International Journal of Science and Research (IJSR)
ISSN: 2319-7064
Impact Factor 2024: 7.101

MACH Architecture and Composable Systems: Revolutionizing E-Commerce for the Digital Age

Elby Mooken

Senior Engineering Manager, Levi Strauss & Co., San Francisco CA, USA

Abstract: ***The e-commerce landscape is undergoing a profound transformation, driven by rapidly evolving customer expectations and technological advancements. Traditional monolithic systems, once the backbone of digital commerce, are being replaced by more agile and flexible architectures specifically, MACH (Microservices, API-First, Cloud-Native, Headless) architecture and composable commerce. These innovative approaches offer unprecedented flexibility, scalability, and agility, making them indispensable for modern e-commerce platforms. The need for MACH and composable systems arises from their ability to enhance customer experiences and improve business competitiveness. MACH architecture allows marketers and developers to modify the presentation layer without involving backend developers, enabling rapid changes and A/B testing to meet evolving customer needs. Similarly, composable commerce extends the MACH principles by enabling businesses to build customized e-commerce architectures using best-of-breed solutions. This approach involves assembling independent components or microservices to create unique customer experiences and adapt to emerging trends, such as AI. Therefore, proposed review paper uniquely synthesizes the benefits of MACH and composable commerce, providing insights into how these technologies can enhance customer experiences and future-proof e-commerce platforms. By exploring successful case studies and discussing limitations and future recommendations, the paper offers a holistic view of the transformative potential of MACH and composable systems in the digital age. Future recommendations include continuous monitoring and feedback to refine and improve the e-commerce experience, ensuring businesses remain responsive to customer needs and technological advancements.***

Keywords: MACH architecture, Composable Architecture, E-commerce

1. How MACH architecture is different from Monolithic Architectures

The e-commerce landscape is constantly changing based on consumer demands for seamless, personalized experiences across multiple channels. Businesses, in return, are utilizing innovative architectural methods to remain on top of their game. A few methods are MACH architecture and composable systems MACH architecture is designed to improve agility, scalability, and resilience in digital commerce platforms. MACH stands for Microservices, API-First, Cloud-Native, and Headless [9] and it allows businesses to create flexible, modular systems that quickly integrate new technologies and adapt to changing market trends.

Microservices are at the core of MACH architecture, which involves breaking monolithic systems down into smaller, independent components. Each microservice can be designed, deployed, and scaled independently, which eliminates some of the complexity and risk associated with more traditional monolithic architectures. By embracing an API-First methodology, microservices can each communicate with each other and be integrated easily with third-party services (payment gateways, shipping, etc). Being Cloud-Native allows MACH-based systems to leverage many benefits of cloud computing (scalability, cost savings, etc). The headless aspect of MACH decouples the front-end presentation layer from back-end logic, allowing marketers and developers to quickly create innovation and personalize customer experiences across multiple channels without needing to call backend developers on each change made.

Composable systems build on the MACH principles by allowing companies to build their custom e-commerce by using best-of-breed solutions. Composable systems are built on independent parts or microservices, creating personalized customer experiences, and being able to pivot with different trends like Artificial Intelligence (AI). Composable commerce is built on solutions that are business centric, modular architecture, and an open ecosystem. This allows retailers to select the best of breed solutions from third-party vendors, and build a store-specific tech stack for their business. The benefit of having modular design, and the low-code API integration of composable architectures allows for a simpler way of collecting, analysing, and utilizing data to take action and drive business growth. AI and composable commerce can have significant benefits to e-commerce businesses. Better key performance indicators (KPIs) and improved conversion rates, increased engagement, and deep customer intelligence are some examples of this. AI can be used for personalized product recommendations, dynamic pricing, and custom portals for different clients. Making AI driven personalized experiences for customers makes them feel cared for and taken into consideration leading to a better customer experience and strength maternal relationships. Also, due to the flexibility and scalability of MACH and composable systems, businesses can accelerate the use of new tools and technologies to club their online stores with AI engines, creating a competitive advantage in their market.

Therefore, owing to these impeccable characteristics of MACH and composable systems, proposed survey focuses on reviewing the different architectures in e-commerce setup as most existing survey emphasizes on monolithic or only microservices. The proposed survey distinguishes itself by focusing on a comprehensive review of MACH and composable architectures within the e-commerce sector, a domain that has received limited attention in existing literature. This distinct focus allows the paper to delve into the uncharted territory of how MACH's microservices, API-first, cloud-native, and headless commerce components can enhance scalability, flexibility, and customer experience in e-commerce platforms. By examining the modularity and composability of these architectures, the survey aims to

highlight their potential to accelerate innovation, reduce time-to-market, and facilitate seamless integrations with various technologies. This, in turn, contributes to a more robust and adaptable e-commerce ecosystem. Furthermore, the survey will explore how MACH architectures enable businesses to leverage best-of-breed tools, ensuring that each component of the e-commerce system is optimized for performance and customer satisfaction, thereby setting a new standard for modern e-commerce solutions.

