YUSUF EMRE BAYRAKCI

22118080006

BM401 STAJ II

STAJ RAPORU

ASELSAN ELEKTRONİK SANAYİ ve TİCARET A.Ş.

Öğrencinin Onaylı Fotoğrafı
Certified Photograph of the Student

Öğrencinin Adı ve Soyadı :.....
Student's Name and Surname :.....

Sınıf ve No :.....
Year and Number :.....

Kurumun Adı ve Adresi :.....
Name and Address of the Company :.....
:.....
:.....
:.....

Staja Başlayış ve Bitiş Tarihleri :.....
Starting and Ending Dates of the Practice :.....

Raporun Sunulduğu Tarih :.....
Submission Date of the Report :.....

Öğrencinin İmzası :.....
Student's Signature :.....

Raporu İnceleyen Öğretim Üyesi :.....
Faculty Member Graded the Report :.....

Verilen Not : B/K :.....
Grade Awarded : S/U

Tarih :.....
Date :.....

İmza :.....
Signature :.....

Raporu İnceleyen Öğretim Üyesi :.....
Faculty Member Graded the Report :.....

Verilen Not : B/K :.....
Grade Awarded : S/U :.....

Tarih :.....
Date :.....

İmza :.....
Signature :.....

İÇİNDEKİLER

Sayfa

ÇİZELGELERİN LİSTESİ

Çizelge **Sayfa**

ŞEKİLLERİN LİSTESİ

Şekil **Sayfa**

SİMGELER VE KISALTMALAR

Bu çalışmada kullanılmış simgeler ve kısaltmalar, açıklamaları ile birlikte aşağıda sunulmuştur.

Kısaltmalar	**Açıklamalar**
Ar-Ge	Araştırma Geliştirme
CI/CD	Continious Integration / Continious Deployment
DI	Dependency Injection
DOM	Document Object Model
DTO	Data Transfer Object
GB	Gigabyte
HTTP	Hypertext Transfer Protocol
JVM	Java Virtual Machine
MS	Milisaniye
MT	Megatransfer
MVC	Model-View-Controller
REST	REpresentational State Transfer
SPA	Single Page Application
SQL	Structured Query Language
TTL	Time To Live

1. GİRİŞ

Bilgisayar Mühendisliği lisans programı kapsamında zorunlu olan ikinci staj, 16 Haziran – 11 Temmuz 2025 tarihleri arasında, ASELSAN Elektronik Sanayi ve Ticaret A.Ş. Savunma Sistem Teknolojileri Sektör Başkanlığı bünyesinde, Gölbaşı Yerleşkesi'nde gerçekleştirilmiştir. Bu stajın amacı modern yazılım geliştirme teknolojileriyle deneyim kazanması ve profesyonel iş ortamını yakından tanınması olmuştur.

Staj süresi boyunca geliştirilen proje, çok oyunculu sistemlerde kritik bir bileşen olan liderlik tablosu (leaderboard) yapısının farklı bellek içi veri yönetim çözümleriyle performans, esneklik ve ölçeklenebilirlik açısından karşılaştırılması üzerine odaklanmıştır. Bu kapsamda, Java Spring Boot tabanlı bir RESTful API tasarlanmış ve üç farklı repository yaklaşımı (Plain Java In-Memory, Caffeine Cache ve Redis Sorted Set) aynı kullanım senaryosu altında test edilmiştir. Projede, Docker altyapısı kullanılarak Redis, Prometheus ve Grafana gibi servisler konteyner tabanlı bir yapıda çalıştırılmış; Micrometer kütüphanesi ile uygulama içi performans metrikleri toplanmış ve Grafana panelleri aracılığıyla görselleştirilmiştir.

Projenin önemi, günümüzde özellikle oyun ve büyük ölçekli web uygulamalarında rekabetin merkezinde yer alan liderlik tablolarının, çok yüksek sayıda eşzamanlı okuma ve yazma isteği altında dahi düşük gecikme süreleriyle çalışabilmesini sağlamaktır. Bu tür sistemlerde, anlık skor güncellemeleri ve hızlı sıralama işlemleri kullanıcı deneyimi açısından kritik öneme sahiptir. Bu nedenle, farklı veri yönetim stratejilerinin test edilmesi, yalnızca performans ölçütlerini karşılaştırmakla kalmamış, aynı zamanda hangi yöntemin hangi senaryolarda tercih edilmesi gerektiğine dair somut sonuçlar da sunmuştur.

Çalışmanın sınırlılığı, testlerin kurumsal ölçekte bir sunucu ortamında değil, laboratuvar koşullarında kişisel bir iş istasyonu üzerinde yapılmış olmasıdır. Bu durum, üretim ortamında karşılaşılabilecek ağ gecikmeleri, donanım farklılıkları ve çok düğümlü dağıtık mimari senaryolarını sınırlı biçimde yansıtmaktadır. Buna rağmen, elde edilen veriler farklı teknolojilerin güçlü ve zayıf yönlerini ortaya koymakta ve akademik/mesleki açıdan değerli bir karşılaştırma sağlamaktadır.

2. FİRMA HAKKINDA BİLGİ

2.1. Kuruluşun Adı

ASELSAN ELEKTRONİK SANAYİ ve TİCARET A.Ş.

2.2. Kuruluşun Yeri

ASELSAN savunma sanayiinde faaliyet gösteren Türkiye'nin en köklü ve stratejik kurumlarından olarak yalnızca başkentte değil İstanbul ve Kocaeli gibi önemli sanayi bölgelerinde de yerleşkelere sahiptir [1].

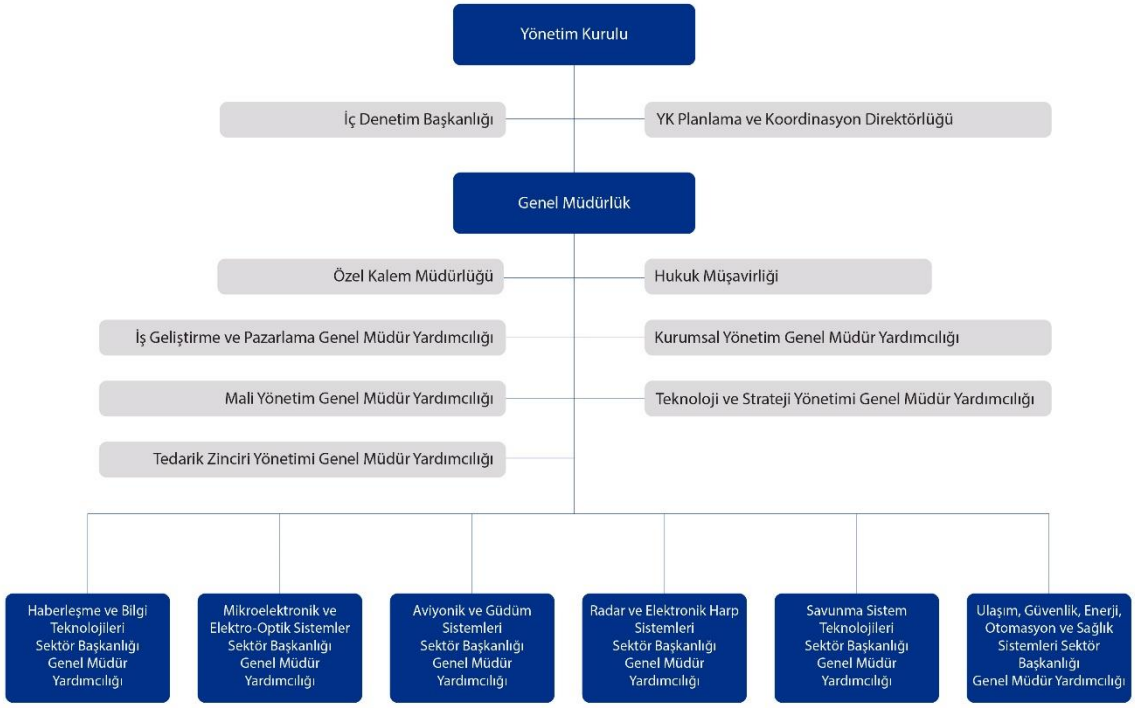
Kurum, Macunköy, Akyurt, Gölbaşı, ODTÜ Teknokent (SATGEB), ODTÜ Teknokent (TİTANYUM), İvedik Teknopark, Hacettepe Teknokent, İstanbul Tuzla, İstanbul Teknopark ve Gebze Bilişim Vadisi olmak üzere toplamda 10 farklı yerleşkede faaliyet göstermektedir [2].

Bu staj ASELSAN'ın Gölbaşı tesislerinde Savunma Sistem Teknolojileri Sektör Başkanlığı'nda gerçekleştirilmiştir.

ASELSAN Gölbaşı Yerleşkesi: Konya Yolu 8.km Oğulbey Mahallesi 3051 Sokak No: 3 06830 Gölbaşı, ANKARA

2.3. Kuruluşta Çalışanlar Hakkında Bilgi

ASELSAN, 12.000'in üzerinde çalışanıyla Türkiye'nin en büyük savunma sanayi işverenlerinden biridir. Bu büyük insan kaynağının %64'ü mühendis kadrosunda görev yapmaktadır. Bu durum kurumun mühendislik tabanlı bir yapıya sahip olduğunu ve yüksek teknolojiye dayalı projelerin ağırlığını ortaya koymaktadır.
Ar-Ge faaliyetleri için özel olarak ayrılmış yaklaşık 7.000'den fazla Ar-Ge personeli bulunmaktadır. Akademik dağılım incelendiğinde çalışanların %43'ü lisans, %27'si yüksek lisans ve %3'ü doktora derecesine sahiptir. Kurum çalışanlarının teknik ve kişisel gelişimlerine de sürekli yatırım yapmaktadır [3].



Şekil 2.1. ASELSAN organizasyon şeması

2.4. Kuruluş Faaliyet Alanı

Radar ve Elektronik Harp Sistemleri: ASELSAN kara, deniz ve hava platformlarına yönelik radar sistemleri ile elektronik harp çözümleri geliştirmektedir. Bu sistemler, hedef tespiti, izleme, karıştırma ve sinyal bastırma gibi görevlerde kullanılmaktadır. Askeri birliklerin elektronik tehditlere karşı korunmasında kritik rol oynar [4].

Haberleşme ve Bilgi Teknolojileri: Taktik saha iletişimi uydu haberleşmesi ve kamu güvenliği için geniş bant çözümleri sunulmaktadır. Kriptolu iletişim sistemleriyle güvenli veri aktarımı sağlanır. Siber güvenlik çözümleri ile ağ altyapıları da korunmaktadır [4].

Aviyonik ve Güdüm Sistemleri: Hava araçları için seyrüsefer, uçuş kontrol ve görev bilgisayarı sistemleri geliştirilmektedir. Güdüm sistemleri sayesinde hassas vuruş kabiliyeti artırılmaktadır. Bu ürünler milli hava platformlarında yaygın olarak kullanılmaktadır [4].

Mikroelektronik ve Elektro-Optik Sistemler: Gece görüş cihazları, termal kameralar, lazer mesafe ölçerler ve optik nişangâhlar üretmektedir. Mikroelektronik birimlerde özgün çip ve entegre devre tasarımları yapılmaktadır. Bu sayede sistemlerin dışa bağımlılığı azaltılır [4].

