SE Project 3

CineStream Platform

Name	Roll Number	Email
Yash Pathak	2022201026	yash.pathak@students.iiit.ac.in
Gaurav Khapekar	2022201055	gaurav.khapekar@students.iiit.ac.in
Vivek Kirpan	2022201071	vivek.kirpan@students.iiit.ac.in
Adarsh	2022201081	adarsh.2022@students.iiit.ac.in
Prasad Kawade	2022202006	prasad.kawade@students.iiit.ac.in
Date of Submission – 18/04/2023		

➢ Prerequisites

What is CineStream Platform?

CineStream is an OTT platform providing a wide range of movies in all languages for users to watch from home. Users can also rent movies from other OTT platforms like Netflix, Amazon Prime, etc. and request movies on the platform. Users can rate, review and provide likes/dislikes for the movies they watch. The platform provides personalized recommendations to users. Users can subscribe to the platform on a monthly or yearly basis and discounts are offered to active users based on their involvement. User metrics are collected to enhance the overall experience.

Main Constraints of CineStream platform

The CineStream platform must handle various types of workloads, such as streaming games, movies, and popular series from different OTT platforms. Users must only stream content from one platform on a single device at any given time.
The platform must handle continuous data inflow from various streaming platforms and offer seamless user experience. The system must not store any sensitive user data due to privacy concerns. Only two devices per user are allowed to be logged in simultaneously.
The platform must provide admins/maintenance teams with easy access to analytics such as number of users, response time, server utilization, and throughput. The system must be environment friendly by using means to reduce energy consumption and carbon footprint. The OTT providers have exposed APIs for content gathering.

Task 1: Requirements and Subsystems

➢ Requirement Refinement

Functional Requirements*

1. Provide a platform to users to watch movies in all languages, including latest and classic movies. Users should also be able to live streaming games, some movies, some popular series and each can be from different OTT platforms.
2. Allow users to search for movies by title, actor, director, genre, language, and other relevant criteria.
3. Allow users to stream movies from other OTT providers' websites by paying a nominal rent on a pay-per-movie basis.
4. Provide a flexible subscription model to users, allowing them to pay a monthly or yearly charge to the platform.
5. The system should support multiple payment options for users to rent the movies and subscribe to the platform, such as credit card, PayPal, or other payment gateways.
6. The system should be able to support a wide range of devices and platforms, including desktops, laptops, tablets, and smartphones, running different operating systems like Windows, MacOS, iOS, Android, etc. The platform should also be able to seamlessly stream content across different platforms without any interruption or loss of quality.
7. Allow users to provide ratings, likes/dislikes, and reviews for the movies they have watched. They should also be able to view ratings and reviews submitted by other users. Overall ratings should be available for each content.
8. Allow users to request movies on the platform, and make them available to users who have raised the request, based on availability.
9. Present personalized movie recommendations to users based on their viewing history and preferences. Allow users to create playlist of such movies to be watched later.
10. Provide discounts on the subscription fee to active users based on their involvement in the platform.
11. Collect user metrics such as likes, time spent on movies, number of movies watched, etc., to enhance the overall experience.
12. Users should be able to create playlists where they can add content of similar type which can be made public to other users also.
13. OTT platform admin should be able to login to add new contents and change settings, if needed.
14. Analytics: The system should provide easy means for admins/maintenance team to get access to the different analytics with respect to platform use.

Non-Functional Requirements*

1. High Availability and Scalability: The system should be able to handle at least 10,000 concurrent users, and should be designed to scale horizontally up to 50,000 concurrent users.
1. Low Latency and Fast Response Times: The system should have a maximum latency of 500ms, and should be designed to deliver content to users within 3 seconds of their request.
2. High throughput: The system should have a minimum throughput of 1000-10,000 requests per second and should be designed accordingly.
3. High Scalability: The system should be scalable to accommodate future growth without requiring significant changes in design and significant infrastructure investment.
4. High Maintainability and Supportability: The system should be designed with maintainability and supportability in mind, with modular architecture and easy-to-use tools and interfaces for monitoring and maintenance tasks.
5. Device Limitation: The system should only support a maximum of 2 devices per user.
6. Time to market: The app should be available for initial testing with architecturally significant requirements developed within 4 months.
7. Security and Data Privacy: The system should use strong encryption mechanisms (e.g., AES-256) for all user data, and should have access controls in place to restrict data access to authorized personnel only. The system should also comply with relevant regulations, such as GDPR or CCPA.
8. Environment Friendly: The system should aim to reduce energy consumption and carbon footprint by using efficient hardware and software, and by implementing techniques such as content caching and data compression. The system should also have a plan in place for responsible disposal of hardware and electronic waste.
9. Use of Open-Source Software Components: The system should open-source software components wherever possible to minimize licensing cost by 80% and improve flexibility as the open-source components can be changed according to the system.

* Order in terms of priority

Architecturally Significant Requirements

Architecturally significant requirements are the ones that have a significant impact on the system's architecture and design decisions. These requirements are key because they form the backbone of the CineStream platform and are essential for providing users with a seamless and high-quality streaming experience.

Architecturally significant functional requirements will guide the selection of technologies and components required to implement the system architecture.

Architecturally significant non-functional requirements will guide the selection of appropriate infrastructure, data storage and processing technologies, security protocols, and other architectural decisions.

Combining these two ensures that the platform is able to meet the needs of the users and provide a high-quality user experience while being scalable, maintainable, and secure.

Based on the requirements listed, the following are the architecturally significant requirements and why:

1. High Availability and Scalability: This requirement will drive the need for a distributed architecture that can scale horizontally or vertically based on demand. It will require the use of load balancing, clustering, and other techniques to ensure high availability and performance.
2. Low Latency and Fast Response Times: This requirement will drive the need for a fast and efficient data processing and retrieval mechanism to minimize latency and improve response times. It will require the use of techniques such as content delivery networks (CDNs), caching, and data partitioning to ensure fast access to content.
3. Security and Data Privacy: This requirement will drive the need for encryption, access controls, and other security measures to protect user data and ensure compliance with relevant regulations. It will require the use of techniques such as authentication, authorization, and encryption to ensure data privacy and security.
4. Device Limitation: This requirement will drive the need for a session management mechanism that can track user activity and limit access to the platform to a maximum of 2 devices per user.
5. Analytics: This requirement will drive the need for a data analytics platform that can collect, store, and analyse data related to platform use, such as number of users, response time, server utilization, throughput, etc. It will require the use of techniques such as data warehousing, data mining, and reporting to provide insights into platform use.
6. Environment Friendly: This requirement will drive the need for techniques and technologies that can reduce the energy consumption and carbon footprint of the system, such as the use of energy-efficient hardware, virtualization, and cloud-based solutions.

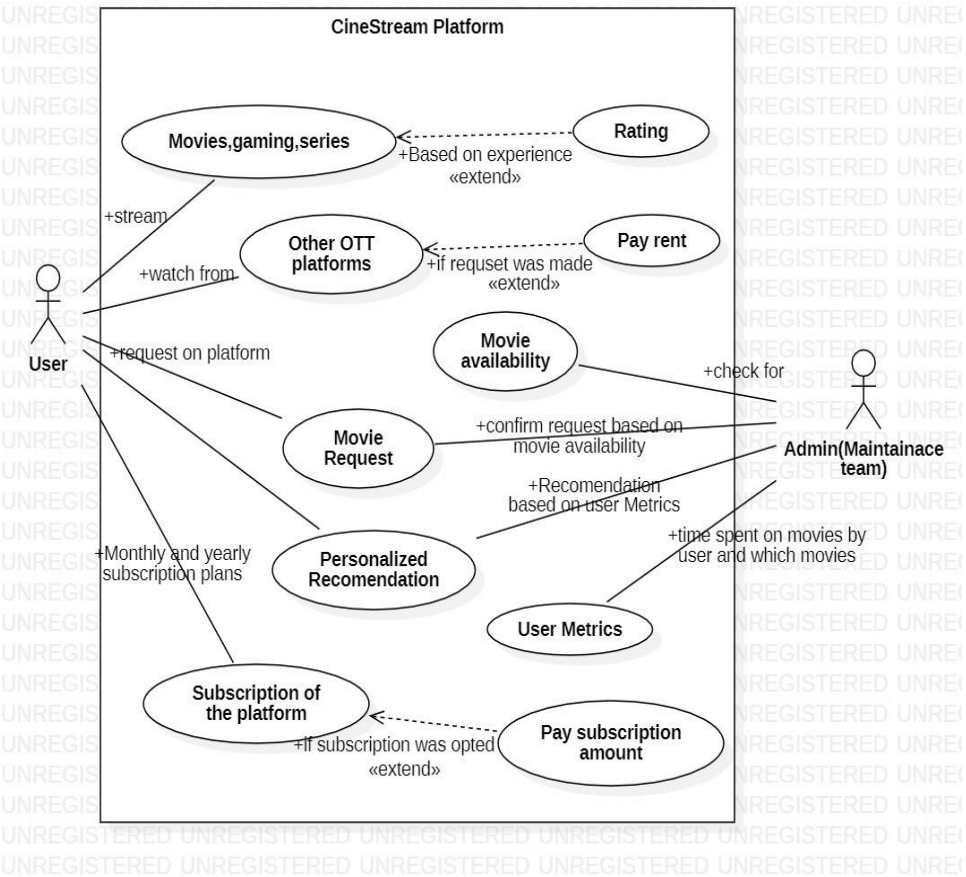


Fig 1: Use Case Diagram

User Stories

1. User Stories for User/Subscriber:
    - As a user, I want to be able to sign up for the time when I visit the platform and be able to login later on my further visits
    - As a user, I want to be able to search for movies by title, actor, director, genre, language, and other relevant criteria so that I can easily find the content I am interested in watching.
    - As a user, I want to be able to rent movies from other OTT providers' websites by paying a nominal rent on a pay-per-movie basis through multiple payment options such as credit card, PayPal, or other payment gateways.
    - As a user, I want to be able to subscribe to the platform with a flexible subscription model, allowing me to pay a monthly or yearly charge so that I can access the content without having to pay for each movie.
    - As a user, I want to be able to request movies on the platform and have them made available to me based on availability, so that I can watch the movies I want to see.
    - As a user, I want to receive notifications of new contents, recommendations, expiring subscriptions and new reviews.