Objectives of the Paper

The objectives of the present review paper is listed as follows,

- To review works of MACH and composable architecture's for e-commerce environment.
- To explore different works on microservices, API First, cloud native, headless architecture as well as composable architecture.
- To explore case studies on different organizations for using MACH and Composable architecture principles.
- To discuss challenges and provide future recommendations with the aim of overcoming limitations faced by the state-of-the-art approaches.

2. How MACH architecture is different from Monolithic Architectures

When comparing monolithic architectures with MACH architecture in the context of e-commerce, several key differences emerge that significantly impact scalability, innovation, and customer experience.

a) **Monolithic Architecture:** Monolithic architectures are traditional, self-contained systems where all components are interconnected and interdependent. This means that any change or update requires a full redeployment of the entire system, which can be time-consuming and risky. Monolithic systems are often inflexible and difficult to scale, as adding new features or handling increased traffic can lead to bottlenecks and performance issues. While they are simpler to develop initially, their rigidity makes them less suitable for modern e-commerce environments where adaptability and speed are crucial.

b) **MACH Architecture:** In contrast, MACH architecture offers a modular and flexible approach. It breaks down the system into independent microservices, each handling a specific business function. This modularity allows for scalability, as individual services can be scaled independently without affecting the entire system. MACH also supports an API-first approach, enabling seamless integration with various services and technologies, such as payment gateways and shipping providers, without disrupting the core system. The cloud-native aspect ensures that resources can be dynamically allocated, reducing costs and enhancing performance during peak periods. Additionally, the headless nature of MACH allows for a decoupled frontend and backend, giving marketers and developers the freedom to innovate and customize the customer experience across multiple channels without needing to involve backend developers.

Therefore, the upcoming section focuses on breaking down MACH principles, thereby demonstrating the principles in detail.

3. Microservices Architecture

Microservices architecture is a modern approach to software development that divides applications into smaller, autonomous components, each responsible for specific functionality. These independent services can be developed, deployed, and scaled separately, enabling greater reliability and flexibility by eliminating single points of failure. By exposing frequently used operations as services, this architecture supports automatic scaling and enhances extensibility. Microservices operate like callable functions within an application and can be hosted remotely on virtual machines or containers. This decoupling allows applications to interact with microservices regardless of the underlying technology or programming language used in their creation. Furthermore, microservices enable applications to seamlessly integrate with diverse backend databases, including cloud-based, relational, and NoSQL systems. Popular tools like Docker and Kubernetes are often employed for deploying microservices. The architecture is characterized by its decentralized nature, fault isolation, scalability, and technology diversity. Each service operates independently, ensuring that failures in one do not disrupt the entire application. This design encourages faster development cycles, easier maintenance, and improved agility by allowing teams to focus on individual services without affecting others.

3.1 Advantages of Microservices Architecture

a) **Flexibility in Technology Stack:** Unlike traditional monolithic architectures, microservices architecture liberates e-commerce development from rigid technology stacks. It allows businesses to leverage modern technologies and frameworks, facilitating the creation of beautifully designed and high-performing e-commerce applications.

b) **Resilience and Fault Isolation:** E-commerce applications built on a microservices architecture are inherently more resilient. Failures or malfunctions in one microservice do not cascade to other parts of the system, ensuring that online sales and operations remain unaffected. This fault isolation improves system reliability and uptime, contributing to a seamless and uninterrupted shopping experience for customers.

c) **Scalability and Performance:** Microservices architecture offers inherent scalability and performance benefits. Each microservice can be independently scaled based on demand, allowing e-commerce platforms to handle fluctuations in traffic and transaction volumes more efficiently. Additionally, microservices enable the use of containerization technologies like Docker, which provide lightweight and isolated environments for testing and deployment, further enhancing scalability and performance.

d) **Quicker deployment time:** In monolithic architectures, variations demands re-deploying the entire application. Microservices architecture permits quicker releases as each service progresses and set out independently, reducing the risk and time related with coordinating variations across complete application. Decoupling services in this manner improves agility. By doing so, updates or fixes with minimal disruption can be carried

out to the overall system. Thus, additional benefits are demonstrated in figure 1.

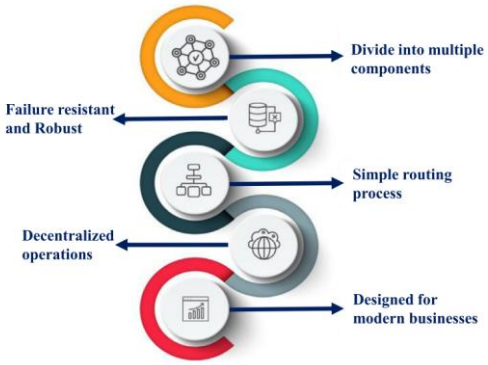

Figure 1: Benefits of Microservice Architecture

4. Architecture of API-First

API- first development prioritizes designing and building application programming interfaces (APIs) before any other components of the application. This technique ensures that APIs, which allow different software systems to communicate, are central to the development process. The API-first approach prioritizes API planning as the initial step in the programming process. Subsequent development stages aim to uphold the original API design while ensuring consistency and reusability. This methodology promotes consistency and reusability through the use of easily understandable and navigable API specification languages.