Uydu Sistemleri: Uydu haberleşme, yer istasyonu altyapıları ve yük kontrol sistemleri geliştirilir. ASELSAN uydu projelerinde görev alan ekipmanları yerli olarak üretmeyi hedeflemektedir. Bu alanda TÜRKSAT ve diğer kuruluşlarla iş birlikleri sürdürülmektedir [4].

Kara, Deniz ve Hava Sistemleri: Zırhlı araçlar için komuta kontrol, atış kontrol ve görev sistemleri geliştirilmektedir. Deniz platformları için sonar, elektro-optik ve radar sistemleri sunulmaktadır. Hava sistemleriyle entegre çalışabilecek görev yönetim altyapıları kurulmaktadır [4].

Komuta Kontrol Sistemleri: Taktik ve stratejik düzeyde görev planlama, durumsal farkındalık ve emir-komuta yapılarının yönetimi için yazılım ve donanım çözümleri sağlanır. Sistemler, çoklu birimlerin eşgüdüm içinde hareket etmesini sağlar [4].

Ulaşım, Güvenlik, Enerji ve Sağlık Sistemleri: ASELSAN akıllı trafik sistemleri, enerji izleme sistemleri, kamu güvenliği kameraları ve medikal cihazlar da geliştirmektedir. Sivil alanda da kritik teknolojiler üreterek teknolojik yetkinliğini genişletmektedir [4].

2.5. Kuruluşun Kısa Tarihçesi

ASELSAN Türkiye'nin savunma elektroniği alanındaki ihtiyaçlarını milli imkanlarla karşılamak üzere, 14 Kasım 1975 tarihinde Türk Silahlı Kuvvetlerini Güçlendirme Vakfı tarafından Ankara'da kuruldu. Kuruluşundan kısa süre sonra Macunköy tesislerinde üretim faaliyetlerine başladı ve 1980’li yılların başında telsiz ve iletişim sistemleri üretimini gerçekleştirerek Türk Silahlı Kuvvetleri’nin ihtiyaçlarını karşılamaya başladı [5].
1980’li yıllarda ürün portföyünü genişleterek radar sistemleri, elektronik harp ve veri terminalleri gibi kritik savunma teknolojilerinde uzmanlaşmaya başladı. Bu dönemde ASELSAN ilk ihracatını gerçekleştirerek uluslararası pazara adım attı. 1990’lı yıllarda elektro-optik sistemler ve radar teknolojileri üzerine Ar-Ge merkezleri kuruldu ve bu sistemlerde önemli teknolojik ilerlemeler kaydedildi [5].
1990'lı yılların sonlarında ASELSAN askeri uygulamalar yanında sivil projelere de yöneldi. Karayolları acil yardım haberleşme sistemleri, otoyol ücret toplama sistemleri ve mobil telefon üretimi gibi sivil alanlarda faaliyet göstermeye başladı. Ayrıca termal kameralar ve gece görüş sistemleri gibi gelişmiş elektro-optik cihazları Türk Silahlı Kuvvetleri’ne sağlamaya başladı [5].

2000’li yıllarda ASELSAN’ın faaliyet alanları daha da genişledi. Bu dönemde aviyonik ve güdüm sistemleri, insansız hava araçları sistemleri, uydu haberleşme teknolojileri ve kara platformları için atış kontrol sistemleri geliştirildi. Aynı dönemde, sivil teknoloji çözümleri olarak enerji yönetimi, akıllı ulaşım sistemleri ve sağlık elektroniği gibi yeni alanlarda faaliyetler başlatıldı [5].

2010 sonrası dönemde ASELSAN ihracat kapasitesini ve uluslararası rekabet gücünü artırarak dünya çapında bilinirliğini güçlendirdi. 2015 yılında Ankara Gölbaşı Yerleşkesi açıldı ve bu tesisle birlikte şirketin üretim ve Ar-Ge kapasitesi büyük ölçüde genişletildi. 2020’li yıllarda savunma sanayindeki büyüme ivmesini sürdürerek savunma sektöründe dünyanın en büyük şirketleri arasına girdi ve uluslararası savunma şirketleri sıralamasında ilk 50 içerisinde yer aldı [5].

3. STAJ RAPORU

Staj süresince ASELSAN bünyesinde gerçekleştirilen projenin amacı, Redis, Caffeine ve Java yöntemlerinin performansını karşılaştıran bir Java Spring Boot uygulaması geliştirmek olmuştur. Proje kapsamında temel hedef farklı önbellekleme çözümlerinin sistem performansına etkisini karşılaştırmalı olarak analiz etmek ve bu çözümlerin uygun kullanım senaryolarını ortaya koymaktır.

Uygulama geliştirme sürecinde Java programlama dili temel alınmış, Spring Boot framework'ü ile RESTful API yapısına sahip bir sistem oluşturulmuştur. Docker teknolojisi kullanılarak uygulama bileşenleri izole edilerek yönetilebilir bir geliştirme ortamı sağlanmıştır.

Uygulamada öncelikle Redis veri tabanı entegre edilerek dış kaynaklı ve kalıcı önbellekleme sağlanmıştır. Ardından Java tabanlı Caffeine kütüphanesi kullanılarak bellek içi, yüksek hızlı önbellekleme mekanizması uygulanmıştır. Her iki yöntem, herhangi bir önbellekleme yapılmayan bir referans yapı (Plain Java) ile birlikte çeşitli senaryolar altında performans açısından karşılaştırılmıştır. Yapılan analizlerde işlem süresi, yanıt süresi ve kaynak kullanımı gibi metrikler değerlendirilmiştir.

3.1. Kuruluşta Kullanılan Teknolojiler

Aşağıda belirtilen teknolojiler modern yazılım geliştirme süreçlerinde ve ASELSAN'da staj yapılan müdürlükte yazılım geliştirme süreçlerinde yaygın olarak tercih edilmekte ve her biri farklı bir problemi çözmek veya geliştirme sürecini daha etkin hale getirmek amacıyla kullanılmaktadır. ASELSAN'da diğer müdürlüklerde kullanılan teknolojiler farklılık gösterebilir.

Java Programlama Dili: Java, platformdan bağımsız, nesne yönelimli bir programlama dilidir. Java Virtual Machine (JVM) sayesinde farklı platformlarda aynı kodu çalıştırabilme yeteneği, geniş bir topluluk desteği ve güçlü güvenlik mekanizmaları nedeniyle özellikle kurumsal ölçekteki uygulamalarda sıkça kullanılmaktadır. Java Android uygulamalarından web uygulamalarına, masaüstü yazılımlardan büyük veri çözümlerine kadar geniş bir kullanım alanına sahiptir [11].

Spring Boot: Java tabanlı uygulama geliştirmeyi basitleştiren bir uygulama çatısıdır. Spring Framework'ün üzerine inşa edilmiş olup otomatik yapılandırma, hazır bağımlılık yönetimi

ve entegre sunucu özellikleri sayesinde geliştiricilerin uygulamalarını hızlı ve kolay bir şekilde üretim ortamına taşımalarını sağlamaktadır. Mikroservis mimarileri ve RESTful API geliştirmede oldukça yaygındır [20].

Gradle: Modern ve esnek bir yapı (build) otomasyon aracıdır. Özellikle Java, Kotlin ve Android projelerinde tercih edilmekte olup Maven ve Ant gibi alternatiflere kıyasla daha hızlı ve özelleştirilebilir bir yapı sunmaktadır. Bağımlılıkları yönetmek, derleme süreçlerini otomatikleştirmek ve projeyi modüler hale getirmek için yaygın olarak kullanılmaktadır [9].

Kotlin: JVM üzerinde çalışan, modern, statik tipli ve çok amaçlı bir programlama dilidir. Java ile tamamen uyumlu olması sayesinde özellikle Android geliştirme başta olmak üzere sunucu tarafı uygulamalarda ve multiplatform projelerde tercih edilmektedir. Kotlin daha sade ve okunabilir kodlar yazmaya olanak sağlayarak geliştiricilere verimlilik avantajı sunar [15].

TypeScript: JavaScript dilinin üzerine inşa edilmiş, açık kaynaklı ve statik tip desteğine sahip bir dildir. Büyük ölçekli uygulamalarda tip güvenliği ve hata yönetimini kolaylaştırarak kodun sürdürülebilirliğini ve okunabilirliğini artırır. Frontend geliştirme süreçlerinde, özellikle karmaşık ve kapsamlı projelerde yaygın olarak kullanılmaktadır [21].

React: Kullanıcı arayüzü geliştirme için yaygın olarak tercih edilen açık kaynaklı JavaScript kütüphanesidir. React'ın temel özelliği bileşen tabanlı yapısı ve sanal DOM (Virtual DOM) sayesinde yüksek performans sağlamasıdır. Tek sayfa uygulamaları (SPA) ve modern web uygulamalarının oluşturulmasında yaygın biçimde kullanılmaktadır [18].

Redis: Hızlı ve bellek içi çalışan NoSQL tabanlı bir anahtar-değer (key-value) veri deposudur. Genellikle önbellekleme ,(caching) oturum yönetimi ve mesaj kuyrukları gibi hızlı veri erişimi gereken senaryolarda tercih edilmektedir. Redis, performans kritik uygulamalarda yüksek hız ve düşük gecikme avantajı sunmaktadır [19].

Kafka: Yüksek kapasiteli ve dağıtık bir olay akış platformudur. Kafka, gerçek zamanlı veri işleme, olay tabanlı mimariler ve log toplama sistemleri için tasarlanmıştır. Yüksek performansı, ölçeklenebilirliği ve dayanıklılığı nedeniyle büyük veri ve mesajlaşma sistemlerinde standart çözüm haline gelmiştir [12].

Docker: Uygulamaların konteyner (container) içinde çalıştırılmasını sağlayan bir konteyner teknolojisidir. Docker uygulamaların ortamdan bağımsız olarak kolayca taşınmasını ve

tutarlı çalıştırılmasını sağlar. Yazılımların üretim ortamına taşınmasını kolaylaştırarak geliştirme ve dağıtım süreçlerini hızlandırır [7].

Kubernetes (k8s): Docker gibi konteynerlerin yönetimini otomatikleştirmek için geliştirilmiş açık kaynaklı bir orkestrasyon aracıdır. Kubernetes konteynerlerin otomatik ölçeklendirilmesi, yük dengelemesi, hata durumunda kurtarma ve yüksek erişilebilirlik gibi görevleri kolaylaştırır ve bu süreçleri standart hale getirir [14].

Prometheus: Açık kaynaklı bir izleme ve alarm sistemidir. Zaman serisi verilerini toplamak, analiz etmek ve görselleştirmek için kullanılır. Mikroservis mimarilerinde uygulama metriklerinin merkezi olarak izlenmesi ve performans sorunlarının tespit edilmesi amacıyla yaygın olarak tercih edilmektedir [17].

Grafana: Özellikle Prometheus gibi zaman serisi verilerini görsel hale getiren bir analiz ve izleme aracıdır. Çeşitli veri kaynaklarından alınan metrikleri gösterişli panolar (dashboards) ile sunarak kullanıcıların sistem durumunu ve performansını hızlı bir şekilde analiz etmelerini sağlar [10].

