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

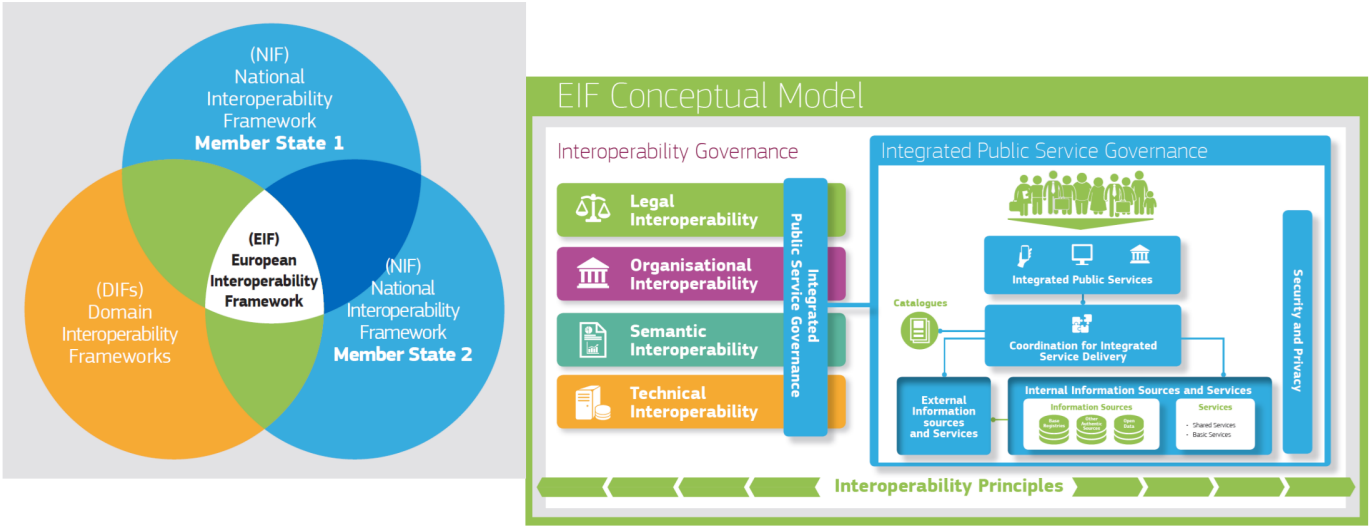
1.4 Enterprise Integration

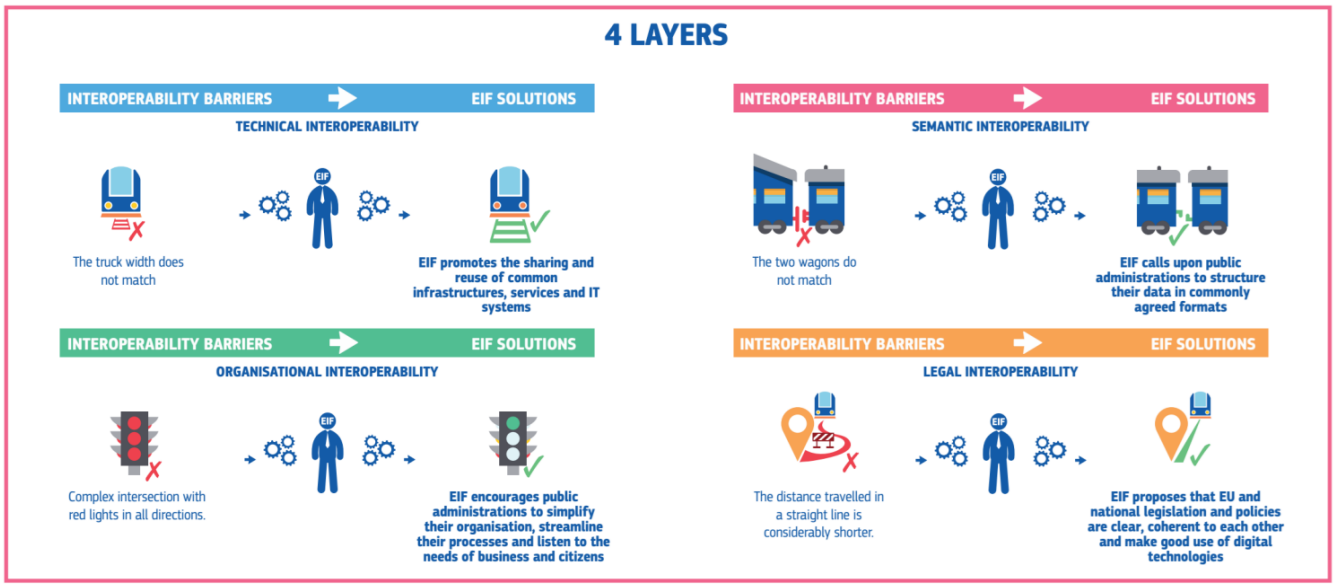
It is the process of ensuring the interaction between enterprise entities necessary to achieve domain objectives. Can be approached in various manners and at various levels, e.g., (i) physical integration,(ii) application integration, (iii) business integration, (iv) through enterprise modeling, and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

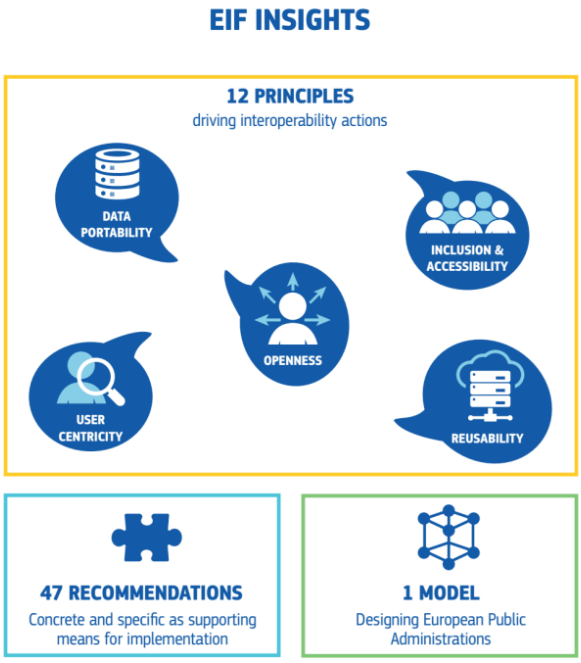
1.5 Enterprise Interoperability

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.

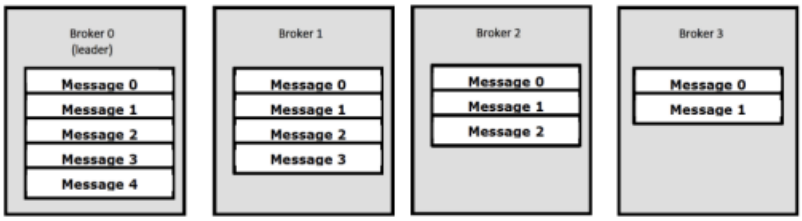
4.4 Multiple choices

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:

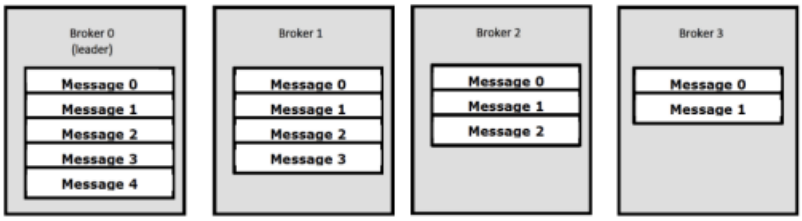
a) As in a Message Oriented Middleware, in Kafka you can send a message to a particular consumer group b) Partitions allow parallelism for the senders of events c) Topic and partitions are alternative concepts; one can send message to a topic or to a partition independently d) The writing offset is incremented 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.

IE Cheatsheet - MAP2

June 2, 2025

Contents

1 Containers & Cloud

1.1 Cloud services properties

- Broadband access - Consume the services from anywhere
- On-demand self-service - Consume the services when you want
- Resource pooling and virtualization - Pool the infrastructure, virtual platforms and applications
- Rapid elasticity - Pooled resources with horizontal scalability
- Measured service - Pay only for what you consume when you consume

1.2 Cloud services

1.2.1 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.

1.2.2 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. An example is Youtube.

1.2.3 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.

1.2.4 FaaS

A Serverless computing platform where the unit of computation is a function that is executed in response to triggers such as events or HTTP requests. The cloud provider handles the infrastructure, scaling, and execution, charging only for actual usage time. Examples are Cloudflare workers, AWS Lambdas

1.2.5 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.

1.2.6 Cloud styles

Clouds may be hosted and employed in different styles depending on the use case, respectively the business model of the provider: **Private cloud** (A cloud environment dedicated to a single organization, hosted either on-premises or by a third party. Used for enhanced security and control over data.); **Community cloud** (A cloud infrastructure shared by several organizations with common concerns, such as security or compliance. Ideal for groups with similar needs, like government agencies or healthcare institutions.); **Public cloud** (A cloud service offered by third-party providers over the internet and shared among multiple customers. Cost-effective and scalable, but less control over infrastructure.); **Hybrid cloud** (A combination of two or more cloud types (private, public, or community) that remain distinct but are integrated. Flexible approach that balances security and scalability.); **Special purpose clouds** (Clouds designed for specific tasks or industries, such as high-performance computing or scientific simulations. Optimized for niche or specialized workloads.).

1.3 Physical Servers

Slow-iteration and slow-deployment. Single tenancy. Unfriendly for friendly for multi programming languages. Deploy in weeks. Typically, alive for years.