4.1 How API-First Development works

API-first development represents a strategic methodology that emphasizes the initial creation of APIs as the cornerstone of a project. This approach is not merely about the order to development activities, but rather about adopting a mind-set where APIs are regarded as the fundamental building blocks of the entire development process. By prioritizing APIs from the outset, this methodology sets a robust foundation for all subsequent development efforts ensuring a cohesive and well-integrated system.

a) **Defining the API Contract:** The initial stage of API-first development focuses on defining the API contract, which serves as a blueprint for the entire development process. This step involves detailing the endpoints, request-response structures and data models that the API will utilize. Standardized tools such as Swagger (OpenAI) are commonly employed to draft these specifications in a language agnostic format. Acting as a binding agreement between backend services and their customers, this contract ensures uniformity and clarity across all teams engaged in the development lifecycle.

b) **Mocking and Prototyping:** Once the API contract is established, the next phase involves mocking or prototyping the API to simulate its behaviour. This process entails creating a basic implementation that generates predefined responses to incoming requests, enabling early testing and integration. Tools like swagger and postman are specialized mocking platforms are utilized to replicate API functionality in a controlled environment. By allowing front-end developers, accelerates workflow efficiency and significantly reduces time-to-market for the final product.

c) **Implementation of Business Logic:** Following the API mock-up, the implementation of the underlying business logic commences. This phase involves configuring the server, database and other backend components, followed by coding the functionality that aligns with the commitments outlined in the API contract. The primary focus during this stage is on maintaining strict adherence to the predefined API specification. This ensures that any modifications do not compromise the established agreement with consumers, thereby preserving the integrity of the API and maintaining seamless interactions across all integrated systems.

d) **Continuous Testing and iteration:** API-first development accentuates continuous testing for ensuring the API meets its contract throughout the development process. This process incorporates unit tests, integration tests and contract tests. Contract testing, in specific verifies that the API responses match the expectations set out in the API specification, ensuring computability between the server and client sides. Tools like Dredd and Pact can be used for automating the contract testing, enabling a CI/CD pipeline that maintains API quality and reliability.

Therefore, the steps provided in the section showcases the process involved in developing the API-first architecture.

4.2 Benefits of API-First Development

Present section highlights the benefits of API-first approach, by helping the benefits of microservices-based applications, aids developers ensure that the service are consumed by the broadest range of client/systems in the API economy.

a) **More scalable System:** API-first design supports modular and scalable architecture, making it easier to add, remove, or upgrade features without affecting the rest of the system. APIs designed upfront ensure compatibility across diverse client applications and future integrations, providing adaptability to changing requirements.

b) **Parallel Development:** Teams can work simultaneously on different aspects of the application by using API contracts as a blueprint. Frontend and Backend developers can collaborate early without waiting for another, accelerating the development process. APIs can be mocked for testing dependencies, enabling faster feedback cycles and smoother integrations.

c) **Language and Platform flexibility:** APIs deliver a language and platform agnostic interface that different microservices can interact with. Because API-first development prompts the creation of consistent and reusable APIs, the API-first approach helps the system integrate with a wide range of services-regardless of the language and platforms.

d) **High Availability, fault tolerant systems:** By integrating APIs within a microservices-based system, the architecture facilitates the creation of a highly available and fault-tolerant environment. This resilience stems from the ease with which incoming requests can be load-balanced across service instances, the automation of microservice deployments, the inherent redundancy of critical components, and the implementation of various sophisticated orchestration techniques.

5. Cloud Native Architecture

Cloud native architecture is an innovative software development approach that harness the cloud computing model by merging methodologies from cloud services, DevOps practices and software development principles. Cloud native approach enables organizations to build applications as loosely coupled services using microservices architecture and run them on dynamically orchestrated platforms. Thus, application built on the cloud-native architecture are known to deliver scale, performance and it is known to be reliable. Additionally, it also offer faster time to market. In 2025, cloud-native extends beyond traditional cloud environments for embracing edge computing, server less architectures and AI driven operations (AIOps), enabling business to deliver customer-centric solutions faster than ever. Moreover, it is noted that, cloud-native apps augmented by microservice architecture leverage the highly scalable, flexible and distributed cloud nature for producing customer-centric software products in a continuous delivery environment. The striking feature of the cloud native architecture is that it permits the user to abstract all the infrastructure layers like servers, OS, databases, security and many more.

