Initial architecture design

This document serves to substantiate choices made for the initial architecture design's critical choices that are hard to change later on in the project.

Scalable architecture designs

When talking about making scalable architecture, the first challenge that comes to mind when looking at ICAP's goal is that the application will have near real-time chat functionality and forums that update live on the client. In order to achieve this, a good choice, with scalability in mind, would be to make the architecture be event-driven.

Event-driven architecture

Event-driven architecture (EDA) is a software architecture paradigm that promotes the production, detection, consumption of, and reaction to events. An event represents a change in state or an update that is significant to a system. For example, an item being placed in a shopping cart on an e-commerce website, a user checking in for a flight, or a password being reset can all be considered events. EDA allows systems to detect, process, manage, and react to real-time events as they occur, notifying all the apps, systems, and people that need to know about the event to react in real time. wikipedia, SAP, Confluent, Amazon AWS

There are two main patterns for transmitting events in an EDA: publish/subscribe and event streaming. In the publish/subscribe pattern, event consumers subscribe to messages and channels published by event producers. When an event is published, it is sent directly to all subscribers via a broker. To avoid duplication, events cannot be replayed or accessed once consumed – they are deleted by the broker. In the event streaming pattern, producers publish entire streams of events to a broker. Consumers subscribe to the stream and can read from any part of it, consuming only the events that are relevant to them. With this pattern, events are retained by the broker even after they are consumed. SAP

EDA can be built on four logical layers: the sensing of an event, the creation of its technical representation in the form of an event structure, the selection and execution of the appropriate reaction by an event processing engine, and the downstream activity where the consequences of the event are shown. This approach promotes flexibility, scalability, and resilience, and is suitable for modern applications with complex workflows, real-time event processing, and event-driven integrations. wikipedia, Confluent

There are three general styles of event processing: simple, stream, and complex. Simple event processing concerns events that are directly related to specific, measurable changes of condition. Stream event processing involves processing a constant flow of data in real time using a data streaming platform. Complex event processing allows patterns of simple and ordinary events to be considered to infer that a complex event has occurred. wikipedia, SAP

The benefits of EDA derive from how systems and components are loosely coupled. This can facilitate independent development and deployment of systems, improved scaling and fault tolerance, and integration with external systems. However, it's important to note that adopting an EDA may require rethinking how you view your application design. For example, workloads that require consistent low-latency performance are not good candidates for EDA. Also, with many events flowing through different services in an architecture at any

given point of time, such workloads often require eventual consistency, making it more complex to process transactions, handle duplicates, or determine the exact overall state of a system. Amazon AWS

Prioritize microservices or serverless compute?

Both Microservices and serverless compute can be set up to work with EDA, but which one should be chosen when it comes to ICAP's requirements? To summarize the general consensus around the choice between the two architectures is that serverless architecture is a good choice if you need a fast, event-driven application. Microservices are a good choice if you need a highly available application to handle a lot of traffic.

The microservice architecture is more distributed, where each service is its self-contained unit. This can make development and deployment more complex and provide more flexibility and scalability.

On the other hand, serverless architecture is more focused on individual functions or pieces of code that are executed in response to events. This makes it simpler to develop and deploy but can limit the flexibility and scalability of the overall system.

Serverless functions have a set time limit that they can run for, known as their runtime. Every provider has different limitations on how long a function can run - on AWS Lambda, for example, a function can run for 15 minutes. The reason for this is that functions are meant to be short-term processes that use up minimal RAM.

Microservices have an advantage over serverless architectures because vendors' requirements do not limit them to runtime, storage space, and RAM. This is especially beneficial for complex, long-term operations that involve storing and manipulating huge amounts of data.

Conclusion about architecture design choices

When it comes to ICAP's goal and requirements and the information given above, the use of microservices should be prioritised over serverless compute. This is due to the needs of creating message functionality and persisting and live updating lots of forum data. Serverless compute also has its niche and will also most likely be used within the project for certain functions.

Next steps

The next step would be to choose an appropriate programming language that fits the needs for the project and the semester. When choosing a programming language for microservices architectures combined with event-driven architecture, there are several languages that stand out due to their robust support for concurrency, distributed systems, and event-driven programming. Here are a few of them:

- Java: Java, with the Spring Boot framework, provides support for building microservices and can handle event-driven architectures effectively. The Spring framework has projects like Spring AMQP and Spring for Apache Kafka that help in creating Kafka- or RabbitMQ-based messaging solutions spring.io. Java's robustness, maturity, and wide community support make it a good choice for such architectures.
- C# (.NET): .NET provides a framework for building microservices. It supports asynchronous programming and event-driven architecture with the help of message brokers and service buses like Azure Service Bus, RabbitMQ, NServiceBus, MassTransit, or Brighter learn.microsoft.com.
- Node.js: Node.js is built around an event-driven, non-blocking I/O model, making it suitable for data-intensive, real-time applications. Its ability to handle multiple concurrent connections with a single server process makes it a great choice for microservices architecture.

It's also worth noting that the choice of language will depend on the main developer's expertise and how they want to challenge themselves within the current semester.