1.4 Virtual Machines

VMs work by operating on top of a hypervisor, which is stacked on top of a host machine. A VM monitor (VMM) or hypervisor intermediates between the host and guest VM. By isolating individual guest VMs from each other, the VMM enables a host to support multiple guests running different OSes. Each VM carries their own virtualized hardware stack

that comprises network adapters, storage, applications, binaries, libraries and its own CPU. **Advantages:** VMs allow to consolidate applications onto a single server, Faster-iteration and deployment, Multi-tenancy, Somewhat friendly for multi programming languages, Deploy in minutes, Typically, alive for weeks. **Disadvantages:** Having multiple VMs with their owned OS adds substantial overheads in terms of RAM, CPU, I/O and storage.

1.5 Containers

A container is a self-contained execution environment that shares the kernel of the host system and which is (optionally) isolated from other containers in the system. One of the major advantages of containers is resource efficiency, because you don't need a whole operating system instance for each isolated workload. When a process is running inside a container, there is only a little bit of code that sits inside the kernel managing the container. Contrast this with a virtual machine where there would be a second layer running. In a VM, calls by the process to the hardware or hypervisor would require bouncing in and out of privileged mode on the processor twice, thereby noticeably slowing down many calls.
Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), can handle more applications and require fewer VMs and Operating systems. **Advantage**: more lightweight than VM's

1.5.1 Docker image

An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run. You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

1.5.2 Docker container

A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state. By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine. A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

1.5.3 Docker Hub

It is a service provided by Docker for finding and sharing container images with others. It's the world's largest repository of container images with an array of content sources including container community developers, open source projects and independent software vendors (ISV) building and distributing their code in containers.
Docker Hub provides the following major features:

- **Repositories:** Push and pull container images.
- **Teams & Organizations:** Manage access to private repositories of container images.
- **Docker Official Images:** Pull and use high-quality container images provided by Docker.
- **Docker Verified Publisher Images:** Pull and use high-quality container images provided by external vendors.
- **Docker-Sponsored Open Source Images:** Pull and use high-quality container images from non-commercial open source projects.
- **Builds:** Automatically build container images from GitHub and Bitbucket and push them to Docker Hub.
- **Webhooks:** Trigger actions after a successful push to a repository to integrate Docker Hub with other services.

1.5.4 Docker compose

It is a tool for defining and running multi-container Docker applications. A YAML file is used to configure the application's services. Then, with a command, you create and start all the services from your configuration. Compose works in all environments: production, staging, development, testing, as well as CI workflows. It also has commands for managing the whole lifecycle of applications:
• Start, stop, and rebuild services
• View the status of running services
• Stream the log output of running services
• Run a one-off command on a service

1.6 Kubernetes

It is an open source orchestrator for deploying, scaling, and management of containerized applications (different from terraform which is applicable to any cloud resource). It allows to run Docker containers and workloads and helps tackling some of the operating complexities when moving to scale multiple containers, deployed across multiple servers.

- The service delivery velocity founded on
  - **Immutability:** rather than incremental updates and changes, an entirely new, complete image is built, where the update simply replaces the entire image with the newer image in a single operation.
  - **Declarative configuration:** describing the state of the desired world, instead of specifying the series of instructions.
  - **Online self-healing systems:** e.g., if you assert a desired state of 3 replicas, Kubernetes creates exactly 3 replicas. If a fourth is manually created, Kubernetes destroys it.
- Scaling (software and teams)
  - Decoupling
  - Clusters
  - Microservices
  - Separation of concerns
- Abstracting the infrastructure
- Resource efficiency usage

1.6.1 Kubernetes Architecture

A **node** is a machine where containers (workloads) are deployed. Every node in the cluster must run a container runtime such as Docker, as well as the below-mentioned components, for communication with the primary for network configuration of these containers. A **Kubelet** is responsible for the running state of each node, ensuring that all containers on the node are healthy. It takes care of starting, stopping, and maintaining application containers organized into pods as directed by the control plane. A **Kube-proxy** is an implementation of a network proxy and a load balancer, and it supports the service abstraction along with other networking operation. It is responsible for routing traffic to the appropriate container based on IP and port number of the incoming request.

1.6.2 Kubernetes Pods

A Pod represents a collection of application containers and volumes running in the same execution environment. A Pod is the smallest deployable artifact in a Kuberneted cluster. All of the containers in a Pod always land on the same machine. Applications running in the same Pod share the same IP address and port space (network namespace), have the same hostname, and can communicate using native interprocess ccommunication channels. On the opposite, applications in different Pods are isolated from each other; they have different IP addresses, different hostnames,... Containers in different Pods running on the same node might as well be on different servers. In general, "Will these containers work correctly if they land on different machines?"
No - use a Pod to group the containers
Yes - use multiple Pods

1.7 Terraform (IaC)

Terraform is a solution to create cloud environments using code instead of using the cloud providers User Interfaces. Therefore, the management of the cloud resources takes less effort and the creation, or update, process is faster. Terraform state can be shared through cloud environments like S3 buckets in AWS. For example, I can have 2 different terraform projects that use the same bucket, meaning I could get the hostnames of machines in other projects.

1.8 Serverless Computing

Serverless computing is a platform that hides server usage from developers and runs code on-demand automatically scaled and billed only for the time the code is running. Cloud Native Computing Foundation (CNCF) defines serverless computing as "the concept of building and running applications that do not require server management. It describes a finer grained deployment model where applications, bundled as one or more functions, are uploaded to a platform and then executed, scaled, and billed in response to the exact demand needed at the moment." **Serverless essential qualities:**

1. Providing an abstraction that hides the servers and the complexity of programming and operating them.
2. Offering a pay-as-you-go cost model instead of a reservation-based model, so there is no charge for idle resources.
3. Elasticity - automatic, rapid, and unlimited scaling resources up and down to match demand closely, from zero to practically infinite.

1.8.1 AWS Lambda Functions

- **Function**: is a resource that you can invoke to run your code in AWS Lambda. A function has code that processes events, and a runtime that passes requests and responses between Lambda and the function code. You provide the code, and you can use the provided runtimes or create your own.
- **Runtime**: Lambda runtimes allow functions in different languages to run in the same base execution environment. You configure your function to use a runtime that matches your programming language. The runtime sits in between the Lambda service and your function code, relaying invocation events, context information, and responses between the two. You can use runtimes provided by Lambda or build your own.
- **Event**: is a JSON formatted document that contains data for a function to process. The Lambda runtime converts the event to an object and passes it to your function code. When you invoke a function, you determine the structure and contents of the event. When an AWS service invokes your function, the service defines the event.
- **Concurrency**: is the number of requests that your function is serving at any given time. When your function is invoked, Lambda provisions an instance of it to process the event. When the function code finishes running, it can handle another request. If the function is invoked again while a request is still being processed, another instance is provisioned, increasing the function's concurrency.
- **Trigger**: is a resource or configuration that invokes a Lambda function. This includes AWS services that can be configured to invoke a function, applications that you develop, and event source mappings. An event source mapping is a resource in Lambda that reads items from a stream or queue and invokes a function.

1.8.2 Serverless Drawbacks

Not suitable in terms of cost and execution platform for long-running processes. Vendor lock-in (becoming heavily dependent on a specific cloud provider's proprietary services and technologies, making it difficult and expensive to switch to a different provider later). Cold starts can cause an overhead that is unacceptable on low-latency applications. Monitoring and debugging is more complex when compared with other architectures.

2 Business Process Management

Business Process Management (BPM) is a method for analyzing, improving, and automating business processes to make them more efficient and effective. It helps organizations streamline operations and adapt to change.

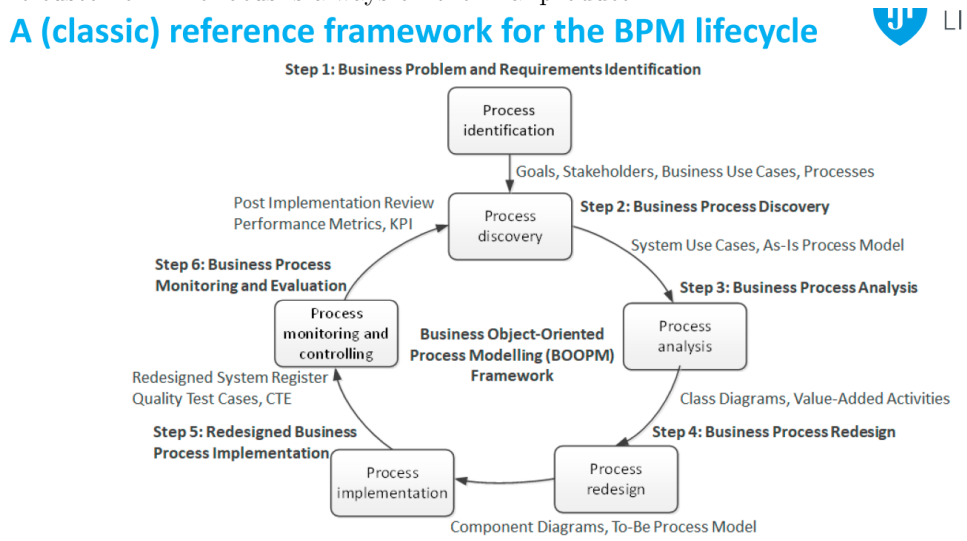
Camunda Engine is the business process executor that consumes the available microservices in the enterprise. The goal is to enable the integration with all the remaining enterprise systems and to enable the representation and execution of the sequence of business behaviours in an easy way. The process-oriented nature allows a closer approach to the business development of the enterprise.

2.1 Workflow Systems

Workflow Management System is a software platform that supports the design, development, implementation and analysis of workflows. Create new applications as an exercise of composition (existing functions) rather them the traditional development of a new application by conventional programming. Programming based on models, represented as straight forward visual constructs (visual programming). Workflows were considered, since the 90's also as a strategic approach for application integration. For all of the above reasons considered as: programming in the large

2.2 Business Process

Archimate: A business process represents a sequence of business behaviors that achieves a specific result such as a defined set of products or business services. A business process describes the internal behavior performed by a business role that is required to produce a set of products and services. For a consumer, the products and services are relevant and the required behavior is merely a black box, hence the designation "internal".
Slides: Set of interrelated activities that transform inputs into outputs in order to produce a service or product to a specific customer. The focus is always on the final product!!!