Traditionally, enterprise applications have been designed as monolithic structures, wherein all functionalities are integrated within a singular codebase. This approach presents several structural challenges. Firstly, development teams are required to build and test the entire application as a cohesive unit, which significantly impedes developer efficiency. Any modifications necessitate a compute recompilation and retesting of the entire application to ensure that no new issues have emerged. The process of producing documentation is often cumbersome and time-consuming. Large-scale applications can experience prolonged start-up times and potentially sluggish performance. Minor bugs can also lead to unforeseen complications due to the high degree of interdependence among application components.

In contrast, cloud-native architecture is fundamentally modular and distributed. Furthermore, teams can integrate pre-built components without enduring lengthy testing phases. Furthermore, applications can be modularized, allowing developers to collaborate on different components concurrently. This approach results in a substantial increase in developer productivity. Thus, cloud native architecture use internet services, thereby protecting the data against external attacks. Additionally, it is noted that, resilience is important for cloud security native architecture, especially when microservices are used, with applications deployed across distributed nodes, the system must be resilient to a failure affecting one node. Likewise, a well-designed cloud native application should have the capability to keep running or recover quickly in the event of node failures.

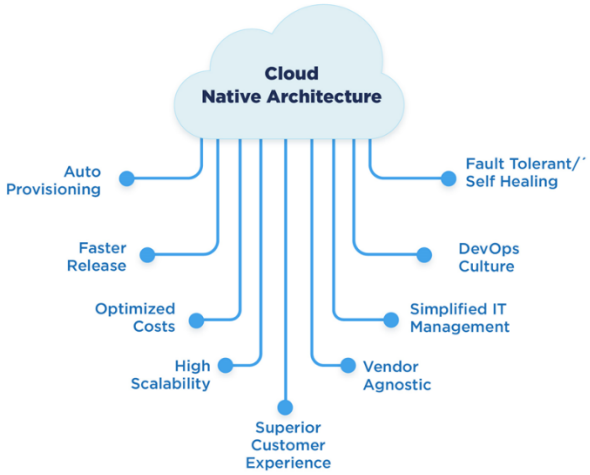

Figure 2: Merits of Cloud Native

Figure 2 highlights the various benefits of cloud native architecture, where the benefits demonstrated are auto provisioning, faster release, optimized costs, high scalability, superior customer experience, vendor agnostic, simplified IT management, DevOps culture and Fault tolerant/self-healing. Along with it some of the advantages of cloud native architecture is listed as follows,

- Customizability: Exploiting loosely coupled services rather than technology stacks enables DevOps teams to choose ideal framework, system and language for the projects.
- Portability: Containerised microservices are portable, enabling organizations to move between cloud environments. It can avoid vendor-lock as it does not exclusively rely on single vendor.
- Flexibility: Cloud native architecture supports a wide range of tools and framework, allowing organizations to adopt new technologies without being tied to specific vendors.
- Disaster Recovery: Cloud native systems can quickly restore operations after catastrophic events, minimizing business disruptions.
- Business efficiency: Cloud native approaches combine organizational and technical changes to deliver value faster while improving overall efficiency through automation and streamlined processes.

6. Headless Architecture

Headless architecture, a contemporary software design philosophy, establishes a clear division between the front-end presentation layer and the back-end functionality. This decoupling empowers developers to construct adaptable, scalable, and consistent digital experiences across all touchpoints. Consequently, the term "headless" describes a system or software operating without a graphical user interface, enabling remote interaction and management without visual rendering. Characteristically, a headless architecture integrates diverse, specialized back-end services through API connections, often referred to as a "best-of-breed" approach. The central objective is complete modularity, ensuring that all components function autonomously with minimal or no interdependencies. This independence leads to enhanced flexibility, improved scalability, and simplified maintenance, as modifications to

one component have a reduced impact on the others. The fundamental elements that constitute a Headless architecture include:

a) **Back-end (headless CMS):** This is where the content and data reside. It focuses on storing, managing and delivering content through APIs. A headless system does not have a built-in-front-end. It concentrates on offering the raw data. Thus, backend provides a content management interface that allows creators to define content models, input data and manage assets. Furthermore, for e-commerce applications such as product management, inventory tracking and order processing and customer data management. This separation allows the back-end to concentrate on its core function, providing a reliable and scalable data source.
b) **APIs:** API act as the communication bridge between the back-end and the front-end. It allows the front-end to request and receive content from the back-end in a structured format, typically JSON or XML. RESTful APIs and GraphQL are commonly used. Thus, API delivery forms a crucial communication bridge between the back-end and several front-end applications. When a front-end application needs content, it sends a request to the back-end via the API. The API then retrieves the requested data from the back-end's database and transforms it into a format suitable for the requesting application. This transformation capability ensures that each device, whether it's a mobile phone, a smart display, or a website, receives data in a format it can efficiently process. Security is also paramount in API delivery. Authentication and authorization protocols are implemented to protect sensitive data and prevent unauthorized access, ensuring that only trusted applications and users can retrieve content.
c) **Front-end (head):** Finally, the front-end rendering layer is where the user experience is crafted. This "head" of the headless architecture is entirely decoupled from the back-end, giving developers the freedom to use any front-end technology they prefer, such as React, Vue.js, or Angular. When a front-end application receives data from the API, it dynamically renders the content for the user. This dynamic rendering allows for highly customized and interactive user experiences tailored to specific devices and platforms. Front-end developers also have the flexibility to optimize the presentation for different screen sizes and device capabilities, ensuring a consistent and seamless user experience across all touchpoints. This flexibility, combined with the power of the headless back-end, empowers businesses to create innovative and engaging digital experiences that can adapt to the ever-evolving digital landscape.

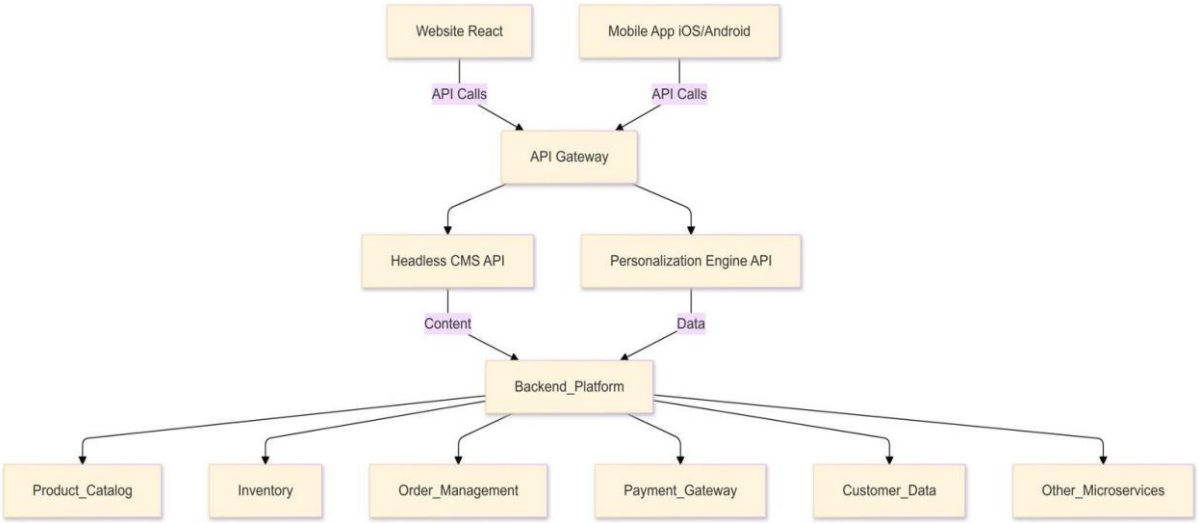

Figure 3: Headless Architecture

In figure 3, the individual engaged with website or other mobile apps. Then, front-end application transmitted the API request to the API gateway. Later, the API Gateway directed the request to the pertinent back-end service and the backend service processed the request and retrieved data from its DB. Further, the back-end service requested content from the headless CMS via its API if necessary and the back-end service provided data to the API gateway in a structured format like JSON. In the next step, the API gateway transmitted data to the front-end application and eventually, the front-end application processed data and modified the user interface as required.

6.1 Benefits of headless architecture

a) **Streamlined content management:** Headless CMS system allow marketers and editors to manage content centrally and distribute it across different platforms.
b) **Scalability:** Backend resources can be scaled independently of the frontend allowing systems to handle high traffic loads efficiently without overhauling the entire architecture. Additionally, it deliver optimized resource allocation where each component can be optimized separately for better performance and scalability.
c) **Flexibility and Future Proofing:** The decoupled mechanism of frontend and backend allows developers to update or replace either the frontend or backend independently, enabling adaptability to new technologies and requirements. Likewise, the developers can pick the

best tools and frameworks for each layer, ensuring the system remains relevant as technologies evolve.

d) **Enhanced Performance**: Faster load times is considered as one of the advantages of headless architecture, as it deliver only necessary data via APIs, due to which headless architecture ensures faster pages loads and smoother user experience. Decoupling reduces the bloat associated with monolithic systems, improving overall system efficiency.

e) **Faster Innovation and Experimentation**: teams can test new features or designs independently without disrupting the core system, fostering continuous innovation and also businesses can quickly respond to changing customer demands by iterating on specific layers of the architecture.

Hence, due to the various advantages of headless architecture, it is known as an ideal choice for businesses aiming for agility, scalability and omni-channel capabilities while maintaining robust security and high performance. It empowers developers with unparalleled flexibility in building innovative digital experience at scale.