- As a user, I want to be able to provide ratings, likes/dislikes, and reviews for the movies I have watched, and view ratings and reviews submitted by other users so that I can make informed decisions about what to watch.
- As a user, I want to receive personalized movie recommendations based on my viewing history and preferences, so that I can discover new content that I am likely to enjoy.
- As a user, I want to be able to access the platform on a wide range of devices and platforms, including desktops, laptops, tablets, and smartphones, running different operating systems like Windows, MacOS, iOS, Android, etc., and seamlessly stream content across different platforms without any interruption or loss of quality.

2. User Stories for Software Developers and Architects (called as admins below):
    - As an admin, I want to be able to monitor the platform's performance with respect to number of users, response time, server utilization, throughput, etc., so that I can optimize the system for better user experience.
    - As an admin, I want to be able to access user metrics such as likes, time spent on movies, number of movies watched, etc., so that I can understand user behaviour and enhance the overall experience.
    - As an admin, I want to be able to provide movies to the users based on their requests raised on the platform, so that users can watch movies that are not available on the platform.
    - As an admin, I want to ensure that the system is environment friendly, using appropriate means to minimize the energy consumed and carbon footprint of the system, so that we can be responsible and sustainable in our operations.
    - As an admin, I want to minimize the infrastructure cost by reducing the resource utilization by the system.

➢ Brief description of the system and its architecture

CineStream is an online streaming platform that allows users to watch movies, TV shows, and live events from different OTT providers. The system should be designed to support a wide range of devices and platforms, including desktops, laptops, tablets, and smartphones running different operating systems like Windows, MacOS, iOS, and Android.

The CineStream architecture should be designed to be highly scalable and resilient, with a distributed architecture that can scale horizontally or vertically based on demand. The system should also use caching and optimization techniques to minimize latency and improve response times, ensuring a fast and efficient data processing and retrieval mechanism.

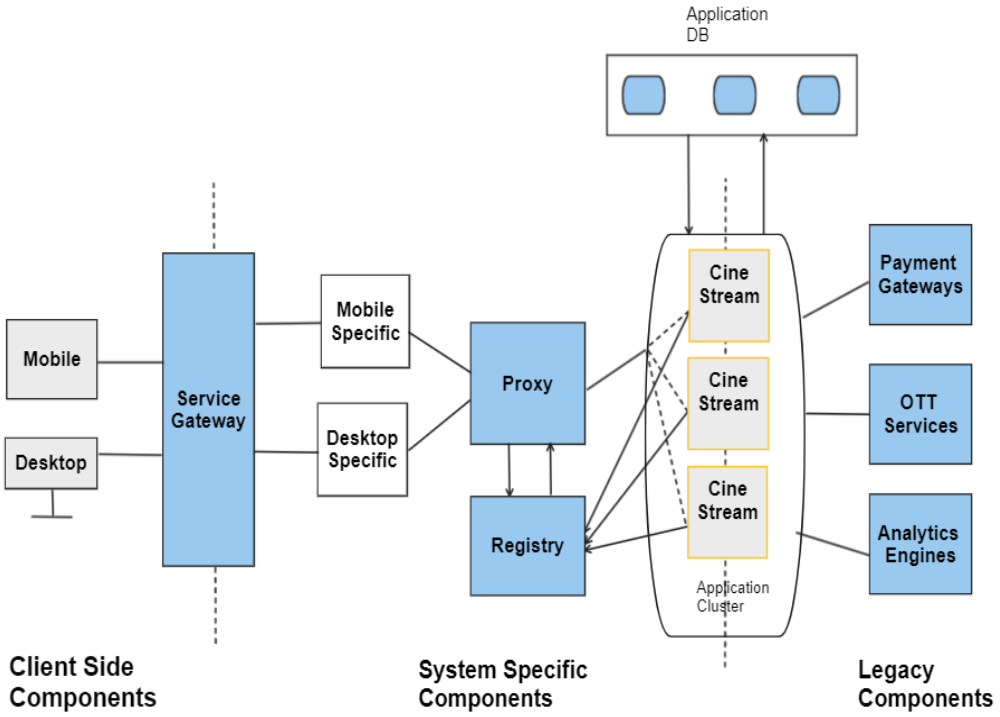


Fig 2: High-Level Architecture Diagram

The main big picture of the system is as shown above. The users can access the application via multiple types of devices and these devices should show the contents based on the device type. BFF helps to satisfy this. We need a proxy and service registry for Application instances to register and thus used by clients via proxy. These services are also connected to external services such as payment gateways etc.

Dotted lines in the diagram shows distinction between client side, legacy components and the system specific components. We will focus on the system specific components and how to architecturally design them to satisfy the requirements mentioned and thus develop a good quality application.

➢ Main Subsystems

The CineStream system is composed of several sub-systems, including a user management system, a content management system, a search and recommendation system, a payment processing system, and a streaming system. Each sub-system is responsible for performing specific tasks and communicating with other sub-systems to provide a unified user experience. Following are the few subsystems:

1. User Management System: This subsystem is responsible for user registration, login, and authentication.
2. Content Management System: This subsystem manages the content of the platform, including movies, series, and games. It also handles the search functionality, categorization, and recommendation of movies based on the user's preferences.
3. Payment Management System: This subsystem is responsible for handling all financial transactions, including movie rental charges, subscription charges, and discount management.
4. Streaming Management System: This subsystem handles the streaming of movies, games, and series across different platforms and devices.
5. Review and Rating System: This subsystem handles user reviews, ratings, and feedback for the movies and series available on the platform.
6. Analytics System: This subsystem collects and processes data on user behaviour, engagement, and feedback, and provides insights for the platform owners to improve the user experience.

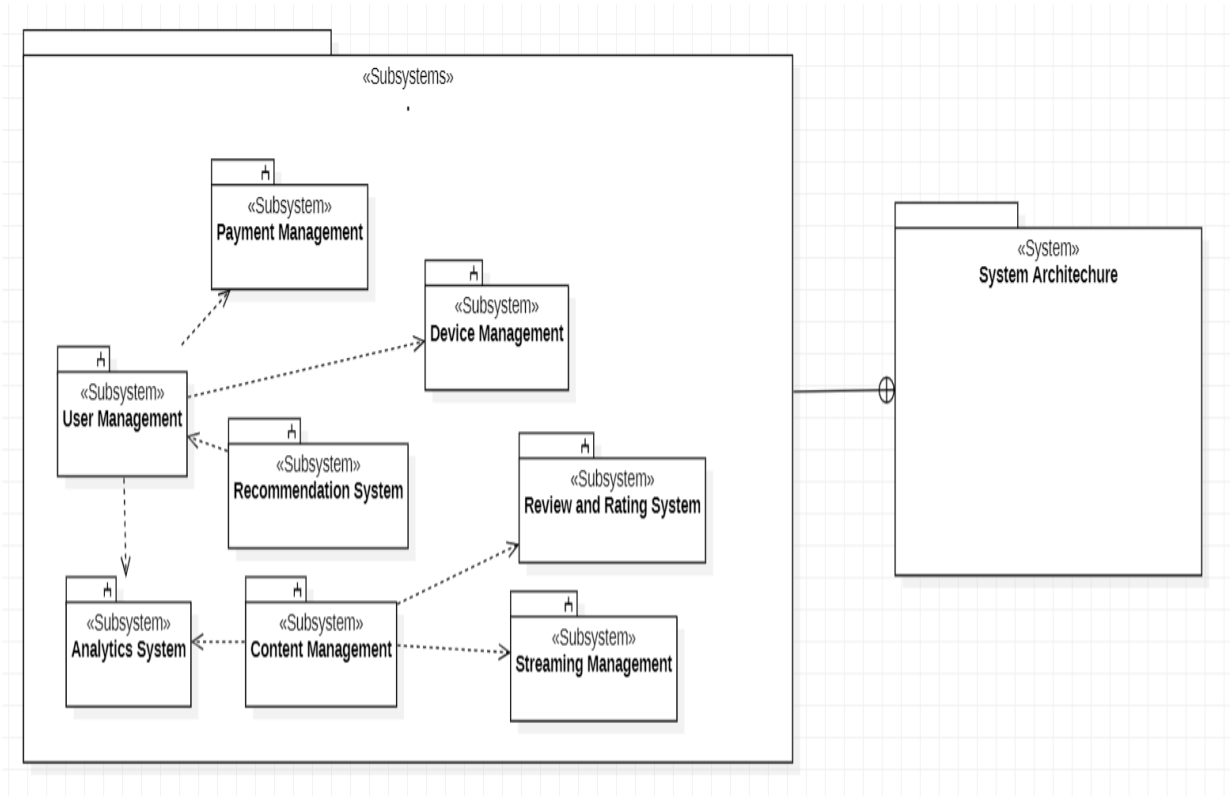


Fig 3: Subsystem Diagram

➢ How solution satisfies the requirements

The solution described for CineStream appears to satisfy the requirements in the following ways:

1. Support for multiple devices and platforms: The architecture is designed to support a wide range of devices. This ensures that users can access the platform on any device of their choice. Use of BFF helps in this.
2. Highly scalable and resilient: The architecture is designed to be highly scalable and resilient, with a distributed architecture that can scale horizontally or vertically based on demand. This ensures that the platform can handle a large number of users and traffic spikes without compromising on performance.
3. Caching and optimization techniques: The system uses caching and optimization techniques to minimize latency and improve response times, ensuring a fast and efficient data processing and retrieval mechanism. This improves the user experience and ensures that the platform is responsive.
4. Use of Subsystems: The CineStream system is composed of several sub-systems, each responsible for performing specific tasks and communicating with other sub-systems to provide a unified user experience. This ensures that the platform is modular, easy to maintain, and can be scaled independently.
5. User Management System: The user management subsystem handles user registration, login, and authentication. This ensures that users can access the platform securely.
6. Content Management System: The content management subsystem manages the content of the platform, including movies, series, and games. It also handles the search categorization, and recommendation of movies based on user's preferences. This ensures that users can easily find the content they are interested in.
7. Payment Management System: The payment management subsystem is responsible for handling all financial transactions, including movie rental charges, subscription charges, and discount management. This ensures that users can make payments securely and easily.
8. Streaming Management System: The streaming management subsystem handles the streaming of movies, games, and series across different platforms and devices. This ensures that users can access the content they want to watch, regardless of the device they are using.
9. Review and Rating System: The review and rating subsystem handles user reviews, ratings, and feedback for the movies and series available on the platform. This ensures that users can provide feedback on the content they have watched, which can be used to improve the platform.
10. Analytics System: The analytics subsystem collects and processes data on user behaviour, engagement, and feedback, and provides insights for the platform owners to improve the user experience. This ensures that the platform can be continuously improved based on user feedback and data analysis.

