RESEARCH DOCUMENT

Individual Project

Abstract

A research document answering the research questions for the MyWatchList project.

Niels Roefs

Contents

Problem Definition

MyWatchList should be able to handle a million concurrent users. There are various aspects that need to be addressed to allow this to happen. Some of these aspects are scalability and performance optimization. In this research plan, I want to learn more about how to make MyWatchList scalable and how to keep high performance.

Main Question

How can MyWatchList be effectively prepared for deployment while considering diverse factors such as scalability, and performance optimization?

Sub Questions & Methods

Scalability:

a. What are the key principles and strategies for designing a scalable architecture and how can MyWatchList benefit from these?
   Design pattern research (Library)
   IT architecture sketching (Workshop)

b. What are the best practices for load balancing and auto-scaling configurations?
   Best good and bad practices (Library)
   Non-functional Test (Lab)

c. How can cloud-based infrastructure contribute to the scalability of MyWatchList?
   Available product analysis (Library)
   Multi-criteria decision making (Workshop)

Performance:

a. What are the common performance bottlenecks in web applications and how can they be mitigated in the MyWatchList application?
   Community research (Library)
   Problem analysis (Field)

b. How can caching mechanisms and CDNs enhance MyWatchList's performance?
   Component testing (Lab)
   Literature study (Library)
   Prototyping (Workshop)

Results

What are the key principles and strategies for designing a scalable architecture and how can MyWatchList benefit from these?

Design Pattern Research (Library)

After doing research, I found the following key principles and strategies:

1. **Microservices Architecture**:
    - **Principle**: This involves breaking down the application into smaller, loosely coupled services that can be developed, deployed, and scaled independently. Each service encapsulates a specific business function and communicates with other services through well-defined APIs. (Lewis & Fowler, 2014)
    - **Benefits**:
        - **Independent Scaling**: Each microservice can be scaled independently according to its load and resource requirements. This is particularly useful for services that experience variable loads.
        - **Fault Isolation**: Failures in one microservice do not directly impact others, enhancing the overall resilience of the application.
        - **Technology Diversity**: Different microservices can be developed using different programming languages and technologies, allowing the best tool for each job.
    - **Cons**:
        - **Increased Complexity**: Managing a large number of microservices can be complex, requiring robust orchestration and monitoring tools.
        - **Inter-Service Communication Overhead**: Communication between microservices typically occurs over the network, which can introduce latency and increase the risk of partial failures.
2. **Horizontal Scaling**:
    - **Principle**: Horizontal scaling involves adding more instances of a service to handle increased load, as opposed to vertical scaling, which involves adding more resources (CPU, memory) to a single instance. (Digital Ocean, n.d.)
    - **Benefits**:
        - **Cost-Effectiveness**: Commodity hardware can be used to scale out, which is often more cost-effective than scaling up.
        - **Flexibility**: Easily accommodates growth by adding more instances as needed.
        - **Fault Tolerance**: Distributing the load across multiple instances reduces the impact of a single instance failure.
    - **Cons**:
        - **Load Balancing Required**: Effective load balancing strategies are essential to distribute traffic evenly across instances.
        - **Data Consistency Challenges**: Maintaining data consistency across multiple instances can be challenging, particularly for stateful applications.

3. **Asynchronous Communication**:

- **Principle**: Utilizing asynchronous communication mechanisms, such as message queues, to decouple services and improve performance. This allows services to communicate without waiting for each other to process requests. (System Design by CHK, 2023)
- **Benefits**:
  - **Improved Performance**: Reduces latency and improves throughput by allowing services to continue processing other tasks while waiting for responses.
  - **Decoupling**: Services are loosely coupled, making it easier to update or replace them without affecting the entire system.
  - **Scalability**: Asynchronous messaging systems can handle a large number of messages, supporting high loads and bursty traffic patterns.
- **Cons**:
  - **Complex Error Handling**: Managing failures and retries in an asynchronous system can be more complex compared to synchronous systems.
  - **Message Ordering**: Ensuring the correct order of messages can be challenging in a distributed system.

