Migration of Monolithic Applications to Microservices

Point of View
**What is Microservice?**

**Definition**
A microservices architecture consists of a collection of small, autonomous services. Each service is self-contained and should implement a single business capability.

**Characteristics**
- Many smaller (fine grained), clearly scoped services
  - Single Responsibility Principle
  - Domain Driven Development
  - Bounded Context
  - Independently Managed
- Clear ownership for each service
  - Typically need/adopt the “DevOps” model

**Principles**
- Scalability
- Resiliency
- Decentralized Governance
- Auto - Provisioning
- Availability
- Independent & Autonomous
- Failure Isolation
- Continuous Delivery through DevOps
# Pros & Cons of Monolithic Apps

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Simplicity</strong></td>
<td><strong>Reliability</strong></td>
</tr>
<tr>
<td>• Simple to build, test and deploy.</td>
<td>• An error in any of the modules in the application can bring the entire application down</td>
</tr>
<tr>
<td>• Scale horizontally, in one direction by running several copies of the application behind a load balancer</td>
<td></td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td><strong>Tight Coupling</strong></td>
</tr>
<tr>
<td>• Monolithic applications have performance advantages due to shared-memory access which is faster than inter-process communication (IPC)</td>
<td>• Monolithic app services tend to get tightly coupled and entangled as the application evolves, making it difficult to isolate services for purposes such as independent scaling or code maintainability</td>
</tr>
<tr>
<td><strong>Cross-cutting Concerns</strong></td>
<td><strong>Understandability</strong></td>
</tr>
<tr>
<td>• Apps typically have a large number of concerns, such as logging, rate limiting, and security features such as audit trails and DOS protection</td>
<td>• Architecture is harder to understand, because there may be dependencies which are not obvious when we’re looking at a particular service or controller</td>
</tr>
<tr>
<td>• In monolithic everything is running through the same app, it’s easy to hook up components to those cross-cutting concerns</td>
<td></td>
</tr>
<tr>
<td><strong>Updates</strong></td>
<td><strong>Updates</strong></td>
</tr>
<tr>
<td>• Due to a single large codebase and tight coupling, the entire application would have to deploy for each update</td>
<td></td>
</tr>
</tbody>
</table>

[Source: Enterprise Productivity]
When to use Microservices?

Maintainability
• Application Code Maintainability
• Constant changes to business logic & demand
• Unmanageable & time consuming complex code changes

Coordination
• Coordination across different teams
• Applicability of frequent break-fixes in coordination with different teams
• Bigger the application, the larger the risk of coordination

Plugging & Unplugging
• Business need of couple or decouple a logic
• Lot of overhead as the entire application will have to be unit tested & debugged

Life Cycle Management
• Longer lifecycle management of monolithic application
• Every functional piece of the application will have to be tested, irrespective of whether they have been modified or not

Deprecated Features & Attributes
• Feature & attributes within code become deprecated
• Code refactoring become tedious task if the size of the application is considerable bigger
Challenges solved by Microservices

1. Removes dependencies between teams and enable faster code to production
2. Highly modular approach of microservices ensure better maintenance, troubleshooting by identifying issue directly in the modules.
3. Improves Developers Efficiency by reducing development time with small code bases and provides a reusability benefits over time.
4. Independent deployable components which allows new features into production without any dependencies.
5. Organizational alignment: Reduces developers learning curve and encourages teams to build more complex products and features iteratively.
6. Integration Facades: Enables integration with multiple tools like API, Message Queue, Service Bus without modifying entire application.
7. The distributed architecture of microservices enables increased performance and scalability of application.
8. Microservices can quickly adapt to market conditions like cloud transformation and containerization.
Tradeoffs by Microservices

1. **Performance**
   Microservices decreases the performance of application as the message flow & communication increases when number of Microservices increases.

2. **Asynchronous**
   Asynchronous calls when it comes to microservices are a big performance gain but comes at a price. Asynchronous programming is hard and even harder to debug.

3. **Resiliency**
   Microservices require writing Circuit breaker and Retry communication patterns to achieve resiliency. Thus increases overall complexity and maintenance of code to avoid application disruption.