Task 2: Architecture Framework

➢ Design Decisions

Software Architecture are highly complex and cost to change is huge. Design decisions are embedded in architecture. This leads to problems. Ideally, architecture should be a composition of sets of explicit design decisions.

Definition of Design Decisions: “A description of the set of architectural additions, subtractions and modifications to the software architecture, the rationale, and the design rules, design constraints and additional requirements that (partially) realize one or more requirements on a given architecture.” []

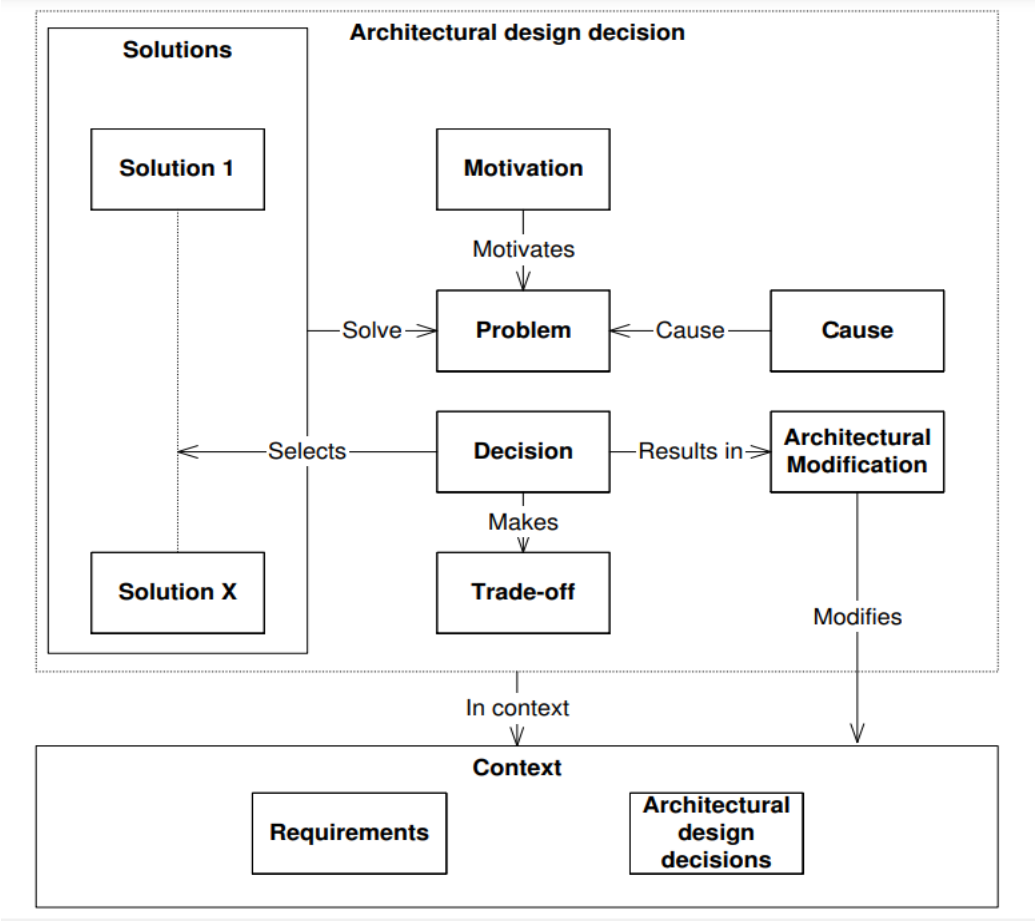


Fig 4: Conceptual Model for Architectural Design Decisions [1]

Following are the most *important* and *accepted* architecture design decisions using the Architecture Decision Records (ADR):

I. Use of Microservices Architecture
   - Issue: A monolithic system can become unwieldy and difficult to scale as the system grows in complexity and size. SOA architecture will not serve the purpose as we have to maintain high availability.
   - Change: Breaking down the monolithic architecture into smaller, more manageable microservices.
   - Benefit: Microservices can be independently deployed, scaled, and managed, making the system more resilient and easier to maintain.

II. Use of Service Registry
   - Issue: With multiple microservices, it can become difficult to manage service discovery and load balancing.
   - Change: Introducing a service registry to keep track of all microservices in the system and their endpoints.
   - Benefit: The service registry can automatically route requests to the appropriate microservice instance, improving reliability and reducing latency.

III. Use of BFF
   - Issue: With multiple front-end clients (e.g., mobile, desktop), it can become difficult to manage the presentation logic for each client separately.
   - Change: Implementing a Backend For Frontend (BFF) layer to manage the presentation logic for each front-end client.
   - Benefit: The BFF layer can simplify the logic of the front-end clients, making it easier to maintain and modify the presentation logic.

IV. Use of Caching
   - Issue: With a large number of users and high traffic, the system can become slow and unresponsive.
   - Change: Introducing caching to store frequently accessed data in memory.
   - Benefit: With caching, the system can respond more quickly to user requests, reducing latency and improving overall performance.

V. Use of external and distributed Analytics engine and Recommender system
   - Issue: With a large number of users and high traffic, the system can become slow and unresponsive due to processing and analysis of large amounts of data.
   - Change: Introducing an external and distributed Analytics engine and Recommender system to offload the processing and analysis workload from the system.

- Benefit: By utilizing an external and distributed Analytics engine and Recommender system, the system can handle large amounts of data more efficiently, reducing latency and improving overall performance.

VI. Use of Notification System

- Issue: With a large number of users and high traffic, the system can become slow and unresponsive due to the high volume of notifications that need to be delivered to users.
- Change: Introducing a Notification System to manage and deliver notifications to users more efficiently.
- Benefit: By utilizing a Notification System, the system can deliver notifications to users more quickly and efficiently, reducing latency and improving overall performance. Additionally, users can have a better experience with timely and relevant notifications, which can increase engagement and retention.

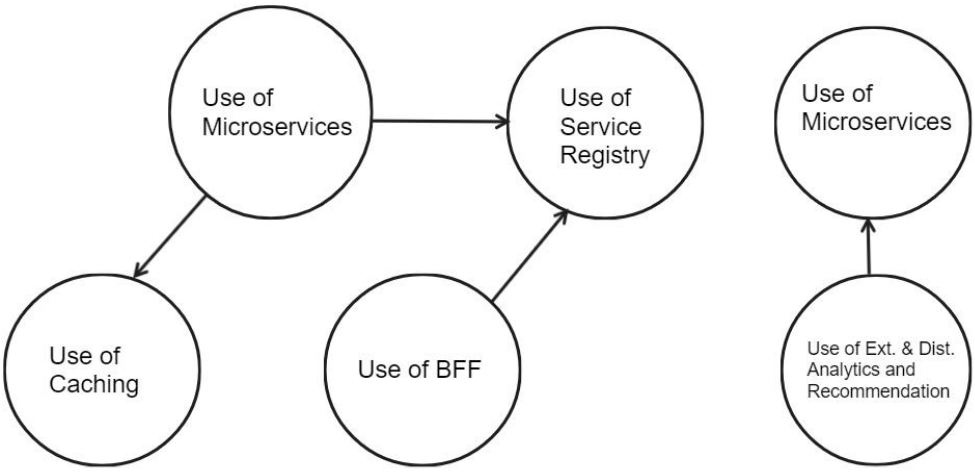


Fig 5: Important Design Decisions and their dependencies

Taking one design decision creates *concerns* and hence requires thinking of another design decision. This creates a dependency *view* for the architect (as shown above) to add it in the software architecture. There can be multiple such *views* and based on the *stakeholders*, the views can change which in turn is called *viewpoints*.

Let's look at *stakeholders*, *concerns*, *view* and *viewpoints* according to the IEEE 42010 standards in the next section.

➢ Views, Stakeholders, Concerns and Viewpoints

In IEEE 42010, views in architectural design refer to different perspectives from which the architecture can be viewed, analysed, and designed. Views provide stakeholders with different representations of the architecture that emphasize different aspects of the system, allowing them to understand and communicate about the system's structure and behaviour

IEEE 42010 identifies four fundamental views of software architecture:

1. *The logical view*: This view focuses on the functional requirements of the system and how they are implemented in terms of components and their interfaces.
2. *The process view*: This view focuses on the concurrency, synchronization, and communication aspects of the system.
3. *The physical view*: This view focuses on the mapping of the software architecture onto the hardware infrastructure.
4. *The development view*: This view focuses on the organization and structure of the software development process.

