Milestone 3 SDT Project
SkillHub

by FINICHIU Eduard-Adelin & JERCAU Hadasa-Stefana

Milestone 3 Requierment:

Investigate, describe and evaluate three different software architectures for your system, each one using different architectural styles.

Overview of the Approach of Milestone 3 for SkillHub project:

1. Monolithic Architecture
2. Microservices Architecture
3. Event-Driven Distributed Architecture

---

1) Monolithic Architecture

Description of structure:

All functionality (UI, business logic, persistence, and notification logic) lives in a single deployable application (one jar/war or a container image). Internally the app is modular (packages or layers): Presentation (web UI / REST controllers), Service layer (user management, job management, bidding, notifications facade), Data Access layer (repositories/ORM), and shared utilities (configuration singleton, factories, builders, facades).

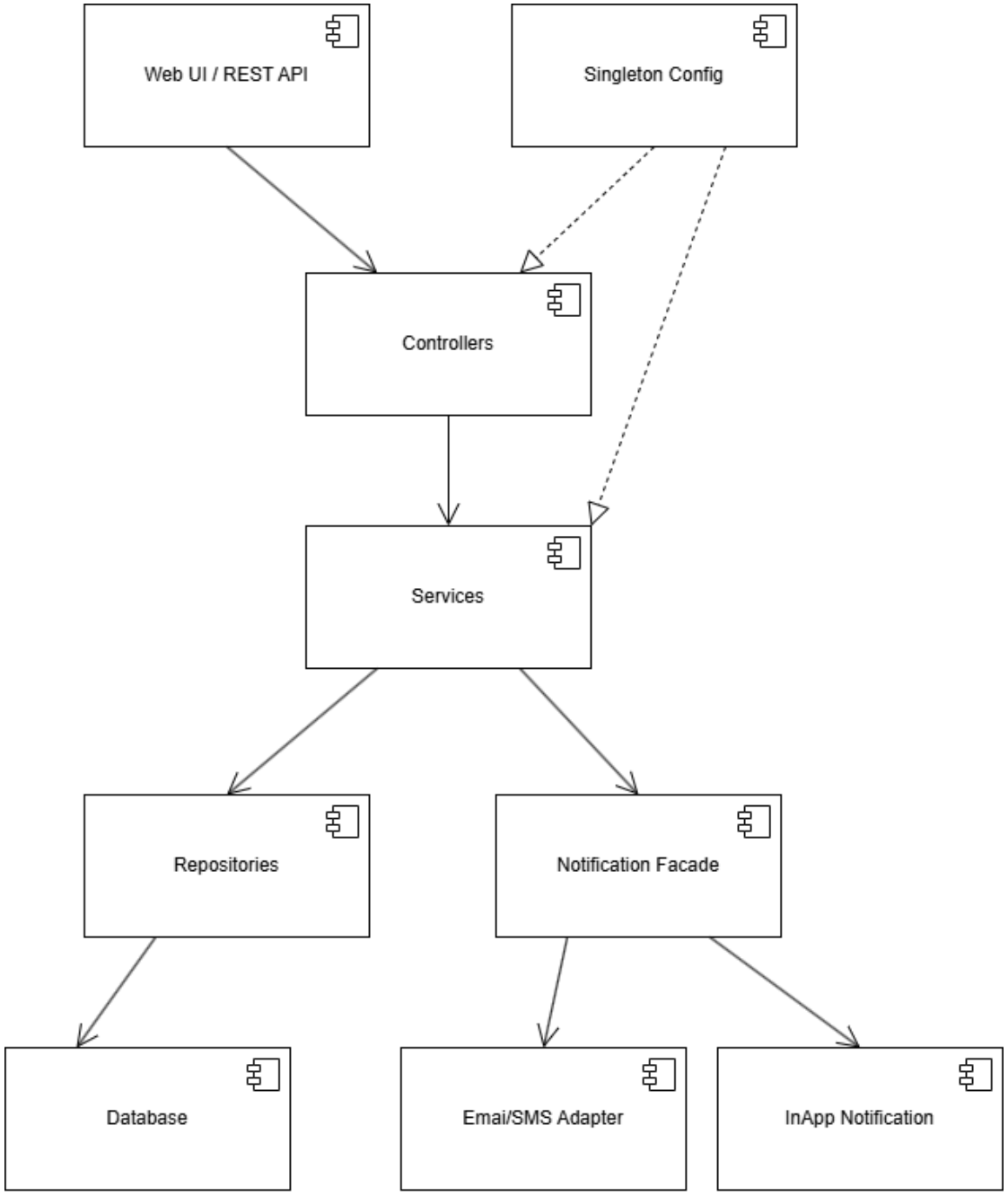
Data flow:

Example for New job posting:

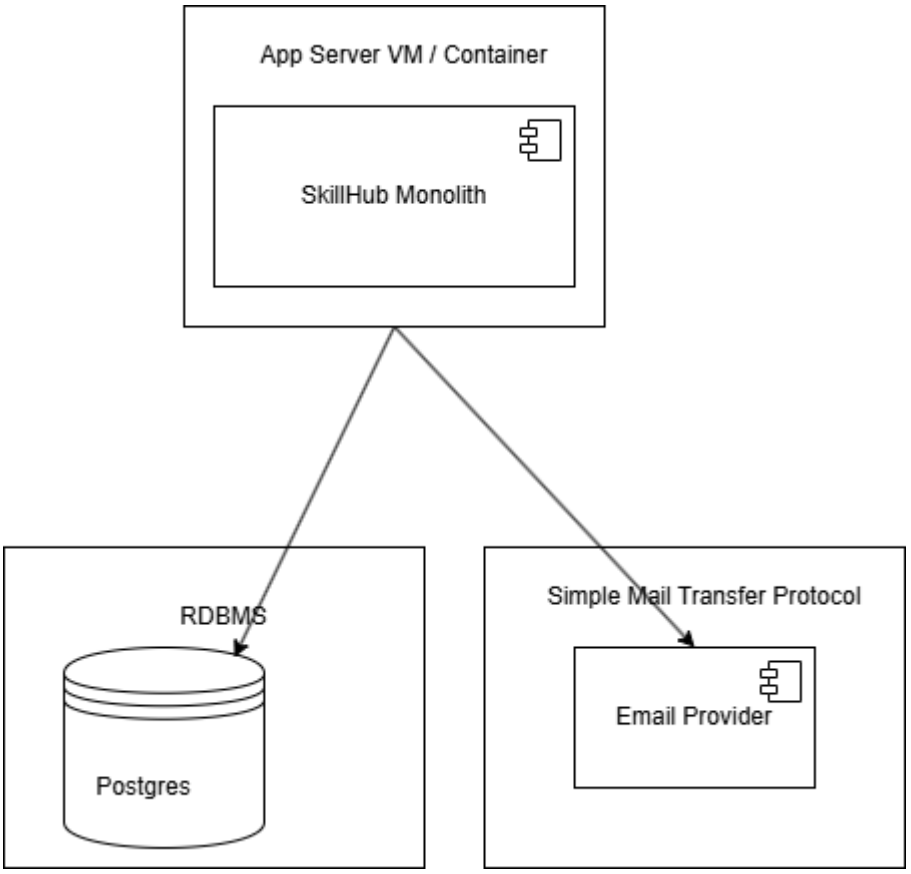
1. Client submits job form → HTTP POST to JobController.
2. The controller calls JobService (Service layer) which uses JobBuilder to construct the job entity.
3. JobRepository persists the job to the single relational DB.
   NotificationFacade triggers NotificationService (internal) which queries interested freelancers and creates notifications (stored in DB or pushed).
4. UI and APIs return success.

