Potential Solution Architecture

Introduction

To develop the puzzle game using the scaled agile framework, it’s important to choose an architecture that's flexible, scalable and efficient and allows efficient and smooth collaboration between the three groups that are working simultaneously. We aim to select an architecture that not only helps us work simultaneously but also to accommodate the specific needs for the puzzle game.

Potential Architectures

Potential Architecture 1: <Microservices Architecture>

Microservices architecture is chosen because of its ability to break down the application into smaller and independently deployable services where each service focuses on executing a specific function [1]. This architecture aligns with the principles of the scaled agile framework by supporting separate workflows for each group to develop and test their services independently. Furthermore, it also allows rapid iteration on game mechanics and it’s scalable [1].

Advantages and disadvantages

Property	Advantage	Disadvantage
Usage of API	Loose coupling between services which enhances modularity and flexibility.	Requires careful API management as microservices heavily relies on APIs and API gatewars to facilitate communication between components and other applications [2].
Scalability	Each service can be independently scaled which allows for efficient resource utilisation and also enables precise scaling of only the components that require it [3].	Complexity increases with each services added, requiring more sophisticated understanding of the overall system.
Development speed	Enables parallel development which can speed up feature releases.	Potential for overhead in coordinating changes across teams.
Technology diversity	Allows to choose the best technology stack for each service.	Potential fragmentation and challenging to maintain consistency across the application.

Risks

1. Risk event 1: Service interdependency and communication failure
2. Risk event 2: Inconsistent Data management across services
3. Risk event 3: Security vulnerabilities due to increased attack surface

Risk Event	**Probability**	**Impact**	**Risk Response**
Risk event 1	40%	High	Establish clear contracts for APIs and user versioning to manage changes and regularly test inter-service communication and have a fallback mechanism to ensure system resilience.
Risk event 2	50%	High	Standardise on a set of data management policies and technologies to ensure consistency.
Risk event 3	30%	High	Implement robust authentication and authorization mechanisms for service to services communications.

Potential Architecture 2: <Event driven architecture>

An event driven architecture is a software architecture for applications that detect and respond to events in real time or near real time instead of periodically polling for updates. An event is a significant change in the state of a system or its environment [4]. The occurrence of an event in the past, or its current unfolding, or a prediction of an event in the future is deduced from data [4].

It was chosen as this pattern replaces the traditional “request/response” architecture where services would have to wait for a reply before they could move onto the next task. The flow of event-driven architecture is run by events and it is designed to respond to them or carry out some action in response to an event. It is often referred to as “asynchronous” communication [5].

Additionally, Event-driven architectures have three main components:
1) Event producers: publish events to the router [6]
2) Event routers: filters, processes and routes the events to consumers [6]
3) Event consumers: receive the event and take action on them [6]

Advantages and disadvantages

Property	Advantage	Disadvantage
Usage of API	Provides standardised communication and seamless integration with external systems.	Dependency on external services, vulnerability to disruptions or changes in API functionality.
Asynchronous Communication	Allows for real-time processing and responsiveness.	Complexity in managing event ordering and debugging.
Loose Coupling	This decoupling helps in the logical separation of production and consumption of events [6].Enables scalability and flexibility in system design.	May lead to difficulties in maintaining consistency and debugging.
Scalability	It scales dynamically with increased traffic and also increases the number of concurrent executing functions [6]. This is possible as operations are smaller and can be loaded into any of the free executors available.	Complexity in managing system-wide scalability, especially in distributed environments.

Risks

1. Risk event 1: System Downtime
2. Risk event 2: Data Inconsistency
3. Risk event 3: Performance Degradation
4. Risk event 4: Security Breach

Risk Event	Probability	Impact	Risk Response
Risk event 1	40%	High	Implement redundancy and failover mechanisms. Regularly monitor system health and performance.
Risk event 2	70%	High	Implement data validation and consistency checks. Ensure proper error handling and rollback mechanisms.
Risk event 3	50%	Moderate	Conduct thorough performance testing and optimization. Implement caching, load balancing, and resource allocation strategies.
Risk event 4	30%	High	Implement robust security measures such as encryption, authentication, and access control. Regularly update and patch software components. Conduct security audits and penetration testing.