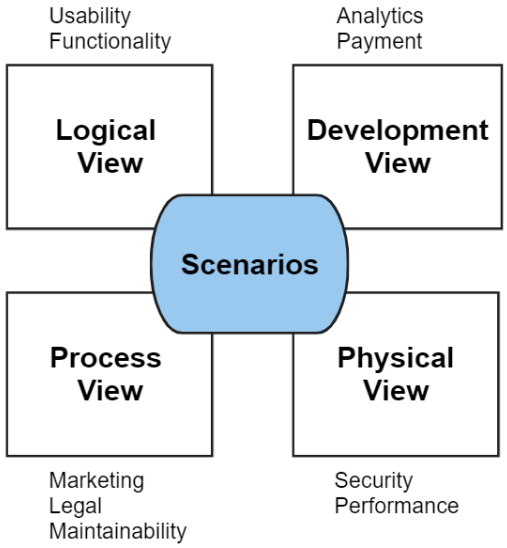


Fig 6: 4 fundamental views of software architecture and concerns

We will follow the IEEE 42010 standard to identify different stakeholders, concerns and corresponding viewpoints and views:

<u>Stakeholders</u>

1. *Customers*: end users who watch movies through the CineStream platform.
2. *Content providers*: companies or individuals who provide movies for the platform.
3. *Administrators*: those responsible for maintaining and managing the platform.
4. *Developers*: those responsible for building and maintaining the platform.
5. *Marketing*: those responsible for promoting and advertising the platform.

6. *Payment gateway providers*: third-party providers responsible for processing payments.
7. *Legal*: those responsible for ensuring compliance with relevant laws and regulations.
8. *Analytics team*: those responsible for monitoring and analysing user behaviour and platform performance.

Concerns and Viewpoints

1. *Functionality*
    - Customer viewpoint: how easy is it to browse and search for movies? Can users easily rent or buy movies?
    - Content provider viewpoint: how easy is it to upload new movies? Can providers easily manage their content?
    - Administrator viewpoint: how easy is it to manage users, payments, and other administrative tasks?
    - Developer viewpoint: how easy is it to modify and extend the platform's functionality? Usability

2. *Usability*
    - Customer viewpoint: how intuitive is the user interface? Can users easily find what they're looking for?
    - Content provider viewpoint: how easy is it to navigate the content management system? Can providers easily add metadata and descriptions to their movies?
    - Administrator viewpoint: how easy is it to manage user accounts and settings?
    - Developer viewpoint: how easy is it to build and modify the user interface?

3. *Security*
    - Customer viewpoint: how secure is the platform? Are customers' personal and payment information protected?
    - Content provider viewpoint: how secure is the content management system? Are only authorized providers able to upload movies?
    - Administrator viewpoint: how secure is the admin interface? Are only authorized users able to access it?
    - Developer viewpoint: how can the platform be designed to prevent and detect security vulnerabilities?

4. *Performance*
    - Customer viewpoint: how quickly do movies load and buffer? Are there any issues with playback or streaming quality?
    - Content provider viewpoint: how quickly are new movies processed and made available on the platform?
    - Administrator viewpoint: how quickly do administrative tasks (such as user management) take to complete?
    - Developer viewpoint: how can the platform be optimized for fast and efficient performance?

5. *Payment*
    - Customer viewpoint: how easy is it to complete payments and transactions? Are there any issues with payment methods or currency conversion?
    - Content provider viewpoint: how are revenue sharing and payment processing handled? Are there any issues with delayed or missing payments?
    - Administrator viewpoint: how easy is it to manage payment gateway providers and payment settings?
    - Developer viewpoint: how can the platform be designed to ensure secure and reliable payment processing?

6. *Analytics*
    - Customer viewpoint: how are recommendations and personalized content suggestions generated? Are there any issues with accuracy or relevancy?
    - Content provider viewpoint: how are movie ratings and popularity measured? Are there any issues with content discovery or exposure?
    - Administrator viewpoint: how are user engagement and platform performance tracked and monitored? Are there any issues with data integrity or completeness?
    - Developer viewpoint: how can the platform be designed to optimize data collection, analysis, and reporting?

7. *Marketing*
    - Customer viewpoint: how attractive and appealing is the platform's design and branding? Are there any features that differentiate the platform from competitors?
    - Content provider viewpoint: how easy is it to promote and market their movies on the platform? Are there any opportunities for cross-promotion or collaboration?
    - Payment gateway provider viewpoint: how easy is it to integrate payment processing with the platform? Are there any issues with payment processing or fraud?
    - Analytics team viewpoint: how can the platform be designed to capture and analyse user engagement and conversion metrics?

8. *Legal*
    - Customer viewpoint: how are user data and privacy protected? Are there any issues with data collection or storage?
    - Content provider viewpoint: how are intellectual property rights and licensing agreements handled? Are there any legal risks associated with content distribution?
    - Administrator viewpoint: how are payment processing and refunds handled? Are there any legal requirements for user or content management?
    - Developer viewpoint: how can the platform be designed to ensure compliance with relevant laws and regulations?

Concerns-Stakeholder Traceability

		Stakeholders							
		Customer	**Content Provider**	**Administrator**	**Developer**	**Marketing**	**Legal**	**Payment Gateway**	**Analytics**
Concerns	**Functionality**	X	X	X	X				
	Usability	X	X	X	X				
	Security	X	X	X	X		X		
	Performance	X	X	X	X				
	Marketing	X	X			X		X	X
	Legal	X	X	X	X		X		
	Payment	X	X	X	X			X	
	Analytics	X	X	X	X				X

Fig 7: Concern vs. Stakeholder Table

Task 3: Architectural Tactics and Patterns

➢ Architectural tactics

Architectural tactics are fundamental design decisions. They are the building blocks for both architectural design and analysis.

Architectural patterns and styles have been proposed as a way to manage the unconstrained nature of the architectural design process but it is still a challenge. Patterns are complex and their interactions with other patterns are not always clear. Furthermore, patterns are always underspecified, and so the designer still needs to add in considerable amounts of detail to reify these into an implementable design.

Hence, we use a more fine-grained approach to architectural design, employing tactics. Tactics are the building blocks of architectures, and hence the building blocks of architectural patterns. We will define sets of tactics that address six quality attributes: *performance*, *usability*, *availability*, *modifiability*, *testability*, and *security*.

Please note: There are a few tactics which can be used to address multiple tactics also.

<u>Performance</u>

1. Caching
   Caching can help improve the performance of CineStream platform in several ways. First, it can reduce the amount of data that needs to be transmitted over the internet by caching frequently accessed content and metadata on servers closer to the user's device. This reduces the latency and improves the speed of content delivery, resulting in a better user experience.

2. Load Balancing
   Multiple users may be accessing different content simultaneously movies, series, etc. resulting in a high volume of traffic. If this traffic is not distributed evenly across multiple servers, it can lead to server overload, which can cause slow performance (too much buffering or low quality) or even crashes.

   Load balancing ensures that the incoming traffic is distributed across multiple servers to prevent any one server from being overloaded. This can be achieved through various load balancing techniques such as round-robin, least connections, setting different servers for different content.

   In addition to improving performance, load balancing also enhances the availability of the CineStream platform. By distributing the traffic across multiple servers, the platform can continue to function even if one or more servers fail.

3. Asynchronous Processing
   Asynchronous processing can be used to execute time-consuming tasks in the background while the user continues to interact with the platform. This frees up resources on the server and allows it to handle other requests while the task is being processed in the background. For example, asynchronous processing can be used to transcode video files or generate thumbnails while the user continues to browse the platform.

   Asynchronous processing can also improve the scalability of out CineStream platform. By processing requests asynchronously, the platform can handle a large number of requests without overloading the server. This is especially important in cases where there are sudden spikes in traffic or when the platform is being used by a large number of users simultaneously.

   Another benefit is that it can help to reduce the response time of the platform. By processing time-consuming tasks in the background, the user does not have to wait for the task to complete before continuing to use the platform.

4. Data Compression and Optimization
   In our platform, large amounts of data need to be transmitted over the internet to the user's device in real-time. This data includes video and audio content streaming which required huge data transmission, as well as metadata such as subtitles and thumbnails. Data Compression and Optimization can be used to reduce the size of this data, making it faster and more cost-effective to transmit.

   Optimization techniques can also be used to improve the performance of CineStream platform. For example, optimizing images and other media assets can reduce the size of these assets, improving page load times and reducing the amount of data that needs to be transmitted over the internet. This helps to reduce the response time and have low latency for the platform.

5. CQRS (Command Query Responsibility Segregation)
   CQRS can be used to separate the read and write operations of the platform into separate components. This separation allows the platform to handle read and write operations independently, which can improve scalability and performance. For example, a high volume of read operations can be handled by read replicas, while write operations can be handled by the master database.

   CQRS can also help to improve the maintainability of an OTT platform. By separating read and write operations, it is easier to modify or update one component without affecting the other. This can help to reduce the risk of errors and improve the overall stability of the platform.

Usability

1. CDN (Content Delivery Network) (Consistency, Responsiveness)
   CDN can be used to cache frequently accessed content in multiple locations, which allows users to access the content from a server that is closer to their location. This reduces the amount of data that needs to be transmitted over long distances, which can significantly improve the speed and reliability of content delivery. It improves the responsiveness of the platform as it can respond to user requests more quickly, resulting in faster user experience.

2. BFF (Accessibility)
   BFF can be used to provide a custom backend for each type of user interface. For example, the backend for a mobile application may be different from the backend for a smart TV application or a web application. By providing a custom backend for each type of user interface, the platform can optimize the user experience for each device and ensure that users have access to the content and features that are relevant to them.

   BFF can also help to improve the scalability of the platform. By providing a custom backend for each type of user interface, the platform can distribute the workload across multiple servers, which can help to improve performance and reduce the risk of downtime.