Loki: Log kayıtlarının merkezi ve etkili biçimde toplanmasını sağlayan açık kaynaklı bir log yönetim çözümüdür. Grafana ekosisteminin bir parçası olarak çalışır ve log kayıtlarını Prometheus tarzında depolayarak hızlı sorgulama ve analiz imkânı sunar [16].

Azure DevOps: Microsoft tarafından sağlanan yazılım geliştirme süreçlerini uçtan uca yöneten bir DevOps platformudur. Kaynak kontrolü, sürekli entegrasyon ve sürekli teslim (CI/CD) süreçlerini otomatikleştirme, proje yönetimi, test otomasyonu ve paket yönetimi gibi fonksiyonları entegre bir şekilde sunarak yazılım ekiplerinin iş birliği ve üretkenliğini artırır [6].

3.2. Staj Süreci: İlk Hafta (1 - 5. Gün)

Staj süresi boyunca ilk 1 hafta proje ile ilgilenilmemiştir, proje yerine yapılanlar aşağıda gün gün belirtilmiştir.

16 Haziran 2025 1.gün: Stajın ilk günü ASELSAN Macunköy Yerleşkesi'nde başlamıştır. Sabah saatlerinde stajyerler Macunköy’e giderek, giriş işlemleri kapsamında avuç içi biyometrik tanımlama yapılmış ve kurumsal yaka kartları teslim alınmıştır. Bu kartlar ile ASELSAN yerleşkelerine erişim sağlanabilmektedir. Giriş işlemleri tamamlandıktan sonra tüm stajyerler, stajın gerçekleştirileceği Gölbaşı Yerleşkesi'ne yönlendirilmiştir.

Gölbaşı'na ulaşıldığında ilk olarak stajyerler için kahvaltı ikramı gerçekleştirilmiştir. Kahvaltının ardından staj başlangıç işlemleri yürütülmüş; stajyerlerin kimlik ve kayıt bilgileri kontrol edilmiş, gerekli belgelerin teslimi yapılmıştır. Ardından servis kullanımına ilişkin kayıt işlemleri gerçekleştirilmiş ve personel servis sistemi hakkında bilgilendirme yapılmıştır. Günün devamında ise, staj boyunca uyulması gereken kurallar, tesis içi hareket düzeni ve iletişim prosedürleri gibi konuları kapsayan bir oryantasyon oturumu gerçekleştirilmiştir. Bu oturumda ayrıca ASELSAN'ın temel kurumsal yapısı, vizyonu ve stajyerlerden beklentileri de paylaşılmıştır.

17 Haziran 2025 2.gün: İkinci gün, iş sağlığı ve güvenliği eğitimi kapsamında çeşitli teorik ve uygulamalı içeriklerin sunulmasıyla geçmiştir. Sabah oturumunda, iş hukuku konusu ele alınmış; işverenin işçiye karşı yükümlülükleri, işçinin işverene karşı sorumlulukları, iş sözleşmelerinin kapsamı, çalışma koşulları, izin ve mola hakları gibi başlıklar detaylı şekilde incelenmiştir. Katılımcılara iş hukukuna ilişkin temel yasal çerçeve aktarılmış, örnek olaylar üzerinden uygulamalı değerlendirmeler yapılmıştır. Oturumun sonunda bilgileri ölçmek amacıyla kısa bir sınav uygulanmıştır.

Öğleden sonra gerçekleşen oturumda ise ilk yardım eğitimi verilmiştir. Bu bölümde temel yaşam desteği, solunum kontrolü, bilinç kaybı, kanamalar, kırık-çıkık burkulmalar ve zehirlenmeler gibi acil durumlara müdahale teknikleri anlatılmıştır. Eğitim, görsel sunumlar ve eğitmen eşliğinde uygulamalı olarak desteklenmiş; stajyerlerin acil durum farkındalığını artırmaya yönelik bilgiler pekiştirilmiştir. Eğitimin sonunda bu bölüme özel ayrı bir değerlendirme sınavı gerçekleştirilmiştir.

18 Haziran 2025 3.gün: Üçüncü günün sabah bölümünde iş sağlığı ve güvenliği eğitimine devam edilmiştir. Eğitimin ilk kısmında, kişisel koruyucu donanımların kullanımı, iş güvenliği ekipmanlarının özellikleri, ergonomik çalışma düzeni, riskli durumların tespiti ve iş kazalarının önlenmesine yönelik temel tedbirler detaylı bir şekilde anlatılmıştır. Bu teorik bilgiler, çeşitli görsel ve uygulamalı örneklerle desteklenmiş, eğitimin sonunda konuları kapsayan kısa bir sınav gerçekleştirilmiştir.

Eğitimin ikinci kısmında ise yangın güvenliği ve itfaiye eğitimi yapılmıştır. Bu bölümde yangın sınıfları, yangın söndürme cihazlarının türleri ve kullanım şekilleri, tahliye planları ve acil durumlarda izlenecek adımlar ayrıntılı biçimde açıklanmıştır. Katılımcılara yangın anında yapılması ve yapılmaması gerekenler anlatılmış, çeşitli senaryolar üzerinden değerlendirmeler yapılmıştır. İkinci oturumun ardından yine bir değerlendirme sınavı

gerçekleştirilmiştir.
Tüm eğitim oturumlarının sonunda sabah ve öğleden sonra gerçekleştirilen bölümlerin içeriğini kapsayan genel bir sınav uygulanarak iş sağlığı ve güvenliği eğitim süreci tamamlanmıştır.
Öğleden sonra stajyerler görevli oldukları birimlere geçiş yapmış, bireysel bilgisayar temini sağlanmış ve kullanıcı hesapları tanımlanmıştır. İlgili birim yetkilileri tarafından bölümün genel işleyişi, kullanılan teknolojik araçlar ve staj boyunca yürütülecek projeler hakkında bilgilendirme yapılmıştır.

19 Haziran 2025 4.gün: Dördüncü gün teknik hazırlık ve geliştirme ortamının yapılandırılması çalışmaları ile geçmiştir. Sabah saatlerinde stajyer bilgisayarına yazılım geliştirme sürecinde kullanılacak olan çeşitli programlar kurulmuştur. Bu kapsamda Git, IntelliJ IDEA, Java JDK, Docker, Google Chrome, Node.js ve Windows Subsystem for Linux (WSL) gibi araçlar sistemlere yüklenmiştir. Kurulumlar sırasında karşılaşılan bazı teknik sorunlar, özellikle kurum içi internet erişimindeki kısıtlamalar nedeniyle çözüm gerektirmiş ve bu sorunlar araştırılarak giderilmiştir.

Öğleden sonra Docker teknolojisine dair temel bilgiler edinilmeye başlanmıştır. Docker'ın çalışma prensibi, konteyner yapısı, imaj oluşturma ve yönetme gibi temel kavramlar üzerinde durulmuş; örnekler üzerinden uygulamalı öğrenme sağlanmıştır. Bu süreçte ayrıca, geliştirilecek projede Docker kullanımının sağlayacağı faydalar ve kullanım senaryoları hakkında fikir sahibi olunmuştur.

20 Haziran 2025 5.gün: Beşinci gün yazılım geliştirme ortamının son yapılandırmaları ve teknik engellerin çözümü ile geçmiştir. Günün ilk saatlerinde, kurumun güvenlik politikaları gereği telefonların teslim alınması nedeniyle Google hesabına erişim sağlanamamış ve iki aşamalı kimlik doğrulama tamamlanamamıştır. Bu durum Google ve Github gibi bazı servislerde erişim sorunlarına neden olmuştur.

Bununla birlikte Docker hakkında öğrenilen bilgiler derinleştirilmiş ve komut satırı üzerinden pratikler yapılmıştır. Ardından Java Spring Boot teknolojisini öğrenmeye yönelik çalışmalar başlatılmıştır. Bu doğrultuda basit bir REST API yapısına sahip yapılacaklar listesi uygulaması üzerinde çalışılmıştır. Uygulamanın bağımlılık yönetiminde kullanılan Maven aracı ile çalışırken karşılaşılan sertifika doğrulama hatası incelenmiş, özel depo tanımı .m2/settings.xml dosyasına eklenerek hata başarılı şekilde giderilmiştir.

3.3. Geliştirdiğim Proje

Proje çok oyunculu sistemler için liderlik tablosu (leaderboard) uygulaması üzerine odaklanmıştır. Liderlik tabloları modern oyun ve uygulamalarda rekabetin merkezinde yer alan kritik bileşenlerdir. Saniyeler içinde binlerce skor güncellemesi ve sorgu gerçekleştiği için, bu tür sistemlerin yüksek performans, düşük gecikme süresi ve ölçeklenebilirlik sağlaması zorunludur.

Bu bağlamda proje kapsamında temel amaç, Redis, Caffeine Cache ve Java (In-Memory) yaklaşımlarının aynı problem senaryosu altında karşılaştırılması olmuştur. Her üç çözüm de Spring Boot tabanlı bir RESTful API çatısı altında uygulanmış, ardından k6 aracıyla gerçekleştirilen yük testleriyle performans metrikleri analiz edilmiştir.

Sistem oyuncu oluşturma, skor güncelleme, en yüksek skora sahip oyuncuları sıralama ve oyuncu verilerini temizleme gibi temel işlevleri sağlamaktadır. Bu işlevler farklı veri saklama stratejileri üzerinde test edilerek, okuma-yazma performansı, gecikme süreleri ve sistemin ölçeklenebilirliği ölçülmüştür.

Projede ayrıca Micrometer, Prometheus ve Grafana teknolojileri kullanılarak sistem metrikleri toplanmış ve görselleştirilmiştir. Böylece yalnızca işlevsel bir uygulama geliştirmekle kalınmamış, aynı zamanda dağıtık sistemlerde gözlemleme ve performans analizi konularında da pratik bir deneyim elde edilmiştir.

Sonuç olarak bu proje, bellek içi veri yapılarıyla lider tablosu yönetimi üzerine deneysel bir çalışma olmuş; performans, esneklik ve kullanım senaryoları açısından Java, Caffeine ve Redis çözümlerinin avantaj ve dezavantajlarını ortaya koymayı hedeflemiştir. Proje bir Github deposuna yüklenmiştir [8].

3.3.1. Proje gereksinimlerinin belirlenmesi

Projede temel amaç çok oyunculu sistemlerde kullanılan liderlik tablosu (leaderboard) yapısının farklı bellek içi veri yönetim çözümleriyle performans, esneklik ve ölçeklenebilirlik açısından karşılaştırılması olmuştur. Bu hedef doğrultusunda gereksinimler fonksiyonel, performans ve teknolojik boyutlarıyla tanımlanmış ve uygulanacak çözümler bu gereksinimlere göre şekillendirilmiştir.

Fonksiyonel gereksinimler: Projede oyuncuların skor yönetimini sağlayacak temel işlevlerin yerine getirilmesi öngörülmüştür. Sistem üzerinde oyuncuların benzersiz kimliklerle temsil edilmesi, skorlarının güncellenebilmesi, en yüksek skora sahip oyuncuların belirli sayıda listelenebilmesi ve gerektiğinde tüm oyuncu verilerinin temizlenebilmesi amaçlanmıştır. Ayrıca hatalı veya geçersiz istekler için uygun hata mesajlarının dönülmesi gerektiği belirlenmiştir. Bu şekilde uygulamanın hem işlevsel açıdan tam hem de kullanıcı deneyimi açısından güvenilir olması hedeflenmiştir.

