higher computational consumption because they handle deserialization, processing logic, offset management, and possible storage, while Producers mostly serialize and send.
Kafka Consumer Group rebalance: Occurs when consumers join/leave a group or topic partitions change. Kafka reassigns partitions to consumers to ensure balanced consumption, temporarily pausing message processing. It keeps track of the offset of each member so after rebalancing, they can keep going where it was left off.
How to size the number of Kafka brokers of a Kafka cluster: Based on: desired parallelism (number of consumers), throughput, and key distribution. More partitions = higher parallelism, but also more overhead. Start small and scale with tests.

Quarkus unification of reactive and imperative models? From a programmatically point of view, is it possible to use both models in Quarkus? How to do it in the JAVA code? Yes. Quarkus allows both non-blocking (reactive) and blocking (imperative) handlers to coexist. You can offload blocking logic to worker threads using @Blocking, and keep lightweight non-blocking logic on the I/O thread using @NonBlocking. This helps maintain high throughput.

AWS S3 storage service Amazon S3 (Simple Storage Service) is a scalable object storage service provided by AWS. It's designed to store and retrieve any amount of data, at any time, from anywhere on the web. It's commonly used for backups, static website hosting, data lakes, and media storage.

4.2 Is it (...) ? Why or why not?

Possible to delete or update the Kafka Commit log: No, it's not directly possible to delete or update the Kafka commit log. Kafka is designed to be immutable. Once a message is written to the log, it cannot be changed. However, logs can be deleted based on retention policies (time-based or size-based).

Possible to include any data type in the Kafka message key: Yes, you can include any data type, but it must be serialized. Kafka treats keys and values as byte arrays. You can use custom serializers (e.g., JSON, Protobuf, String, etc.) to include complex or custom data types. So technically, any data type is possible as long as it's converted to bytes.

Possible to store any data type in the Kafka message: Yes, similar to message keys, any data type can be stored in the message value if properly serialized. Kafka does not restrict the format, as it just stores bytes. You must use matching serializers/deserializers on producer and consumer sides.

Relevant to have a timestamp for stream processing: Yes, having a timestamp is very relevant for stream processing. Timestamps are used for event-time processing, windowing, and ordering of events. Kafka supports both log append time (time of arrival in broker), event time (producer-supplied timestamp, when it is create) and processing time (when a processing application receives event).

In a kafka cluster, is it possible to consume an unsynchronized message:

Usually, Kafka ensures that only committed messages (those replicated to in-sync replicas) are visible to consumers. If the cluster is in "unclean leader election" mode, this is not ensured.

In a kafka in-sync cluster, is it possible to consume an unsynchronized message from one broker:

The in-sync option guarantees that nobody can read a message that has been commited. A message is considered commited only when all followers have it.

4.3 Advantages and pitfalls

Topic	Advantages	Disadvantages
Fire-and-Forget	• Fast, low latency.	• No guarantee of delivery.
Synchronous Send (Wait for Acknowledgment)	• Ensures delivery to Kafka.	• Slower due to blocking.
Asynchronous Send (Callback)	• Non-blocking, high performance with acknowledgment.	• More complex logic for error handling and callbacks.

Table 1: Kafka Messaging Modes: Advantages and disadvantages.

Topic	Advantages	Disadvantages
Distributed System	• Scalable – can handle increased load by adding more nodes. • Fault-tolerant – system continues working even if one node fails. • Resource sharing across multiple machines.	• Complex to develop, test, and maintain. • Network latency and communication overhead. • Difficult to ensure data consistency.
File Integration	• All operating systems and programming languages support files • Many applications have ways of exporting or importing files • Easy to transfer remotely with file transfer protocols, e.g., FTP • Support disconnected interaction	• The complexity of encoding and decoding increases exponentially with complexity of the information to transfer • Performance is limited (No real-time processing)
Web Service Integration	• Platform and language independent. • Standardized protocols like SOAP and REST. • Enables remote system communication.	• Network latency due to HTTP (for REST). • Versioning can be complex. • Requires high availability of services.
REST API Integration	• Lightweight and fast. • Easy to scale and cache. • Stateless – simplifies server-side design.	• Lacks standards for complex operations. • Inconsistent error handling. • Requires good API documentation.
ESB (Enterprise Service Bus) Integration	• Centralized integration logic. • Supports multiple protocols and formats. • Enables message routing and transformation. • Allows accelerating the integration between heterogeneous applications and thus respond to market needs more quickly	• Can become a single point of failure. • Configuration and maintenance are complex. • May introduce performance bottlenecks.
Messaging Streaming vs. Consumer/Producer	• **Advantages:** High throughput, scalable, durable, supports event replay. • **Pitfalls:** Requires careful design, more complex to manage.	• Easy to integrate with legacy systems. • Useful for scheduled or batch jobs.
File Connector	• Easy to integrate with legacy systems. • Useful for scheduled or batch jobs.	• Not suitable for real-time integration. • Sensitive to format changes. • Poor visibility and error recovery.
Screen Scraping	• Works when no API is available. • Can quickly automate tasks.	• Fragile – breaks if UI changes. • Hard to maintain and debug. • Potential legal or ethical concerns.