Availability

1. Redundancy (Replication and Failover)
   Redundancy can be achieved through replication of data and services across multiple servers or data centres. This ensures that if one server or data centre fails, the platform can continue to operate without interruption. Failover mechanisms can also be put in place to automatically redirect user requests to a functioning server or data centre in the event of a failure.

2. Auto-Scaling
   Auto-scaling involves automatically adding or removing resources based on the current demand for the platform. For example, during periods of high traffic, the platform may automatically add additional servers to handle the increased load, and during periods of low traffic, the platform may automatically reduce the number of servers to save on costs. By automatically adding or removing resources based on demand, the platform can ensure that it is only using the resources that are needed, which can help to reduce costs

Modifiability

1. Microservices Architecture (Modularisation, Loose coupling)
   Microservices Architecture involves breaking down the platform into smaller, independent services that can be developed, deployed, and scaled independently.

   The main benefit of Microservices Architecture is that it enables a more modular and flexible system. Each microservice can be developed and deployed independently, which allows for faster development cycles and easier maintenance.
   Another benefit of Microservices Architecture is that it enables loose coupling between the components of the platform. Each microservice communicates with other microservices through APIs, which enables them to be developed and deployed independently. This reduces the risk of system failures, as a problem in one microservice does not necessarily affect the entire system.

2. DDD (Domain Driven Design)
   DDD can help to ensure that the platform's design is aligned with the needs of the users and the business objectives. By modelling the platform around the core business domain, DDD can help to ensure that the platform is designed to meet the needs of the users and provide a seamless user experience.

   DDD can also help to improve the flexibility and scalability of the platform. By modelling the software around the core business domain, DDD can help to ensure that the software is modular and loosely coupled, making it easier to modify and scale as the needs of the business evolve over time.

3. CI/CD
   CI/CD is important because users expect the platform to be available and functioning properly at all times. By using CI/CD, developers can ensure that software updates are thoroughly tested and deployed in a controlled and predictable manner, minimizing the risk of outages or other issues that could negatively affect the user experience.

   Another benefit of architectural tactic CI/CD is that it allows developers to quickly respond to user feedback and make iterative improvements to the platform.

Testability

1. Logging and Monitoring
   Together, Logging and Monitoring provide important insights into the performance and stability of the OTT platform. By tracking user behaviour and monitoring the system in real-time, the team can identify and resolve issues quickly, leading to improved user experience and increased user satisfaction.

Additionally, Logging and Monitoring can help to ensure compliance with regulatory requirements and provide important data for analysis and optimization of the platform.

Security

1. Authentication & Authorisation
   By implementing strong authentication and authorization mechanisms, the CineStream platform can ensure that only authorized users have access to sensitive data and content. It also helps to prevent unauthorized access to user data and content, such as personal information, payment details, or copyrighted material.

2. Encryption
   Encryption is used to secure user data such as login credentials, payment information, and personal information. By encrypting this data, the platform can prevent unauthorized access and ensure that the data remains confidential and secure.

3. Secure Communication Protocols
   Secure Communication Protocols such as HTTPS, SSL, and TLS are used to encrypt user data and protect it from interception or eavesdropping by third parties. This helps to ensure that user data, such as login credentials, payment information, and personal information, remains confidential and secure.

4. Data backup and recovery
   In our CineStream platform, large amounts of data are generated and stored, including user data, content data, and operational data. Data backup and recovery involves regularly backing up the data to a secure location. This ensures that the platform can quickly resume its services without significant downtime or loss of data.

➢ Architectural Patterns

1. Microservices Pattern
2. Broker Pattern
3. Event Bus Pattern

Microservices Pattern: The microservices pattern can be used to break down the CineStream application into smaller, independent services that can be developed, deployed, and scaled independently. Each microservice can be responsible for a specific functionality, such as user management, content delivery, or recommendation engine. By using this pattern, the application can be more resilient, scalable, and maintainable.

Broker Pattern: The broker pattern can be used to decouple the communication between services in CineStream. This can be achieved by using a message broker such as RabbitMQ or Apache Kafka, which can handle the communication between services asynchronously. By using this pattern, services can be developed and deployed independently, without worrying about the availability or state of other services.

Event Bus Pattern: The event bus pattern can be used to implement a pub/sub system in CineStream that can be used to notify different services about events and changes in the application. This can be achieved by using an event bus such as Apache Kafka or AWS Event Bridge, which can handle the distribution of events to different subscribers. By using this pattern, services can be more reactive and responsive to changes in the application.