4. **Eventual Consistency**
   Eventual consistency typically yields better performance, easier operation, and better scalability while requiring programmers to understand a more complicated data model. However, eventual consistency pattern has issues. Let’s say a REST API data is updated in your application which uses Entity Framework. Further the get operation in MVC’s view is supposed to reflect the changes. Microservices require multiple resources to update and relies on distributed transactions.

5. **Operational Cost**
   There is higher cost while running each of the microservices in each container as it will utilize more CPU and memory. Also reliable connectivity is required for communication between services interacting with each other. Cost of monitoring microservices in production environment is higher.
### Monolithic to Microservices - Planning

<table>
<thead>
<tr>
<th><strong>Establish an application architecture baseline</strong></th>
<th><strong>Domain Driven Design</strong></th>
<th><strong>Avoid Disruption</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Understanding of monolithic applications with business functionality</td>
<td>1. Assessing the size of the namespace</td>
<td>1. To shield users from disruption and ensure that no important business application goes offline during the process</td>
</tr>
<tr>
<td>2. Understanding the importance of a module and what business purpose it solves.</td>
<td>2. If any single component in the architecture has more 10% of functionality, it’s suggested to break it down into smaller chunks.</td>
<td>2. Decouple the presentation logic into a remote UI with API access.</td>
</tr>
<tr>
<td>3. To have a wholistic approach and not a fine-grained one.</td>
<td>2. Monolithic application should be translated to services with namespaces organized by domains and sub-domains.</td>
<td>2. Install proxies between API consumers and services and Set up facades.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Self Contained</strong></th>
<th><strong>Split front-end and back-end</strong></th>
<th><strong>Best Practices</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Identify self contained modules as part of the overall planning.</td>
<td>1. A typical flaw with monolithic services is about splitting the front-end with the back-end and the business layer.</td>
<td>1. Identify the most independent module and migrate that first.</td>
</tr>
<tr>
<td>2. Modules are loosely integrated in monolithic applications and should be easier to transform.</td>
<td>2. Plan Microservices architecture by keeping these three layers into consideration.</td>
<td>2. Never allow the migrated Microservices module code to continue in monolithic application.</td>
</tr>
<tr>
<td>3. Self-contained systems contain both backend and UI.</td>
<td>2. Follow the MVC model for breaking up the service.</td>
<td>2. Do not hesitate to break the existing Microservice to further smaller Microservices</td>
</tr>
</tbody>
</table>

---

**Avoid Disruption**
- To shield users from disruption and ensure that no important business application goes offline during the process.
- Decouple the presentation logic into a remote UI with API access.
- Install proxies between API consumers and services and Set up facades.
- Release new code with toggles and dark launches rather than cutovers.

**Best Practices**
- Identify the most independent module and migrate that first.
- Never allow the migrated Microservices module code to continue in monolithic application.
- Do not hesitate to break the existing Microservice to further smaller Microservices.
- Create a parallel design if apps are business critical.
Cost Consideration

Cost of Getting Started

- **Personnel**: Not everyone is familiar with Microservices. What is the cost of bringing those personnel up to date with Microservices?

- **Organizational Expenses**: Benefits are best realized when administered by small cross functional teams.

- **Tools**: Containerization and other supporting technologies which can bring down the cost.

Cost of Maintenance

- A microservices architecture, with fewer application dependencies and simple APIs, will immediately reduce the time and money spent on application maintenance.

- Savings on application maintenance have proven to be more cost effective within a few years of adoption.

Cost of Speed

- Microservices can increase the speed of development, ensuring that the organization remains competitive. This however depends on how well the application is modularized and thought through.

- A well architected microservices architecture will have faster turnaround time for any changes that business wants depending of the market requirements

Total Cost of Ownership

- Software that creates microservice directly from the legacy or on-premise system, we can bypass complex ESB/SOA layers that were previously used for legacy integrations.