Diagrams:

Component Diagram:



Deployment Diagram:



Pros & Cons:

Pros (for SkillHub):

1. Simplicity: Easy to implement and reason about, good for the academic milestone 2.
2. Lower overhead: Single codebase, single deployment pipeline; fewer infrastructure components, easy to observe and work on.
3. Transactions: Simpler to maintain ACID (Atomicity, Consistency, Isolation, and Durability) transactions across operations (e.g. job creation + initial bid). Easier to link files between them, no complicated API needed.
4. Easier local dev & testing: No need to run multiple services.

Cons (for SkillHub):

1. Scalability limits: Harder to scale parts independently (e.g. notification bursts).
2. Deployment risk: Any change requires redeploying the whole app. If something breaks, the whole thing breaks.
3. Long-term complexity: As features grow, the codebase can become large and harder to maintain. Not possible to separate stuff and work independently.
4. Barrier to polyglot tech: Cannot mix different technologies per service easily. (not advised for a big scale project like SkillHub intends to be)
5. Testing: Unit tests fine, but integration tests may be heavier. Problems are also harder to identify, without being able to track very fast the source of error. Solving problems may result in creating other problems (common monolit behaviour).

---

2) Microservices Architecture

Description of structure:

System split into small, independently deployable services by business capability. In the case of the SkillHub project, the following small / independently deployable services might be a very good idea. The list includes the following:

1. Auth/User Service: user accounts, profile management.
2. Job Service: job creation, editing, browsing.
   Bidding Service: proposals and bid lifecycle.
3. Notification Service: sending notifications, interfaces to email/push.
4. Gateway / API Aggregator: single entrypoint for clients (HTTP gateway / API Gateway).
5. Shared DB per service (each service owns its data) or a combination where necessary.
6. Service Discovery (if dynamic), and Config Service for centralized configuration.

Data flow:

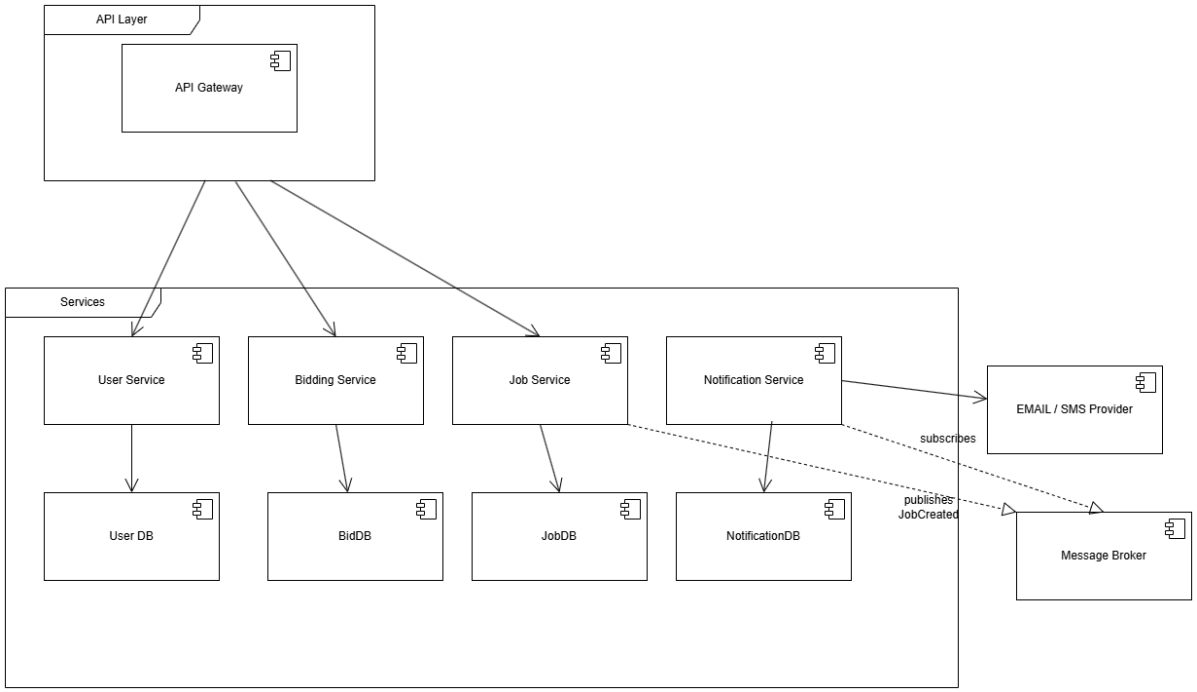
Example for New job posting:

