**1. Main & Subroutines**

* **Structure**: Decomposes functions into a main program and subroutines connected by function/procedure calls.
* **Pros**: Good modularity if interfaces are maintained.
* **Cons**: Poor scalability; hard to maintain as it grows.
* **Best For**: Small programs with simple workflows.

**2. Object-Oriented**

* **Structure**: Encapsulates data and behavior through objects with method calls and polymorphism.
* **Pros**: Data integrity, modularity, real-world entity representation.
* **Cons**: High coupling, not ideal for distributed applications.
* **Best For**: Complex applications requiring modular components (e.g., games, GUIs).

**3. Dataflow (Batch-Sequential & Pipe-and-Filter)**

* **Batch-Sequential**:
  + **Structure**: Runs separate programs in sequence; output from one is input for the next.
  + **Best For**: Financial transactions, batch processing.
* **Pipe-and-Filter**:
  + **Structure**: Independent filters connected by data streams (pipes).
  + **Pros**: Reusable filters, effective for text processing.
  + **Cons**: Inefficient for complex interactions or interactive tasks.
  + **Best For**: Unix utilities, data transformation tasks.

**4. Layered**

* **Structure**: Hierarchical layers provide services to upper layers; strict communication with adjacent layers.
* **Pros**: Simplified design, clear separation of concerns.
* **Cons**: Performance bottlenecks if all layers are accessed for each request.
* **Best For**: Systems with clear, separated functionalities (e.g., web applications, operating systems).

**5. Modular Monolith**

* **Structure**: Similar to layered but with modular components that can be independently developed.
* **Pros**: Simplified deployment, stronger data consistency, supports scaling towards microservices.
* **Cons**: Complexity in fully implementing encapsulation, potential tight coupling.
* **Best For**: Applications aiming for modularity without full microservices.

**6. Client-Server**

* **Structure**: Clients request resources from a centralized server.
* **Pros**: Centralized control, easy to maintain and scale.
* **Cons**: Bandwidth limitations, latency concerns.
* **Best For**: Web services, applications with straightforward server-client interactions.

**7. Implicit Invocation (Event-Based & Publish-Subscribe)**

* **Publish-Subscribe**:
  + **Structure**: Asynchronous communication; publishers send messages to subscribers via a broker.
  + **Pros**: Decoupled communication, scalable.
  + **Cons**: Difficult to guarantee message delivery.
  + **Best For**: Distributed systems with diverse components (e.g., financial systems, IoT).
* **Event-Based**:
  + **Structure**: Components emit and process events asynchronously.
  + **Pros**: Highly scalable, flexible.
  + **Cons**: Event loss or overload possible.
  + **Best For**: Systems requiring real-time updates (e.g., sensor networks, UIs).

**8. Peer-to-Peer (P2P)**

* **Structure**: Decentralized; each peer acts as both client and server.
* **Pros**: Scalable, resilient to node failures.
* **Cons**: Security challenges, resource discovery.
* **Best For**: Distributed file sharing, messaging systems.

**Tutorial Case**

**Reasons for Layered Architecture Fit**

1. **Clear Separation of Concerns**:
   * Your C4 diagrams illustrate a clear separation of responsibilities across different layers, especially in Level 2 and Level 3. For instance, you have distinct layers for user interactions (Web Application, Main Page Application), business logic (e.g., User API Application, Employee API Application), and data access (User Database and Shop Database). This aligns well with the layered architecture style, where each layer provides services to the layer above it.
2. **Hierarchical Structure**:
   * The system is designed so that user-facing applications (Web Application, Employee Web Application) interact with backend APIs (User API Application, Employee API Application), which in turn interact with databases. This structure exemplifies the hierarchical, layer-by-layer service provision typical of layered architecture.
3. **Modularization and Scalability**:
   * Layered architectures support modularization, making it easier to develop, maintain, and scale each part of the system independently. Your model’s design allows for each layer to scale or modify independently, such as expanding database capacity or scaling the web applications without affecting other layers directly.
4. **Alignment with Web Applications**:
   * The Layered architecture is well-suited for web applications that involve presentation, business logic, and data storage layers. Since your system involves multiple user roles and complex functionalities, like order management and payment processing, the layered architecture offers a straightforward, manageable approach.

**Microservices Justification:**

1. **Scalability Needs**: Mr. Mushroom’s high order volume (8,000+ daily) and peak season demands require independent scaling. Microservices allow us to scale only critical services (e.g., Order and Payment) without affecting the whole system.
2. **Fault Isolation**: Microservices help contain failures to individual services, preventing issues in one part (e.g., Payment) from impacting others, which is crucial for a reliable e-commerce experience.
3. **Complex User Requirements**: The system serves different user roles (Customers, Employees, Managers) with specific functions. Microservices let us build and manage these roles separately, improving modularity and flexibility.
4. **Future Expansion**: The architecture supports future growth (e.g., international expansion) and easy integration with new services or technologies, ensuring long-term adaptability.
5. **Team Efficiency**: Microservices enable parallel development and faster feature releases, allowing teams to work independently and deploy updates quickly.

**Summary**: Microservices offer the scalability, resilience, and agility that Mr. Mushroom’s platform needs, making it a better fit than a monolithic approach for handling varied users, high load, and growth potential.