- When decommission ESB/SOA and replace it with independent, directly connected microservice-based APIs, we will always achieve exponential savings by comparison

Cost of Quality

- DevOps, Agile, and other modern development practices rely heavily on automated testing.

- Microservices make for a much cleaner testing process. They’re built simpler, so it’s easier to review their code.
High Level Architecture Considerations

Simple Decouple Capability
- Start with capabilities that are decoupled from the monolith, that don’t require changes to many client facing applications that are currently using the monolith and possibly don’t need a data store.
- Decoupling edge services gives us a good idea about operational prerequisites that is needed for the other monolithic services that need decoupling.

Minimize Dependency back to Monolithic
- A major benefit of microservices is to have a fast and independent release cycle.
- There will be scenarios where dependencies with monolithic applications cannot be avoided.
- It’s suggested that create an interface with which Microservice interacts and has the abstraction from the monolith to reduce the dependency.

Split Sticky Capabilities Early
- One of the preferred best practices is to identify the stickiness in your monolithic application and extract those sticky business logic at the very early stages.
- Ensuring that those sticky capabilities are converted into domain driven design and converting them into producing microservices is a good practice

Decouple Vertically and Release the Data Early
- The general concept of a monolithic application is with having tightly coupled layers.
- There are also dependencies on other external systems. If the data layer schema changes releasing those data dependencies so that those entities can live on their own becomes a significant factor of how well your microservices are architected and more importantly

Decouple Capability and not Code
- Understanding a particular module within the monolithic service and deconstructing it based on capability and not trying to re-engineer the code is a preferred and acceptable mode.
- Translating the capability into modern API by not looking at how the code was built helps to stay within the best practice’s realm.
## Architecture Patterns for Transition

<table>
<thead>
<tr>
<th>Pattern Type</th>
<th>Pattern Name</th>
<th>Pattern Outline</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Decomposition</strong></td>
<td>Domain-driven Design</td>
<td>Decomposition of business domain into sub-domains and identification of bounded contexts (groups of functional units capable of autonomous existence)</td>
</tr>
<tr>
<td></td>
<td>Strangler</td>
<td>Coexistence of legacy / monolith components along with decomposed microservices. This is commonly used for incremental modernization / transition needs</td>
</tr>
<tr>
<td><strong>Integration</strong></td>
<td>API Gateway</td>
<td>An API Gateway serves as a single entry-point to numerous microservices offering abstraction, aggregation, protocol independence and security</td>
</tr>
<tr>
<td></td>
<td>Aggregation</td>
<td>Aggregation of results from multiple microservices eliminating need for interaction with multiple granular services</td>
</tr>
<tr>
<td></td>
<td>Back-end for Front-end</td>
<td>Layered microservices implementation with a layer dedicated for specific front-end needs (consumers interact only with this layer). Example – microservices for mobile, desktop etc.</td>
</tr>
<tr>
<td><strong>Database</strong></td>
<td>Database per Service</td>
<td>Dedicated database for each service not accessed by another service</td>
</tr>
<tr>
<td></td>
<td>Shared Database</td>
<td>Database shared by 2-3 microservices; used mostly in modernization implementations</td>
</tr>
<tr>
<td></td>
<td>Command Query Responsibility Segregation (CQRS)</td>
<td>CQRS splits applications into Command (Create, Update, and Delete requests) side and Query side</td>
</tr>
<tr>
<td></td>
<td>Saga</td>
<td>Use of Event Driven architectures to coordinate multiple microservices (each having its own database)</td>
</tr>
<tr>
<td><strong>Isolation / encapsulation</strong></td>
<td>Sidecar</td>
<td>Deploy helper components (monitoring, logging, configuration etc.) of an application into a separate process or container to provide isolation and encapsulation</td>
</tr>
<tr>
<td><strong>Others</strong></td>
<td>Circuit Breaker</td>
<td>Invocation of service calls by dependent services via a proxy which forces a timeout after a regulated number of attempts in case of an error in the providing service</td>
</tr>
</tbody>
</table>
Microservices – Azure Service Fabric

Need for Containerized Microservices Deployment
Containerized microservices deployment offers numerous benefits in terms of agility, scalability, rapid capabilities addition etc. An effective container orchestrator is needed for maximizing containerization benefits in Azure (and other cloud environments).