Potential Architecture 3: <Monolithic Architecture>

A monolithic architecture is developing the entire application as a single unified codebase, this approach simplifies the development process and management as it is just one simple codebase [7].

The reason monolithic architecture is one of the choices for the puzzle game development project is because it enables rapid prototyping and it has quick deployment which is both crucial for early testing and feedback gathering. Furthermore, considering that the puzzle game is not as complex so monolithic architecture might be viable as monolithic design facilitates easy to understand and management of the operation by the team.

Advantages and disadvantages

Property	**Advantage**	**Disadvantage**
Simplicity	Easy to develop, test, deploy and manage as all the components are within a singular application.	As the application scales, the complexity can increase. Furthermore, when everyone works on a singular codebase, merging might be an issue,
Scalability	Scaling can be simple as there is only one codebase to deal with.	If a single requirement in the codebase needs to be scaled, the whole system has to be scaled this leads to resource wastage because not all parts of the application needs to be scaled [8].
Deployment speed	In the early stages of deployment, it can be faster to launch and iterate due not needing to deal with multiple services [9].	Over time, the speed of deploying can slow down as the codebase becomes larger and harder to work with.
Technology stack	Unified technology stack which simplifies decision making and reduces the need for developers to learn multiple technologies	Limits flexibility in adopting the best technology for any specific functionalities leading to not optimal performance in certain areas.

Risks

1. Risk event 1: Difficulty scaling as user base grows
2. Risk event 2: Single point of failure
3. Risk event 3: Difficulty in adopting new technologies

Risk Event	Probability	Impact	Risk Response
Risk event 1	80%	High	Plan for modularization of the monolith into manageable pieces that can be independently scale or transition to a microservice.
Risk event 2	50%	High	Implement backup and recovery procedures including regular snapshots of the application state and database.
Risk event 3	50%	Medium	Regularly review and refactor the codebase to ensure it remains modular within the monolith

Potential Architecture 4: <Component-Based Architecture>

Component-based architecture is a framework for building software based on reusable parts. Each component contains well-defined functionality into a binary unit that is stored in a library and then dropped into an application without modifying other components [10]. The benefits include reduced development and testing time, enhanced reliability (because components are pre-tested), and the flexibility to change applications by adding or replacing components without significant disruption [10].

It provides a higher level of abstraction and divides the problem into sub-problems, each associated with component partitions. There are many standard component frameworks such as COM/DCOM, JavaBean, EJB, CORBA, .NET, web services, and grid services. These technologies are widely used in local desktop GUI application design [11].

Advantages and disadvantages

Property	**Advantage**	**Disadvantage**
Reusability	Components can be reused across different projects, saving development time and effort.	Poorly designed or documented components may not be reusable, leading to duplicated efforts.
Modularity	Components are modular and can be developed, tested, and maintained independently, promoting code organisation and scalability.	Complex dependencies between components may introduce coupling, making it challenging to modify or replace individual components.
Usage of API	Allows components to communicate and integrate seamlessly with external systems or services, enhancing interoperability.	Dependency on external APIs can introduce vulnerabilities and increase the risk of disruptions or changes in API functionality.
Scalability	Components can be scaled independently, enabling better performance and resource utilisation.	Scalability challenges may arise when components are not designed with scalability in mind, leading to performance bottlenecks.

Risks

1. Risk event 1: Dependency Management
2. Risk event 2: Integration Challenges
3. Risk event 3: Poor Component Reusability

Risk Event	Probability	Impact	Risk Response
Risk event 1	60%	High	Implement strict version control and dependency management practices. Regularly update and review dependencies.
Risk event 2	40%	Moderate to High	Establish clear interface specifications and communication standards. Conduct thorough integration testing.
Risk event 3	40%	Moderate	Encourage proper documentation and design guidelines for building reusable components. Conduct code reviews.