Performans gereksinimleri: Liderlik tablolarının doğası gereği yoğun eşzamanlı işlemlerle karşılaşacağı göz önünde bulundurulmuştur. Sistem binlerce eşzamanlı okuma ve yazma isteğini düşük gecikme süreleriyle yanıtlayabilecek şekilde tasarlanmıştır. Ölçeklenebilirliğin sağlanması ve kullanıcı sayısındaki artışa rağmen performans kaybının önlenmesi gerekliliği vurgulanmıştır. Bu amaçla farklı senaryolarda sistemin davranışını gözlemlemek üzere yük testlerinin yapılması planlanmıştır.

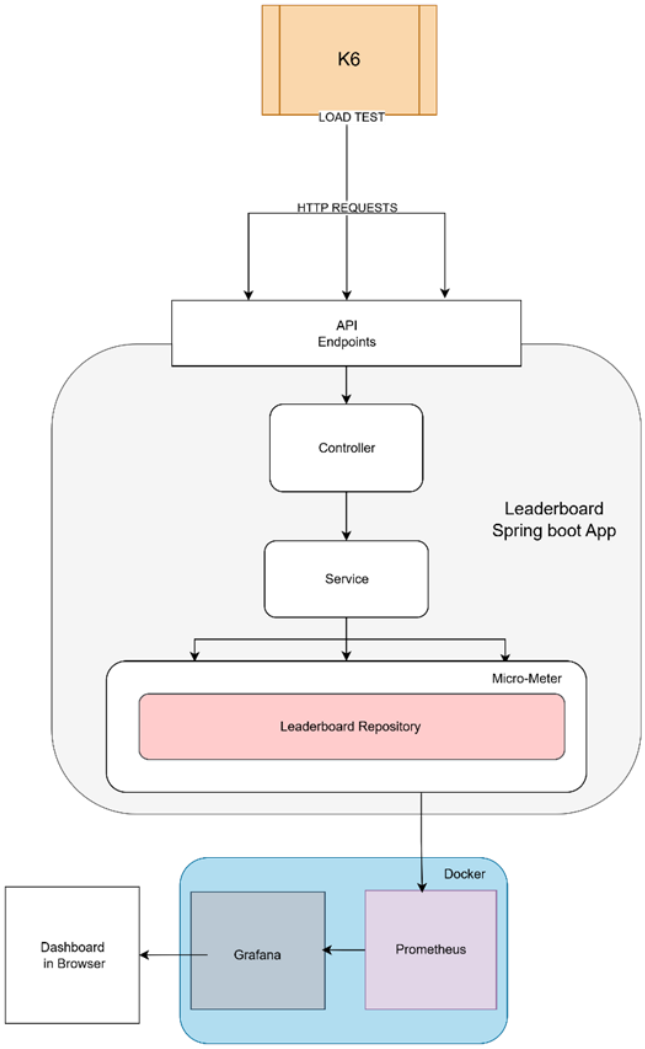
Teknolojik gereksinimler: Java, Spring boot ve Redis haricinde başlangıçta doğrudan bir teknoloji seçimine gidilmemiş; ihtiyaçların tespiti ve bu ihtiyaçlara en uygun çözümlerin araştırılması sonucunda teknolojiler belirlenmiştir. Öncelikle uygulamanın performansının düzenli olarak izlenebilmesi için metrik toplama ihtiyacı doğmuştur. Yapılan araştırmalar sonucunda Spring Boot ile uyumlu çalışan Micrometer kütüphanesinin uygun olduğu, ayrıca Prometheus'un zaman serisi verilerini depolamada, Grafana'nın ise bu verileri görselleştirmede yaygın kullanım sunduğu görülmüştür.

Veri saklama gereksinimleri doğrultusunda, Java tabanlı HashMap'in yüksek hız sağladığı ancak kalıcılık sunmadığı, Caffeine Cache'in bellek içi önbellek mantığıyla düşük gecikme sağladığı, Redis Sorted Set'in ise sıralama desteği ve dağıtık kullanım avantajları sunduğu tespit edilmiştir. Böylece aynı işlevin üç farklı yöntemle gerçekleştirilerek karşılaştırılması planlanmıştır.

Son olarak sistemin yüksek yük altında nasıl davranacağını test etmek için uygun bir yük testi aracına ihtiyaç duyulmuştur. Araştırmalar sonucunda, REST API performans testleri için güçlü bir seçenek olan k6 aracı tercih edilmiştir. Bu araç ile farklı okuma-yazma senaryoları oluşturularak sistemin dayanıklılığı gözlemlenmiştir.

3.3.2. Projenin Mimarisi ve Tasarım Deseni

Projenin Projede geliştirilen liderlik tablosu uygulamasının mimarisi, modülerlik, genişletilebilirlik ve gözlemlenebilirlik ilkeleri dikkate alınarak tasarlanmıştır. Şekil 3.1'de görüldüğü üzere sistem; istemciden gelen yük testleri, Spring Boot tabanlı uygulama bileşenleri, veri saklama katmanı ve gözlemleme araçlarından oluşmaktadır.



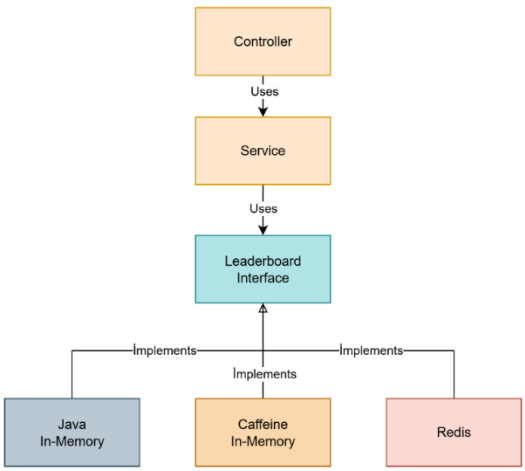
Şekil 3.1. Lider tablosu uygulama genel şeması

Katmanlı mimari: Projede üç katmanlı mimari tercih edilmiştir. Katmanlı mimari yazılım geliştirme sürecinde uygulamanın farklı işlevsel parçalarının ayrı katmanlarda toplanarak sorumluluklarının ayrıştırılması esasına dayanır. Böylece her katman kendi görevine odaklanır, bu da sistemin okunabilirliğini, sürdürülebilirliğini ve test edilebilirliğini artırır [13].

Şekil 3.1'de görüldüğü üzere, kullanıcıdan gelen HTTP istekleri ilk olarak API uç noktaları aracılığıyla Controller katmanına ulaşmaktadır. Controller istemci ile uygulama arasındaki arayüzdür; gelen istekleri kabul eder, uygun servislere yönlendirir ve elde edilen yanıtları istemciye geri döndürür. Burada örnek olarak LeaderboardController sınıfı, oyuncu ekleme, skor artırma, sıralama ve silme işlemleri için uç noktalar sağlamaktadır [8].

Service katmanı, iş mantığının uygulandığı yerdir. Katmanlı mimaride servis katmanı, uygulamanın ne yapması gerektiğini belirler. Örnek olarak şekil 3.2'de gösterildiği üzere bir oyuncunun skorunu artırma isteği geldiğinde Controller bu isteği Service'e iletir; Service gerekli kontrolleri yapar ve Repository'den aldığı veriler üzerinde işlemler gerçekleştirir. Böylece iş mantığı, hem Controller'dan hem de Repository'den bağımsız bir şekilde yönetilmiş olur.

Repository katmanı ise verinin yönetildiği bölümdür. Repository deseni, uygulamanın veri erişim detaylarını soyutlayarak iş mantığını veri kaynağından bağımsız hale getirir. Projede LeaderboardRepository adlı bir arayüz tanımlanmış ve bu arayüz, farklı veri kaynakları için ayrı ayrı somutlaştırılmıştır. Böylece aynı işlev, farklı veri yönetim yaklaşımlarıyla gerçekleştirilebilmiştir [8].



Şekil 3.2. Lider tablosu uygulama genel şeması

Repository tasarımı: LeaderboardRepository arayüzü, sistemin temel işlevlerini kapsayacak şekilde tanımlanmıştır: oyuncu ekleme, skor güncelleme, en iyi N oyuncuyu getirme ve oyuncu verilerini temizleme. Bu arayüz, şekil 3.2'de gösterildiği üzere üç farklı implementasyon ile somutlaştırılmıştır. InMemoryLeaderboardRepository, Java'nın kendi veri yapıları ile basit ve hızlı bir çözüm sunmuştur; CaffeineLeaderboardRepository, bellek

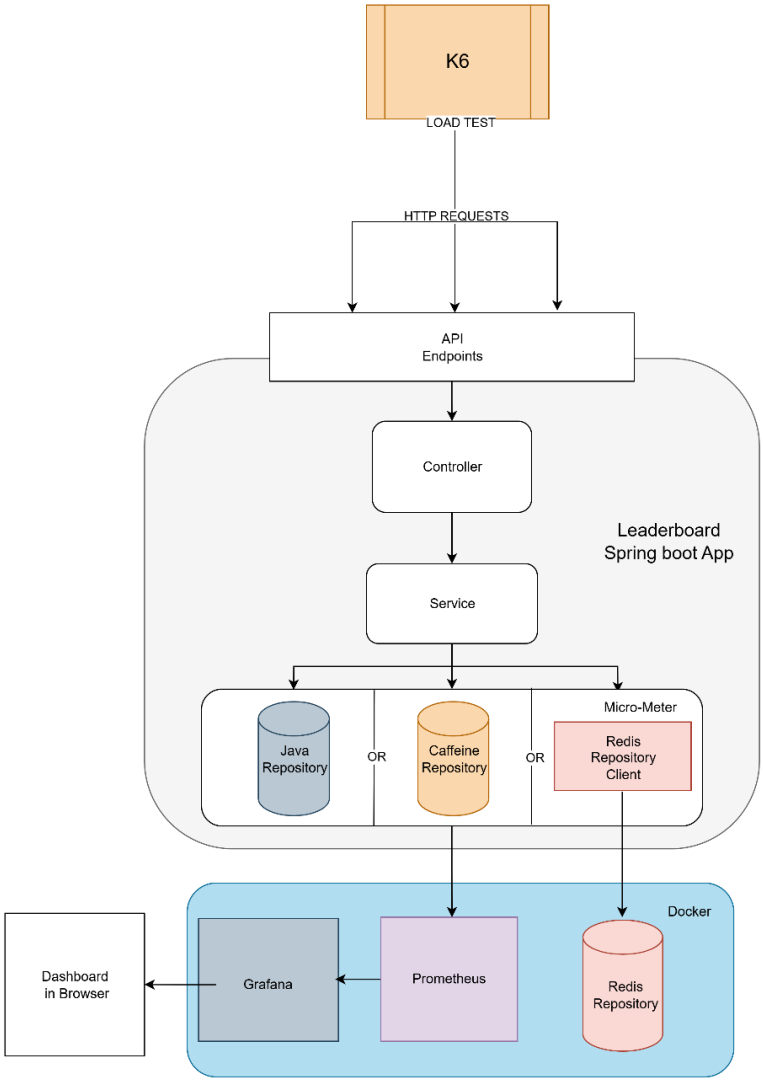
içi önbellek mekanizmaları sayesinde düşük gecikmeli bir çözüm sağlamıştır; RedisSortedSetLeaderboardRepository ise Redis'in sıralı küme (sorted set) veri yapısını kullanarak kalıcılık ve dağıtık kullanım avantajları sunmuştur [8].