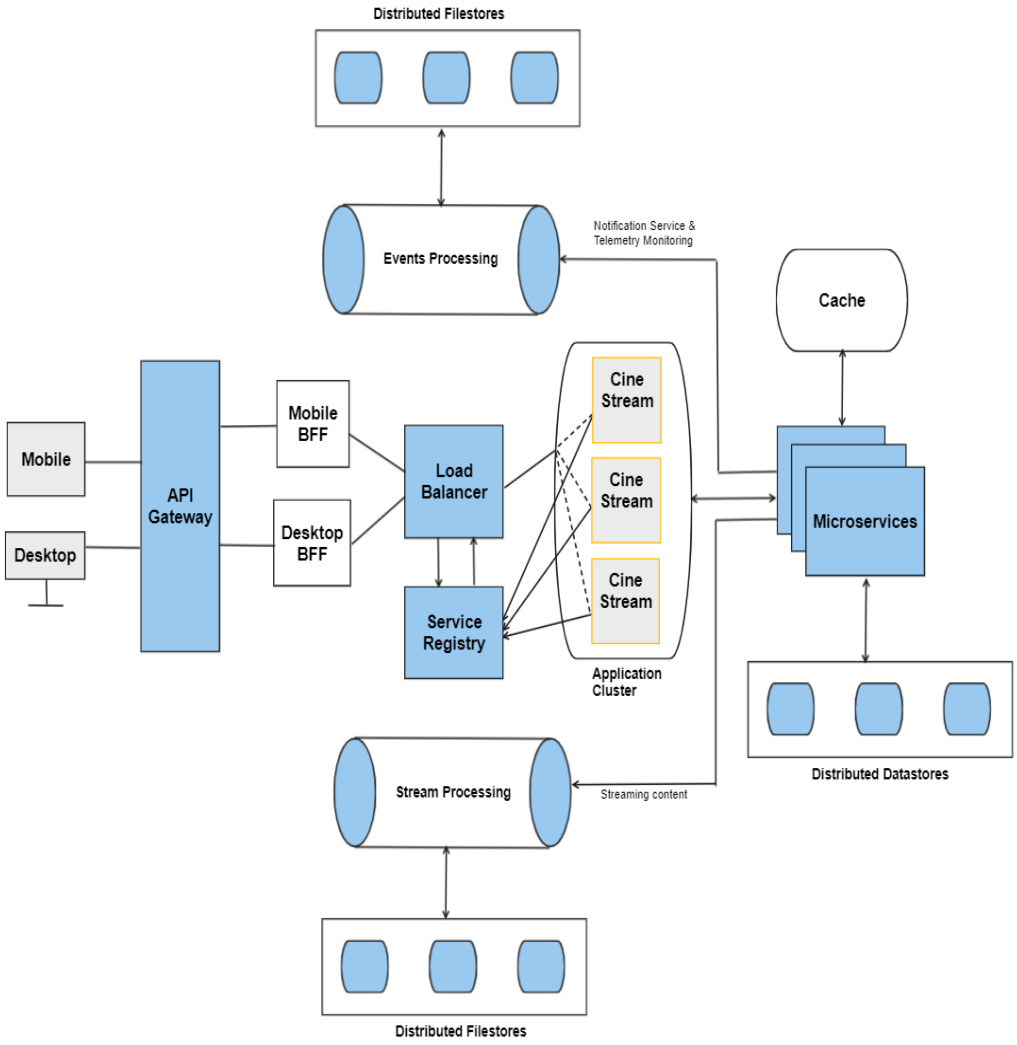


Fig 8: Architecture Diagram using 3 patterns for CineStream

➢ Architecture UML Diagram

1. Component Diagram depicting the major components of CineStream Platform

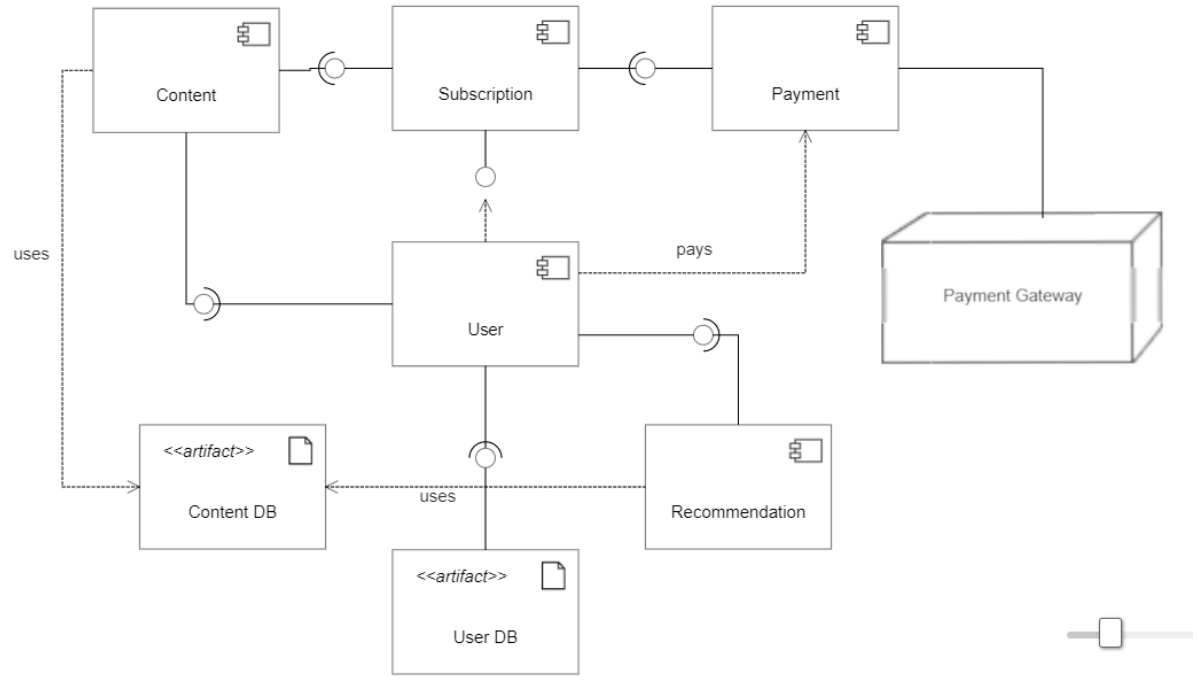


Fig 9: Component Diagram

2. Deployment Diagram depicting the major deployable nodes of CineStream Platform

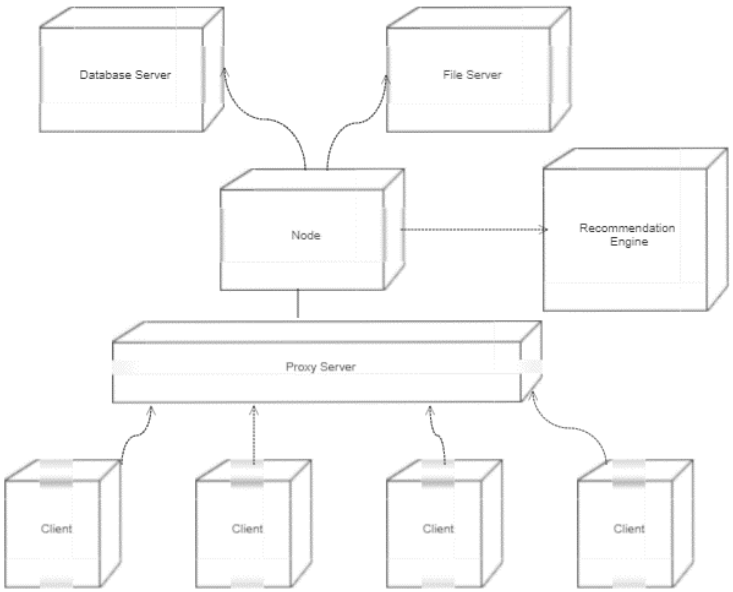


Fig 10: Deployment Diagram

Task 4: Architectural Analysis

➢ What part of system is going to be analysed?

We are going to analyse a part of the system where User Management, Content Management & Subscription Management systems are being used.

Implementation of this part of system is done such that user can login to the system or can create a new account. Hence, login and signup services are available.

User can view all OTT providers and the contents which are provided by them. User (supposed to be Admin) can also add new OTT platforms and contents for other actual users to see and subscribe.

Users can subscribe to the contents provided by these platforms and they can view the contents to which they have subscribed to.

PFB some screenshots on the working of the system implemented: (Will be using these two for all the further NFR measurements)

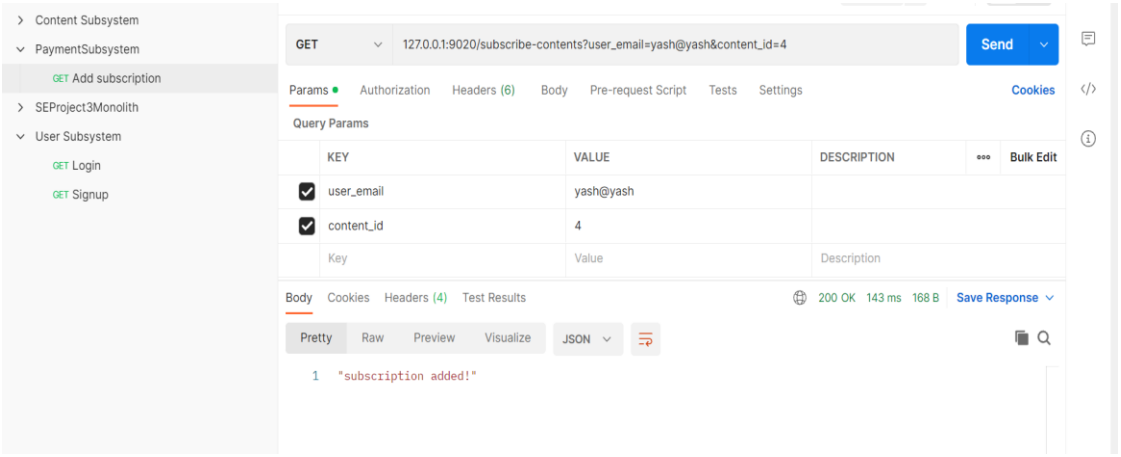


Fig 11: User subscribing to a content

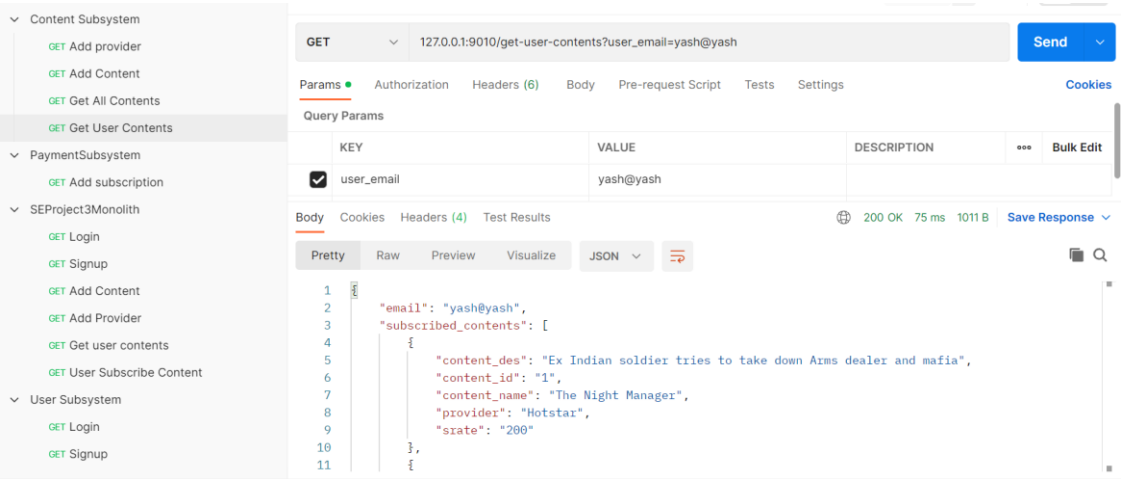


Fig 12: User viewing subscribed contents

➢ What quality attributed will be analysed?

We are going to perform quantitative analysis on the following quality attributes:

1. Latency
2. Resource Usage
3. Throughput
4. Response Time

Quantitative analysis involves the use of mathematical and statistical techniques to analyse data and make predictions. It relies on objective, measurable data to draw conclusions and make informed decisions.

➢ Mention the pattern being used for comparison and why?

Monolithic layered pattern and microservices pattern are two different architectural approaches, each with its own strengths and weaknesses. Therefore, the choice between the two depends on the specific needs and requirements of the CineStream platform, especially the non-functional requirements. Both can be used to implement CineStream, but they differ in their approach of handling the various components of the platform.

For a monolithic layered pattern, the application is structured as a single, self-contained unit that contains all the modules and components required to run the platform. Each layer in the application is responsible for a specific set of tasks, and the layers are tightly coupled to one another.

On the other hand, a microservices architecture is based on the concept of breaking down the application into smaller, independent services, each responsible for a specific function. This approach allows for greater flexibility and scalability, as each service can be developed and deployed independently of the others

To implement CineStream using a monolithic layered pattern, you could organize the application into layers based on its different functions, such as user management, content management, and streaming. Each layer would have its own set of modules and components, and the layers would communicate with each other through well-defined interfaces.

To implement CineStream using a microservices architecture, you could break down the platform into a set of smaller, independent services. For example, you could have one service for user management, another for content management, and a third for streaming. Each service would have its own set of APIs and interfaces, and they would communicate with each other using standardized protocols such as REST.

➢ Results of Analysis and Inference

1. Resource Usage

We have analysed resource usage by running multiple instances of Monolithic and Microservices versions of system. CPU% utilization and Memory% utilization was look upon while analysing this non-functional requirement.

Windows Task Manager was used to capture the utilization %. After running each instance, the values were captured and plotted on the graph.

The below graphs show plot of no. of instances (X-axis) and CPU% utilization (Y-axis) & Memory% utilization (Y-axis). We have three services in the microservices implementation. When we say 'x' instances, that means 'x' instances of monolithic system and 'x' instances of 3 services each.

CPU% utilization measures the percentage of time that the CPU is busy executing instructions, while Memory% utilization measures the percentage of physical memory currently being used by the system.

When comparing 10 instances of a monolithic system with 10 instances of 3 microservices each, the CPU% utilization is more than twice as high for the monolithic system, and the memory % utilization is also lower for microservices. This is because all the services in a monolithic system are tightly coupled and run on the same instance, making any changes to one service affect all the other services. On the other hand, microservices can be independently auto-scaled, which is not possible in a monolithic system. Hence, in general, CPU% utilization and Memory% utilization will be higher for a monolithic system compared to microservices.