2.3 Process vs Business Process

Process:
• "A set of interrelated and cooperative activities that transform inputs into outputs".
Business Process:
• "A collection of activities that takes one or more kinds of input and creates an output that is of value to the customer."

2.4 BPMN (Business Process Model and notation)

BPMN (Business Process Model and Notation) is a standardized graphical language used to model and visualize business processes. It helps stakeholders understand and communicate how processes work using flowchart-like diagrams.

2.4.1 Process types

- **Descriptive**: Concerned with visible elements and attributes used in high-level modeling.
- **Analytic**: Contains all of Descriptive and in total about half of the constructs in the full Process Modeling Conformance Class.
- **Common Executable**: Focuses on what is needed for executable process models.

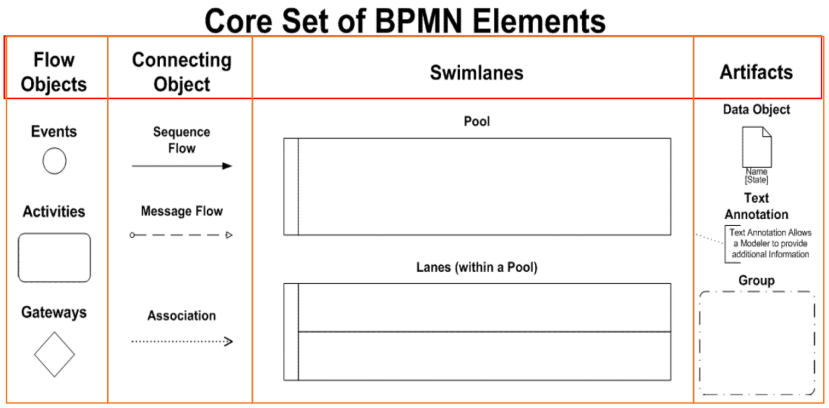
2.4.2 Types of BPMN Diagrams

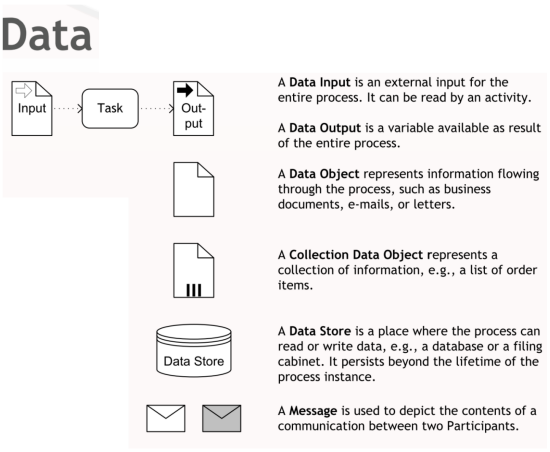
- **Process**
  - Represents the public or private processes of a participant.
  - Focus on representing the (internal) orchestration of a process.
  - The participant can be subdivided into multiple Lanes.
  - All external pools (if any) must be black-box.

- A Process focuses on a single Participant.
- A Private Process focuses on a Participant internal to the organization (a single Participant with multiple lanes is a private process).
- A Public process means more than one participant in it.

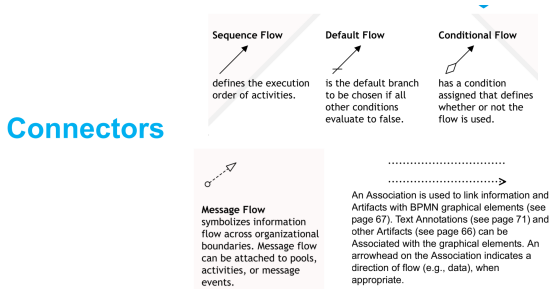
- **Collaboration**
  - Represents the message exchange between two or more participants.
  - Focus on representing the orchestration of a process across multiple participants.
- **Choreography**
  - Represent the information pertaining to each participant in the choreography.
  - The focus is not on orchestrations of the work performed within these Participants, but rather on the exchange of information between Participants.
- **Conversation**
  - Conversation diagrams visualize messages exchange between pools.
  - Design workflow with business process diagram and visualize communications with BPMN conversation diagrams.

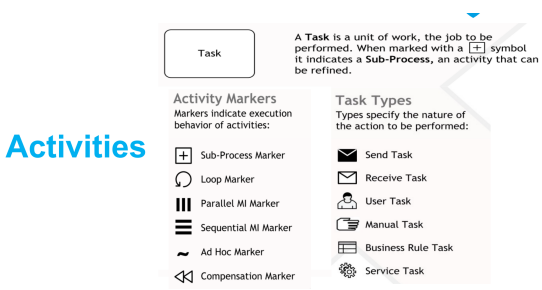
2.4.3 Core BPMN Elements





Pool: represents a process Participant. Sequence flows cannot cross Pool boundaries. Message flows can cross Pool boundaries.
Lane: a sub-division of a Pool. Used to organize and categorize the activities of a Participant. Sequence flows can cross Lane boundaries.
Activity: a unit of work.
Event: an occurrence during a business process (Like start, one line, intermediate, 2 lines, end, bold line).
Gateway: controls the flow of activities.





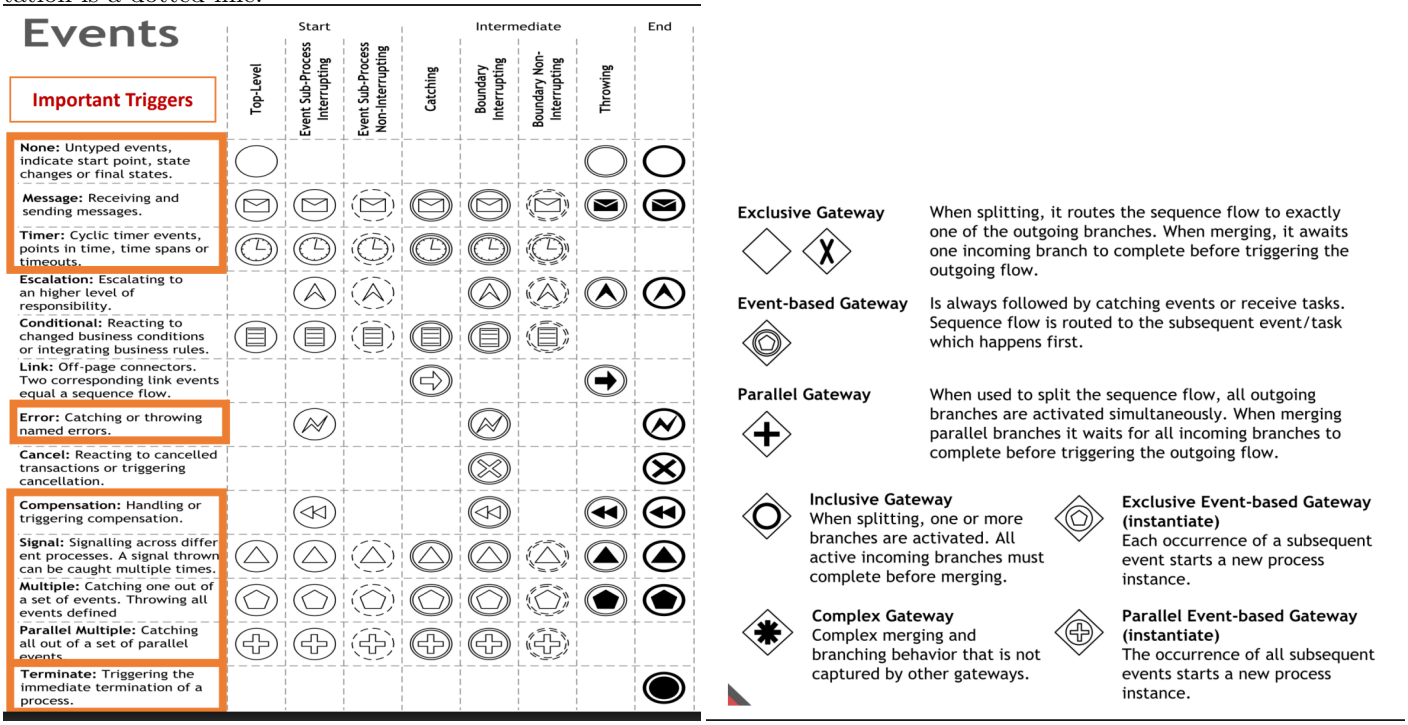
2.4.4 Event Triggers/Behaviour

A Trigger specifies what causes the Event. The Trigger is shown as an icon inside the Event symbol. BPMN defines several types of Triggers: None (i.e. no Trigger) and 12 other (Message, Timer, Error, Signal,...)
A Trigger may have two different behaviours:
Throw a Trigger (throw/send a message) (Black)
Catch a Trigger (White)

• When throwing a Trigger the Event waits for a token and then produces the Trigger. The notation for throw is a filled Trigger.
• When catching a Trigger the Event waits for a token and then waits for the Trigger. The notation for catch is an outlined Trigger. "Catch" is blocking while it waits for the Trigger and the Token.

Start events, can only catch, intermediate events, can catch and throw, end events can only throw.

Interrupting Event: when the event is thrown or caught the corresponding activity is interrupted and is not completed. Notation is a solid line.
Non-Interrupting Event: when the event is thrown or caught the corresponding activity continues its execution. Notation is a dotted line.



2.4.5 Compensation

Sometimes you need to undo the effects of a business process. You do this by **rolling it backwards**, one completed Activity at a time

There are three options for undoing each completed Activity:

1. **Do nothing** - there are no data changes to undo
2. **Overwrite** - restore all data to its original state
3. **Undo** - perform a specific Activity to undo the data changes - this is known as *compensation*

2.5 Gateways

Photo above.