1. Client → API Gateway → Job Service endpoint.
2. Job Service writes to its DB and emits an event JobCreated.
3. Notification Service subscribes to JobCreated event and creates notifications for matching freelancers (reads data either from the event or queries User Service - can be also called Profile Service - if more info needed).
4. Bidding Service is notified or remains passive until freelancers bid.

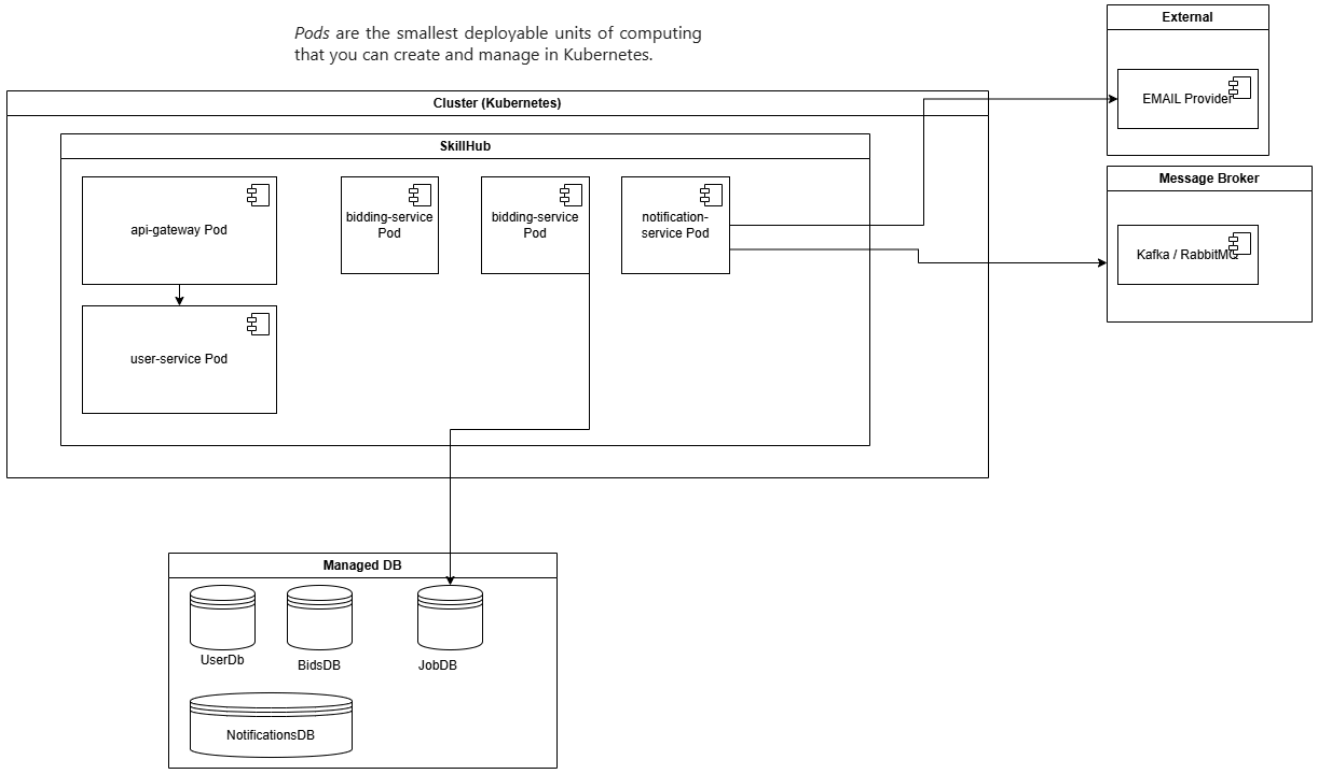
This example showcases both the data flow but also highlights the core advantages of the Microservices architecture.