7. Composable Architecture

Composable architecture is a software design that accentuates building systems with modular, self-contained components with clear functionalities and well-defined interfaces. This modularity allows developers to easily assemble and combine these components for creating intricate applications. Present mechanism contrasts with traditional monolithic architectures by breaking systems into smaller, independent modules. Components are loosely coupled, allowing for easier updates and replacements, and depend profoundly on APIs for communication. Thus, this paradigm focuses on the idea of breaking down complex applications into smaller, self-contained units, often referred to as microservices or packaged business capabilities (PBCs). Each component encapsulates a specific business function, allowing for targeted development and deployment. The core of this approach lies in the principle of modularity, where these components are designed to be reusable and replaceable. Instead of building a single, monolithic application, developers construct applications by assembling these pre-built, specialized components.

A critical aspect of composable architecture is the emphasis on loose coupling and API-driven communication. These components are designed to operate independently, interacting with each other through well-defined APIs. This API-first approach ensures that components can communicate seamlessly, regardless of the underlying technology or programming language. By prioritizing API design, organizations create a robust integration layer that facilitates the smooth exchange of data and functionality between different components. This interoperability is crucial for enabling the integration of best-of-breed software from various vendors, allowing organizations to select the most suitable tools for each specific business function. Furthermore, the headless nature of many composable systems, powered by APIs, enables content and functionality to be delivered across various channels and devices, ensuring a consistent user experience regardless of the platform.

Composable architecture also aligns with MACH, ensuring modern, flexible, and scalable software solutions. Here, the three key principles of composable architecture is demonstrated as follows,

a) **Modularity:** A core tenet of composable architecture is the strategic decomposition of complex systems into discrete, independent modules. Each module encapsulates specific functionalities, enabling autonomous development, testing, and maintenance. This modular approach, akin to the building block paradigm exemplified by Jamstack and micro frontend architectures, facilitates streamlined system management and enhances codebase clarity, thereby promoting long-term maintainability.

b) **Reusability and Flexibility:** Reusability in composable system is a significant advantage, as it refers to design, develop and deploy of the individual component in such a way that, it can readily utilized across different applications. Besides, a primary advantage of composable architecture lies in its inherent adaptability. By decoupling system components, organizations gain the capacity to rapidly modify and evolve their digital infrastructure without necessitating comprehensive system redesigns. This modularity facilitates the seamless replacement or augmentation of individual components, enabling applications to demonstrate agility in response to evolving market dynamics and technological innovations.

c) **Scalability:** Composable architecture delivers significant scalability advantages. Independent component operation permits scaling based on specific requirements, without impacting the overall system. This optimizes resource utilization and facilitates integration of new technologies as business expands. Individual component scalability enables efficient load management and cost-effective operations.

Owing to these factors, composable architectures are used in the study for overcoming the drawbacks of server-based architecture. Hence, Q-learning based composable architecture has incorporated in the study for consolidating the workloads that spread over many underutilized nodes onto fewer nodes. Incorporation of Q learning based reinforcement learning approach has yielded an approximate pareto font, providing a set of ideal solutions catering to various preferences for the two objectives. Usage of Q learning integrated composable architecture resulted in minimizing the nodes and the no. of migrated workload elements. Despite its advantage, there exist few drawbacks such as high complexity and lack of high inaccurate predictions to mitigate negative effects. Likewise, a DL characterization on a composable infrastructure has used in the study for providing flexibility for serving a variety of workloads and offers a dynamic co-design platform that allows experiments and measurements in a controlled manner.

7.1 Relationship with MACH Architecture

Typically, MACH and composable architecture are closely interlinked in the modern software design, particularly within the realm of building digital experiences and e-commerce platforms. Thus, table-1 shows the relationship between MACH and Composable Architecture.

Table 1: Difference between MACH and Composable Architecture

Aspect	MACH Architecture	Composable Architecture
Definition	It is a subset of composable architecture, which is defined by 4 principles.	It is a modular approach to system design where components can be easily combined, replaced or scaled independently.
Components	Microservices, API, cloud-native application and headless architecture.	Packaged business capabilities, microservices and modular components.
Scope	It is identified as narrow framework which emphasizes specific technical principles for modern software development.	It is a broad philosophy, which is applicable across industries and use cases, focusing on modularity and adaptability.
Relationship	MACH acts as an enabler for composable architecture in contexts like digital commerce.	MACH is inherently composable but represents a specific implementation of composable architecture principles.
Key principles	Microservices, API first design, cloud native deployment and headless architecture.	Modularity, flexibility, reusability and scalability.
Technical Backbone	MACH principles provide the backbone for composable enterprises with SaaS models.	This rely on modularity and integration frameworks such as APIs or packaged business components.
Flexibility	Allows for rapid innovation and deployment of new services.	Enables businesses to swap out components as needs evolve.
Industrial Application	MACH is primarily used in digital commerce and related applications	Applicable across various industries and organizational scales.