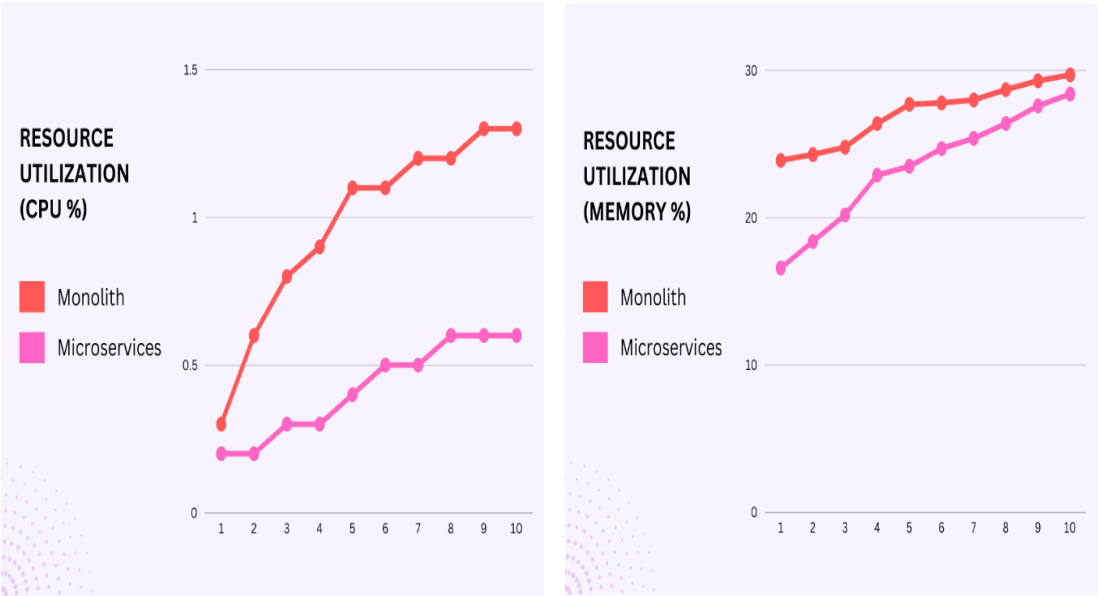
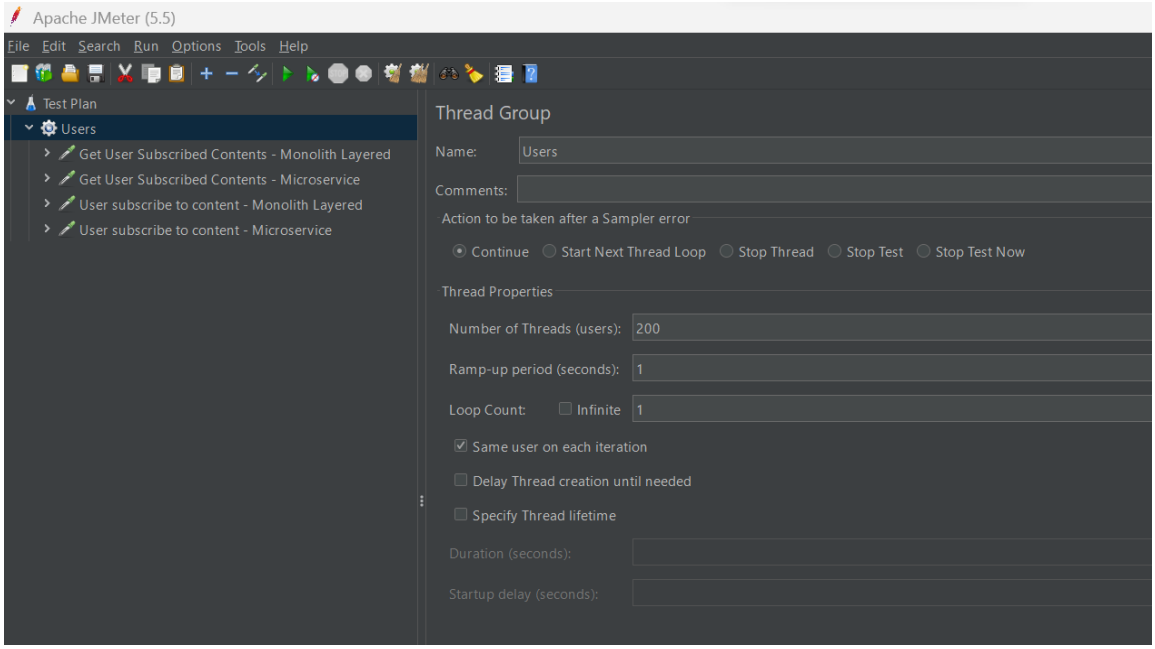


Fig 13 & 14: Resource Utilisation Graphs

For the next three performance metrics (Latency, Throughput, Response Time), we have analysed by sending multiple requests simultaneously to the monolithic and microservices system to do load and performance test and hence calculating these metrics and comparing them. (Metrics here are analogous to NFRs)

We have used *Apache Jmeter* to send these requests simultaneously and perform testing.



We analysed two functions: *Get User Subscribed Contents* and *User Subscribe to Content*. We sent 200 requests simultaneously to these functions in microservices and monolithic layered system to analyse these for each API and type of system. Both functions in microservices system have a call chain of 3 i.e., they interact with other services via REST APIs. All calls in monolith are local.

2. Latency

Latency is the duration that a request is waiting to be handled – during which it is latent, awaiting service. Latency measurements are used for diagnostic purposes.

Following are the results for Get User Subscribed Contents (Monolithic and Microservices):

Sam...	Start Time	...	Label	Sample Ti...	Status	Bytes	Sent By...	Latency	Connect Time(ms)
187	00:12:38.204	...	Get User Subscribed Contents - Monolith...	2522		1250	157	2522	1
188	00:12:38.209	...	Get User Subscribed Contents - Monolith...	2527		1250	157	2527	0
189	00:12:38.219	...	Get User Subscribed Contents - Monolith...	2544		1250	157	2543	1
190	00:12:38.224	...	Get User Subscribed Contents - Monolith...	2567		1250	157	2567	1
191	00:12:38.229	...	Get User Subscribed Contents - Monolith...	2589		1250	157	2589	0
192	00:12:38.234	...	Get User Subscribed Contents - Monolith...	2588		1250	157	2588	1
193	00:12:38.239	...	Get User Subscribed Contents - Monolith...	2602		1250	157	2602	1
194	00:12:38.243	...	Get User Subscribed Contents - Monolith...	2603		1250	157	2603	1

Sam...	Start Time	...	Label	Sample Time...	Status	Bytes	Sent Byt...	Latency	Connect Time(ms)
187	00:12:40.726	...	Get User Subscribed Contents - Microservice	4645		1256	157	4645	1
188	00:12:40.735	...	Get User Subscribed Contents - Microservice	4662		1256	157	4662	1
189	00:12:40.818	...	Get User Subscribed Contents - Microservice	4619		1256	157	4619	0
190	00:12:40.791	...	Get User Subscribed Contents - Microservice	4684		1256	157	4684	0
191	00:12:40.841	...	Get User Subscribed Contents - Microservice	4643		1256	157	4642	0
192	00:12:40.821	...	Get User Subscribed Contents - Microservice	4674		1256	157	4674	1
193	00:12:40.762	...	Get User Subscribed Contents - Microservice	4753		1256	157	4753	1
194	00:12:40.857	...	Get User Subscribed Contents - Microservice	4693		1256	157	4692	1

We can clearly see that the latency for function in microservices system is higher as compared to the monolithic layered system.

A monolith has no network latency, as all calls are local. Even in a perfectly parallelizable world, the monoliths will still be faster. The way to fix this issue is reducing call chain length, using fan-out and keeping data as local as possible.
Similarly for the other function:

Sam...	Start Time	...	Label	Sample Time...	Status	Bytes	Sent Bytes	Latency	Connect Time(ms)
187	00:12:45.372	..	User subscribe to content - Mono...	2353		187	171	2352	0
188	00:12:45.397	..	User subscribe to content - Mono...	2353		187	171	2353	1
189	00:12:45.437	..	User subscribe to content - Mono...	2337		187	171	2337	1
190	00:12:45.475	..	User subscribe to content - Mono...	2332		187	171	2332	0
191	00:12:45.484	..	User subscribe to content - Mono...	2342		187	171	2342	0
192	00:12:45.515	..	User subscribe to content - Mono...	2346		187	171	2346	0
193	00:12:45.550	..	User subscribe to content - Mono...	2356		187	171	2356	0
194	00:12:45.495	..	User subscribe to content - Mono...	2454		187	171	2454	1

Sa...	Start Time	...	Label	Sample Time...	Status	Bytes	Sent Byt...	Latency	Connect Time(ms)
187	00:12:47....	..	User subscribe to content - Mic...	9820		181	171	9820	1
188	00:12:47....	..	User subscribe to content - Mic...	9613		181	171	9612	1
189	00:12:47....	..	User subscribe to content - Mic...	9700		181	171	9700	1
190	00:12:47....	..	User subscribe to content - Mic...	9583		181	171	9583	1
191	00:12:47....	..	User subscribe to content - Mic...	9766		181	171	9766	1
192	00:12:47....	..	User subscribe to content - Mic...	9660		181	171	9660	1
193	00:12:47....	..	User subscribe to content - Mic...	9646		181	171	9646	1
194	00:12:47....	..	User subscribe to content - Mic...	9624		181	171	9624	1

3. Throughput

Throughput indicates the number of transactions per second an application can handle, the number of transactions produced over time during a test.
Below are the throughputs of both functions for each system (monolith & microservice):

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
Get User Subscribed Contents - Monolith ...	200	1336	61	2672	778.27	0.00%	54.5/sec	66.58	8.36	1250.0
TOTAL	200	1336	61	2672	778.27	0.00%	54.5/sec	66.58	8.36	1250.0

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/s...	Sent KB/sec	Avg. Bytes
Get User Subscribed Contents - Microservice	200	3911	302	4825	1198.98	0.00%	24.1/sec	29.53	3.69	1256.0
TOTAL	200	3911	302	4825	1198.98	0.00%	24.1/sec	29.53	3.69	1256.0