Potential Architecture 5: <Client-Server Architecture>

Client-server architecture is a foundational model in computing where computational tasks are divided between client devices and server systems, facilitating efficient data processing and communication over a network [12]. In this architecture, clients, such as personal computers, smartphones, or IoT devices, initiate requests for services or resources from servers, which are powerful machines or systems dedicated to providing those services [12]. The server responds to client requests by processing the requested data or executing the required tasks and then sending the results back to the client [12].

This division of labour allows for centralised management and storage of data, as well as scalability and flexibility in distributing computing resources [13]. Additionally, client-server architecture enables the development of distributed applications that can be accessed from various devices connected to the network, offering users a seamless and consistent experience across different platforms [13].

Advantages and disadvantages

Properties	**Advantages**	**Disadvantages**
Decentralised	Centralised control and management of resources.	Server downtime affects all clients.
Client-Initiated	Servers can be upgraded to accommodate more clients.	Server may struggle to handle numerous requests
Scalability	Centralised security measures can be implemented.	Setting up and maintaining server infrastructure.
Interoperability	Supports communication between different systems.	Requires expertise in network protocols and administration.
Usage of API	APIs enable seamless communication between client and server.	API versioning and maintenance can be challenging.

Risks

1. Risk event 1: Server Downtime
2. Risk event 2: Network Congestion
3. Risk event 3: Data Loss

Risk Event	**Probability**	**Impact**	**Risk Response**
Risk event 1	80%	High	Implement redundancy and failover mechanisms to minimise downtime.
Risk event 2	50%	Medium	Optimise network resources and upgrade infrastructure as needed.
Risk event 3	20%	High	Implement data backup and recovery procedures, and employ redundancy in critical systems.

Potential Languages and Frameworks

Potential Frontend Framework

Frontend Framework	Advantages	Disadvantages
React	● Large online community support which means it will be easy to pick up. ● Allows developers to view immediate changed done in the code without restarting the app ● Uses javascript which many programmers are familiar with and it is easy to pick up.	● implementation might require setting up additional tools, which can add complexity for beginners. ● Although React Native provides access to many native APIs through native modules or third-party libraries, there may still be limitations in accessing certain platform-specific features or APIs ● React Native applications may experience performance limitations compared to native applications, especially for complex and highly interactive UIs.
Flutter	● Allows cross platform development for a single codebase. ● Allows developers to view immediate changed done in the code without restarting the app ● Many customizable widget for more interactive and engaging user interface and experience.	● Has a bigger learning curve compared to react ● it may lack certain native functionalities or APIs available in platform-specific development environments. This can sometimes lead to challenges when integrating with device-specific features or accessing platform-specific resources. ● Flutter apps tend to have larger file sizes compared to native applications because they include the Flutter engine and framework libraries, which may not be fully optimised for each platform.
Svelte	● Svelte shifts much of the work to compile time rather than runtime. This approach results in smaller bundle sizes and optimised code, leading to faster load times and improved performance for web applications.	● Compared to more established frameworks like React or Angular, Svelte has a smaller ecosystem. This means there are fewer third-party libraries, plugins, and community resources available.

	● Svelte offers a clean and intuitive syntax that emphasises simplicity and readability.	● While Svelte offers a compiler that optimises code during build time, the tooling support is not as extensive as other frameworks.
Angular	● Angular promotes a modular and component-based architecture, making it easier to organise and manage complex web applications. ● Any changes made to the model are immediately reflected in the view, and vice versa	● Angular is a comprehensive framework with a lot of features and concepts to grasp, which can result in a steep learning curve for developers, especially those new to front-end development or JavaScript frameworks. ● Setting up an Angular project can be complex, especially for smaller or simpler applications.