Thus, from table-1 it can be identified that, composable Architecture serves as a broad, modular design philosophy emphasizing the independent combination, replacement, and scaling of components for adaptable systems across diverse industries, while MACH Architecture represents a specific technical instantiation of this philosophy, particularly within digital commerce, defined by its four key principles of Microservices, API-first, Cloud-native, and Headless, which act as an enabler for building composable solutions by providing a concrete framework for achieving modularity, flexibility, and rapid innovation through its distinct technical backbone and focus.

8. Challenges and Future Directions

8.1 Challenges

Challenges encountered by the existing MACH and composable architecture is listed as follows,

a) Increased Complexity

- **Distributed Systems:** Moving from a monolithic architecture to a collection of independent microservices inherently introduces complexity. Managing numerous services, the interdependencies and communication pathways becomes a significant undertaking.
- **Operational Overhead:** Monitoring, deploying and maintaining a distributed system with many moving parts requires sophisticated tooling, processes and expertise. This can lead to increased operational overhead compared to managing a single application.
- **Data consistency:** Ensuring data consistency across various independent databases used by different microservices can be complex. Thus, better strategies can be used for careful implementation and comprehending the implications.

b) Security Considerations

- **Increased Attack Surface:** A distributed system with numerous APIs presents a larger attack surface compared to a monolithic application. Each service and its API endpoint needs to be secured independently.
- **Complexity of Security Management**: Managing security policies and configurations across a multitude of services can be more complex than in a monolithic environment.
- **Governance and Security**: MACH architecture decentralizes services and APIs increasing potential vulnerabilities and governance challenges. Besides, the decentralized nature of MACH can make it harder to enforce standards and maintain overall architectural coherence without strong governance processes.

c) Reliability and performance of the System

- **Network Latency:** Communication between distributed services introduces network latency, which can impact overall application performance. Careful consideration of service location and communication patterns is necessary.
- **Inconsistencies across microservices:** Each microservice may have its own codebase and development team, leading to potential inconsistencies in design and functionality.
- **Monitoring and Observability:** Comprehensive monitoring and observability are essential to track the health, performance, and behavior of individual services and the overall system. This requires robust logging, tracing, and metrics collection.

Thus, in order to overcome these drawbacks, future advancements must be made to uplift the performance of MACH and composable architecture.

8.2 Future Recommendations

a) ***Enhanced integration with Emerging Technologies:*** As technology continues to evolve at an unprecedented pace, the integration of emerging technologies such as Artificial Intelligence, Machine Learning, Deep Learning, Internet of Things, and Blockchain with composable architecture is becoming increasingly crucial. This integration is essential for creating a flexible foundation that supports innovative solutions across various industries. Composable architecture, by its nature, allows for the modular design of software systems, making it easier to incorporate these emerging technologies.

b) ***Low-Code/No-Code Integration:*** Composable architectures are poised to increasingly incorporate low-code and no-code platforms, empowering business users to assemble and manage certain components and experiences with less reliance on developers. This shift democratizes application development, allowing non-technical stakeholders to contribute to the creation and customization of digital experiences. Low-code platforms, with their intuitive drag-and-drop interfaces, simplify the development process, making it accessible to a broader range of professionals. The integration of low-code and no-code tools with composable architecture enables businesses to accelerate their digital transformation by streamlining application development and reducing the time-to-market for new features. It also allows for more agile responses to changing market conditions and customer needs. However, challenges such as complexity limits and integration with legacy systems must be addressed to fully leverage these platforms.

c) ***Edge Computing Integration:*** For applications requiring low latency and local data processing, composable components will be deployed and orchestrated at the edge. Edge computing involves processing data closer to its source, reducing latency and improving real-time responsiveness. This is particularly important for applications in industries such as manufacturing, transportation, and healthcare, where timely data processing can be critical. Composable architecture is well-suited for edge computing because it allows for the modular deployment of components in distributed environments. This means that businesses can ensure seamless user experiences even in scenarios where data needs to be processed locally and quickly. For instance, in a smart city scenario, composable components can be used to manage traffic flow by processing real-time sensor data at the edge, reducing congestion and improving safety. The integration of composable architecture with edge computing also enables more efficient management of IoT devices. By processing data locally, organizations can reduce the amount of data that needs to be transmitted to central servers, improving network efficiency and reducing costs.

d) ***Focus on Developer Experience (DX):*** As the number of components in composable systems increases, there is a growing need for tools and platforms that simplify the development, deployment, and management of these solutions. Improving developer experience (DX) is crucial for ensuring that developers can efficiently work with composable architectures, thereby enhancing productivity and reducing complexity. Emerging tools and platforms are designed to streamline the lifecycle of composable components, from creation to deployment. These tools often include features such as automated testing, continuous integration/continuous deployment (CI/CD) pipelines, and observability tools to monitor performance metrics. By focusing on DX, organizations can ensure that their developers are empowered to build and maintain complex systems efficiently, which is essential for driving innovation and competitiveness in the digital landscape. Moreover, as composable architecture becomes more prevalent, the demand for skilled developers who can manage and integrate these modular systems will increase. Therefore, investing in developer experience not only improves productivity but also helps attract and retain top talent in the industry. This focus on DX will be a key factor in the successful adoption and implementation of composable

Therefore, these are some of the key future recommendations that need to be considered for better performance of MACH and composable architecture.