Java In-Memory yaklaşımı: Şekil 3.2'de sol kısımda yer alan Java bellek içi çözümü, verilerin doğrudan uygulamanın belleğinde tutulmasına dayanmaktadır. Bu yöntem, ConcurrentHashMap gibi veri yapıları kullanılarak hayata geçirilmiştir. Bellek içi yaklaşım, dış bağımlılıklara gerek duymadan yüksek hız sağladığı için özellikle performans testlerinde referans noktası olarak kullanılmıştır. Ancak kalıcılık sağlamadığı için uygulama yeniden başlatıldığında tüm veriler kaybolmaktadır.

Caffeine In-Memory yaklaşımı: Şekil 3.2'nin ortasında gösterilen Caffeine ise modern bir Java tabanlı önbellek kütüphanesidir. Caffeine, Time-To-Live (TTL) ve maksimum boyut kontrolü gibi özelliklerle otomatik bellek yönetimi sağlamaktadır. Bu yaklaşım, özellikle sıcak veriye hızlı erişim gerektiren senaryolarda avantajlıdır. Caffeine kullanımı sayesinde, Redis gibi harici bir sunucuya ihtiyaç olmadan düşük gecikmeli veri erişimi mümkün hale gelmiştir. Bununla birlikte Caffeine de tıpkı Java yaklaşımı gibi verilerin kalıcılığını garanti etmez.

Redis yaklaşımı: Şekil 3.2'nin sağ tarafında yer alan Redis, yüksek performanslı, bellek içi çalışan bir veri tabanı çözümüdür. Projede Redis'in sıralı küme veri yapısı kullanılarak skorların sıralı tutulması sağlanmıştır. Redis, kalıcılık (persistence) ve dağıtık kullanım desteği sunarak çok oyunculu senaryolarda veri kaybı olmadan güncel skor tablolarının tutulmasını mümkün kılmıştır. Bununla birlikte, harici bir sunucu gerektirmesi ve ağ üzerinden erişim sağlandığı için gecikme süreleri Java bellek içi veya Caffeine'e göre nispeten daha yüksektir.

Bu tasarım, uygulamanın iş mantığını repository'den soyutlayarak, hangi veri kaynağının kullanıldığından bağımsız bir yapı kurmuştur. Örneğin, uygulama Redis üzerinde çalıştırılmak istendiğinde yalnızca konfigürasyonda Redis seçilmiş; iş mantığı veya controller katmanında herhangi bir değişikliğe gerek duyulmamıştır.



Şekil 3.3. Lider tablosu uygulama tam şeması

Şekil 3.3'te görüldüğü üzere, sistemin son hâlinde repository katmanı üç farklı seçenek ile uygulamanın içerisine entegre edilmiştir. Java bellek içi ve Caffeine repository'leri doğrudan Spring Boot uygulamasının içerisinde çalışırken Redis repository'si Docker ortamında ayrı bir sunucu üzerinde çalışmaktadır. Bu farklılık, Redis'in Linux tabanlı çalışmaya ihtiyaç duyması ve dağıtık kullanım için optimize edilmiş bir çözüm olmasıyla ilişkilidir.

Şekil 3.3'te üst bölümde görülen K6 yük testi aracı, istemciyi simüle ederek API uç noktalarına yoğun HTTP istekleri göndermektedir. Bu istekler uygulamanın Controller katmanı tarafından karşılanmakta, Service katmanında işlenmekte ve sonrasında repository katmanına iletilmektedir. Repository katmanında, seçilen veri yönetim stratejisine göre (Java In-Memory, Caffeine veya Redis) veriler işlenmektedir.

Java ve Caffeine repository'leri uygulamanın belleğinde çalıştığı için düşük gecikmeli yanıtlar üretmektedir. Redis repository'si ise ağ üzerinden Docker konteynerinde çalışan Redis sunucusuna bağlanmakta ve verileri sıralı küme veri yapısı üzerinde işlemektedir. Bu süreçte performans metrikleri Micrometer aracılığıyla toplanmakta, Prometheus tarafından depolanmakta ve Grafana üzerinden görselleştirilerek kullanıcıya sunulmaktadır.

Şekil 3.3 ayrıca sistemin gözlemlenebilirliğini de açıkça göstermektedir. Docker üzerinde çalışan Prometheus, uygulamadan gelen metrikleri toplamakta; Grafana ise bu metrikleri kullanıcıya anlaşılır paneller halinde sunmaktadır. Böylece farklı repository implementasyonlarının performans sonuçları gerçek zamanlı olarak takip edilebilmektedir.

Sonuç olarak, Şekil 3.3'te gösterilen yapı; yük test aracından başlayan API uç noktaları üzerinden işlenen repository katmanında farklı veri kaynakları üzerinde gerçekleştirilen ve Prometheus ile Grafana üzerinden gözlemlenen bütünleşik bir mimariyi ortaya koymaktadır.

Decorator deseni kullanımı: Projenin gözlemlenebilirliğini artırmak amacıyla Decorator tasarım deseni uygulanmıştır. Decorator deseni mevcut bir nesnenin davranışını değiştirmeden ona yeni işlevler kazandırmaya olanak tanır. Bu kapsamda InstrumentedLeaderboardRepository geliştirilmiştir. Şekil 3.1'de görüldüğü üzere, bu yapı repository katmanının üzerine eklenmiş, Micrometer aracılığıyla performans metriklerinin toplanmasını sağlamıştır. Böylece repository'nin temel işlevleri korunurken metrik toplama gibi ek özellikler de sisteme entegre edilmiştir. Ayrıca ölçüm yaparken HTTP istekleri ve spring kaynaklı gecikmelerin önüne geçilmiştir [8].

Konfigürasyon ve özelleştirilebilirlik: Sistemin farklı veri yönetim çözümlerini destekleyebilmesi için esnek bir yapı kurulmuştur. LeaderboardRepositoryConfig bileşeni, uygulamanın çalışma zamanında hangi repository implementasyonunun kullanılacağını belirlemiştir. Bu yaklaşım yazılım mühendisliğinde Dependency Injection (DI) olarak bilinen yöntem ile gerçekleştirilmiştir. Şekil 3.2'de gösterildiği gibi Service, Leaderboard Interface yapısını çağırır ve DI ile konfigürasyonda önceden belirlenmiş olan repository çağırılır. Böylece sistem farklı test ortamlarında veya performans ihtiyaçlarına göre yalnızca konfigürasyon değişiklikleriyle uyarlanabilir hale gelmiştir [8].

Hata yönetimi ve istisna yakalama: Projede hata yönetimi merkezi olarak ele alınmıştır. GlobalExceptionHandler bileşeni, uygulamada oluşabilecek istisnaları yakalayarak bunların kullanıcıya uygun HTTP yanıtları olarak dönülmesini sağlamıştır. Örneğin, sistemde

bulunmayan bir oyuncuya skor eklenmek istendiğinde PlayerNotFoundException fırlatılmış, kullanıcıya bu durum 404 Not Found olarak bildirilmiştir. Bu sayede sistem hem güvenilirlik hem de kullanıcı deneyimi açısından güçlendirilmiştir [8].

Performans testleri ve gözlemleme: Şekil 3.3'te görüldüğü üzere, sistemin performansını ölçmek ve gözlemlemek amacıyla mimariye ek bileşenler dâhil edilmiştir. Üst kısımda yer alan K6 yük testi aracı, API uç noktalarına eşzamanlı çok sayıda HTTP isteği göndermekte ve sistemin farklı senaryolar altındaki davranışlarını analiz etmektedir.

Performans testleri bir test betiği aracılığıyla gerçekleştirilmiştir. Bu betikte testler başlamadan önce belirli sayıda oyuncu oluşturulmuş, testler sırasında ise okuma ve yazma işlemleri senaryolaştırılmıştır. Testler, aynı anda binlerce sanal kullanıcı çalıştırılarak sistemin dayanıklılığı ölçülmüştür.

K6 tarafından oluşturulan bu yük, uygulamanın Controller ve Service katmanları üzerinden repository katmanına yönlendirilmiş, seçilen repository implementasyonuna göre veriler işlenmiştir. Bu süreçte Micrometer aracılığıyla performans metrikleri toplanmıştır. Micrometer, uygulama içinden elde edilen bu metrikleri Prometheus'un okuyabileceği formatta sunmuştur.

Alt kısımda yer alan Prometheus, uygulamadan gelen metrikleri düzenli aralıklarla çekerek kendi veri tabanına kaydetmiştir. Prometheus'un zaman serisi veri tabanı yapısı, hem geçmiş hem de anlık performans verilerinin saklanmasını sağlamıştır. Grafana ise Prometheus'a bağlanarak bu verileri görselleştirmiştir. Hazırlanan paneller aracılığıyla yanıt süreleri, hata oranları, bellek kullanımı ve istek sayıları gibi önemli metrikler gözlemlenmiştir.

Bu yapı sayesinde, farklı repository çözümlerinin (Java, Caffeine ve Redis) performansları karşılaştırılabilir hale gelmiştir. Redis üzerinde pipelining tekniği uygulanarak ağ üzerinden yapılan çoklu komutların tek seferde gönderilmesi sağlanmış, bu da gecikme sürelerinde önemli iyileştirmeler sunmuştur. Java ve Caffeine çözümleri uygulamanın belleğinde çalıştıkları için daha düşük gecikmeli yanıtlar verirken, Redis kalıcılık ve dağıtık kullanım avantajıyla öne çıkmıştır.

3.3.3. Projede Kullanılan Paketler

Paketler yazılım geliştirmede belirli fonksiyonları sağlayan hazır kod parçacıklarıdır. Tekrar tekrar kullanılabilir ve geliştiriciye önemli faydalar sağlar. Projeye dışardan eklenir. Çeşitli görevleri yerine getirmesi için kullanılır. Projede kullanılan paketler, hem uygulamanın temel işlevlerini yerine getirmesi hem de performans ve gözlemlenebilirlik hedeflerinin karşılanması amacıyla seçilen teknolojiler içindir. Kullanılan bağımlılıklar, pom.xml dosyasında tanımlanmış olup her biri proje için belirli bir ihtiyacı karşılamaktadır.

Spring Boot Starter Web: Projede RESTful API geliştirilmesi için spring-boot-starter-web paketi tercih edilmiştir. Bu bağımlılık, Spring MVC çatısını içermekte olup HTTP isteklerinin karşılanması, JSON veri dönüşümleri ve temel web servis altyapısının sağlanması için kullanılmıştır. Controller katmanında tanımlanan uç noktaların istemciden gelen istekleri alabilmesi bu paket sayesinde mümkün olmuştur.

Springdoc OpenAPI Starter: Uygulamanın dokümantasyonu için springdoc-openapi-starter-webmvc-ui bağımlılığı eklenmiştir. Bu paket sayesinde uygulama uç noktaları otomatik olarak belgelenmiş ve Swagger UI üzerinden test edilebilir hale gelmiştir. Böylece uygulamanın işlevleri hem geliştirici hem de kullanıcı tarafından kolaylıkla gözlemlenebilmiştir.