How Azure Service Fabric fits in?
Azure Service Fabric is a Container Orchestration platform offering auto-scaling, auto-fault-healing, rolling upgrades etc. It allows for developers to focus on business capability creation and takes care of infrastructure and resource management needs itself.

Key Offerings
• Support for multiple container types
• Application Health Monitoring
• Auto scale-in / out of microservices
• Rolling upgrades
• Services Discovery
• Security
Serverless microservices implementation is gaining traction due to increasing maturity of Azure Serverless offerings

A summary of key Azure serverless microservices offerings:

<table>
<thead>
<tr>
<th>Offering</th>
<th>Outline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Azure Functions</td>
<td>Azure Functions can be purposed as serverless microservices. They can be exposed as HTTP based API end-points and allow for event-driven invocation</td>
</tr>
<tr>
<td>Azure Logic Apps</td>
<td>Logic Apps can be leveraged to effectively orchestrate Azure Function implemented microservices. They also automate routine tasks like notifications.</td>
</tr>
<tr>
<td>Azure Container Instances (ACI)</td>
<td>ACI helps to orchestrate containerized microservices in a serverless fashion (though limited in capabilities relative to Regular Container Orchestrators like AKS / Service Fabric)</td>
</tr>
</tbody>
</table>

Serverless good for:
1. Greenfield Development
2. Relatively small logic chunks (services)
3. Operations that do not require long-duration execution

Serverless may not be good for:
1. Inter-cloud portability
2. Modernization of applications with significant legacy dependencies
3. Long-running operations
4. Performance-intensive applications
• **Identify decomposition pattern** - A combination of decomposition patterns including capability-driven, domain-driven, UI-driven etc. are leveraged for monolith decomposition into services

• **Firm up dependencies strategy** - A decomposition strategy is firmed up to make sure dependencies shift from the monolith to microservices being implemented, progressively

• **Decompose in increments** - Decomposition is performed in increments. Capabilities / components would be identified based on the current state of decoupling, frequency of alterations etc. Peripheral capabilities that are frequently altered are gain priority. **Granularity** of decomposition is driven by factors like current level of decomposition maturity, time available for an increment etc.

• **Plan increment implementation** - Implementation is planned in iterations in accordance with time available and the current maturity of the implementation team

• **Execute increment and move to the next decomposition increment** - Increment Implementation is executed as planned

• **Manage / Contain Microservices Complexity** - Microservices complexity is continually assessed and complexity containment is strategized on
## Application Migration Approach

<table>
<thead>
<tr>
<th>Apply Domain Driven Design</th>
<th>Use glue code (Anti-corruption layer)</th>
<th>Create Presentation layer</th>
<th>Manage the data</th>
<th>Retire the monolith</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Stop adding functionality to the monolith.</td>
<td>- &quot;Glue code&quot; allows the monolithic application to proxy calls to the new service to obtain new functionality.</td>
<td>- Separate the presentation layer from the backend layer.</td>
<td>- Embrace eventual consistency.</td>
<td>- Peel away the monolith by applying Strangler pattern.</td>
</tr>
<tr>
<td>- Split the front end from the back end.</td>
<td>- Ensure that the new service is not polluted by data models required by the monolithic application</td>
<td>- Develop this tier as Single Page Application or MVC application.</td>
<td>- Use patterns like “Scheduler Agent Supervisor” and “Compensating Transaction” for data consistency.</td>
<td>- Over time, existing functionality is moved into microservices, the monolith will shrink in size and complexity, until it no longer exists.</td>
</tr>
<tr>
<td>- Decompose and decouple the monolith into a series of microservices.</td>
<td>- Using domain analysis to model microservices</td>
<td>- These applications interact with the microservices via the gateway, using standard HTTP calls.</td>
<td>- Keep your services loosely coupled.</td>
<td>- At this point, the anti-corruption layer can safely be removed.</td>
</tr>
<tr>
<td>- Using domain analysis to model microservices</td>
<td></td>
<td>- Embrace eventual consistency.</td>
<td>- Use Event driven architecture style to propagate changes to its models or entities.</td>
<td></td>
</tr>
</tbody>
</table>
Data Migration Approach