Diagrams:

Component Diagram:



Deployment Diagram:



Pros & Cons:

Pros (for SkillHub):

1. Independent scaling: For example: Scale Notification service separately during high load (e.g. many new jobs are posted on the platform in a short period of time).
2. Technological freedom: Different services can use best-fit tech stacks (e.g. Node for real-time notifications). This ensures that in the future we will not be stuck with the same technology.
3. Fault isolation: A failure in Bidding service doesn't necessarily take down Job browsing. The Bidding service does cooperate with other services but it is not riding in the same boat. As long as a service does not fully rely on the affected service, the first mentioned will not have serious issues.
4. Team autonomy & parallel development: If a project expands, teams can own services and manage them with great care. For example, I will be able to build the Bidding service while my colleague will build the User Service. This also ensures better cooperation between teams.
5. Easier CI/CD (Continuous Integration and Continuous Delivery/Deployment) for individual components: Faster iteration on small services.

Cons (for SkillHub):

1. Operational complexity: Need containers/orchestration, logging, monitoring, tracing, service discovery. Attention will be split into many directions, the project manager shall not neglect any service in favour of others since the final functionality of the project consists in the ability to merge all services into one functioning application.
2. Distributed transactions: Harder to maintain consistency; eventual consistency patterns needed.

3. Higher infrastructure cost: Multiple services lead to more infrastructure & management overhead. Will they run in the Cloud, on our Servers, how will we connect them, how do we optimize them, etc. ?
4. Testing complexity: Integration tests require many services or mocks. The integration is essential in order to have a functional app.
5. Design overhead: Requires careful API design.

---

3) Event-Driven Distributed Architecture

I choose Event-Driven Architecture because SkillHub benefits from decoupled reaction to domain events.

Description of structure:

Core idea: services react to domain events delivered through a message broker (e.g. Kafka). Command queries can still be implemented via HTTP endpoints, but side effects are handled via events.

The following list of services can represent the foundation of the Event-Driver Architecture in the case of SkillHub project:

1. Job Command Service (synchronous API for job CRUD): persists job and emits JobCreated.
2. Notification Worker: consumes JobCreated to compute recipients and send notifications (email, in-app).
3. Search/Indexing Worker: consumes events to update search index.
4. Activity Feed Worker: builds activity streams.
5. Bidding Service: receives events or provides its own commands/events (e.g., BidPlaced).
6. User/Profiles Service: emits UserUpdated.
7. Event Store / Broker: Kafka or event bus; may also include an event store for event sourcing if desired.