Spring Boot Starter Data Redis: Veri yönetiminde dağıtık ve kalıcı bir çözüm sağlamak amacıyla spring-boot-starter-data-redis bağımlılığı tercih edilmiştir. Bu paket Redis sunucusuna bağlantı kurmayı ve Redis üzerinde veri işlemleri yapmayı mümkün kılmıştır.

Caffeine Cache: Bellek içi (in-memory) önbellek çözümü için caffeine bağımlılığı kullanılmıştır. Caffeine düşük gecikmeli veri erişimi sağlayan modern bir Java kütüphanesidir. Projede sık erişilen oyuncu verilerinin hızlı bir şekilde tutulması ve okunması için kullanılmıştır.

JNanoid: Oyuncular için benzersiz kimlikler üretmek amacıyla jnanoid bağımlılığı projeye eklenmiştir. Bu kütüphane güvenli ve kısa kimliklerin hızlı bir şekilde oluşturulmasını sağlamış, böylece her oyuncunun sistemde eşsiz bir ID'ye sahip olması garanti edilmiştir.

Micrometer Core ve Micrometer Prometheus Registry: Uygulamanın performansının gözlemlenebilmesi için micrometer-core ve micrometer-registry-prometheus bağımlılıkları kullanılmıştır. Micrometer uygulama metriklerini toplamak için kullanılırken, Prometheus

registry eklentisi sayesinde bu metriklerin Prometheus tarafından toplanabilir formatta sunulması sağlanmıştır. Şekil 3.1'de görüldüğü üzere, toplanan metrikler Prometheus'a aktarılmış ve Grafana üzerinden görselleştirilmiştir.

Spring Boot Starter Actuator: Uygulamanın durumu ve sağlığı hakkında bilgi sağlayan spring-boot-starter-actuator bağımlılığı projeye entegre edilmiştir. Bu kütüphane, hazır uç noktalar aracılığıyla uygulamanın çalışma durumu, bellek kullanımı ve performans verileri hakkında bilgi sunmuştur. Actuator, Micrometer ile entegre çalışarak gözlemleme kapasitesini artırmıştır.

Commons Pool2: Redis bağlantılarında etkin kaynak yönetimi sağlamak için commons-pool2 bağımlılığı kullanılmıştır. Bu kütüphane bağlantı havuzlama mekanizmaları sunarak Redis bağlantılarının verimli kullanılmasını sağlamış ve yüksek yük altındaki sistem performansını artırmıştır.

3.3.4. Projenin İçeriği

Sistemde oyuncu verilerinin tutulması: Sistemde her oyuncu LeaderboardEntry nesnesi aracılığıyla temsil edilmektedir. Bu nesne; oyuncunun kimliği, kullanıcı adı, seviyesi, skoru ve liderlik tablosundaki sırası gibi bilgileri barındırmaktadır [8].

Çizelge 3.1. Lider tablosunda tutulan oyuncu kaydı özellikleri

Alan Adı	Veri Tipi	Açıklama
playerId	String	Oyuncuya sistem tarafından atanan benzersiz kimlik. jNanoid kütüphanesi ile üretilmektedir.
username	String	Oyuncunun kullanıcı adı. Toplu üretim sırasında otomatik, manuel eklemede kullanıcıdan alınmaktadır.
level	Integer	Oyuncunun bulunduğu seviye bilgisi.
score	Double	Oyuncunun mevcut skoru. Skor artırma işlemleriyle güncellenmektedir.
rank	Long	Oyuncunun liderlik tablosundaki mevcut sırası. *GET /leaderboard/{palayadır}/rank* uç noktası üzerinden elde edilmektedir.

Lider tablosunun temsili: Lider tablosu, sistemde kayıtlı oyuncuların tümünü barındıran bir yapı olarak List<LeaderboardEntry> şeklinde tutulmaktadır. Bu yapı, her oyuncunun LeaderboardEntry nesnesi olarak temsil edildiği sıralı bir koleksiyon görevi görmektedir [8].

Başka bir ifadeyle, her bir oyuncu tek başına bir veri nesnesi iken; lider tablosu, bu oyuncuların sıralı bir listesi şeklinde organize edilmektedir. Sıralama kriteri oyuncuların skorlarıdır.

Player ID	Username	Score	Level
player:9582	CrimsonHunter7236	999871	47
player:6130	IronWizard9173	999827	97
player:6616	WildViking3589	999776	95
player:7782	NeptuneNinja4086	999670	146
player:8933	NeonSpecter8764	999589	26
player:5436	GrimFish613	999566	90
player:9958	ShadowRacer2548	999552	78
player:2311	FierceSeeker5099	999457	48
player:8729	NovaSpecter873	999422	81
player:745	EmeraldStalker1118	999419	71
player:816	CrimsonPhoenix7846	999303	90
player:5495	SilentCrusader8948	999126	115
player:403	ZephyrSamurai6569	999120	63

Şekil 3.4. Lider tablosu örnek görüntüsü

Şekil 3.4'te olduğu gibi sistemdeki her oyuncu lider tablosunda bir satırı temsil etmekte, tüm liste bir bütün olarak lider tablosunu oluşturmaktadır. Kullanılan repository implementasyonuna (Java In-Memory, Caffeine veya Redis) bağlı olarak bu listenin elde edilme şekli farklılık göstermektedir, ancak mantıksal temsil değişmemektedir.

REST API uç noktaları: REST API (Representational State Transfer Application Programming Interface), istemci ve sunucu arasında HTTP protokolü üzerinden iletişim kurulmasını sağlayan bir yaklaşımdır. Bu yöntem, modern web ve mobil uygulamalarda yaygın olarak kullanılmaktadır. Projede REST API, oyuncuların oluşturulması, skorlarının güncellenmesi, lider tablosunun görüntülenmesi ve verilerin yönetilmesi için kullanılmıştır.

Çizelge 3.2. Lider tablosu uygulama genel şeması

Uç Nokta (Endpoint)	**HTTP Metodu**	**Açıklama**
/leaderboard/top	GET	En yüksek skora sahip ilk *n* oyuncuyu listeler.
/leaderboard/{playerId}	GET	Belirli bir oyuncunun bilgilerini getirir.

Uç Nokta (Endpoint)	HTTP Metodu	Açıklama
/leaderboard/{playerId}/rank	GET	Belirli bir oyuncunun mevcut sırasını döndürür.
/leaderboard/{playerId}/score/incrementBy	POST	Belirli bir oyuncunun skorunu artırır.
/leaderboard/player	POST	İstemciden gelen bilgilerle tekil bir oyuncu ekler.
/leaderboard/players	POST	Belirtilen sayıda oyuncu oluşturur.
/leaderboard/player	DELETE	Belirli bir oyuncuyu sistemden siler.
/leaderboard/players	DELETE	Sistemdeki tüm oyuncuları siler.

GET /leaderboard/top: Sistemde kayıtlı oyuncular arasından en yüksek skora sahip ilk n oyuncuyu sıralı şekilde döndürmektedir. n parametresi, istemcinin kaç oyuncu görmek istediğini belirler. Yanıt, JSON formatında hazırlanmış LeaderboardEntry nesnelerinden oluşur. Bu işlem, özellikle Redis Sorted Set gibi veri yapılarının hızlı sıralama yeteneği sayesinde düşük gecikme süreleriyle gerçekleştirilmektedir [8].

GET /leaderboard/{playerId}: Belirtilen oyuncunun kimliği (playerId) kullanılarak o oyuncunun bilgileri döndürülmektedir. Eğer oyuncu sistemde mevcut değilse, özel bir hata (PlayerNotFoundException) tetiklenir. Dönen bilgi, oyuncunun kimliği, kullanıcı adı, seviyesi ve mevcut skorunu içermektedir. Bu uç nokta, özellikle testlerde tekil oyuncuların takibini yapmak için önemlidir [8].

GET /leaderboard/{playerId}/rank: Rank ifadesi (sıra), bir oyuncunun skoruna göre liderlik tablosundaki konumunu ifade etmektedir. Bu uç nokta, belirtilen oyuncunun sırasını sayısal olarak döndürmektedir. Örneğin, oyuncu 5000 kişi arasında 12. sıradaysa, istemciye bu değer döner. Bu özellik, rekabetin takibi için kritik öneme sahiptir [8].

POST /leaderboard/{playerId}/score/incrementBy: Belirli bir oyuncunun skorunu artırmak için kullanılmaktadır. İstek gövdesinde JSON formatında IncrementRequestDTO nesnesi bulunur. DTO (Data Transfer Object), istemci ile sunucu arasında veri aktarımını düzenleyen yapılardır. Buradaki DTO, skorun ne kadar artırılacağını belirler. İşlem

tamamlandığında oyuncunun güncellenmiş skoru istemciye döndürülür. Bu uç nokta, özellikle k6 ile yapılan yük testlerinde sıkça çağrılarak gerçekçi bir güncelleme trafiği oluşturmuştur [8].

POST /leaderboard/player: Tekil bir oyuncunun manuel olarak eklenmesini sağlar. İstek gövdesinde LeaderboardEntryDTO nesnesi bulunur; bu nesne oyuncunun kimliği, kullanıcı adı, seviyesi ve başlangıç skorunu içerir. DTO kullanımı, verilerin doğruluğunu ve formatını güvence altına alır [8].

POST /leaderboard/players: Toplu oyuncu oluşturma işlemleri için kullanılır. İstek gövdesinde PlayerGenerationRequestDTO nesnesi gönderilir; bu nesne, oluşturulacak oyuncu sayısını belirtir. PlayerGenerator sınıfı devreye girerek istenen sayıda oyuncuyu üretir. Oyuncu kimlikleri jNanoid kütüphanesi ile rastgele ve benzersiz olarak oluşturulur. Bu uç nokta, testler sırasında on binlerce oyuncunun hızlı bir şekilde oluşturulabilmesi için kritik rol oynamıştır [8].

DELETE /leaderboard/player: İstek parametresinde belirtilen playerId değerine sahip oyuncuyu sistemden siler. Silme işlemi tamamlandığında istemciye bilgilendirici bir mesaj döndürülür [8].

DELETE /leaderboard/players: Sistemde kayıtlı tüm oyuncuları siler. Bu uç nokta genellikle testlerin başlangıcında veya bitişinde sistemin sıfırlanması amacıyla kullanılmaktadır. Silme işlemi sonrası istemciye başarı durumu bildirilir [8].

Oyuncu oluşturma ve ID yönetimi: Oyuncuların toplu olarak oluşturulması için POST /leaderboard/players uç noktası kullanılmıştır. Bu uç noktaya yapılan isteklerde, istemciden gelen PlayerGenerationRequestDTO nesnesi, oluşturulacak oyuncu sayısını belirtmektedir. Controller katmanı, bu isteği aldıktan sonra bilgileri PlayerGenerator sınıfına iletmekte; PlayerGenerator belirtilen sayıda oyuncuyu sistemde üretmektedir [8].
Her oyuncunun kimliği (playerId), kullanıcı adı (username), seviyesi (level) ve başlangıç skoru (score) otomatik olarak belirlenmektedir. Normal koşullarda oyuncular için kimlik üretimi jNanoid kütüphanesi aracılığıyla yapılmıştır. jNanoid, kısa ve güvenli rastgele karakter dizileri üreterek her oyuncunun sistemde benzersiz olarak tanımlanmasını sağlamaktadır. Ancak test senaryolarında jNanoid kullanımı önemli bir sorun doğurmuştur. jNanoid kimlikleri tamamen rastgele olduğundan, on binlerce oyuncu arasından belirli bir oyuncunun seçilip skorunun artırılması mümkün olmamıştır. Rastgele ancak tahmin

edilemez kimlikler nedeniyle test yükleri sırasında oyunculara erişim imkânsız hale gelmiştir [8].