Potential Backend Framework

Python Framework:

Python Backend Framework	**Advantages**	**Disadvantages**
Flask	● Flask is a micro-framework, meaning it has minimal built-in features and dependencies, resulting in a smaller footprint compared to full-stack frameworks. This makes it faster to set up and deploy. ● Flask provides a simple and flexible architecture, allowing developers to choose and integrate only the components they need for their specific application, resulting in greater control over the project's structure and functionality. ● Flask offers built-in support for creating RESTful APIs, making it ideal for developing web services and microservices. Its lightweight nature and simplicity make it well-suited for building scalable and efficient backend systems for web and mobile applications. ● Flask integrates seamlessly with the Jinja2 templating engine, allowing developers to create dynamic and reusable HTML templates. This enables efficient development of web applications with clean and maintainable code, enhancing productivity and scalability.	● Flask is a micro-framework, which means it offers minimal built-in functionality compared to full-stack frameworks like Django. Developers may need to rely on third-party extensions or write custom code to implement certain features, potentially increasing development time and complexity. ● Flask does not provide built-in security features such as user authentication and authorization, input validation, or protection against common web vulnerabilities like CSRF (Cross-Site Request Forgery) and XSS (Cross-Site Scripting). Developers need to implement these security measures themselves or rely on third-party extensions, increasing the risk of security vulnerabilities if not implemented correctly. ● While Flask is suitable for building small to medium-sized applications, it may face scalability challenges when handling high traffic or large datasets. Developers need to carefully design and optimise their Flask applications to ensure scalability, which may require additional effort and expertise.
Django	● With its scalable architecture and built-in tools, Django can easily	● While Django's monolithic structure provides many built-in features, it can

	handle growing user bases and increasing traffic without compromising performance. ● Django comes with built-in security features such as protection against common security threats like SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF), helping developers create secure web applications. ● Django offers a powerful admin interface that allows developers to manage site content, user authentication, and other administrative tasks without writing additional code. ● Django includes built-in authentication and authorization features, making it easy to add user authentication, permissions, and user management to web applications.	also lead to bloated codebases and make it challenging to integrate with certain third-party libraries or technologies. ● Django's built-in features and conventions may limit flexibility for developers who require highly customised or unconventional solutions, as they may need to work within Django's framework constraints. ● While Django's ORM (Object-Relational Mapping) simplifies database interactions, it may lack some advanced features or optimizations found in standalone ORMs, potentially impacting database performance in certain scenarios. ● Managing dependencies and updates within a Django project, including third-party packages and Django itself, can sometimes be challenging and require careful coordination to avoid conflicts or compatibility issues.

Javascript Framework:

Javascript Backend Framework	Advantages	Disadvantages
Node.js	● Node.js enables full-stack JavaScript development, allowing developers to use the same language (JavaScript) for both frontend and backend development, streamlining the development process and reducing context switching. ● Node.js has a vast ecosystem of open-source libraries and modules available through NPM (Node Package Manager), providing developers with access to a wide range of tools, frameworks, and utilities to accelerate development and extend functionality. ● Node.js is ideal for building real-time applications such as chat applications, online gaming platforms, and collaborative tools, due to its event-driven architecture and support for WebSockets, enabling bidirectional communication between clients and servers in real-time. ● Node.js offers excellent performance, particularly for I/O-intensive applications, thanks to its event-driven, non-blocking architecture, which minimises overhead and maximises throughput, resulting in fast response times and efficient resource utilisation. ● Node.js is well-suited for building microservices-based architectures, allowing developers to create modular, decoupled services that can be independently developed, deployed, and scaled, facilitating agility, maintainability, and scalability.	● Despite its asynchronous nature, Node.js may not be suitable for CPU-bound tasks or highly concurrent applications due to its single-threaded nature, limiting its scalability compared to multi-threaded frameworks. ● Error handling in Node.js can be complex, especially with asynchronous operations, as errors must be manually propagated through callbacks or handled with additional modules, increasing the risk of overlooked errors and potential bugs. ● Node.js heavily relies on callback-based programming, which can lead to callback hell and makes code harder to understand, especially for developers accustomed to synchronous programming paradigms. ● While NPM (Node Package Manager) provides a vast ecosystem of packages and modules, it can also lead to dependency management issues, such as dependency conflicts, security vulnerabilities, and package version mismatches, requiring careful management. ● Node.js applications are susceptible to memory leaks, especially in long-running processes or applications with frequent asynchronous operations, requiring careful memory management and monitoring to prevent performance degradation over time.