Table 2: Advantages and disadvantages.

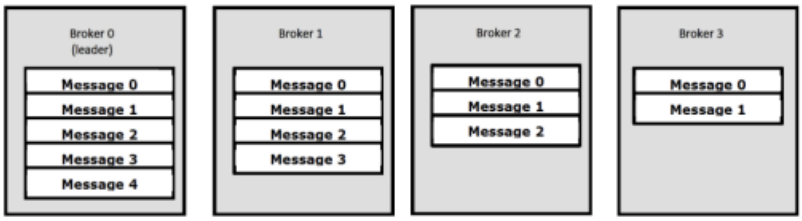
4.4 Multiple choices

Some examples of "What is the correct answer?" for Kafka

Integration using files and Message Oriented Middleware have one common feature: a) The same performance b) Broker based on the content of the file/message c) Publish and subscribe pattern of communication d) The receiver may crash, but the communication may still occur later
The main characteristics of a Message Oriented Middleware system is: a) Both the client and the servers must be running simultaneously b) The interaction is always request-reply c) The integration tolerates server crash failures d) The system tolerates client failures but not cluster failures
The main characteristics of a synchronous message system is: a) Both the client and the servers must be running simultaneously b) The interaction requires a callback c) The integration tolerates server crash failures d) The system tolerates client failures but not cluster failures
Regarding Message Oriented Middleware Publish and Subscribe: a) A Consumer subscribes a topic then it may send messages b) A Producer sends a message to a topic and all the subscriber servers are notified c) A Producer sends a message to a static list of servers defined in the MOM and all servers are notified. d) A Producer sends a message to a queue associated with a topic, where all published messages can be read by the Consumers
Regarding Kafka: a) As in a Message Oriented Middleware, in Kafka you can send a message to a particular queue or instead to a topic b) Partitions allow parallelism for the senders of events c) Topic and partitions are alternative concepts; one can send message to a topic or to a partition independently d) The reading offset is incremented by Kafka whenever a message is consumed
Regarding Kafka:

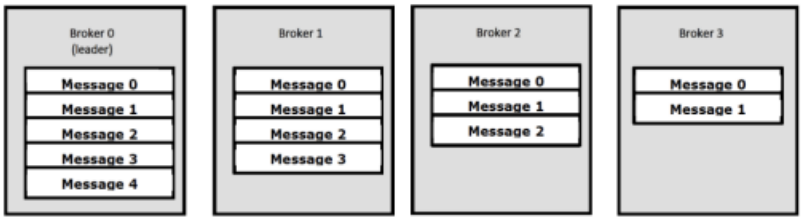
a) As in a Message Oriented Middleware, in Kafka you can send a message to a particular consumer group b) Partitions allow parallelism for the senders of events c) Topic and partitions are alternative concepts; one can send message to a topic or to a partition independently d) The writing offset is incremented by Kafka whenever a message is produced
Regarding Kafka and fault tolerance mechanisms: a) Having multiple partitions on a Topic improves availability b) Brokers replicate the entire Topic to improve reliability c) A Partition may be replicated and updated with a primary backup protocol that assures availability d) The replication protocol assures that all replicas are always coherent
Regarding Kafka and consumer group: a) A consumer group is a set of consumers, and all those consumers can read events from all partitions b) Kafka management is unaware of the number of consumers in a consumer group c) When a consumer fails, the rebalance may create inconsistency in the stream of events consumed d) When a consumer fails, Kafka detects the occurrence and synchronizes the offset in all consumers
Regarding Kafka and consumer group: a) A consumer group is a set of consumers, and all those consumers can read events from all partitions b) Kafka management is aware of the number of consumers in a consumer group c) When a consumer fails, the rebalance may create inconsistency in the stream of events consumed d) A consumer within a consumer group can produce new messages
Consider that a Kafka topic has been created with 5 partitions: a) A producer must know the number of partitions to send a message to the topic b) A consumer reading from all the partition cannot be assured of the strict order of the messages it reads c) The system must have 5 brokers one for each partition d) A consumer group reading the topic must have 5 consumers

Consider that a KAFKA cluster is configured to be all in-sync replicas mode. And a topic has one partition replicated on 4 brokers, accordingly to the figure bellow:



a) Message 3 can be read by a consumer as it is replicated in 50% of the brokers
b) Message 2 can be read by a consumer has it is replicated in a majority of brokers
c) Only message 0 and 1 can be consumed
d) For improving load distribution, a consumer can read from Broker 1