4. **Load Balancing**:
   - **Principle**: Distributing incoming network traffic across multiple servers to ensure no single server becomes a bottleneck. Load balancers can use various algorithms such as round-robin, least connections, or IP hash to distribute the load. (Newman, 2024)
   - **Benefits**:
     - **Enhanced Availability**: Improves system reliability and uptime by ensuring traffic is spread evenly across available servers.
     - **Scalability**: Makes it easy to add or remove servers based on demand without affecting the end-user experience.
     - **Performance Optimization**: Prevents any single server from being overwhelmed, thereby maintaining optimal performance.
   - **Cons**:
     - **Single Point of Failure**: The load balancer itself can become a single point of failure if not properly managed or duplicated.
     - **Configuration Complexity**: Configuring load balancers to handle different types of traffic efficiently can be complex.
5. **Database Sharding**:
   - **Principle**: Partitioning the database into smaller, more manageable pieces (shards), which can be spread across multiple database servers. Each shard contains a subset of the data, and queries are routed to the appropriate shard. (Awati & Denman, n.d.)
   - **Benefits**:
     - **Improved Performance**: Distributes the load across multiple servers, reducing the query load on any single server and improving response times.
     - **Enhanced Scalability**: Allows the database to scale horizontally by adding more shards as data volume grows.
     - **Fault Isolation**: Issues in one shard do not affect the entire database, enhancing overall system resilience.
   - **Cons**:
     - **Complex Query Logic**: Queries that span multiple shards can be complex to implement and may require additional coordination.

- **Operational Overhead**: Managing multiple database instances and ensuring data consistency can increase operational complexity.

IT architecture sketching (Workshop)

See architecture diagrams in the Technical Design Document under ‘Scalable Architectures’.

Conclusion

Based on these findings, MyWatchList adopts a microservices architecture to leverage the benefits of independent scaling, and fault isolation. Horizontal scaling is implemented in the Kubernetes cluster to manage increased loads efficiently. Asynchronous communication using RabbitMQ enhances performance and scalability, supporting load balancing as well. Ocelot .NET serves as the API gateway, providing load balancing capabilities. While database sharding is not currently implemented, Azure CosmosDB PostgreSQL supports sharding when it becomes necessary, ensuring future scalability and performance optimization.

What are the best practices for load balancing and auto-scaling configurations?

Best good and bad practices (Library)

Load Balancing Best Practices:

- Implementing a load balancer is essential for distributing incoming network traffic across multiple servers to ensure no single server becomes a bottleneck. It enhances the availability and reliability of the application by balancing the load efficiently.
- Different algorithms such as round-robin, least connections, and IP hash can be used depending on the application's needs. Each algorithm has its strengths and is suitable for different scenarios. (Cloudflare, n.d.)

Load Balancing Bad Practices (Santos, n.d.):

- Overlooking or underestimating communication overhead and synchronization issues.
- Using a load balancing algorithm that is too complex, too simple, or too rigid for the workload and system characteristics.
- Applying a load balancing algorithm that is incompatible or inconsistent with the parallel application logic or data dependencies.
- Ignoring or neglecting performance variability and reliability issues that can affect dynamic load balancing outcomes.
- Relying on inaccurate or outdated workload information or feedback to make load balancing decisions.

Auto-Scaling Best Practices:

- Defining clear auto-scaling policies that decide when and how to scale resources based on metrics such as CPU utilization and memory usage. For MyWatchList, Horizontal Pod Autoscaler (HPA) is used in the Kubernetes cluster, with scaling triggered when CPU usage reaches 50% or memory usage reaches 70%. (Besedin, n.d.)
- Continuous monitoring of the auto-scaling configuration makes sure that it remains aligned with the current usage patterns. Adjustments may be needed as MyWatchList grows. For monitoring, MyWatchList uses Prometheus to track metrics and ensure the auto-scaling policies are effective. (Besedin, n.d.)