Bu sebeple, yalnızca test amaçlı olmak üzere, oyunculara sıralı ve tahmin edilebilir kimlikler verilmiştir. Örneğin, 10.000 oyuncu oluşturulduğunda kimlikler 0 ile 9999 arasında atanmıştır. Böylece test aracı olan K6, bu aralıktan rastgele bir sayı üreterek herhangi bir oyuncuyu seçebilmiş ve skor artırma işlemini gerçekleştirebilmiştir. Bu yöntem, sistemin gerçekçi koşullar altında test edilmesini mümkün kılmıştır.

```
private static final String[] NAME_PREFIXES = {
        "Swift", "Night", "Golden", "Cyber", "Silent", "Fierce",
        "Crimson", "Lone", "Wild", "Dark", "Pixel", "Ghost",
        "Rapid", "Stealth", "Ancient", "Shadow", "Storm", "Turbo",
        "Drift", "Neon", "Venom", "Inferno", "Bright", "Silver",
        "Iron", "Neptune", "Jade", "Rogue", "Vortex", "Nova",
        "Titan", "Omega", "Lunar", "Solar", "Meteor", "Blaze",
        "Hyper", "Frost", "Glacial", "Cobalt", "Emerald", "Onyx",
        "Vivid", "Zephyr", "Primal", "Abyss", "Astral", "Nova",
        "Grim", "Phantom"
};

2 usages
private static final String[] NAME_SUFFIXES = {
        "Eagle", "Wolf", "Tiger", "Fish", "Hawk", "Dragon",
        "Falcon", "Ninja", "Bot", "Hunter", "Knight", "Phoenix",
        "Raider", "Sniper", "Striker", "Viper", "Gamer", "Racer",
        "Raven", "Wizard", "Sorcerer", "Guardian", "Titan", "Spartan",
        "Viking", "Rogue", "Paladin", "Samurai", "Champion", "Wizard",
        "Shifter", "Stalker", "Crusader", "Patriot", "Seeker",
        "Blaster", "Nomad", "Phantom", "Marauder", "Specter",
        "Ranger", "Cyborg", "Enforcer", "Crusher", "Slayer",
        "Avenger", "Gladiator", "Oracle", "Warden", "Zealot"
};
```

Şekil 3.5. Oyuncu kullanıcı adları için kullanılan diziler görüntüsü

Oyuncu kullanıcı adları (username), Şekil 3.5'te görülen diziler ile rastgele oluşturulmuştur. Bu amaçla kullanılan diziler NAME_PREFIXES ve NAME_SUFFIXES adında iki dizidir. Oyuncu oluşturma sırasında bu dizilerden rastgele birer önek ve sonek seçilmiş, bunlar birleştirilmiş ve sonuna 0 ile 9999 arasında rastgele bir sayı eklenmiştir. Örneğin, "Dark + Slayer + 3721" şeklinde bir oyuncu adı üretilmiştir. Bu yaklaşım sayesinde oyuncu adları çeşitlendirilmiş ve gerçekçi bir görünüm elde edilmiştir.

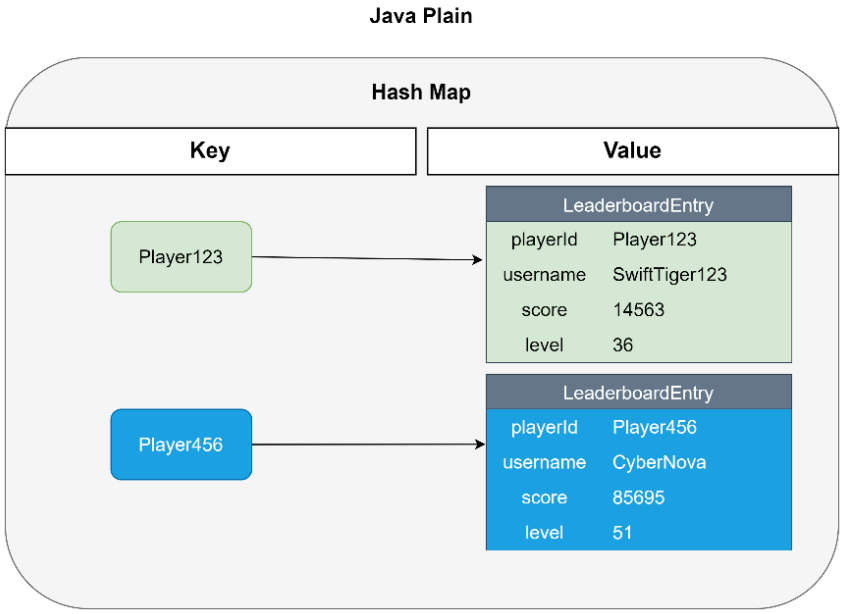
Oyuncuların seviye ve skor değerleri de rastgele belirlenmiştir. Bu değerler, liderlik tablosunun başlangıçta homojen olmayan ve gerçekçi bir şekilde dağıtılmış skorlarla doldurulmasını sağlamıştır. Böylece testlerde farklı seviyelerdeki oyuncuların sıralamadaki davranışları gözlemlenebilmiştir.

Manuel olarak tek bir oyuncu eklenmesi gerektiğinde ise POST /leaderboard/player uç noktası kullanılmıştır. Bu uç nokta, istemciden gelen LeaderboardEntryDTO nesnesi

aracılığıyla bir oyuncunun kimliği, kullanıcı adı, seviyesi ve skoru doğrudan alınarak sisteme eklenmesini sağlamaktadır [8].

Oyuncuların silinmesi için iki farklı uç nokta tanımlanmıştır. DELETE /leaderboard/player uç noktası, parametre olarak verilen playerId değerine sahip tek bir oyuncunun silinmesini sağlamaktadır. DELETE /leaderboard/players uç noktası ise sistemde kayıtlı tüm oyuncuları temizlemektedir. Bu uç nokta, testlerin başlaması veya bitmesi sırasında sistemin sıfırlanması için kullanılmıştır [8].

Lider tablosu verilerinin yönetimi - Java: Java bellek içi çözümü, liderlik tablosu verilerinin herhangi bir harici bağımlılık olmadan doğrudan uygulamanın belleğinde tutulmasına dayanmaktadır. Bu yöntem, özellikle basitliği ve yüksek hızda erişim imkânı sağlaması nedeniyle tercih edilmiştir. Şekil 3.6'da görüldüğü üzere, veriler bir HashMap yapısı kullanılarak saklanmaktadır.



Şekil 3.6. Lider tablosunun Java implementasyonu veri yapısı şeması

HashMap veri yapısında, her bir oyuncunun kimliği (playerId) anahtar (Key) olarak kullanılmakta, karşılık gelen değer (Value) ise oyuncunun bilgilerini içeren LeaderboardEntry nesnesi olmaktadır. Bu nesne, oyuncunun kimliği, kullanıcı adı, seviyesi ve skorunu barındırmaktadır. Örneğin, şemada görüldüğü gibi "Player123" anahtarına karşılık gelen değer, kullanıcı adı "SwiftTiger123", skor değeri 14.563 ve seviye bilgisi 36 olan bir oyuncudur. Benzer şekilde "Player456" anahtarına karşılık gelen LeaderboardEntry

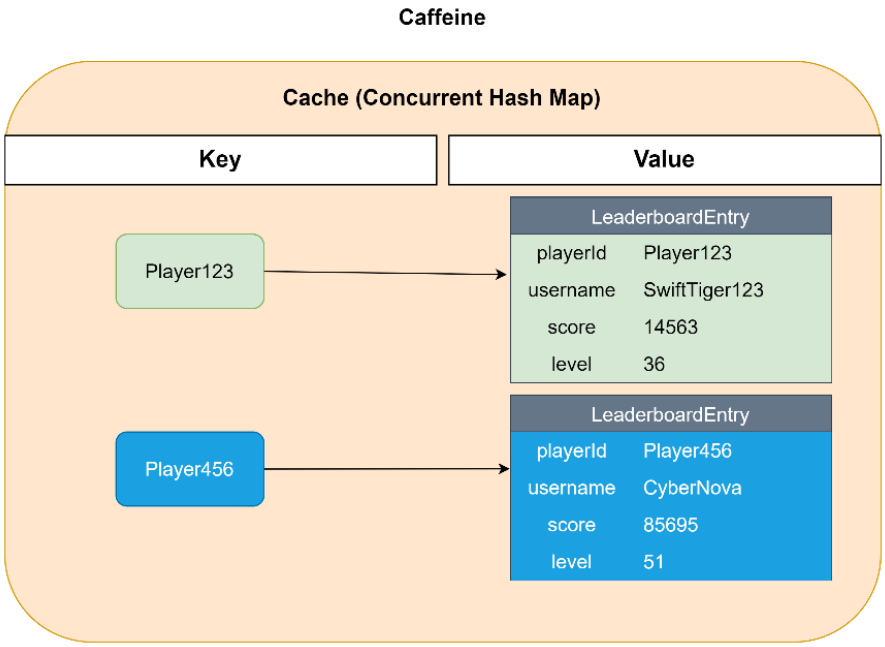
nesnesi, kullanıcı adı “CyberNova”, skor değeri 85.695 ve seviye bilgisi 51 olan bir oyuncuyu temsil etmektedir.

Bu yaklaşım, uygulamanın belleğinde çalıştığı için son derece düşük gecikme süreleriyle veri erişimi sağlamaktadır. Okuma ve yazma işlemleri doğrudan bellekte gerçekleştirildiğinden, herhangi bir ağ gecikmesi yaşanmamaktadır. Bu nedenle performans testlerinde Java çözümü, özellikle hız açısından referans noktası olarak değerlendirilmiştir.

Ancak bu yöntemin belirli sınırlamaları bulunmaktadır. HashMap veri yapısı, verileri eklenme sırasına göre tutmakta, skor değerine göre sıralama yapmamaktadır. Dolayısıyla, lider tablosu bir HTTP isteği ile talep edildiğinde, sistem listenin sıralamasını sonradan gerçekleştirmektedir. Bu işlem için HashMap içeriği alınmakta, skor değerlerine göre sıralanmakta ve istemciye en yüksek skora sahip ilk n oyuncu döndürülmektedir. Yani sıralama işlemi her istek sırasında dinamik olarak yapılmaktadır. Bu, küçük veri kümeleri için performans açısından sorun oluşturmazken, oyuncu sayısının artmasıyla birlikte işlem süresini uzatabilmektedir.

