Milestone 3: Architecture Investigation, Evaluation, and Selection

The following report investigates, describes, and evaluates three different software architectures for the **B2B Land Listing Platform** project, tailored to meet the Grade 10 criteria.

---

1. Monolithic Architecture

The Monolithic architecture is a traditional approach where all application functionalities (UI, business logic, data access layers, etc.) are combined into a single, tightly-coupled codebase and deployed as a single unit.

A. Structure, Interactions, and Data Flow

- **Structure:** All core features—**Listing Management**, **Offer Flow**, **Notifications**, and **Search**—reside within a single application module.
- **Interactions:** Components communicate directly via in-process method calls within the same memory space.
- **Data Flow:** A single, shared relational database (e.g., PostgreSQL or MySQL) is utilized by all application functions.

B. Component and Deployment Diagram

Component	**Description**
B2B Platform Application	A single deployable package (e.g., a .jar or .war file) containing all application logic.
Shared Database (DB)	A single schema used for all persistence needs.
Load Balancer	Distributes incoming traffic across multiple instances of the single application (for horizontal scaling).

C. Pros and Cons for the Project

Pros	**Cons**

Simple Development/Deployment: Easiest to set up and deploy, especially during the initial Proof-of-Concept (PoC) stage.	**Difficult Scaling:** To scale the high-demand **Search** feature, the entire application (including less-used **Offer Management**) must be scaled, leading to resource inefficiency.
Easy Debugging: All code runs within a single process, simplifying tracing and testing.	**Maintainability Risk:** The codebase will become complex and harder to understand and evolve as the platform grows.
Low Initial Cost: Requires less server infrastructure and management overhead.	**Technology Lock-in:** The entire application is tied to a single programming language/framework.

---

2. Microservices Architecture

Microservices architecture is a distributed approach where the application is decomposed into small, independent, business-focused services. Each service is self-contained, owning its own data and lifecycle.

A. Structure, Interactions, and Data Flow

- **Structure:** The system is divided based on business capabilities: **Listing Service**, **Offer Service**, **Notification Service**, and **Search Service**.
- **Interactions:** Services communicate synchronously via lightweight protocols (typically HTTP/REST or gRPC) or asynchronously via message queues.
- **Data Flow:** Each service maintains its own private database. Data sharing occurs through controlled API calls or by publishing events.

B. Component and Deployment Diagram

Component	**Description**
API Gateway	The single entry point for all client traffic, routing requests to the appropriate service.
Microservices	Independent, deployable units: Listing Service, Offer Service, Notification Service, Search Service.

Databases (DBs)	Dedicated databases for each service (enabling Polyglot Persistence).
Service Discovery	Mechanism for services to find each other on the network.

C. Pros and Cons for the Project

Pros	**Cons**
Independent Scaling (Crucial): High-traffic services like **Search** can be scaled independently of the **Offer Service**, maximizing efficiency.	**Operational Complexity:** Requires more robust infrastructure for deployment, monitoring, and network management.
Technology Diversity (Polyglot): Each service can use the best tool for the job (e.g., Java for Offers, Python for GIS integration).	**Distributed Transactions:** Managing transactions that span two or more services (e.g., submitting an offer that updates both Listing and Offer status) is complex.
Better Maintainability: Smaller codebases are easier to understand and faster to develop.	**Increased Development Complexity:** Developers need expertise in distributed systems principles.

---

3. Event-Driven Architecture (EDA)

EDA is a distributed style where system components communicate indirectly by publishing and consuming **events** (records of state change). Components react to events they are subscribed to, rather than making direct synchronous calls.

A. Structure, Interactions, and Data Flow

- **Structure:** Core components (producers and consumers) are organized around a central **Message Broker/Event Bus** (e.g., Apache Kafka).
- **Interactions:** Communication is entirely **asynchronous** and decoupled. A component publishes an event; other subscribed components consume and process it.

- **Data Flow:** When a transaction occurs (e.g., ListingPublishedEvent), the event is sent to the broker. Consumers (e.g., Notification Service, Search Indexer) process the event to update their local data or perform side effects.

B. Component and Deployment Diagram

Component	Description
Message Broker (Event Bus)	The central channel for event distribution (e.g., Kafka).
Producers (Publishers)	Listing Service, Offer Service (generate and publish events).
Consumers (Handlers)	Notification Service, Search Indexer (subscribe to and react to events).

C. Pros and Cons for the Project

Pros	Cons
Extreme Decoupling: Services don't need to know about each other, making the system highly flexible and scalable. Ideal for the **Notification Service**.	**Complex Debugging:** Tracking the flow of a single business transaction across multiple asynchronous events is challenging.
Real-Time Responsiveness: The system can react instantly to state changes (e.g., updating the search index immediately when a listing is published).	**Infrastructure Complexity:** Requires the management of a robust and highly available message broker.
Scalability: The broker buffers the load, allowing services to scale independently based on message throughput.	**Event Idempotence:** Consumers must ensure they process the same event multiple times without causing side effects.

---

4. Final Comparison and Selection

Feature	Monolithic	Microservices	Event-Driven (EDA)
Maintainability	Low	High	High
Scaling Granularity	Low (Scale All)	High (Per Service)	High (Per Event Handler)
Initial Development Speed	Highest	Low	Low
System Complexity	Low	Moderate	High
Best For	Initial PoC stage	Independent Feature Teams	Asynchronous Notifications/Indexing

Most Suitable Architecture: Microservices Architecture

Justification (Meeting Enterprise Requirements)

1. **Independent Scalability:** The project includes components with vastly different load profiles (**High-traffic Search** vs. **Low-traffic Offer Management**). Microservices allow the Search Service to scale aggressively without needing to allocate resources to less-demanding services, directly addressing the requirement for **scalability**.
2. **Maintainability and Modularity:** The design patterns implemented in **Milestone 2** (State, Builder, etc.) are perfectly suited to create clean, encapsulated business logic within each microservice. This ensures the long-term **maintainability** of the large enterprise system.
3. **Flexibility for Integrations:** The modular nature allows for easy integration of external services (GIS/Mapping, KYC verification) by wrapping them in dedicated microservices, utilizing the best-suited technology for the integration, thereby fulfilling the **integration** requirement.
4. **Practical Use of EDA:** Microservices can easily incorporate the benefits of EDA for asynchronous tasks. For example, the Listing Service can publish a ListingPublishedEvent, which the Notification Service consumes, achieving the **decoupling** and real-time reaction of an Event-Driven style where it matters most (notifications), without taking on the full complexity of a pure EDA system.

Therefore, **Microservices Architecture** is the most robust, scalable, and maintainable choice for the B2B Land Listing Platform