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
User subscribe to content - Monolith Layered	200	2235	985	2695	212.53	0.00%	19.2/sec	3.51	3.21	187.0
TOTAL	200	2235	985	2695	212.53	0.00%	19.2/sec	3.51	3.21	187.0

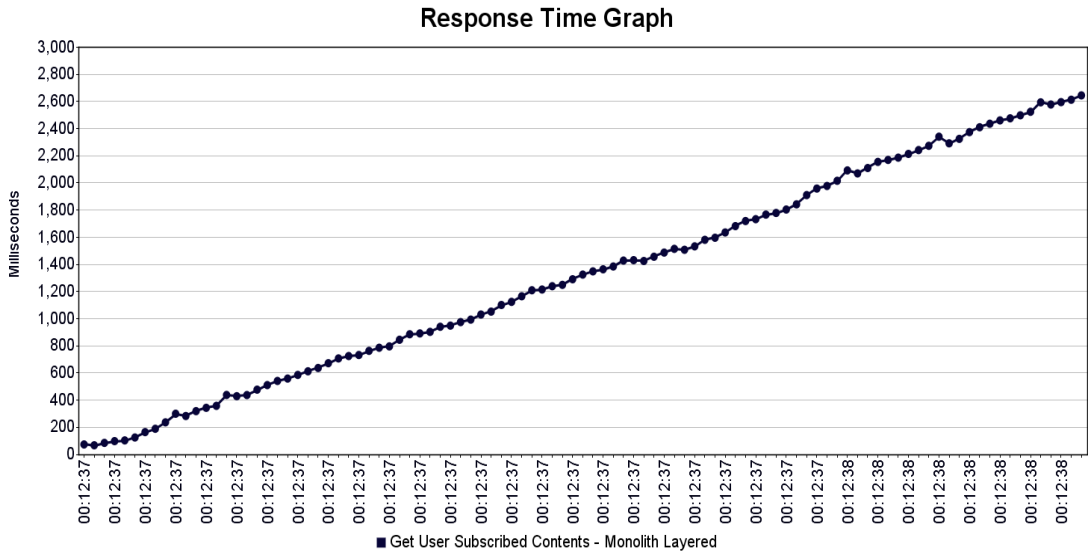
Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
User subscribe to content - Microservice	200	10061	9417	11055	401.15	0.00%	10.5/sec	1.86	1.76	181.0
TOTAL	200	10061	9417	11055	401.15	0.00%	10.5/sec	1.86	1.76	181.0

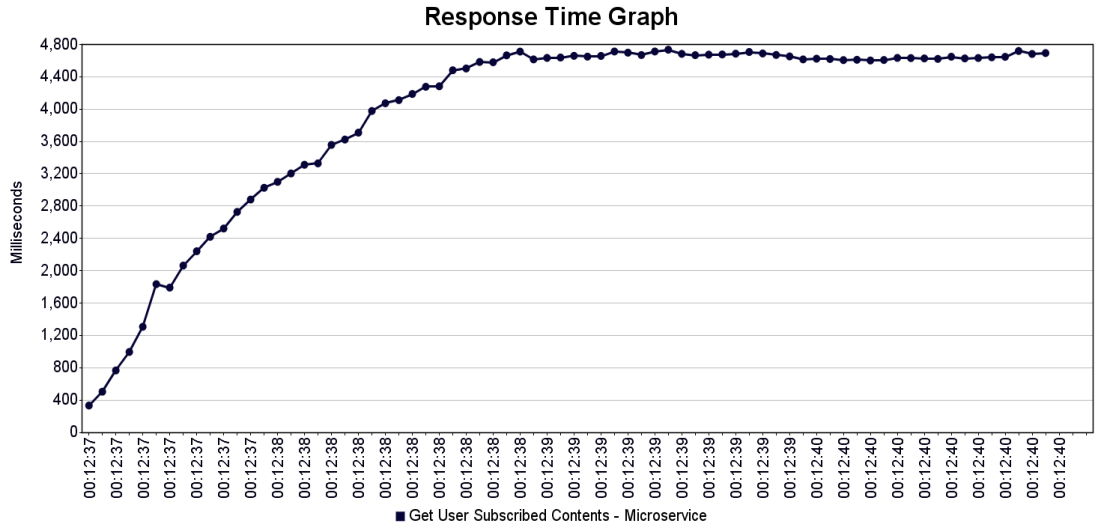
We can see that throughput for Microservices is less than monolith. This is because In workloads that cannot run concurrently across the network, monoliths may deliver better performance. Data needs to sent between services and also all the infrastructure induces a certain overhead. If the workload cannot be scaled to multiple instances, a monolith can deliver a higher throughput.

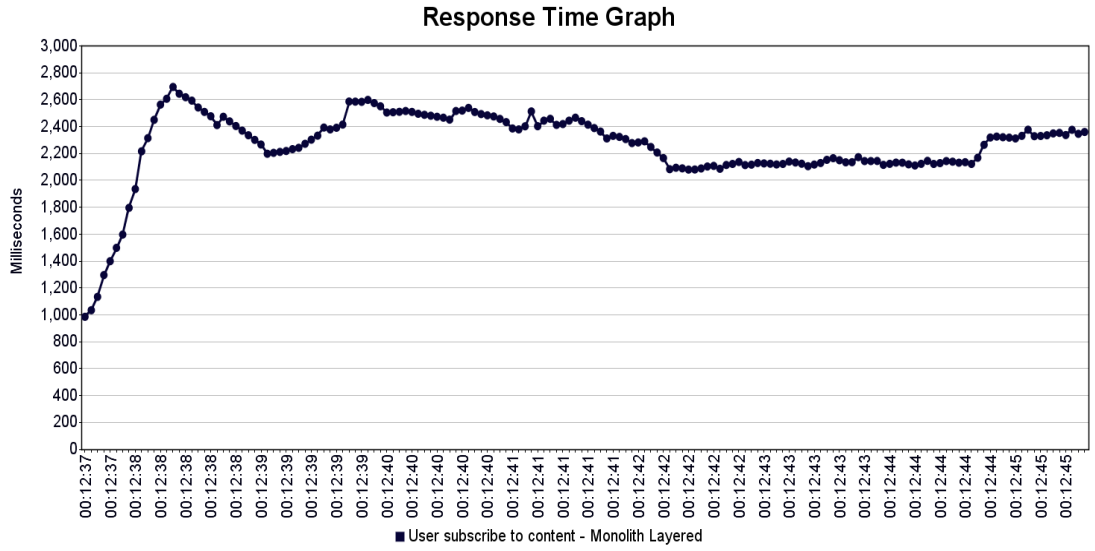
4. Response Time

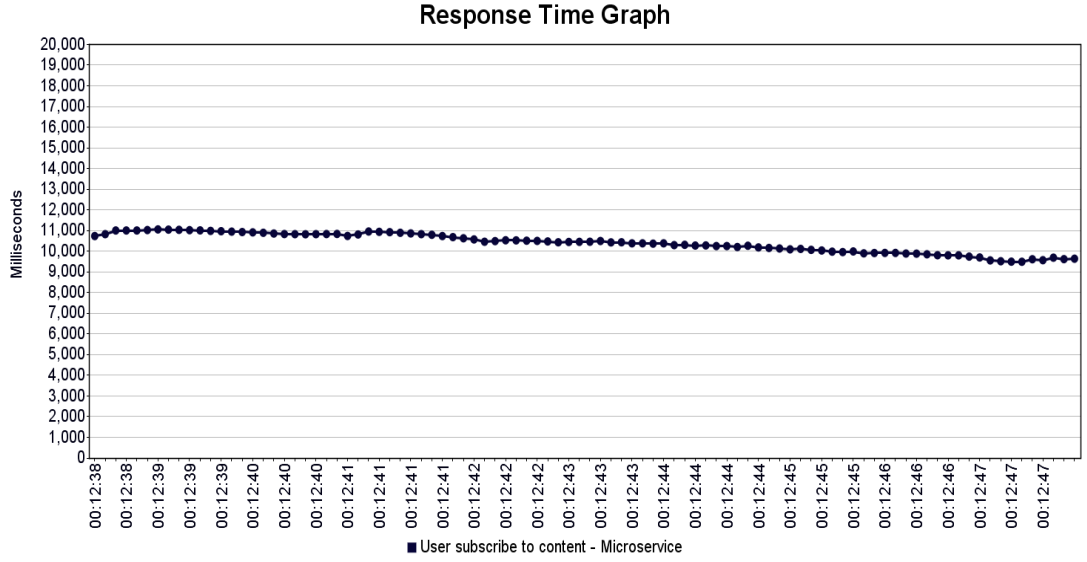
Response time is the amount of time it takes for a service to process a request and send a response back to the client. The response time period begins when the client initiates a request to the service and ends when the client receives a response from the service.

Following graphs shows the response time as the number of simultaneous requests to each service in each system increases:




Response Time Graph
Milliseconds
Get User Subscribed Contents - Microservice


Response Time Graph
Milliseconds
User subscribe to content - Monolith Layered


Response Time Graph
Milliseconds
User subscribe to content - Microservice

The Response time is lower for microservices as compared to monolith. This is because, all the calls in the monolith are local calls with no network latency which is not the case with microservices. Also, as all the data is stored on separate DBs, generating response and sending it back to client takes more time in microservices.

➢ Trade-offs between the two patterns used

Monoliths are simpler, have lower latency, response time and higher throughput is most of the scenarios.

Microservices, on the other hand, are more reliable, have less resources utilization and are easier to scale.

If we consider the heuristic analysis of some NFRs like time to market or accepted complexity in development, microservices perform better as compared to monoliths as they are easier to develop by breaking into multiple tasks which can be done multiple teams independently. Thus, the complexity reduces and the time to market also reduces as multiple teams are working concurrently.

These are the trade-offs between microservice and monolith but our choice will be **Microservices** for developing this application