Sails.js	● Sails.js automatically generates RESTful APIs (Application Programming Interfaces) for data models, simplifying API development and speeding up the backend development process. ● Sails.js offers powerful data handling capabilities, including support for relational databases (MySQL, PostgreSQL) and NoSQL databases (MongoDB), allowing developers to seamlessly integrate and interact with various data sources. ● Sails.js allows developers to extend and customise the framework's functionality using hooks, which are modular components that can be added or removed based on project requirements, providing flexibility and adaptability. ● Sails.js automatically generates CRUD (Create, Read, Update, Delete) operations for models, reducing boilerplate code and speeding up development time. ● Sails.js is designed to be scalable, supporting horizontal scaling by distributing workload across multiple instances or servers, ensuring optimal performance and reliability for applications under heavy load.	● While Waterline ORM provides flexibility by supporting multiple databases, it may also lead to performance bottlenecks or limitations in complex database operations, compared to using database-specific ORMs. ● Sails.js heavily relies on asynchronous programming using callbacks, promises, or async/await, which may lead to callback hell or difficulties in managing asynchronous code, especially for developers new to asynchronous programming paradigms. ● Although Sails.js has an active community, it may not be as large or diverse as other Node.js frameworks like Express.js, resulting in fewer third-party plugins, extensions, or community-driven resources available for developers. ● Sails.js uses a convention-based routing system, which may limit flexibility in defining custom routing rules or handling complex routing scenarios, especially for applications with unconventional URL structures or routing requirements.

Typescript Framework:

Typescript Backend Framework	**Advantages**	**Disadvantages**
Nest.js	● Nest.js encourages a modular structure for applications, making it easy to organise code into reusable and maintainable modules, enhancing scalability and code organisation. ● Nest.js includes built-in middleware support for common tasks such as logging, error handling, authentication, and authorization, reducing the need for third-party libraries and simplifying the development process. ● Nest.js provides flexibility in terms of choosing databases, libraries, and tools, allowing developers to integrate with various technologies based on project requirements and preferences. ● Nest.js is built on top of Express.js, allowing developers familiar with Express to transition smoothly to Nest.js while still leveraging the robustness and performance of the underlying Express framework. ● Nest.js provides built-in support for unit testing and integration testing through various testing utilities and frameworks, enabling developers to write comprehensive test suites and ensure the reliability and stability of their applications.	● While Nest.js provides abstractions that promote code organisation and modularity, this can sometimes lead to additional complexity, especially for developers who prefer simpler, more straightforward frameworks. ● While Nest.js is built on top of Express.js, which is known for its performance, the additional abstractions and features introduced by Nest.js may incur some performance overhead compared to using Express.js directly for simpler applications. ● While Nest.js provides flexibility in terms of choosing databases, libraries, and tools, it also imposes certain conventions and opinionated patterns, which may not align with the preferences or requirements of all developers or projects. ● Due to the use of TypeScript and additional abstractions, the output bundle size of Nest.js applications may be larger compared to applications built with more lightweight frameworks, potentially impacting load times and performance.

Database Management System(DBMS):

Typescript Backend Framework	**Advantages**	**Disadvantages**
Oracle	● Oracle Database can handle large amounts of data and can scale up to accommodate increasing workloads. ● Oracle Database is known for its high performance and efficient data processing capabilities. ● Oracle Database offers features such as data replication, backup, and recovery to ensure data reliability and high availability. ● Oracle Database provides robust security features, including encryption, access controls, and auditing, to protect sensitive data. ● Oracle Database offers a wide range of features for data management, including support for SQL, PL/SQL, and advanced analytics.	● Oracle Database can be expensive, especially for small businesses or startups, due to licensing and support costs. ● Oracle Database can be complex to manage and administer, requiring specialised skills and training. ● Oracle Database may require significant hardware resources, including memory and processing power, to run efficiently. ● Oracle Database may result in vendor lock-in, making it difficult to switch to alternative solutions in the future. ● Oracle Database may require regular maintenance, updates, and patches, which can add to the administrative overhead.
Redis	● Redis is known for its speed and low latency, making it suitable for real-time applications. ● Redis stores data in memory, enabling fast read and write operations. ● Redis supports various data structures such as strings, lists, sets, and hashes, providing flexibility in data modelling. ● Redis provides built-in support for Publish/Subscribe messaging, facilitating communication between components in real-time systems. ● Redis supports master-slave	● Redis primarily supports key-value data model, limiting its use cases compared to relational databases. ● While Redis offers persistence options, it may not be as robust as traditional databases, leading to potential data loss in certain scenarios. ● Scaling Redis horizontally can be challenging compared to some other databases, and vertical scaling may have limitations. ● Redis operates as a single-threaded process, which may limit its ability to fully utilise multi-core processors in high-load scenarios.