Auto-Scaling Bad Practices (Adamson, 2024):

- HPA and VPA may make conflicting scaling decisions if configured with different metrics or thresholds. For instance, HPA might scale out due to high CPU usage while VPA simultaneously tries to scale down pod resources, resulting in resource inefficiencies or performance issues.
- VPA can override manually set resource limits for pods, which can lead to violations of namespace policies or resource quotas. This can cause instability or even pod termination if security policies are enforced.
- Setting narrow upper and lower thresholds for scaling can cause frequent and unnecessary scaling actions, leading to instability and performance degradation. It's essential to maintain a buffer between thresholds to avoid rapid scaling cycles.

- Autoscalers depend on timely and accurate metrics for effective scaling decisions. Missing or delayed metrics, such as CPU or memory usage, can lead to incorrect scaling actions, impacting application performance and resource utilization.
- Using default parameters without tuning for specific workload characteristics can result in suboptimal scaling behavior. Parameters such as scaling thresholds, cooldown periods, and polling intervals should be adjusted based on workload requirements and performance metrics.

Conclusion

In terms of best practices for load testing and autoscaling configurations, MyWatchList uses Ocelot .NET as an API gateway, which also functions as a load balancer, and a Kubernetes cluster with Horizontal Pod Autoscaler (HPA) to manage auto-scaling based on CPU and memory usage. Prometheus is used for monitoring, making sure there is effective tracking and optimization of performance metrics.

How can cloud-based infrastructure contribute to the scalability of MyWatchList?

Multi-criteria decision making (Workshop)

See cloud research under the 'Cloud Native'.

Available Product Analysis (Library)

Cloud-based infrastructure provides various tools and services that can significantly enhance the scalability of applications like MyWatchList. By leveraging the capabilities of leading cloud providers, MyWatchList can efficiently scale to handle a million concurrent users. This section analyzes key cloud-based products and services and how they contribute to scalability.

1. Compute Resources:

a. Virtual Machines (VMs) and Containers:

1. **Azure Virtual Machines**:
    - **Description**: Azure Virtual Machines provide scalable, on-demand computing resources.
    - **Benefits**:
        - **Scalability**: Easily scale up or down based on demand.
        - **Flexibility**: Customize VM configurations to meet specific needs.
        - **Cons**:
            - **Cost**: Pay-as-you-go model can be expensive if not managed properly.
            - **Management Overhead**: Requires ongoing management and maintenance.
2. **Azure Kubernetes Service (AKS)**:
    - **Description**: AKS simplifies the deployment, management, and operations of Kubernetes.
    - **Benefits**:
        - **Automatic Scaling**: Supports Horizontal Pod Autoscaler (HPA) for automatic scaling based on CPU and memory usage.
        - **Orchestration**: Efficiently manages containerized applications.
        - **Cons**:
            - **Complexity**: Requires knowledge of Kubernetes.
            - **Resource Overhead**: Kubernetes itself consumes resources.

2. Load Balancing:

a. Azure Load Balancer:

1. **Azure Load Balancer**:
    - **Description**: Distributes incoming network traffic across multiple VM instances.
    - **Benefits**:
        - **High Availability**: Ensures application remains available by distributing traffic.
        - **Scalability**: Automatically adjusts to handle increased load.
        - **Cons**:
            - **Configuration Complexity**: Requires proper configuration to optimize performance.
            - **Latency**: Additional hop may introduce slight latency.

2. **Ocelot API Gateway**:
    - **Description**: An open-source .NET API gateway that provides a unified entry point to back-end services.
    - **Benefits**:
        - **Load Balancing**: Can act as a load balancer for incoming API requests.
        - **Routing**: Simplifies routing and service discovery.
        - **Cons**:
            - **Performance Overhead**: Can introduce latency if not optimized.
            - **Complexity**: Requires careful setup and management.