Source Monolithic Data Stores

Scan Report

- XML Log to be used for automated Design Input
- PDF Scan Report with Salient Information

Analysis for Design

- Machine Learning guided Design Inputs
- Manual Analysis of Information

Pattern Store

- Saga Pattern
- Strangler Pattern
- API Composition Pattern
- Shared Data Store Pattern
- Command Query Responsibility Segregation Pattern

Disposition Plan based on Source Morphology and Enterprise Parameters

Supplying Enterprise Parameters for Disposition
- Usage Frequency
- Relevance
- Business Value
- Data Criticality

Target Schema

- Migrated Data from Monolithic System to Micro Service Paradigm
- Supported Script based Data Loading in Target Schema

Target Design

- Store(s) Identification
- Monolith Split Design
- Security for Split Design
- Target Data Structure

Analysis for Design

- Automated Pattern Fitment Recommendation

Pattern Store

- Migrated Data from Monolithic System to Micro Service Paradigm
- Supported Script based Data Loading in Target Schema

Target Schema

- Creation of Target Schema in Chosen Azure Platform(s) by Supported Scripts

Source Monolithic Data Stores

Scan Report

- Data Source Scan by respective Supported Script based Crawler

Analysis for Design

- Machine Learning guided Design Inputs
- Manual Analysis of Information

Pattern Store

- Saga Pattern
- Strangler Pattern
- API Composition Pattern
- Shared Data Store Pattern
- Command Query Responsibility Segregation Pattern

Disposition Plan based on Source Morphology and Enterprise Parameters

Supplying Enterprise Parameters for Disposition
- Usage Frequency
- Relevance
- Business Value
- Data Criticality
## Detailed Steps for Data Migration

<table>
<thead>
<tr>
<th>Discovery</th>
<th>Disposition</th>
<th>Analysis</th>
<th>Design</th>
<th>Execution</th>
<th>Load</th>
<th>V&amp;V</th>
<th>Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>▪ Identification of ALL Source Systems</td>
<td>▪ Collect Information for Disposition Decision from Discovery Output</td>
<td>▪ Pre-defined guideline (based on Micro Service Best Practices) driven Analysis of Discovered Information in light of Disposition Plan and Categorization</td>
<td>▪ Identification of Target Data Platforms</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▪ Scanning each and every Source System for Data Structure, Relationships, Constraints, Procedures and Functions, Jobs, Usage Information and other Data Elements</td>
<td>▪ Shortlist Disposition Parameters as applicable for the target Enterprise Ecosystem [Standard Parameters include Usage Frequency, Relevance, Business Value, Data Criticality]</td>
<td>▪ Appropriate Pattern Fitment based on Pattern Guidelines</td>
<td>▪ Alignment with Architecture Principles of Chosen Pattern</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▪ Generating XML Log for further Machine Consumption towards Analysis</td>
<td>▪ Decision on Boundary for Shortlisted Parameters</td>
<td>▪ Additional Analysis Information identified manually moderating automatically recommended Analysis</td>
<td>▪ Design of SPLIT of Monolith Data Source(s) into one or multiple Target Data Container(s) in an optimal way towards zero or minimal redundancy</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▪ Generate PDF and Excel Report for Manual Reference towards Manual Inputs during Design</td>
<td>▪ Rule based Categorization [To be Migrated, To be Archived, To be Retired etc.] of Source Data based on Disposition Parameters</td>
<td></td>
<td>▪ Design of Security for SPLIT Data Targets at same or higher level of Security</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Next Slide
## Detailed Steps for Data Migration