2.5.1 Exclusive Gateway (XOR-split/merge)

XOR-split selects one and only one of the 2..* output flows. The conditions are evaluated top-to-bottom; the output flow associated to first condition evaluated true is selected. Optionally, a default flow (–/–) may be included. The default flow is unconditional and is selected whenever all the other conditions are evaluated false. The output flows of a XOR-split may be merged using a XOR-merge that continues as soon as one input flow arrives.

2.5.2 Parallel (AND-split/merge)

AND-split forks one input flow into 2..* parallel (concurrent in time) output flows. AND-merge waits for all input flows before joining them into a single output flow.

2.5.3 Event-Based Gateway

Follow the flow of the corresponding event that happenened.

2.5.4 Complex Gateway

These have no default semantics!

- Splitting - modeller provides an IncomingCondition
- Merging - modeller provides an OutgoingCondition

They *must* be supported by Text Annotations that describe their semantics, otherwise the BPD is unreadable!

2.6 Modelling an executable BPMN

Process automation is about automating tasks within a process as well as the automation of the control flow between these tasks. Three different combinations possible which define a kind of maturity levels of process automation:
1. The human controls the process, often passing paper around. You're probably familiar with this from physical inbox folders on your desk in an office environment. Software and office tools can, of course, make this process more efficient.
2. The computer controls the process and involves people whenever necessary, for example using task list user interfaces.
3. The whole process is fully automated and only requires manual intervention if something happens beyond the expected normal operations.

Two types of automation:
1. Automation of the control flow: The interactions between tasks are automated, but the activities itself might not. In the previous example, this was the humans cooking the food. A process might not be fully automated as it cannot run without human interaction. This is often known as human task management.
2. Automation of the tasks: The task itself is automated. In the previous example, this would be the robots who cook the food. Automating the tasks leads to fully automated processes, which is typically named STP (straight through processing). Very often this is the most ideal path, where everything runs smoothly and is fully automated. Humans are involved in case any unforeseen exceptions or special cases occur. This approach is actually quite powerful, as you can invest the effort on automation on the majority of cases and save additional (manual) effort only on the rare cases.

2.7 Business Process Execution Engine (Camunda)

2.7.1 Camunda Architecture

• Process Engine Public API: Service-oriented API allowing Java applications to interact with the process engine. The different responsibilities of the process engine (i.e., Process Repository, Runtime Process Interaction, Task Management, ...) are separated into individual services. The public API features a command-style access pattern.
• BPMN 2.0 Core Engine: this is the core of the process engine. It features a lightweight execution engine for graph structures (PVM - Process Virtual Machine), a BPMN 2.0 parser which transforms BPMN 2.0 XML files into Java Objects and a set of BPMN Behavior implementations (providing the implementation for BPMN 2.0 constructs such as Gateways or Service Tasks).
• Job Executor: the Job Executor is responsible for processing asynchronous background work such as Timers or asynchronous continuations in a process.
• The Persistence Layer: the process engine features a persistence layer responsible for persisting process instance state to a relational database.

2.7.2 Camunda Deployment & Operation environment

- Clustering – active/ative, with shared database for process execution
- Kubernetes for dynamic installation
- Separation between execution data and historical data
- Optimize available with Enterprise Edition

2.7.3 Asynchronous Continuations in Camunda BPM

Asynchronous continuations are break-points in the process execution. They are used as transaction boundaries and allow another thread than the currently active thread to continue execution.
• Async is used for placing a safe-point before an activity such that the execution state is committed. If the activity then fails to execute, the transaction is rolled back only up to the safe point.

• Async also comes in handy if you have longer-running computations and do not want to block the calling thread (eg. HTTP Thread) but instead want to delegate the heavy lifting to a background thread.
• Finally, due to the fact that asynchronous continuations are executed by the job executor, the retry mechanism can be used in order to retry a failed activity execution.

2.7.4 Handling data in process

When using Camunda, you have access to a dynamic map of process variables, which lets you associate data to every single process instance (and local scopes in case of user tasks or parallel flows).
• Input mappings can be used to create new variables. They can be defined on service tasks and subprocesses. When an input mapping is applied, it creates a new local variable in the scope where the mapping is defined.
• Output mappings can be used to customize how job/message variables are merged into the process instance. They can be defined on service tasks, receive tasks, message catch events, and subprocesses.
• Can be used to create new variables or customize how variables are merged into the process instance.

2.7.5 Camunda Authorization

• Assignee a specific user who must perform the task
• Candidate users a list of specific users who can perform a task
• Candidate groups a list of user groups who can perform the task

3 Application Programming Interfaces (API) Management

3.1 API

APIs are by definition interfaces to be used as entry points to reusable software entities with well-defined contracts. They are not independent software entities; they are instead packaged with the software libraries, frameworks, or Web services, that offer them.

Second definition: An API represents an abstraction of the underlying implementation. An API is represented by a specification that introduces types. Developers can understand the specifications and use tooling to generate code in multiple languages to implement an API consumer (software that consumes an API). An API has defined semantics or behavior to effectively model the exchange of information. Effective API design enables extension to customers or third parties for a business integration. In process API invocation: e.g., a java RMI call is handled by the same process from which the call was made. Out of process API invocation: e.g., a .NET application invocating an external REST-like API using an HTTP library is handled by an additional external process other than the process from which the call was made.

Due to their omnipresence and evolution, APIs greatly impact software development. Understanding, mitigating, and leveraging the impact of APIs and API evolution on software development is necessary to design and use software APIs APIs evolve for various reasons, such as increasing complexity, and continuous change. However, due to their nature as a connection point between software modules, API evolution is not without side effects On the one hand, as predicted by Lehman, continuing change means that API developers must determine ways to keep their APIs useful, cutting edge, and competitive with other pieces of software and API users must adapt to these API changes and new API releases. On the other hand, conservation of familiarity, or existing API usages, constrain the evolution of an API to avoid breaking changes while improving the API (i.e., security or performance improvements). The evolution of APIs therefore involves a balancing act of constant improvement and maintaining existing functionality. Maintaining existing functionality requires in-depth knowledge of use cases and architectural foresight and flexibility, while keeping up with rapid release cycles requires modifications to user applications as well as learning about new APIs and changes to existing APIs.

3.2 Differences from SOA

From an architectural perspective they are similar: “a logical representation of a repeatable activity that has a specific outcome, the usual notion of a service” But the focus is different:
• APIs were focused in simplifying the consumption, developer centric, humanreadable contract
• SOA focus in shielding the client from back-office applications changes

3.3 API trends

New tools and techniques typically seek to help with API evolution by resolving problems that it can cause for API users (e.g., API migration tools) or to help reduce the development burden on API developer (e.g., automatic API documentation tools). Empirical studies related to API evolution typically employ large data, case studies, or user studies

to provide evidence of existing problems, the impacts of API evolution, or potential solutions to existing problems. These problems typically centre around the impacts of API evolution on API usability and API maintainability.

3.4 API Management

It is a set of tools and services that enable developers and companies to build, analyze, operate, and scale APIs in secure environments. API management can be delivered on-premises, through the cloud, or using a hybrid on-premises – SaaS.

3.4.1 Important features of API management tools

- **API ACCESS CONTROL:** APIs should be built using access controls, commonly known as authentication and authorization, that grant users permission to access certain systems, resources, or information.
- **API PROTECTION:** API protections include API keys for identification, API secrets, and application authorization tokens that can be verified.
- **API CREATION AND DESIGN:** APIs allow web applications to interact with other applications. You can create and define different types of APIs such as RESTful APIs and WebSocket APIs.
- **SUPPORT FOR HYBRID MODELS:** A RESTful API is a group of resources and methods, or endpoints, that leverage an HTTP request type. A WebSocket API maintains a persistent connection between connected clients.
- **HIGH PERFORMANCE:** Highly performant APIs depend on code, the separation of functionalities, and on underlying data structure and data architecture.
- **CUSTOMIZABLE DEVELOPER PORTAL:** API developer portals connect API publishers with API subscribers. They enable self-service API publishing and allow potential API customers to easily discover APIs they can use.

3.5 Representation State Transfer (REST)

It is a set of architectural contraints, most commonly applied using HTTP as the underlying transport protocol. To be considered RESTful your API must ensure that:
1. A producer-to-consumer interaction is modeled where the producer models resources the consumer can interact with.
2. Requests from producer to consumer are stateless, meaning that the producer doesn't cache details of a previous request. In order to build up a chain of requests on a given resource, the consumer must send any required information to the producer for processing.
3. Request are cachable, meaning the producer can provide hints to the consumer where this is appropriate. In HTTP this is often provided in information contained in the header.
4. A uniform interface is transmitted to the consumer.
5. It is a layered system, abstracting away the complexity of systems sitting behind the REST interface. For exemple, the consumer should not know or care if they are interacting with a database or other services.

3.5.1 REST data elements

The key abstraction of information in REST is a resource. Any information that can be named can be a resource (document, image, etc.) In other words, any concept that might be the target of an author's hypertext reference must fit within the definition of a resource. A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time.

REST components perform actions on a resource by using a representation to capture the current or intended state of that resource and transferring that representation between components. A representation is a sequence of bytes, plus representation metadata to describe those bytes. Other commonly used but less precise names for a representation include: document, file, and HTTP message entity, instance, or variant.

A representation is a sequence of bytes, plus representation metadata to describe those bytes. The data format of a representation is known as a media type. Some media types are intended for automated processing, some are intended to be rendered for viewing by a user, and a few are capable of both. Composite media types can be used to enclose multiple representations in a single message. Response messages may include both representation metadata and resource metadata: information about the resource that is not specific to the supplied representation.