Consider that a KAFKA cluster is configured to be all in-sync replicas mode. And a topic has one partition replicated on 4 brokers, accordingly to the figure bellow:



a) Any message can be consumed if propagated to all brokers
b) Message 3 can be read by a consumer as it is replicated in 50% of the brokers
c) Message 2 can be read by a consumer has it is replicated in a majority of brokers
d) For improving load distribution, a consumer can read from Broker 1

Consider the following phrase: *"In a consumer group, the members can consume different partitions to create redundancy"*.
Do you consider this sentence correct? Justify your answer
No, its to increase performance

Consider the following phrase: *"Although it's possible to increase the number of partitions over time, one has to be careful if messages are produced with keys"*.
What is the reason to use a key when sending a message to Kafka? To meta-identify the messages and to allow balancing partitions policeies
Why does one have to be careful if the number of partitions is increased over time. Explain your answer. Due to the non-sequence that is offered within the topic. Each partition key increase monotonically, but not in the overall of the partition

Consider the following phrase: *"All partition in a topic must be read by a consumer"*.
What happens when a consumer crashes? Explain clearly what does Kafka in this situation. The reading offset is kept to resume the consuming after a crash. This offset can be stored in kafka cluster explicitly or implicitly

In Kafka you can have different semantics for sending a message: Fire-and-forget, synchronous send and asynchronous send. Consider the following code corresponding to sending a message to Kafka in JAVA:

```
ProducerRecord<String, String> record =
        new ProducerRecord<>("CustomerCountry", "Precision Products", "France");
    try {
        producer.send(record).get();
        } catch (Exception e) {e.printStackTrace();
    }
```

The example corresponds to which semantic? Justify based on the methods used and their behavior in JAVA.
synchronous send, due to the get method that is called after sending the message

In Kafka you can have different semantics for sending a message: Fire-and-forget, synchronous send and asynchronous send. Consider the following code corresponding to sending a message to Kafka in JAVA:

```
ProducerRecord<String, String> record =
        new ProducerRecord<>("CustomerCountry", "Precision Products", "France");
    try {
        producer.send(record);
        } catch (Exception e) {e.printStackTrace();
    }
```

The example corresponds to which semantic? Justify based on the methods used and their behavior in JAVA.
Fire-and-forget send, due to the get method that is not called after sending the message

Some examples of "What is the correct answer?"

An application built with a microServices framework:
a) Can communicate with other microServices using REST which is very well suited to a time decoupled pattern of communication
b) Can communicate with other microServices using events which are very well suited to a time decoupled pattern of communication
c) Can communicate with other microServices using shared databases
d) The programming model using service synchronous invocation is more complex than event invocation

From the point of view of an API that is exposing services:
a) API's are focused are focused on the governance of services
b) They must be defined in REST
c) SOA has exactly the same objective
d) API's expose a business asset that has value for the owner

A microservice framework scope:

a) Microservices are based on the principle that they can be orchestrated by a controller service
b) Microservices in opposite to SOA do not have to be high granularity services
c) An application based on microservices defines a choreography using its interfaces
d) Microservices are highly dependent on an Enterprise Service Bus

Considerer a business service of "Selling a product" on an E-commerce application. The interface of the service is specified in JSON and uses the REST protocol. The first time a user purchases something the services requires authentication and register that internally. After that, on a second invocation, the service already has the user identity and follows for the purchase. From this simple description chose the best answer:

a) Loosely coupled and connection oriented
b) Tightly coupled and connection oriented
c) Loosely coupled and connectionless
d) Tightly coupled and connectionless

4.5 Being informed, and an expert, about the following main key terms

4.5.1 PaaS

A cloud computing model that provides a ready-to-use development and deployment environment. It includes infrastructure (servers, storage, networking) and platform tools (databases, runtime, development frameworks), allowing developers to build and run applications without managing the underlying hardware or software layers. An example is Azure.

4.5.2 SaaS

A cloud delivery model where users access fully functional software applications over the internet. The provider manages everything (infrastructure, platform, and software), and users typically interact through a web browser.

4.5.3 IaaS

A cloud computing model that provides virtualized computing resources (servers, storage, networking) over the internet. Users manage the OS, middleware, and applications, while the provider handles the infrastructure. An example is AWS.

4.5.4 FaaS

A serverless computing model where users deploy individual functions or pieces of code that run in response to events. The cloud provider handles the infrastructure, scaling, and execution, charging only for actual usage time. Examples are Cloudflare workers, AWS Lambdas

4.5.5 IaC

A DevOps practice where infrastructure (servers, networks, etc.) is provisioned and managed using code and automation rather than manual processes. This enables version control, testing, and consistent environments. An example is Terraform.