3. Storage Solutions:

a. Azure Cosmos DB:

1. **Azure Cosmos DB**:
    - **Description**: A fully managed NoSQL database service that offers multiple data models.
    - **Benefits**:
        - **Global Distribution**: Automatically distributes data globally.
        - **Automatic Scaling**: Scales throughput and storage on demand.
        - **Cons**:
            - **Cost**: Can be expensive, especially with global distribution.
            - **Complexity**: Requires understanding of partitioning and consistency models.

4. Monitoring and Alerts:

a. Prometheus:

1. **Prometheus**:
    - **Description**: Prometheus is an open-source monitoring system with a dimensional data model and flexible query language.
    - **Benefits**:
        - **Comprehensive Monitoring**: Tracks various performance metrics across the infrastructure.
        - **Alerting**: Provides alerting capabilities to proactively manage performance issues.
        - **Cons**:
            - **Complex Setup**: Requires configuration and integration.
            - **Resource Usage**: Consumes resources for monitoring and alerting.

5. Additional Considerations: Caching and Content Delivery:

a. Azure Cache for Redis:

1. **Azure Cache for Redis**:
    - **Description**: A fully managed Redis cache service.
    - **Benefits**:
        - **Performance**: Improves application performance by reducing database load.
        - **Scalability**: Scales cache size and throughput as needed.
        - **Cons**:
            - **Cost**: Additional service cost.
            - **Management**: Requires configuration and management.
    - **Future Considerations:** If time allows, I want to look more into Azure Cache for Redis to further optimize database performance.

b. Azure Content Delivery Network (CDN):

1. **Azure CDN**:
    - **Description**: A global CDN solution for delivering high-bandwidth content.
    - **Benefits**:
        - **Low Latency**: Delivers content from the closest edge server.
        - **Scalability**: Automatically scales to handle traffic spikes.
        - **Cons**:
            - **Cost**: Can incur high costs with extensive usage.
            - **Complexity**: Requires proper setup and integration.
    - **Future Considerations:** I want to explore Azure CDN for efficient content delivery and improved user experience across different geographical locations.

Conclusion

Considering MyWatchList is built using C#, Azure's ecosystem is particularly beneficial due to its deep integration with Microsoft tools and technologies. The choice of Azure Kubernetes Service (AKS) for managing containerized applications, and Azure Cosmos DB for PostgreSQL for scalable database needs align well with the development stack. Azure's support for Kubernetes through AKS simplifies deployment and scaling. Azure Key Vault ensures secure key management for .NET applications, and Azure Monitor offers comprehensive monitoring and proactive performance management.

What are the common performance bottlenecks in web applications and how can they be mitigated in the MyWatchList application?

Community Research (Library)

a. Server-Side Bottlenecks:

Server performance depends on how quickly and reliably your web application's logic runs on the server. Factors like CPU, memory, disk, and network can affect server performance. To improve it, you can use load balancing to spread traffic across multiple servers, concurrency to handle multiple requests at the same time, and caching to store data on the server. Tools like Nginx, Apache, HAProxy, threads, processes, async/await, Redis, Memcached, or MongoDB can help improve server reliability, speed, and efficiency. (How do you address performance bottlenecks in web development frameworks?, n.d.)

Database performance is about how fast and reliably your web application can store, retrieve, or change data. This can be affected by factors like schema design, query optimization, indexing, or replication. To improve performance, you can use normalization to organize data into logical tables, denormalization to reduce the number of joins needed to retrieve data, and indexing to create pointers to data for faster searches. Tools like ER diagrams, foreign keys, constraints, views, materialized views, primary keys, unique keys, or composite keys can help with these tasks. Additionally, caching frequently accessed data can further enhance performance. (How do you address performance bottlenecks in web development frameworks?, n.d.)