Control data defines the purpose of a message between components, such as the action being requested or the meaning of a response. It is also used to parameterize requests and override the default behavior of some connecting elements. For example, cache behavior can be modified by control data included in the request or response message. A representation

can be included in a message and processed by the recipient according to the control data of the message and the nature of the media type.

3.5.2 REST over HTTP

An HTTP request is:
• A verb (aka method), most of the time one of GET, POST, PUT, DELETE or PATCH
• A URL
• Headers (key-value pairs)
• Optionally a body (aka payload, data)
An HTTP response is:
• A status, most of the time one of 2xx (successful), 4xx (client error) or 5xx (server error)
• Headers (key-value pairs)
• A body (aka payload, data)
HTTP verbs characteristics:
• Verbs that have a body: POST, PUT, PATCH
• Verbs that must be safe (i.e. that mustn't modify resources): GET
• Verbs that must be idempotent (i.e. that mustn't affect resources again when run multiple times): GET (nullipotent), PUT, DELETE

3.5.3 Richardson maturity heuristics

• Level 0: The starting point for the model is using HTTP as a transport system for remote interactions, but without using any of the mechanisms of the web. Essentially, using HTTP as a tunneling mechanism for remote interaction mechanism, usually based on Remote Procedure Invocation.
• Level 1 – introduce Resources: rather than making all our requests to a singular service endpoint, start talking to individual resources. Tackles the question of handling complexity by using divide and conquer, breaking a large service endpoint down into multiple resources.
• Level 2 - using the HTTP verbs: as closely as possible to how they are used in HTTP itself. Introduces a standard set of verbs so that we handle similar situations in the same way, removing unnecessary variation.
• Level 3 - Hypermedia Controls introduces HATEOAS (Hypertext As The Engine Of Application State).
• Level 3 introduces discoverability, providing a way of making a protocol more self-documenting.

3.6 Open API

REST APIs are technological implementation of services and do not account with contracting issues. Usually an additional document, or wiki documents the API usage. However, those are hard to maintain and testing is not automating. "The OpenAPI Specification (OAS) defines a standard, languageagnostic interface to HTTP APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection. When properly defined, a consumer can understand and interact with the remote service with a minimal amount of implementation logic".

Swagger has evolved into one of the most widely used open source tool sets for developing APIs with rich support for the OpenAPI Specification, AsyncAPI specification, JSON Schema and more.

The contract of an API encompasses:
• Title
• Endpoint
• Method
• URL parameters
• Message payload
• Header parameters
• Response code
• Error code
• A sample request and response
• Tutorials and walkthrough
• Service-level agreement

3.7 GraphQL

It is an API technology developed by Facebook to address some of the challenges faced with RESTful APIs, particularly in complex applications where multiple data sources need to be queried. Allows the client to request only the specific data it needs, using a declarative query language, reducing over-fetching and under fetching issues common with RESTful APIs. Provides a flexible and efficient way for front-end applications, making it a choice for applications with complex data requirements and a variety of front-end interfaces. It allows developers to define their queries precisely, resulting in fewer network requests and less data transfer. However, comes with additional complexity compared to RESTful APIs and requires more careful management of server-side schema

3.8 gRemote Procedure Call (gRPC)

Developed by Google, is a high-performance, open-source API framework that uses HTTP2 for transport and protocol buffers as data format. gRPC is designed to provide efficient communication between microservices, making it highly suitable for blockchain backends that require fast and reliable communication with front-end clients. One of the advantage is its ability to support bi-directional streaming, which is particularly useful for real-time applications where low latency is critical. Unlike RESTful APIs, gRPC allows the server to push updates to the client, making it suitable for applications that require continuous data streams. gRPC's performance and efficiency make it an excellent option for projects that demand low latency and high data throughput, such as real-time trading platforms or blockchain monitoring services. However, gRPC is more complex to implement than REST or GraphQL and may require more setup, particularly for handling serialization and managing HTTP2 connections. Despite this, its performance benefits are invaluable for projects with stringent speed and reliability requirements.

3.9 API Gateway Systems (KONG)

According to Gartner, a gateway in computer networking: "gateway converts information, data, or other communications from one protocol or format to another".
Moreover, it must allow communication in both directions and maintain the connections. A gateway can also be seen as a "supervised entry point" through which messages (carries of information/knowledge) can enter a particular environment undergiven conditions and rules applied.

Provides or integrates with third-party gateways for runtime management, security, policy enforcement, throttling, operational control and usage monitoring for APIs.

Specific:
• Kong Gateway, is a commercial version of its open-source API gateway based on NGINX and OpenResty.
• Kong Gateway works as a reverse proxy that manage, configure, and route requests to other APIs.
• Kong Gateway runs in front of any RESTful API and can be extended through modules and plugins. It's designed to run on decentralized architectures, including hybrid-cloud and multi-cloud deployments.

Advantages of an API Gateway System
• Enable new consumers without coding or configuration (Interface and Volume Scoping, Routing)
• Keep unwanted people & robots out of your systems(Authorization and Authentication, Threat Protection)
• Understand what is happening with your APIs (Discovery, Usage, and Concerns, History)

3.9.1 Kong main concepts

• Service: is the name Kong uses to refer to the upstream APIs and microservices it manages
• Route: specify how (and if) requests are sent to their Services after reaching Kong
• A single Service can have multiple Routes.
• Plugin: extension to the Kong Gateway, written in Lua or Go
• Port 8001 is used for managing APIs
• Port 8000 for service invocation

3.9.2 Full lifecyle of API management

• Design APIs – solutions such as swagger that are able to facilitate the design, build and documentation
• Run APIs – solutions such as Kong that are able to connect to endpoints and return the responses
• Secure APIs – solutions such as Kong plug-ins that are able to apply fine-grained security and traffic policies to services
• Govern APIs – solutions such as Konga that are able to gain real-time visibility of the services

3.10 Identity Management

3.11 AAA (authentication, authorization and accountability) framework

• Authentication (A) - It is the challenge process by which identity claims are tested and validated. The purpose of the authentication is ensuring that a particular user is really who he or she claims to be. Then access is granted to a system if granted by the authorization' process execution
• Authorization (A) - After authentication, the authorization process enforces the system' policies, granular access control, and user privileges. It also establishes the tasks and activities that users can perform within those authorized resources
• Accountability (A) - is about measuring what's happening within the system, collecting and logging data on user sessions, e.g., length of time, type of session, and resource usage. It offers a clear audit trail for compliance and business purposes.

3.11.1 Authentication (A)

It is the challenge process by which identity claims are tested and validated, where challenge is secret and could be:
• Password
• Physical identifies: card, physical key
• Biometric data: face, iris, fingerprint
• Cypher/decipher challenge with keys
Authentication in applications/data stores within an organization:
• Typically, single sign-on using LDAP to interact with Active Directory, Kerberos

But, on the Cloud, multiple options possible:
• Each organization does its authentication leads to multiple identities
• One organization authenticates all participants
• For each business partnership, an organization (s) is chosen to authenticate
• Allow each organization to use its authentication framework and create a loosely coupled mechanism allowing the reuse of identities and authentication

3.11.2 Authorization (A) (Não acho isto necessário?)

Today, countless access control models (ACM) solutions are available in the academy and industry. Some examples:
• Discretionary access control (DAC)
• Mandatory access control (MAC)
• Role-based access control (RBAC)
• Time-role-based access control (TRBAC)
• Attribute-based access control (ABAC)
• Orcon or Chinese wall
• Nevertheless, in the majority of situations, the recognized development of ACM, these solutions specifies and implements the structural security access concerns of a single organizational silo.

3.12 eXtensible Access Control Markup Language (XACML)

XACML is an XML-based standard markup language for specifying access control policies. The standard, published by OASIS defines a declarative fine-grained, attribute-based access control policy language and a processing model describing how to evaluate access requests according to the rules defined in policies.
XACML is an attribute-based access control system:
• Input, e.g., the information about the subject accessing a resource, the resource to be addressed, and the environment
• Output, the decision of whether access is granted or not

• Policy - A set of rules, an identifier for the rule-combining algorithm and (optionally) a set of obligations or advice. May be a component of a policy set
• PAP - Policy Administration Point is the system entity that creates a policy or policy set
• PEP - Policy Enforcement Point is the system entity that performs access control, by making decision requests and enforcing authorization decisions.
• Obligation - An operation specified in a rule, policy or policy set that should be performed by the PEP in conjunction with the enforcement of an authorization decision
• PDP - Policy Decision Point is the system entity that evaluates applicable policy and renders an authorization decision
• PIP - Policy Information Point is the system entity that acts as a source of attribute values

