**Case Study: - Monolith vs Serverless**

Server-side building infrastructure is the various technologies and frameworks used by us to construct and manage the backend side of web applications and services. This might involve setting up and maintaining servers, databases, APIs, and other critical components that operate on the server side of an application. Some of the key elements involved in this infrastructure building are as follows:

1. **SERVERS:** Servers are generally of three types. First being Web Servers that mainly used for handling HTTP requests given by the clients. Second is Application Servers which help us in executing application logic and interact with other web servers. Third and last is Database Servers whose main task is to manage and store data.
2. **SERVER-SIDE LANGUAGES AND FRAMEWORKS:** Many languages and frameworks are used to perform this task. Each having their own pros and cons.
3. **DATABASES:** Use of structured query language (SQL) and including tables with relationship. We also use to handle unstructured or semi-structured data.
4. **APIs AND SERVICES:** We use HTTP requests to interact with web services. It allows clients to request specific data and get precisely that data in response.
5. **AUTHENTICATION AND AUTHORISATION:** It verifies user identity and determines what he authenticated users can access.
6. **HOSTING AND DEPLOYMENT:** It also involves renting and owning physical or virtual servers and using cloud operations to host applications.
7. **INFRASTRUCTURE MANAGEMENT AND AUTOMATION:** We need to automate server setup and configuration and ensure the dependency of packages applications into containers.
8. **SECURITY:** It helps in protecting servers from unauthorised access.

Server-side building infrastructure can be categorized based on various criteria such as deployment models, architecture, and the technology stack used. When we talk about architectural models of server-side building infrastructure, there are many models which we utilise to do so but we can consider Monolith and Serverless as the two main architectural model.

**MONOLITH:**

It is a traditional approach to building software applications where all components and functionalities are integrated into a single, unified system.

Some of its characteristics are:

1. SINGLE CODEBASE: The application is built as a single codebase. All the functionalities are combined in one application.
2. TIGHTLY COUPLED COMPONENTS: All the components of the system are interdependent and share the same resources. This implies that changing anything in one part of a system can alter other parts.
3. UNIFIED DEPLOYMENT: All the components are deployed as a single unit. All components are packed together and can be updated all together.
4. CENTRALIZED DATA ACCESS: It uses a single, central database.

**Advantages**

1. **Simplicity**: Easier to develop and test since everything is in one place. Developers can understand the whole system without having to manage multiple services.
2. **Performance**: Because all components are tightly integrated, communication between them can be faster compared to distributed systems.
3. **Deployment**: Simplified deployment process as there’s only one artifact to deploy. This can make deployment easier, particularly in small-scale applications.
4. **Consistent Environment**: Everything runs in the same environment, reducing issues related to different environments (e.g., development vs. production).

**Disadvantages**

1. **Scalability Challenges**: Scaling a monolithic application can be challenging. Since all components are intertwined, scaling one part of the application often requires scaling the entire application, which can be inefficient.
2. **Maintenance Complexity**: As the application grows, the codebase can become complex and difficult to manage. Changes in one part of the application can unintentionally affect other parts.
3. **Deployment Risks**: A change in any part of the application requires redeploying the entire application. This can lead to more frequent downtime and potential issues with new deployments.
4. **Technology Limitations**: Monolithic applications are often tied to a specific technology stack. Adopting new technologies or frameworks can be more challenging as it may require a complete overhaul of the existing system.
5. **Lack of Flexibility**: It's harder to use different technologies for different parts of the application. For example, you can’t easily use a different programming language for a specific functionality without impacting the entire system.

**Use Cases**

Monolithic architecture can be suitable for:

* **Small to Medium-Sized Applications**: Applications that do not have high scalability requirements and are relatively straightforward.
* **Startups and Prototypes**: Rapid development and deployment for initial releases and proofs of concept.
* **Applications with Low Complexity**: Systems where tight integration of components is beneficial and where high scalability is not a primary concern.