The following seem to be the most important:

1. **Increased CPU Usage**:
    - **Description**: High CPU usage can slow down request processing.
    - **Mitigation**: Use efficient algorithms, optimize code, and implement asynchronous processing.
    - **Solution**: Synchronous code was refactored to be asynchronous using async/await patterns.
2. **Excessive Database Queries**:
    - **Description**: Inefficient database queries can delay responses.
    - **Mitigation**: Optimize queries, use indexing, and implement caching.
3. **High Network Latency**:
    - **Description**: Slow data transmission can impact responsiveness.
    - **Mitigation**: Use CDNs, optimize data payloads, and ensure efficient routing.
    - **Solution**: To be looked into.
4. **High Server Response Times**:
    - **Description**: Slow server responses can degrade user experience.
    - **Mitigation**: Optimize server-side code, use load balancing, and implement caching.

- **Solution**: Deployed Ocelot .NET as an API gateway, which also acts as a load balancer to distribute traffic effectively.

b. Client-Side Bottlenecks:

1. **Slow Page Load Times**:
    - **Description**: Large assets like images, scripts, and stylesheets slow down page loading.
    - **Mitigation**: Optimize images, minify CSS and JavaScript, and implement lazy loading.
2. **Inefficient JavaScript Execution**:
    - **Description**: Long-running JavaScript can block the main thread.
    - **Mitigation**: Break down long tasks, use web workers, and optimize JavaScript execution.
3. **Render-Blocking Resources**:
    - **Description**: Blocking resources delay page rendering.
    - **Mitigation**: Load JavaScript asynchronously, defer non-critical CSS, and inline critical CSS.

(Sharma, 2024)

Problem Analysis (Field)

Through analysis, some bottlenecks were found in MyWatchList.

Bottlenecks

a. Server-Side Bottlenecks:

1. **Synchronous Code Execution**:
    - **Description**: Synchronous operations blocked the main thread, leading to slow response times.
    - **Mitigation**: Refactored code to use asynchronous programming. Asynchronous patterns (async/await) were implemented to handle I/O-bound operations without blocking the main thread.
    - **Benefits**:
        - **Improved Responsiveness**: Non-blocking I/O operations allow the application to handle more concurrent requests.
        - **Better Resource Utilization**: Efficient use of CPU and memory resources.
    - **Cons**:
        - **Complexity**: Asynchronous code is more complex to implement and debug.

- (Microsoft, 2023)

2. **Database Query Performance**:

    - **Description**: Some queries were slow due to lack of indexing and inefficient design.
    - **Mitigation**: Added indexes, optimized query logic, and implemented Redis caching to reduce database reads.

Performance Improvement Measures:

a. Database Optimization:

- I haven't implemented anything to optimize this. Ways to optimize database queries would be to add indexes and implement a caching system, possibly 3rd party like Redis.

b. Monitoring and Alerts:

- **Implementation**: Deployed Prometheus for monitoring key performance.

Conclusion

Common performance bottlenecks in web applications, such as increased CPU usage, inefficient database queries, high network latency, slow page load times, and synchronous code execution, can significantly impact user experience and scalability. For MyWatchList, these bottlenecks were identified through research, and mitigated using server-side optimizations. The most strategies included refactoring synchronous code to asynchronous, using RabbitMQ for asynchronous processing, and deploying monitoring with Prometheus. These measures have enhanced MyWatchList's performance and scalability, ensuring it can handle a lot of users effectively.

How can caching mechanisms and CDNs enhance MyWatchList's performance?

No time to investigate this unfortunately.