3.13 SAML

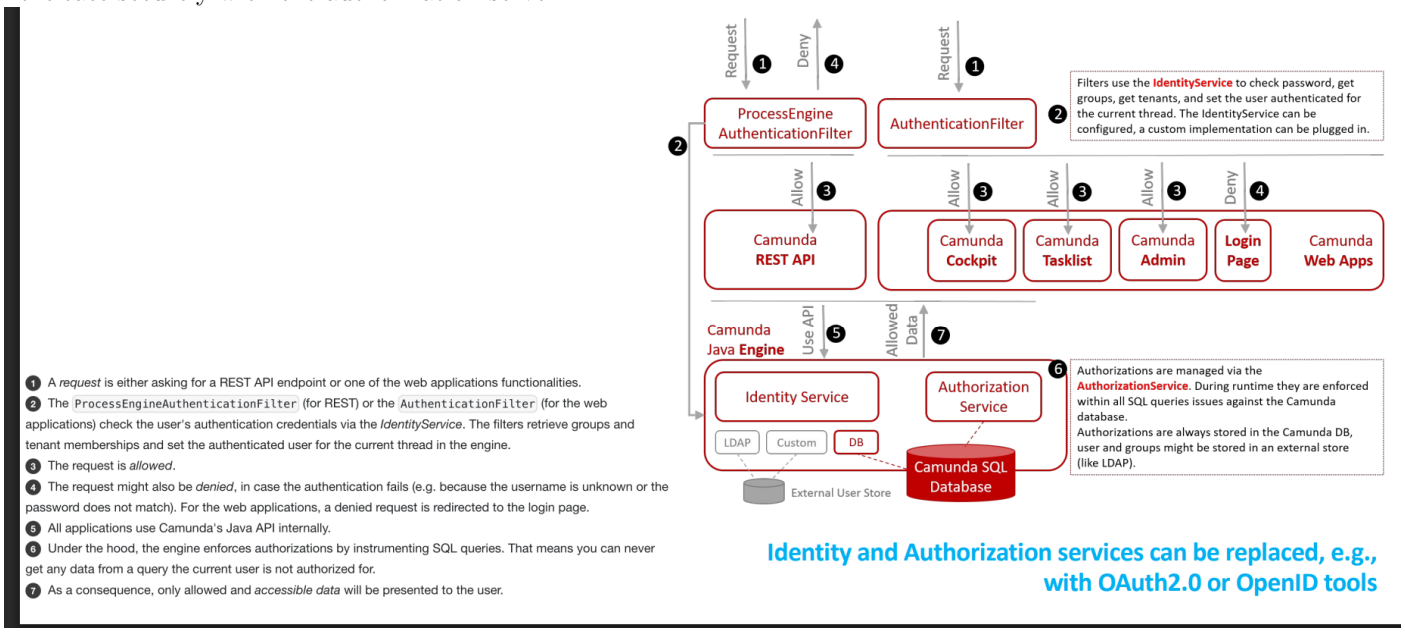
SAML is a well-established standard developed by the OASIS consortium for authentication and authorization. Developed during early 2000s, now in the version 2.0, it is an XML-based standard heavily influenced by the enterprise technologies popular at the time, especially the WS- stack. Similar to some other XML-based standards, it defines a hierarchy of basic concepts - assertions, protocols, bindings and profiles. The most common SAML profile (simplified, profiles are essentially use-cases) is multi-domain web single sign-on and includes interaction between three parties - a user, an Identity provider, and a Service Provider. In this scenario, the user already has an established session with one web site, during which the user is redirected to a new web site. The original website acts as an IdP and relays information to the redirected website that acts as a Service provider. The relayed information effectively authenticates the user through a SAML message allowing for a session between the user and the redirected website to be established. In SAML terms, the first website is generally referred to as the asserting party and the second as the relying party.

3.14 OpenID

OpenID Connect is an authentication protocol specified and maintained by the OpenID Foundation. The foundation's members are mostly international corporations including well-known names like Google, Microsoft, Oracle, PayPal, Verizon, RSA, VMWare, Deutsche Telekom. The protocol is defined through a set of OpenID specifications including the core specification (Sakimura, 2014) and accompanying specifications detailing additional services (like dynamic provider discovery, dynamic registration with providers, response types, etc.). Unlike the previous versions of OpenID, the current one (final specification launched in 2014) is based on the previously described OAuth 2.0 protocol. The specification extends and additionally specifies parts of the OAuth specification that have been left open by OAuth, so OAuth-based mechanisms could be used in authentication scenarios

3.15 OAuth 2.0

OAuth 2.0 standard is published and maintained by IETF through a set of RFC documents, the core consisting of RFC 6749 (Hardt, 2012 (1)), RFC 6750 (Hardt, 2012 (2)) and RFC 6819 (McGloin, 2013). OAuth 2.0 is foremost an delegation protocol. The user provides permission to a third party (a client in OAuth terms) to access some of user's data at the identity. For example, the user allows a site like fit4life.com read-only access the user's fitness data stored at google.com (an authorization server and resource server in OAuth terms), without necessarily divulging any of the user's other confidential data. Being a more modern protocol that targets not only web applications and services but also handheld devices, OAuth makes a clear distinction between public and confidential clients based on their ability to authenticate securely with the authorization server.



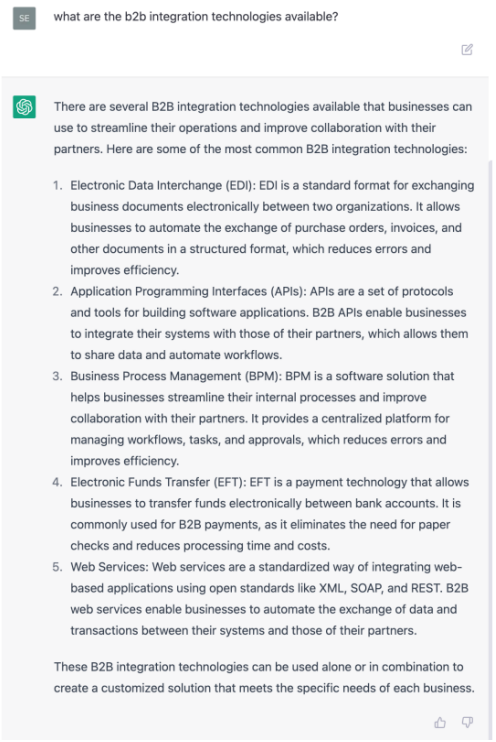
	OAuth2	OpenID	SAML
Token (or assertion) format	JSON or SAML2	JSON	XML
Authorization	Delegation	No	Yes
Authentication	Pseudo-authentication	Yes	Yes
Security risks	Phishing OAuth 2.0 does not support signature, encryption, channel binding, or client verification. Instead, it relies completely on TLS for confidentiality	Phishing Identity providers have a log of OpenID logins, making a compromised account a bigger privacy breach	XML Signature Wrapping to impersonate any user
Best suited for	API authorization	Single sign-on for consumer apps	Single sign-on for enterprise Note: not well suited for mobile

3.16 Consent Management

Consent Management refers to the process and system by which an organization collects, manages, and maintains user consent for the collection and processing of personal data. It ensures compliance with privacy laws and regulations, such as the GDPR (General Data Protection Regulation) and CCPA (California Consumer Privacy Act), by allowing users to make informed decisions about what personal data they are willing to share. The goal is to give users control over their data while also providing transparency about how it is being used. It involves obtaining clear consent, documenting it, and ensuring that it can be easily modified or withdrawn by the user at any time.

4 B2B

'chat.openai.com/chat



- Point to point integration - Increasing cost, Hard to maintain and scale (Different protocols, messages, and security)
- Centralized hub integration - Who implement the bus? Who pays for it? (Trustability problem (maybe solved by blockchain?), Data privacy problem, Cost too high, only applicable to big companies)

- Internet integration - Each company is responsible for his own EAI (Enterprise Application Integration), including all protocol layers, security and audit, Standard EAI connectors simplify interoperability and reduce the deployment cost
- Marketplaces (Like amazon business) - Transaction costs is higher, but companies invest less in IS, Trust is increased, Value chain observability.

5 Perguntas

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

5.1.1 DMN (Decision Model and Notation)

DMN (Decision Model and Notation) is a standardized modeling language used to represent and automate business decisions. It separates decision logic from process flow, allowing organizations to define rules in a clear, structured way. DMN models typically use decision tables and diagrams to capture business rules, making them understandable to both technical and non-technical stakeholders. This helps improve consistency, transparency, and flexibility in decision-making across systems and teams.

5.1.2 Camunda business key

In Camunda, the business key is a unique, human-readable identifier assigned to a process instance, often representing a business entity like an order or customer ID. It helps correlate the process with external systems and makes querying and tracking easier. Once set, it cannot be changed.

5.1.3 Docker

Docker is a platform that enables developers to package applications and their dependencies into lightweight, portable containers. These containers run consistently across different environments, making deployment faster and more reliable.

5.2 What is it and explain in detail how does it work?

5.2.1 Business process engine

A business process engine is software that executes and manages business process models, typically defined in BPMN. It controls the flow of tasks, handles user interactions, manages rules, and integrates with systems to automate business workflows.

5.2.2 Authorization of a business process engine

Authorization in a business process engine controls who can perform specific actions within processes. It ensures that users or systems have the correct permissions to start, view, complete, or manage process instances, tasks, and resources. This helps enforce security, compliance, and role- based access within workflows.

5.2.3 Identity management

Identity management is the process of creating, managing, and verifying digital identities within an organization. It ensures that the right individuals have access to the right resources at the right times by handling user authentication, authorization, roles, and lifecycle management. This is essential for security, compliance, and efficient access control.

5.2.4 Camunda form

In Camunda, a form is a user interface element used to collect input from users during a workflow. Forms can be embedded in tasks to capture data needed to proceed, such as approval decisions or details entry

5.2.5 Camunda http-connector

An http-connector in Camunda is able to make requests to endpoints through http. As definitons, you need to define the method used (POST, GET, etc.), the payload in the body of the request and the url itself to connect to. This is for input, for output, you can use JavaScript, for example, to parse the response and create variables that are outputted. It is ine type of camudna service task.

5.2.6 Camunda user interaction task

A Camunda user task is a step in a business process where human interaction is required. It pauses the automated workflow until a user completes the task, such as filling out a form or approving a request. User tasks enable collaboration between people and automated processes.

5.2.7 Camunda service task

A Camunda service task is an automated step in a business process that executes predefined backend logic without human intervention. It can call external services, run scripts, or execute code to perform tasks like data processing, system integration, or API calls.

5.2.8 Serverless API invocation process. What is cold start and warm execuction?

Cold start happens if the function isn't currently running. The cloud provider must allocate resources and initialize the function environment before execution, causing a delay (latency).

Warm execution occurs when the function is already initialized and ready from a recent invocation, so it runs immediately with minimal delay.

5.2.9 What type of computational offer from cloud providers is depicted when referring to serverful computing?

Serverful computing refers to traditional cloud offerings where you manage and provision servers or virtual machines yourself. You're responsible for configuring, scaling, and maintaining the underlying infrastructure. Examples include virtual machines (VMs), dedicated servers, or managed Kubernetes clusters.