9. Conclusion

The present paper focused on reviewing the principles of MACH and composable architecture. The paper has projected that, MACH architecture offered several key benefits that make it an attractive choice for businesses seeking to modernize their digital solutions. Microservices enabled the development of applications as suites of small, independent services, each handling a specific task. This modular structure simplified development and maintenance, allowing for continuous deployment and delivery practices. API-first ensured seamless integration and interoperability by exposing all functions through standardized APIs, facilitating communication between different areas of the technology stack. Cloud-native applications leveraged cloud infrastructure for scalability, high availability, and automatic updates, optimizing costs and performance. Finally, Headless architecture decoupled the user interface from the backend, allowing different frontends to access the same backend services, thereby enhancing flexibility and reducing vendor lock-in.

Likewise, composable architecture represented a broader paradigm in web development, where applications are constructed from modular, interchangeable components. This approach is characterized by modularity, interoperability, scalability, flexibility, and decentralization. Composable architecture allowed businesses to build systems that are agile, resilient, and adaptable to future technological advancements. While MACH is inherently composable, it focused on specific technical approaches, making it a narrower framework within the broader context of composable architecture. The benefits of adopting a composable approach include enhanced agility, improved scalability, reduced risk, cost efficiency, and future-proofing, as individual components can be updated or replaced without disrupting the entire system. Therefore, these architectures were explored in the present study in e-commerce setup. Additionally, the present work will aid business professionals in the e-commerce sector who are seeking to modernize the digital solutions by providing insights into the principles and benefits of MACH and composable architecture.

References

[1] G.-D. Schwarz, A. Bauer, D. Riehle, N. J. I. Harutyunyan, and S. Technology, "A taxonomy of microservice integration techniques," *Information Software Technology*, p. 107723, 2025.
[2] T. Adewale, "Implementing API-First Data Processing Pipelines in Cloud Environments," 2025.

[3] A. Owen, "Microservices Architecture and API Management: A Comprehensive Study of Integration, Scalability, and Best Practices," 2025.
[4] R. Strauss, "On the Way to Composability: The Changing Role of IT," in *Data-Driven Customer Engagement: Mastering MarTech Strategies for Success*: Springer, 2024, pp. 213-229.
[5] A. Korotenko, "Microservices Architecture: practical implementations, benefits, and nuances," 2024.
[6] G. Blinowski, A. Ojdowska, and A. J. I. a. Przybyłek, "Monolithic vs. microservice architecture: A performance and scalability evaluation," vol. 10, pp. 20357-20374, 2022.
[7] Atlassian, "Advantages of microservices and disadvantages to know."
[8] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle, "Microart: A software architecture recovery tool for maintaining microservice-based systems," 2017, pp. 298-302: IEEE.
[9] M. Raatikainen, E. Kettunen, A. Salonen, M. Komssi, T. Mikkonen, and T. Lehtonen, "State of the practice in application programming interfaces (APIs): A case study," 2021, pp. 191-206: Springer.
[10] E. Koivula, "Building Competitive Advantage with API Strategy–Case Study of Established Enterprises," 2023.
[11] T. Adewale, "Designing Lightweight API-First Data Processing Services for Cloud Computing," 2025.
[12] M. Medjaoui, E. Wilde, R. Mitra, and M. Amundsen, *Continuous API management*. " O'Reilly Media, Inc.", 2021.
[13] K. Lane and A. Asthana, *The API-First Transformation*. Postman, Incorporated, 2022.
[14] V. Baladari, "Cloud Resiliency Engineering: Best Practices for Ensuring High Availability in Multi-Cloud Architectures."
[15] J. Paulsson, "Code Generation for Efficient Web Development in Headless Architecture," ed, 2024.
[16] M. Dudjak, G. J. I. T. Martinović, and Control, "An API-first methodology for designing a microservice-based Backend as a Service platform," vol. 49, no. 2, pp. 206-223, 2020.
[17] Hapio. *What is Headless and Composable Architecture?* Available: https://hapio.io/headless-composable-architecture/
[18] C. Guo, L. Li, and M. J. I. T. o. I. I. Zukerman, "Q-Learning-Based Workload Consolidation for Data Centers With Composable Architecture," *IEEE Transactions on Industrial Informatics*, 2024.