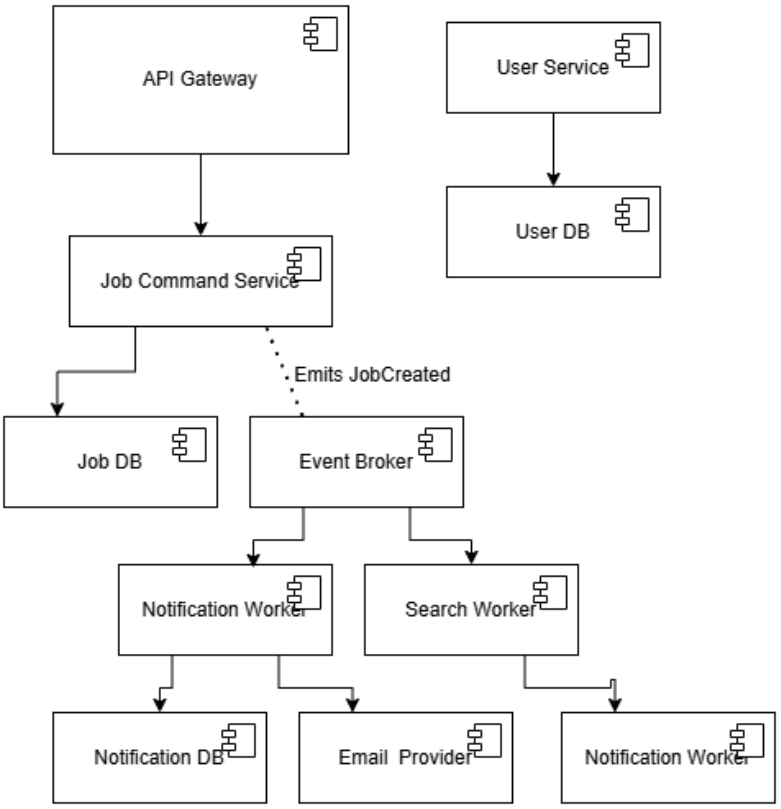
Data flow:

Example of data flow in the case of Job posting:

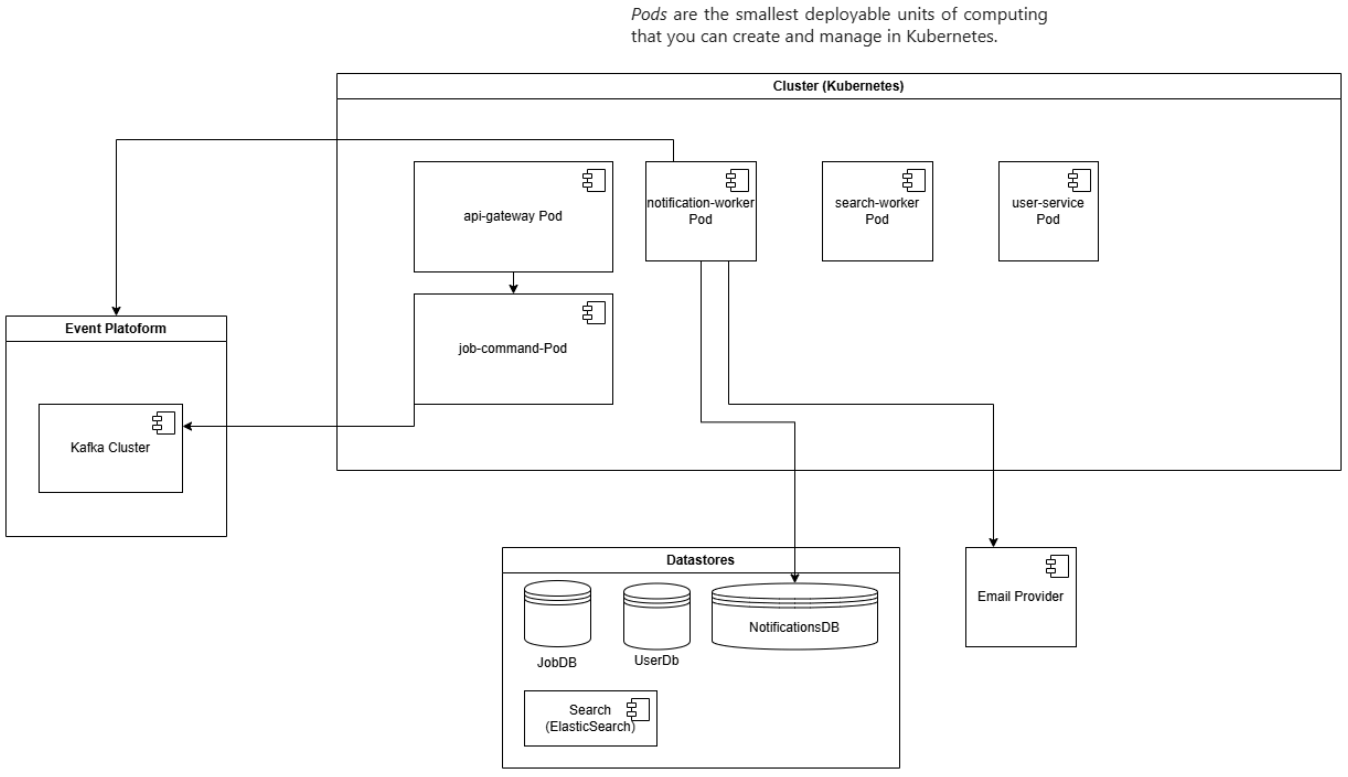
1. JobCommandService receives request, validates, persists job.
2. JobCommandService emits a JobCreated event to broker with job payload.
3. Notification Worker consumes JobCreated, looks up matching freelancer profiles from User Service (or has precomputed matches), and emits NotificationCreated events or calls Notification Provider.
4. Search Worker consumes JobCreated and updates search index.