5.2.10 What does autoscaling means?

Autoscaling is the cloud capability to automatically adjust the number of running resources—like servers or containers—based on current demand. It helps maintain performance during high load and reduce costs during low usage.

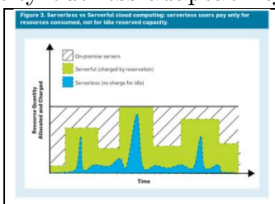
5.2.11 Compare autoscaling and the "premium upgrades" in serverful computing what the main difference between them for the end user is?

The main difference between autoscaling and premium upgrades in serverful computing lies in flexibility and automation:

Autoscaling automatically adjusts resources (e.g., CPU, memory, instances) based on real-time demand, with no manual intervention.

Premium upgrades are manual or static increases in resource capacity (like choosing a larger VM type), requiring user action and often restarts.

For the end user, autoscaling means smoother performance and cost-efficiency, while premium upgrades offer fixed capacity but less adaptability to changing workloads.



Considering the technological dependencies imposed by the following three distinct environments: on-premises servers, serverful and serverless, which were the changes that you need to do if you want to migrate JAVA code:

a) From on-premise to serverful? Adapt to the container operating system and environment + network dependencies

b) From serverful to serverless? Comply to serverless libraries and methods + change to short living processing

5.3 What are the advantages and pitfalls of?

DMN Advantages - Clear, reusable decision logic. Business-friendly and easier to maintain.
DMN Pitfalls - Adds complexity and requires extra learning.

BPMN Gateways Advantages - Simple for basic decisions. No separate tools needed.

BPMN Pitfalls - Hard to manage complex logic. Logic is less reusable and harder to update.

5.4 Pull and push for Camunda task execution

Pull (External Tasks) Advantages - Decoupled and scalable. Good for microservices.

Pull (External Tasks) Pitfalls - Needs polling. Slight latency.

Push (Java Delegates) Advantages - Fast, runs inside engine. Simple setup.

Push (Java Delegates) Pitfalls - Tightly coupled. Harder to scale or externalize.

5.5 What are the differences between the concepts of?

5.5.1 BPMN descriptive process and BPMN executable process

BPMN Descriptive Process is used to visually document and communicate business workflows without technical details. BPMN Executable Process includes all technical details needed to run the process automatically in a BPM engine.

5.5.2 Pull and push for Camunda task execution

Pull (External Tasks) means workers actively fetch tasks from the Camunda engine when they are ready to execute them. This allows better scalability, loose coupling, and easier integration with distributed systems or microservices. However, it may introduce some delay due to polling.

Push (Java Delegates) means the Camunda engine directly invokes the task's code as part of the process execution. This offers faster response and simpler setup but creates tighter coupling between the process engine and the task logic, making scaling and external integration harder.

5.6 How do you? Specify the most relevant steps to perform the following task:

To expose an AWS Lambda function to the internet:
Deploy your Lambda function.
Create an API Gateway and set up an HTTP method linked to your Lambda.
Ensure API Gateway has permission to invoke the Lambda.
Deploy the API to a stage, which provides a public URL.
Use the API Gateway URL to access your Lambda from the internet.

5.7 Multiple Choices

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

Serverless Functions:
a) There is no server executing the function

b) The main objective is to reduce function administration to a minimum
c) It is a model of execution that existed on premises and migrated to the cloud
d) The main objective is speed of execution

Pitfalls related with Serverless functions:
a) More difficult to have high availability
b) Latency on cold starts
c) Can only be used for synchronous invocations
d) Long time execution functions are allowed

Benefits of a Business Process oriented approach:
a) Automatically builds a service hierarchy
b) It produces applications which are more performant than with traditional development
c) Business analysts model the Business Processes in BPMN and then IT people need to create the executable counterparts
d) It creates a complete application which can be integrated with legacy systems and external services

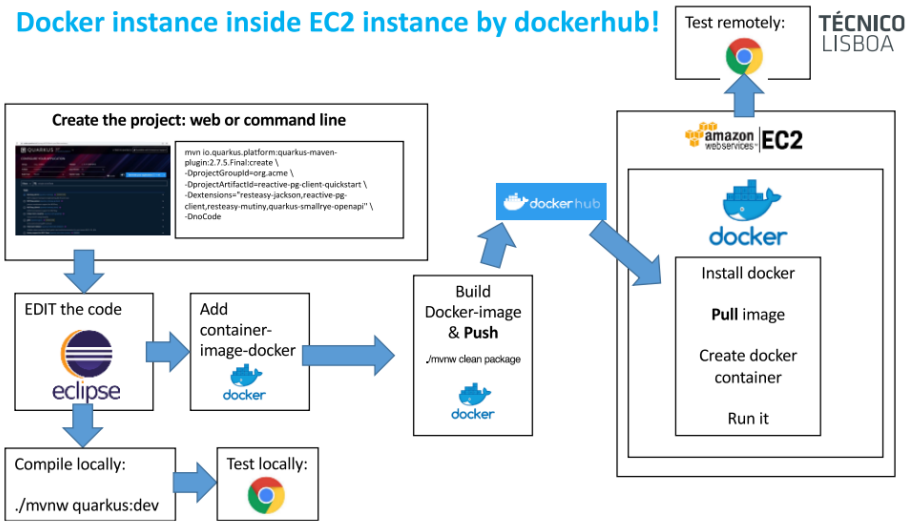
A microservice framework scope:
a) Microservices are based on the principle that they can be orchestrated by a controller service
b) Microservices in opposite to SOA do not have to be high granularity services
c) An application based on microservices defines a choreography using its interfaces
d) Microservices 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

Camunda authentication and authorization. Which sentence is false?
a) Camunda can use an external identity provider
b) Camunda doesn't need to have an authenticated user
c) Camunda has an internal authorization service that enforced authorization based on the role defined in BPMN
d) Camunda can invoke external services providing authentication credentials

5.8 Images (Business process, etc.)



Considering the process depicted in the above figure to deploy a new Quarkus microservice in the AWS cloud environment.

a) Why is Docker used? Explain the reasons referring to at least 2 advantages.

b) Without Docker, what are the other alternatives to achieve the same deployment?

c) What is the technology that could be used to automate the Quarkus microservices unit tests (both locally and remotely)?

a)

Ensures consistency across environments (local and cloud).
Simplifies deployment by packaging the app with its dependencies.

b)

Manual setup with EC2 user data scripts.
Use Elastic Beanstalk or build native binaries and upload them.

c)

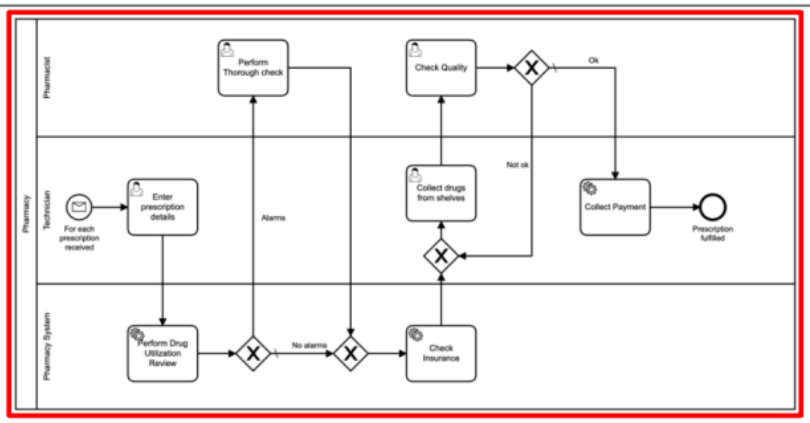
Use CI/CD tools like GitHub Actions, GitLab CI, or Jenkins for running unit tests locally and remotely.

Business processes

Consider the following process model description that the **PharmacyComp** company follows when a client submits a drug prescription.

The process starts when a technician receives a new prescription. The process interest involves three participants: Technician, Pharmacist and Pharmacy System. First, the technician enters the prescription details. Afterwards, the pharmacy system performs a drug utilization review. If an alarm is triggered, then the Pharmacist Perform a Thorough check. Otherwise, the pharmacy system checks the insurance coverage (3rd party company) to verify credit.
If drugs are covered by insurance, then the technician collect the drugs from shelves and the Pharmacist check its quality.
If drugs quality is OK the technician collects the payment and the process ends. Otherwise, technician repeats the drugs collection from shelves and the Pharmacist check its quality again.

1) Design the **PharmacyComp** process using BPMN notation, for a non-executable process, and reusing the existing elements in the following pool:



2) Now, consider that the company wants to automate this process to deal with drug prescription in a more efficient manner.
   a) Considering the CAMUNDA engine capabilities, explain how this part of the previous universe of discourse could be implemented?
   "…the technician enters the prescription details …"
   Explain, in detail, the required steps to implement this part of the process.

   b) Considering the CAMUNDA engine capabilities, explain how this part of the previous universe of discourse could be implemented?
   "…the pharmacy system checks the insurance coverage (3rd party company) to verify credit …"
   Explain, in detail, the required steps to implement this part of the process.

a)

To implement this with Camunda:

- Model a User Task in BPMN (e.g., "Enter Prescription Details").
- Assign the task to a user group (e.g., technicians) via task candidate groups.
- Use Camunda Tasklist to display the task to the technician.
- Technician logs in, fills the form (via embedded or external form), and submits it.
- Data is passed to the next BPMN task via process variables.

b)

To implement this as a service task:

- Create a Service Task in BPMN (e.g., "Check Insurance").
- Configure it as an external task or use a Java/REST connector.
- The pharmacy system (or a worker) polls for this task (if using external task pattern).
- It then calls the insurance API (3rd party system).
- The response (e.g., coverage approved/rejected) is stored as a process variable.
- The workflow proceeds based on the response using a gateway.