	replication, allowing for high availability and fault tolerance.	● Redis may have a steep learning curve for developers unfamiliar with its specific data structures and commands.
Firebase	● Firebase Cloud Firestore provides real-time data synchronisation, enabling instant updates across connected clients. ● Firestore automatically scales to handle growing datasets and user loads, making it suitable for applications with varying demand. ● Firestore offers offline support, allowing applications to function even when users are offline, and automatically syncs data once connectivity is restored. ● Firebase provides built-in authentication and security features, including OAuth, email/password authentication, and role-based access control. ● Firebase Cloud Firestore seamlessly integrates with other Firebase services, such as Cloud Functions, Cloud Storage, and Authentication, simplifying development and deployment workflows.	● Firestore queries have limitations compared to traditional SQL databases, which may impact complex data retrieval operations . ● While Firebase offers a free tier, scaling beyond certain usage limits can lead to increased costs, especially for large-scale applications. ● Firestore's NoSQL data model may require careful consideration of data structure and denormalization, which can be complex for some use cases. ● Firebase Cloud Firestore ties applications to the Google ecosystem, potentially limiting flexibility and portability in the long term. ● Firestore lacks server-side scripting capabilities, which may require additional backend services for complex business logic.

Spikes

A spike is a focused, time-boxed research activity aimed at exploring a specific aspect of a project, typically to gather information, validate assumptions, or mitigate risks.
Here are potential spikes for each scenario:

1. Using Microservices Architecture:
    - Performance Spike where the measure of the performance overhead of microservices is compared to other architectures, which can be done by conducting benchmarks and load tests to evaluate scalability and latency.
    - Security Spike where we would need to assess the security implications of microservices, including network security, data privacy, and authentication/authorization mechanisms.

2. Using TypeScript with NestJS:
    - Compatibility Spike where we would need to evaluate the compatibility of existing libraries and dependencies with TypeScript and NestJS.
    - Learning Spike where each team member would need to invest time in learning TypeScript if your team is not already familiar with it by exploring TypeScript features, syntax, and best practices.

3. Using React Native:
    - Performance Spike is an important aspect to consider to investigate the performance characteristics of React Native for your target platforms (iOS, Android), by measuring the rendering performance, memory usage, and startup times.
    - Cross-Platform Compatibility Spike is considered to test the compatibility of our UI components and third-party libraries across different platforms. We would also have to identify any platform-specific issues and devise strategies for handling them, if any.

4. Using Firebase as a Database Management System (DBMS):
    - Data Modeling Spike where we would need to design and prototype the data model for our application using Firebase and determine how to structure data for efficient querying and storage.
    - Integration Spike is considered to iInvestigate the process of integrating Firebase with our chosen frontend and backend technologies, i.e. React Native and Nest.JS.

These spikes will help us gather valuable insights, validate assumptions, and inform decision-making when considering the chosen technologies for our project.

Conclusions

Recommended Architecture Model

After evaluating the advantages and disadvantages of each architecture model, we decided that Microservices architecture would be the most suitable choice for our development project.

The reason for the chosen architecture model is because as the game grows in complexity and user base, microservice architecture allows for specific aspects of the game like user management, leaderboard and game mechanics to be scaled independently which ensures that resources are used efficiently. Furthermore, microservices supports the principles of SAFE by enabling different teams to work on different services simultaneously and independently.

This architecture als allows each service to use the best technology stack that is the most suitable for its requirements which can lead to optimization of the game. Furthermore, considering the risks it also does not have a single point of failure like monolithic architecture which means if there is an issue in one service it will be less likely to impact the entire application.