Final Conclusion

The research on scaling and optimizing the performance of MyWatchList has led to practical solutions to handle a million concurrent users effectively. Here are the key points:

1. **Microservices Architecture**:
    - MyWatchList uses a microservices architecture, which allows for independent scaling and fault isolation. This setup helps manage increased loads efficiently through horizontal scaling in the Kubernetes cluster.
2. **Horizontal Scaling and Asynchronous Communication**:
    - Horizontal scaling in the Kubernetes cluster ensures that MyWatchList can handle more users smoothly. Using asynchronous communication with RabbitMQ improves performance and scalability, supporting load balancing. The Ocelot .NET API gateway also helps with load balancing.
3. **Cloud-Based Infrastructure and Azure Integration**:
    - Since MyWatchList is built with C#, using Azure is beneficial due to its integration with Microsoft tools. Azure Kubernetes Service (AKS) makes it easier to deploy and scale applications, and Azure Cosmos DB for PostgreSQL offers a scalable database solution. Azure Key Vault provides secure key management for .NET applications, and Azure Monitor helps with monitoring and performance management.
4. **Addressing Performance Bottlenecks**:
    - Common performance issues like high CPU usage, slow database queries, network latency, slow page loads, and synchronous code were identified and fixed. Solutions included converting synchronous code to asynchronous, using RabbitMQ for processing, and using Prometheus for monitoring. These actions have improved MyWatchList's performance and scalability.

By implementing these solutions, MyWatchList is prepared to handle a large user base while maintaining good performance and reliability. Future steps, like exploring caching solutions with Azure Cache for Redis and using content delivery networks (CDNs), will further improve performance. This approach ensures MyWatchList stays scalable, efficient, and provides a good user experience.

Bibliography

Adamson, C. (2024, 1 17). *Autoscaling Pitfalls to Avoid*. Retrieved from LinkedIn: https://www.linkedin.com/pulse/autoscaling-pitfalls-avoid-christopher-adamson-xj21c/

Awati, R., & Denman, J. (n.d.). *sharding*. Retrieved from TechTarget: https://www.techtarget.com/searchoracle/definition/sharding

Besedin, J. (n.d.). *Best Practices for Optimizing Kubernetes' HPA*. Retrieved from intel Granulate: https://granulate.io/blog/optimizing-kubernetes-hpa-best-practices/

Cloudflare. (n.d.). *Types of load balancing algorithms*. Retrieved from Cloudflare: https://www.cloudflare.com/learning/performance/types-of-load-balancing-algorithms/

Digital Ocean. (n.d.). *Horizontal scaling vs vertical scaling: Choosing your strategy*. Retrieved from Digital Ocean: https://www.digitalocean.com/resources/article/horizontal-scaling-vs-vertical-scaling#horizontal-scaling-vs-vertical-scaling

How do you address performance bottlenecks in web development frameworks? (n.d.). Retrieved from LinkedIn: https://www.linkedin.com/advice/1/how-do-you-address-performance-bottlenecks-web

Lewis, J., & Fowler, M. (2014, March 25). *Microservices*. Retrieved from martinFowler.com: https://martinfowler.com/articles/microservices.html

Microsoft. (2023, October 12). *Asynchronous programming scenarios*. Retrieved from Microsoft: https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/async-scenarios

Newman, D. (2024, January 16). *Load Balancing Fundamentals: How Load Balancers Work*. Retrieved from Akamai: https://www.linode.com/docs/guides/load-balancing-fundamentals/

Santos, O. C. (n.d.). *What are the best practices and common pitfalls of dynamic load balancing for parallel applications?* Retrieved from LinkedIn: https://www.linkedin.com/advice/0/what-best-practices-common-pitfalls-dynamic

Sharma, S. (2024, March 5). *How to identify and address performance bottlenecks*. Retrieved from productledalliance: https://www.productledalliance.com/performance-bottlenecks-in-web-applications/

System Design by CHK. (2023, August 12). *System Design —A Comprehensive Guide on Synchronous & Asynchronous Microservice Communication*. Retrieved from Medium: https://medium.com/@systemdesignbychk/system-design-a-comprehensive-guide-on-synchronous-asynchronous-microservice-communication-8bda324943b8