<table>
<thead>
<tr>
<th>Discovery</th>
<th>Disposition</th>
<th>Analysis</th>
<th>Design</th>
<th><strong>Execution</strong></th>
<th><strong>Load</strong></th>
<th><strong>V&amp;V</strong></th>
<th><strong>Base</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>▪ Sanity check for one or multiple Target Environments</td>
<td></td>
<td>▪ Tool based Data Structure Validation</td>
<td>▪ Estimation of the Overall Process keeping all Dependencies in mind</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>▪ Automated Creation of the Script to create the Designed Target Data Structure in Target System(s)</td>
<td></td>
<td>▪ Validation Report</td>
<td>▪ Roadmap Creation for Migration Journey</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>▪ Run the generated Scripts on to Target Data Environment(s) to create the designed Data Structure</td>
<td>▪ Sampling Pilot Data from Source System(s)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>▪ Execution Log in Produce for finding Failed Cases</td>
<td>▪ Script based Data Loading for Pilot Sampling</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>▪ Failed Scripts to be amended manually (if needed)</td>
<td>▪ Export of whole Source Data in a mutually agreed format</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>▪ Re-run Rectified Scripts on Schedule</td>
<td>▪ Data Transformation as per Target Data Structure</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>▪ Loading of Transformed Data in Target Ecosystems</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How to Handle Your data in a Microservice Architecture

**Databases should be treated as private to each microservice**

**Microservices should follow Domain Driven Design and have bounded contexts.**

**Each microservice should have its own database and should contain data relevant to that microservice itself.**

**Follow the Code First approach over Data First approach - hence you design your models first**

**Keep your mind open towards a NoSQL DB as well if it fits your criteria**

**Instead of an ACID transaction, offload the event messages to a queue (follow Pub-Sub model)**

---

Database Design Challenges

**Challenges of Monolithic Database Design**

- Creates a tight coupling and inability to deploy your service changes independently
- It is difficult to scale individual services. You only have the option to scale out the entire monolithic database.
- With a single shared database, over a period, you end up having huge tables. This makes data retrieval difficult.

Need a solid strategy to split database into multiple small databases aligned with applications. In short we need to decouple applications/services from using a single monolith shared database.
### Microservices Best Practices

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Clarity of objectives</strong></td>
<td>Goals of microservices adoption should be very clear. These could be quick technology modernization, faster application release cycles or scalability on cloud, among others. Microservices adoption could be counter-productive in the absence of clear objectives.</td>
</tr>
<tr>
<td><strong>Agility</strong></td>
<td>Microservices implementation needs to be quick for realization of benefits. Each microservice should have an optimal granularity and development cycle. Monolith decomposition should be planned in optimal increments for quick microservices implementation.</td>
</tr>
<tr>
<td><strong>Monitoring &amp; Visibility</strong></td>
<td>Microservices volumes tend to grow rapidly. Effective monitoring and clear visibility across the multitude of microservices is key to microservices manageability</td>
</tr>
<tr>
<td><strong>Automation</strong></td>
<td>Due to relatively large volumes of small components, automation is key to effective value realization of microservices-composed applications. Consequently, release (deployment, rollback and others), monitoring etc. need to be automated.</td>
</tr>
<tr>
<td><strong>Microservices Boundaries</strong></td>
<td>Use of Domain Driven Design (DDD) Bounded Context principles of boundary-delimitation helps define logical microservices boundaries. However, domain evolution must be addressed with boundaries redefinition</td>
</tr>
<tr>
<td><strong>Dynamic Scale-up / down</strong></td>
<td>Cloud compatibility of microservices would help realize the benefits of dynamic / on-demand scale up and scale down in cloud environments. Microservices should be designed with an eye on making them cloud native / cloud-compatible.</td>
</tr>
<tr>
<td><strong>Fault Tolerance</strong></td>
<td>Due to inter-dependencies across a large number of microservices, effective fault tolerance mechanisms would need incorporation. Patterns like circuit-breaker, and asynchronous event-driven architectures help fault tolerance implementation.</td>
</tr>
<tr>
<td><strong>Centralized Logging &amp; Dashboard</strong></td>
<td>Debugging across the multitude of microservices would require putting in place centralized logging and dashboards across services</td>
</tr>
</tbody>
</table>
Thank You