Diagrams:

Component Diagram:



Deployment Diagram:



Pros & Cons:

Pros (for SkillHub):

1. Loose coupling: Producers and consumers evolve independently. It is very good for adding features like analytics, activity streams.
2. Resilience & buffering: Brokers absorb spikes (e.g. many jobs posted), enabling backpressure handling.
3. Extensibility: Add new consumers (e.g. indexing, analytics) without changing producers.
4. Responsiveness: Near real-time processing of side-effects.
5. Good fit for notifications & analytics: Natural mapping of domain flows.

Cons (for SkillHub):

1. Complexity: Requires reliable broker, handling at-least-once/duplicate events, schema evolution for events.
2. Operational overhead: Manage Kafka cluster.
3. Eventual consistency: Data will be eventually consistent; some flows may require compensating actions.
4. Debugging harder: Tracing flows across asynchronous boundaries is more complex.

Comparison & Final Recommendation

Comparison

1. Monolith: Best for speed of initial development, simple deployment, easy transactions. Poor at scaling selected components.
2. Microservices: Best for independent scaling, team autonomy, and long-term maintainability at cost of operational complexity.
3. Event-Driven: Best for extensibility, decoupling, and handling async workflows like notifications and analytics; needs strong ops and careful design.

Reasons to consider when choosing between the 3 architectures

SkillHub is an academic project whose goals include demonstrating good architecture, modularity, scalability and maintainability. A well-structured modular monolith achieves these goals for the course: you can show design patterns (Singleton, Factory, Builder, Facade), produce clean class/sequence diagrams, and provide deployment diagrams, all without the heavy infra baggage.

This was done for Milestone 2, proving that Monolit has been a good decision.

For full production-grade demands (live multi-tenant marketplace), Microservices + Event-Driven would be preferable.

In our case, Monolit would be a good choice. Another pragmatic choice can be Microservices architecture, altho it can be considered as too advanced for a university project who tries to emphatise more things related to SDT and is not solly focussed on microservices architecture.

Final Decision

For a full scale app the Microservices + Db + Event-Driven (Kafka) would be great. In our case, if the teacher agrees, Microservices or Monolith would be better. The most efficient would be the Monolit architecture sincer our app will not need future expansion. The programmers can easily fall into technical decisions that do not really solve the real life problems. Sometimes simple

technology can give better results than complicated ways of doing the same thing. If the SDT Project will evolve, a Monolith architecture can be adapted to become a Microservice architecture by slowly breaking the monolith into multiple microservices.

In our case, the SDT Project involves 2 more milestones involving only microservices. This is a decisive factor for our project. This leads us to choose microservices architecture, being more suitable for the SDT subject that we are doing at the university.