**Example**

An e-commerce website where the user interface, business logic (e.g., processing orders, managing inventory), and data access (e.g., database interactions) are all bundled into a single application would be an example of a monolithic architecture.

**SERVERLESS:**

Serverless architecture is a cloud computing execution model where the cloud provider dynamically manages the infrastructure, allowing developers to focus solely on writing and deploying code. Despite the name, "serverless" does not mean there are no servers involved; rather, it abstracts away the server management tasks from the developer. Here’s an overview of serverless architecture:

**Characteristics**

1. **Event-Driven Execution**: Serverless functions are typically triggered by events such as HTTP requests, file uploads, database changes, or messages from queues.
2. **No Server Management**: Developers do not need to provision, manage, or maintain servers. The cloud provider handles all infrastructure concerns, including scaling, patching, and monitoring.
3. **Automatic Scaling**: Serverless functions automatically scale up or down based on the number of incoming requests or events. You pay only for the compute time you use.
4. **Stateless Functions**: Each function is stateless, meaning that it does not retain any data between executions. If state is needed, it must be stored in external storage systems like databases or object storage.
5. **Micro-Billing**: You are billed based on the actual compute resources consumed (e.g., execution time and memory usage) rather than for reserved server capacity.

**Advantages**

1. **Reduced Operational Overhead**: No need to manage servers, handle updates, or deal with infrastructure scaling issues.
2. **Cost Efficiency**: Pay-as-you-go pricing model where you only pay for the execution time and resources used by your functions, which can be more cost-effective than maintaining dedicated servers.
3. **Scalability**: Automatic scaling in response to workload changes without manual intervention. This is ideal for applications with unpredictable or variable traffic.
4. **Faster Development**: Developers can focus on writing code and implementing business logic rather than managing infrastructure, leading to faster development cycles.
5. **Resilience and Fault Tolerance**: Cloud providers typically offer built-in redundancy and failover capabilities, enhancing the reliability of applications.

**Disadvantages**

1. **Cold Start Latency**: When a function is called after being idle for a period, it may experience a delay (cold start) as the cloud provider initializes the environment. This can impact performance for certain use cases.
2. **Limited Execution Duration**: Functions usually have maximum execution time limits (e.g., AWS Lambda functions have a maximum of 15 minutes). Long-running tasks might require alternative approaches.
3. **Statelessness**: Functions are stateless, so managing state and data persistence requires external services, which can add complexity.
4. **Vendor Lock-In**: Heavy reliance on a specific cloud provider’s services and APIs can make it challenging to migrate to other providers.
5. **Complexity in Debugging and Monitoring**: The distributed nature of serverless functions can complicate debugging and monitoring, as tracking the execution flow across multiple services can be more complex.

**Use Cases**

1. **Microservices**: Ideal for deploying individual services or functions that handle specific tasks within a larger application.
2. **APIs and Webhooks**: Excellent for building and managing APIs or handling webhook requests due to their event-driven nature.
3. **Data Processing**: Suitable for processing data in response to events, such as file uploads or database changes.
4. **Real-Time Applications**: Effective for applications that need to handle real-time events or streaming data, such as chat applications or live data processing.
5. **Automated Workflows**: Useful for creating automation scripts or workflows triggered by events, such as periodic data transformations or scheduled tasks.

**Popular Serverless Platforms**

1. **AWS Lambda**: Offers a wide range of integration with other AWS services and supports multiple programming languages.
2. **Azure Functions**: Provides serverless computing capabilities with integration into Microsoft's cloud ecosystem.
3. **Google Cloud Functions**: Supports event-driven functions and integrates with Google Cloud services.
4. **IBM Cloud Functions**: Based on Apache Open Whisk, it provides a serverless platform with broad language support and integrations.

In summary, serverless architecture simplifies development and scaling by abstracting infrastructure management, though it introduces new considerations such as cold start latency and state management. It’s a compelling choice for modern applications that benefit from event-driven execution and cost-efficient scaling.