Recommended Languages

Group	Task	Language
Group 1	Backend for developing functionality to serialise the current game state, including puzzle progress, and save it to the database.	Typescript - Nest.JS
	Frontend for designing and implement a "Save Game" button in the game menu; implement UI logic to trigger game state serialisation and saving.	React.JS
	Database for storing achievement definitions and criteria. Implement APIs for creating, updating, and retrieving achievement definitions.	Firebase
Group 2	Backend for implementing functionality to retrieve saved game states from the database and deserialize them.	Typescript - Nest.JS
	Frontend for designing and implement a "Resume Game" option in the game menu; implement UI logic to load and resume a saved game state.	React.JS
	Database for fetching user account data from the database and use the relevant data like achievements to create a ranking system among all players	Firebase

Group 3	Backend for implementing API endpoints for saving and resuming game progress, including error handling.	Typescript - Nest.JS
	Frontend for designing and implement user feedback for successful save and resume actions, and error handling for any failures.	React.JS
	Database for developing functionality to track and update player performance in multiplayer matches and integrate the scores with the leaderboard rankings based on number of wins and time taken	Firebase

References

[1] C. Richardson, “Microservices.io,” *microservices.io*, 2017. [Online] Available: https://microservices.io/ [Accessed April 2, 2024]

[2] “What are Microservices?,” *SearchAppArchitecture*. [Online] Available: https://www.techtarget.com/searchapparchitecture/definition/microservices [Accessed April 2, 2024]

[3] “What are Microservices? | IBM,” *www.ibm.com*. [Online] Available: https://www.ibm.com/topics/microservices [Accessed April 2, 2024]

[4] K. Mani Chandy, “Event Driven Architecture,” *Springer eBooks*, pp. 1040–1044, Jan. 2009. [Online] Available: https://doi.org/10.1007/978-0-387-39940-9_570. [Accessed April 2, 2024]

[5] “What is Event-driven Architecture? | TIBCO,” *TIBCO*, 2024. [Online] Available: https://www.tibco.com/glossary/what-is-event-driven-architecture [Accessed April 2, 2024]

[6] S. Kawle, “Advantages And Disadvantages Of Event Driven Architecture,” *www.neebal.com*. [Online] Available: https://www.neebal.com/blog/advantages-and-disadvantages-of-event-driven-architecture [Accessed April 2, 2024]

[7] R. Awati and I. Wigmore, “What is monolithic architecture? - Definition from WhatIs.com,” *WhatIs.com*, May 2022. [Online] Available: https://www.techtarget.com/whatis/definition/monolithic-architecture [Accessed April 3, 2024]

[8] “Monolithic vs Microservices - Difference Between Software Development Architectures- AWS,” *Amazon Web Services, Inc.* [Online] Available: https://aws.amazon.com/compare/the-difference-between-monolithic-and-microservices-architecture/ [Accessed April 3, 2024]

[9] Atlassian, “Microservices vs. monolithic architecture,” *Atlassian*, 2023. [Online] Available: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith#:~:text=A%20monolithic%20architecture%20is%20a [Accessed April 3, 2024]

[10] P. Gillin, “What is Component-Based Architecture?,” *Mendix*, Jan. 15, 2024. [Online] Available: https://www.mendix.com/blog/what-is-component-based-architecture/#:~:text=Component%2Dbased%20architecture%20is%20a [Accessed April 3, 2024]

[11] "Component-Based Architecture - Tutorialspoint," *www.tutorialspoint.com*. [Online] Available: https://www.tutorialspoint.com/software_architecture_design/component_based_architecture.htm [Accessed April 3, 2024]

[12] J. Terra, "What is Client-Server Architecture? Everything You Should Know | Simplilearn," *Simplilearn.com*, Apr. 22, 2022. [Online] Available: https://www.simplilearn.com/what-is-client-server-architecture-article [Accessed April 6, 2024]

[13] IntelliPaat, "What is Client Server Architecture? - Differences, Types, Example," *Intellipaat Blog*, Sep. 24, 2021. [Online] Available: https://intellipaat.com/blog/what-is-client-server-architecture/ [Accessed April 6, 2024]