Business processes

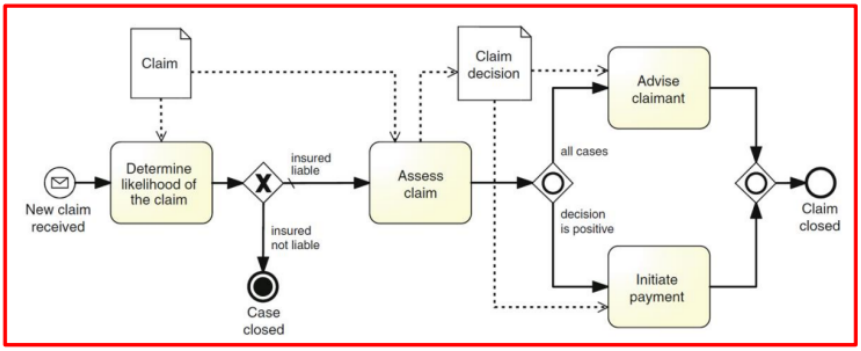
Consider the following process model description that the **InsuranceComp** company follows when a client submits an insurance claim.

```
The process starts when a customer submits a new insurance claim.
Each insurance claim goes through a two-stage evaluation process.
```

[redacted]

```
First of all, the liability of the customer is determined. Secondly, the claim is
assessed in order to determine if the insurance company has to cover this liability and
to what extent.
If the claim is accepted, payment is initiated, and the customer is advised of the
amount to be paid. All tasks except "Initiate Payment" are performed by claim handlers.
There are three claim handlers.
The task "Initiate Payment" is performed by a financial officer.
There are two financial officers.
```

3) Design the **InsuranceComp** process using BPMN notation, for a non-executable process, and reusing the existing elements in the following pool:



4) Now, consider that the company wants to automate this process to deal with claims in a more efficient manner.

1) Considering the CAMUNDA engine capabilities, explain how this part of the previous universe of discourse could be implemented?

```
"…There are three claim handlers…"
```

Explain, in detail, the required steps to implement this part of the process.

2) Considering the CAMUNDA engine capabilities, explain how can the

```
"…the liability of the customer is determined …"
```

be implemented?
Explain, in detail, the required steps to implement this part of the process.

[redacted]

3) Considering the CAMUNDA engine capabilities, propose two different implementation options to inform the client about the **InsuranceComp'** decision?
Explain, in detail, the required steps to implement both options.

1. Configure identity service with three user accounts for claim handlers, Create "claim-handlers" group and assign the three users to it, Set the "Claim decision" task's candidate group to "claim-handlers", Deploy the process and use Camunda Tasklist for task management
2. Several ways: Model the service task in BPMN Create DMN decision table with liability rules, Configure external task worker or REST connector, Define input/output variable mappings, Handle exceptional cases with boundary events
3. Option 1: Automated Email Notification Service Implementation:

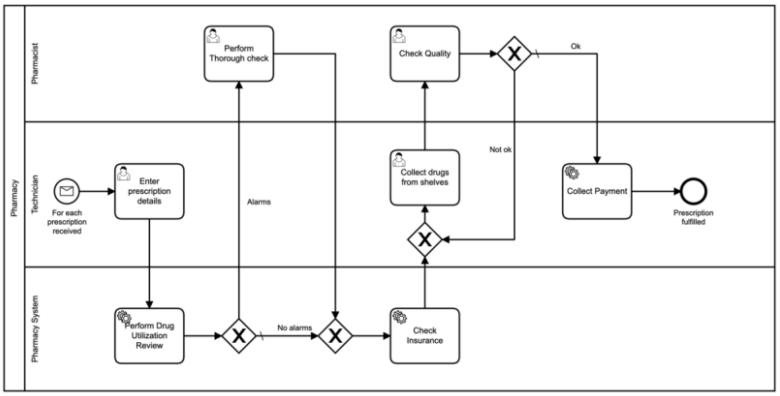
   Create "Send Decision Notification" service task after claim decision Use Camunda's email connector with SMTP configuration Create email templates for approved/rejected claims using process variables Configure as async task with retry mechanism for failed deliveries

Option 2: Manual User Task Communication Implementation:

Create "Notify Customer" user task assigned to customer service group Design form showing claim details and decision rationale Provide communication method options (email, SMS, phone, postal mail) Include documentation fields for communication tracking

Executable BPMN in Camunda

Consider the following process model description that the **PharmacyComp** company follows when a client submits a drug prescription.



Consider that this company wants to automate this process to deal with prescriptions in a more efficient manner using the CAMUNDA engine capabilities.

4) Explain and discuss, in detail, the two alternative solutions (Push and Pull) to implement the *"Collect Payment"* Task?

5) Describe, in detail, two technological alternatives that are available to trigger the instantiation of this process model, as depicted in the next figure.

6) Propose, in detail, a change in the process model to inform the client about the **PharmacyComp'** process result execution. You can design the change directly in the business process model and/or explain it textually.

4) Push: The process engine assigns the task directly to a user or system when it reaches that point in the workflow. The assignee is notified via Tasklist or other means.

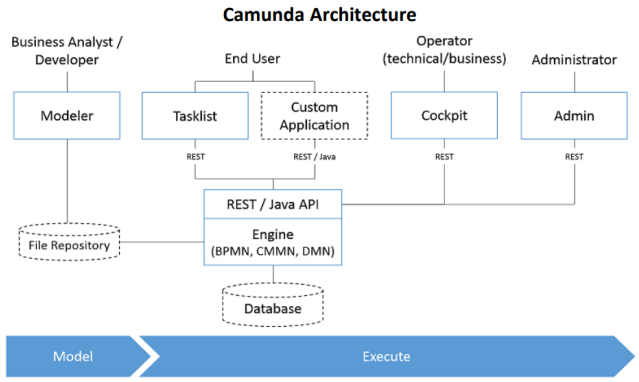
Pull: A worker (e.g., POS system or external task client) polls Camunda for available tasks and "pulls" the Collect Payment task when ready to process it.

5) REST API: An external system sends a POST request to Camunda's /process-definition/key/key/start endpoint when a new prescription is received.

Message Event: A BPMN message start event is used to trigger the process when a message (e.g., PrescriptionReceived) is sent by another system.

6)Model Change: Add a new Service Task (e.g., Notify Client) after the Collect Payment task, using an external system or message to inform the client.

Alternative: Use a Send Task or Message End Event to trigger an email, SMS, or app notification informing the client their prescription is ready or processed.



1) Considering the CAMUNDA architecture depicted in the previous Figure, identify the component (or the components) that is (or are) responsible to execute the instances of a business process?

2) Consider that multiple instances of a given business process, at the same time, are required to be executed. This is a functionality supported by CAMUNDA. What is the CAMUNDA mechanism that can differentiate each instance of the given business process? Explain with an example how it works.

3) Considering the Camunda architecture, what is the difference between an *"assignee"* and a *"candidate groups"* defined within the context of a BPMN task? In which situation should you use one or the other?

4) Considering the following figure, where task B is being implemented as a http-connector, what are the mandatory configurations to enable such connectivity?

---

1)The Engine component (BPMN, CMMN, DMN) is responsible for executing instances of business processes. It interprets the process models and manages the execution flow.

2)Camunda uses Process Instance IDs (unique system-generated) and optionally Business Keys (user-defined identifiers). For example, when starting an "Insurance Claim" process, you can set a business key like "CLAIM-2024-001" for John's claim and "CLAIM-2024-002" for Mary's claim, making instances easily identifiable and searchable by business-meaningful identifiers

3)Assignee: Assigns a task to a specific individual user (e.g., assignee="john.doe")
Candidate Groups: Assigns a task to a group of users who can claim it (e.g., candidateGroups="sales-team")

When to use:
Use assignee when you know exactly who should do the task
Use candidate groups when any member of a team can handle the task (load balancing)

4) For a task implemented as an HTTP connector, the mandatory configurations are:

URL: The endpoint to call
HTTP Method: GET, POST, PUT, DELETE, etc.
Headers: Required HTTP headers (e.g., Content-Type, Authorization)
Input/Output Variable Mapping: How to pass data to/from the HTTP call

BPMN versus DMN

Consider the following DMN decision table:

Meal — Hit Policy: Unique

	When Season (string)	And Number of guests (string)	Then Meal (string)	Annotations
1	"Spring"	<= 4	"Green asparagus"	
2	"Spring"	5	"White asparagus"	
3	"Spring"	[6..8]	"Spinach"	
4	"Spring"	>= 9	"Pasta"	
5	not("Spring")	-	"Lasagne"	
+				

1) Design an equivalent BPMN model using gateways, that is able to perform exactly the same behavior.
2) Discuss advantages and disadvantages of using a DMN decision table instead of using a BPMN model with gateways.
3) How can a DMN decision table be linked with a BPMN process using CAMUNDA? Give an example.

1)Start event, Exclusive gateway checking Season == "Spring", If Spring: Second exclusive gateway checking Number of guests (less or equal 4, =5, [6,8], more or equal 9), Each path leads to a service task setting the meal variable ("Green asparagus", "White asparagus", "Spinach", "Pasta"), If not Spring: Direct path to service task setting meal to "Lasagne" All paths converge to end event

2)DMN decision tables offer several advantages over BPMN gateways: they are business readable allowing non-technical users to understand and modify rules, highly maintainable since you can add or remove rules without changing the process structure, provide centralized logic with all decision rules in one place, and support independent version control of decision tables. However, DMN also has disadvantages including additional complexity requiring understanding of DMN notation, tool dependency needing DMN-capable tools, and less visual flow since decision logic is separated from the process flow.

3)Add Business Rule Task to BPMN process, Set implementation to "DMN", Configure Decision Reference to point to DMN table ID (e.g., "meal-decision"), Map input variables (Season, Number of guests), Map output variable (Meal)