Bir diğer sınırlama, verilerin kalıcı olmamasıdır. Java bellek içi çözümü tamamen bellekte çalıştığı için uygulama yeniden başlatıldığında tüm veriler kaybolmaktadır. Ayrıca, bu yöntem dağıtık kullanım desteği sunmamaktadır; dolayısıyla yalnızca tek bir uygulama örneği üzerinde çalışabilmektedir. Ayrıca bellek tabanlı çalışması sayesinde çok düşük gecikme süreleri sunarken, sıralamanın her istek sırasında yapılması ve verilerin kalıcı olmaması, bu yöntemin temel eksikliklerini oluşturmaktadır.

Lider tablosu verilerinin yönetimi - Caffeine: Caffeine, Java için geliştirilmiş modern ve yüksek performanslı bir önbellekleme (cache) kütüphanesidir. Projede lider tablosu verilerinin düşük gecikme süreleriyle saklanabilmesi ve sık erişilen verilere hızlı erişim sağlanabilmesi için tercih edilmiştir. Şekil 3.7’de görüldüğü üzere, Caffeine önbellek yapısı aslında ConcurrentHashMap temeli üzerinde çalışmakta, her bir oyuncu LeaderboardEntry nesnesi olarak önbellek içerisinde tutulmaktadır.



Şekil 3.7. Lider tablosunun Caffeine implementasyonu veri yapısı şeması

Java yöntemine benzer şekilde, burada da her oyuncunun benzersiz kimliği (playerId) anahtar (Key) olarak kullanılmakta, karşılık gelen değer ise oyuncuya ait bilgileri içeren LeaderboardEntry nesnesi olmaktadır. Örneğin şemada görüldüğü üzere, “Player123” kimliğine sahip oyuncu kullanıcı adı “SwiftTiger123”, skor değeri 14.563 ve seviye bilgisi 36 ile temsil edilmektedir.

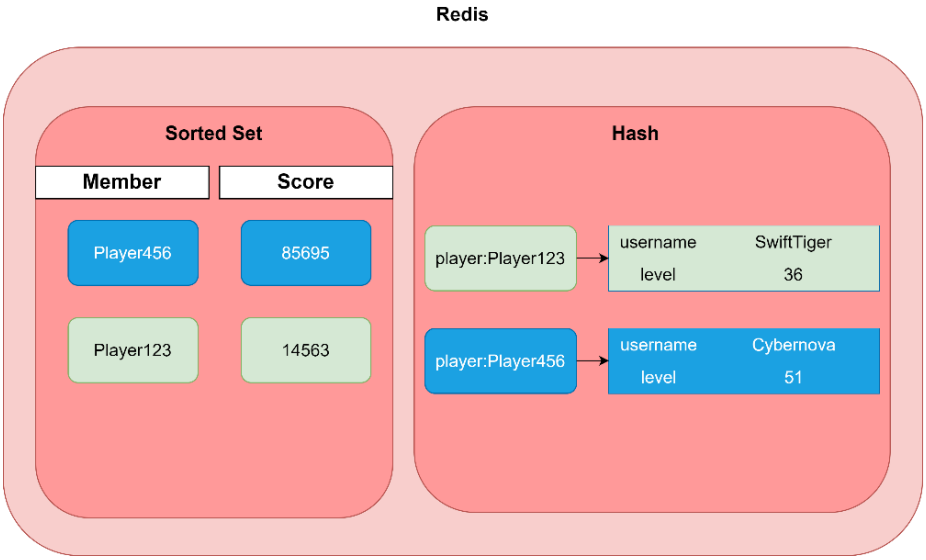
Caffeine’i Java bellek içi yönteminden ayıran önemli özelliklerden biri, bellek yönetimini daha akıllı hale getiren mekanizmalardır. Bunlardan ilki Time-To-Live (TTL) özelliğidir. TTL sayesinde belirli bir süre boyunca kullanılmayan veya güncellenmeyen veriler otomatik olarak bellekten temizlenmektedir. Ayrıca Caffeine, maksimum boyut sınırı tanımlamaya da imkân vermektedir. Bu sınır aşıldığında, en az kullanılan (least recently used) veriler otomatik olarak silinmekte ve yeni verilere yer açılmaktadır. Bu özellikler sayesinde bellek tüketimi dengede tutulabilmektedir.

Caffeine’in en güçlü yanlarından biri, ConcurrentHashMap üzerine inşa edilmiş olmasıdır. Bu sayede, çoklu iş parçacıkları (thread) tarafından eşzamanlı erişim sağlandığında dahi veri bütünlüğü korunabilmektedir. Java yöntemine benzer şekilde bellek içi çalıştığı için hızlı yanıt süreleri sunarken, aynı anda binlerce oyuncunun skorunun güncellenmesi veya okunması durumunda dahi çakışma veya veri kaybı yaşanmamaktadır.

Bununla birlikte, Java yaklaşımında olduğu gibi Caffeine'de de veriler eklenme sırasına göre tutulmakta, skor değerine göre otomatik sıralama yapılmamaktadır. Dolayısıyla, lider tablosu bir HTTP isteğiyle talep edildiğinde, önbellek içeriği alınmakta ve skor değerlerine göre sıralandıktan sonra istemciye sunulmaktadır. Bu işlem küçük ölçekli veri kümelerinde fark edilebilir bir gecikme yaratmazken, oyuncu sayısının artması durumunda sıralama süresi uzayabilmektedir. Bununla birlikte, Caffeine de tıpkı Java bellek içi yaklaşımı gibi kalıcılık sunmamaktadır; sistem yeniden başlatıldığında tüm veriler kaybolmaktadır.

Özet olarak Caffeine teknolojisi kullanımı, Java yöntemine benzer bir temel üzerinde çalışsa da TTL ve maksimum boyut sınırı gibi ek özellikleriyle bellek yönetimini daha etkin kılmıştır. ConcurrentHashMap tabanlı yapısı sayesinde eşzamanlı erişimlerde güvenilir sonuçlar sağlamış, düşük gecikmeli performansı ile liderlik tablosu uygulamasında önemli bir alternatif olmuştur.

Lider tablosu verilerinin yönetimi – Redis: Redis, bellek içi çalışan, yüksek performanslı bir veri tabanı çözümüdür. Liderlik tablosu uygulamasında Redis tercih edilmesinin temel nedeni, verilerin sıralı bir şekilde saklanabilmesi ve sistemin dağıtık mimarilerde güvenilir bir şekilde çalışabilmesidir. Ancak Redis, Java veya Caffeine yöntemlerinde olduğu gibi doğrudan Java nesnelerini tutabilecek bir yapı sunmamaktadır. Redis'te tüm veriler atomik seviyedeki veri yapıları aracılığıyla saklanmaktadır. Bu nedenle, verilerin doğru ve işlevsel bir şekilde yerleştirilebilmesi için birden fazla yapı birlikte kullanılmıştır.



Şekil 3.8. Lider tablosunun Redis implementasyonu veri yapısı şeması

Şekil 3.6'da görüldüğü üzere, projede Redis üzerinde iki temel yapı bir arada kullanılmıştır. Bunlardan ilki sıralı küme (Sorted Set - ZSET) yapısıdır. Bu yapıda yalnızca oyuncunun

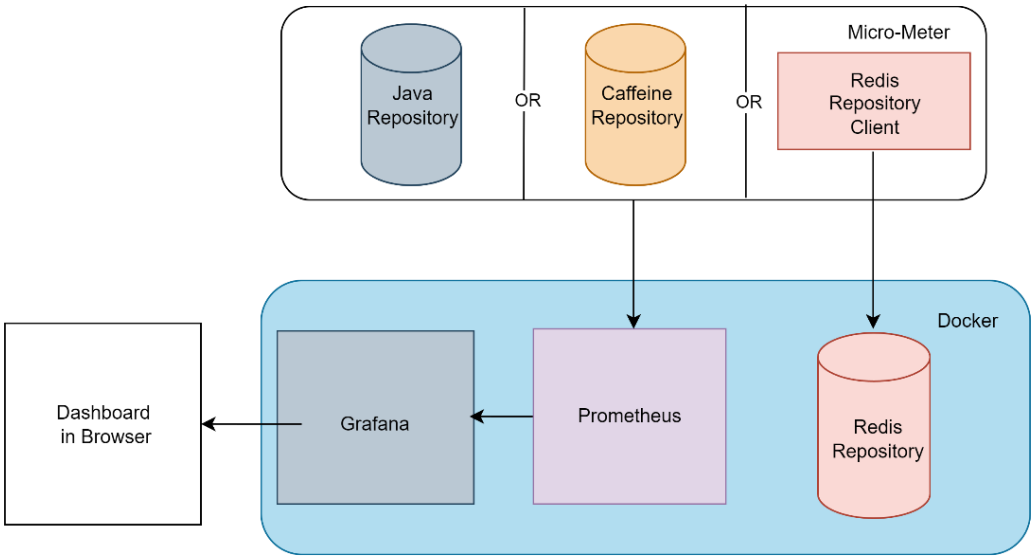
kimliği (playerId) ve skoru saklanmaktadır. Sıralı küme skor değerlerine göre elemanları sıralı olarak tuttuğu için lider tablosunun doğrudan sıralı halde elde edilmesini sağlamaktadır. Bu özellik sayesinde, Java ve Caffeine yöntemlerinde olduğu gibi her HTTP isteğinde listenin baştan sıralanmasına gerek kalmamaktadır. İstemci en yüksek skora sahip ilk n oyuncuyu talep ettiğinde, Redis doğrudan sıralı veriyi döndürmektedir.

Bununla birlikte sıralı küme yalnızca kullanıcı kimliği ve skor alanlarını sakladığından, oyuncuya ait kullanıcı adı (username) ve seviye (level) bilgileri ayrı bir yapıda tutulmaktadır. Bunun için Redis'in Hash veri yapısı kullanılmıştır. Hash yapısında, her oyuncunun player:playerId şeklinde tanımlanan anahtarları bulunmaktadır. Bu anahtarların altında, ilgili oyuncunun kullanıcı adı ve seviye bilgileri alan-değer (field-value) çiftleri olarak saklanmaktadır. Örneğin, player:Player123 anahtarının altında username = SwiftTiger123 ve level = 36 bilgileri tutulmaktadır.

Bu yaklaşım, sıralama maliyetini sıralı küme üzerinden ortadan kaldırırken, aynı zamanda oyuncunun ek bilgilerine ulaşabilmek için sıralı kümedeki playerId üzerinden Hash'e başvurulması gerekliliğini doğurmaktadır. Yani lider tablosu istenirken önce sıralı kümeden en yüksek skora sahip n oyuncunun kimlikleri alınmakta, ardından bu kimlikler kullanılarak Hash yapılarından ilgili kullanıcı adı ve seviye bilgileri çekilmektedir. Dolayısıyla, sıralama avantajı sağlanırken, Hash üzerinden ek alanlara erişim için ek bir işlem maliyeti ortaya çıkmaktadır.

Redis'in bu iki yapıyı birlikte kullanması bazı ek yükler getirse de, sistemin güvenilirliğini ve sürdürülebilirliğini artırmaktadır. Redis, veri kaybetmeme özelliği (persistence) sayesinde uygulama yeniden başlatıldığında verilerin korunmasını sağlamaktadır. Ayrıca Redis, dağıtık mimarilerde birden fazla düğüm üzerinde çalıştırılabildiği için, yüksek kullanıcı yükünü karşılayacak ölçeklenebilirlik sunmaktadır.

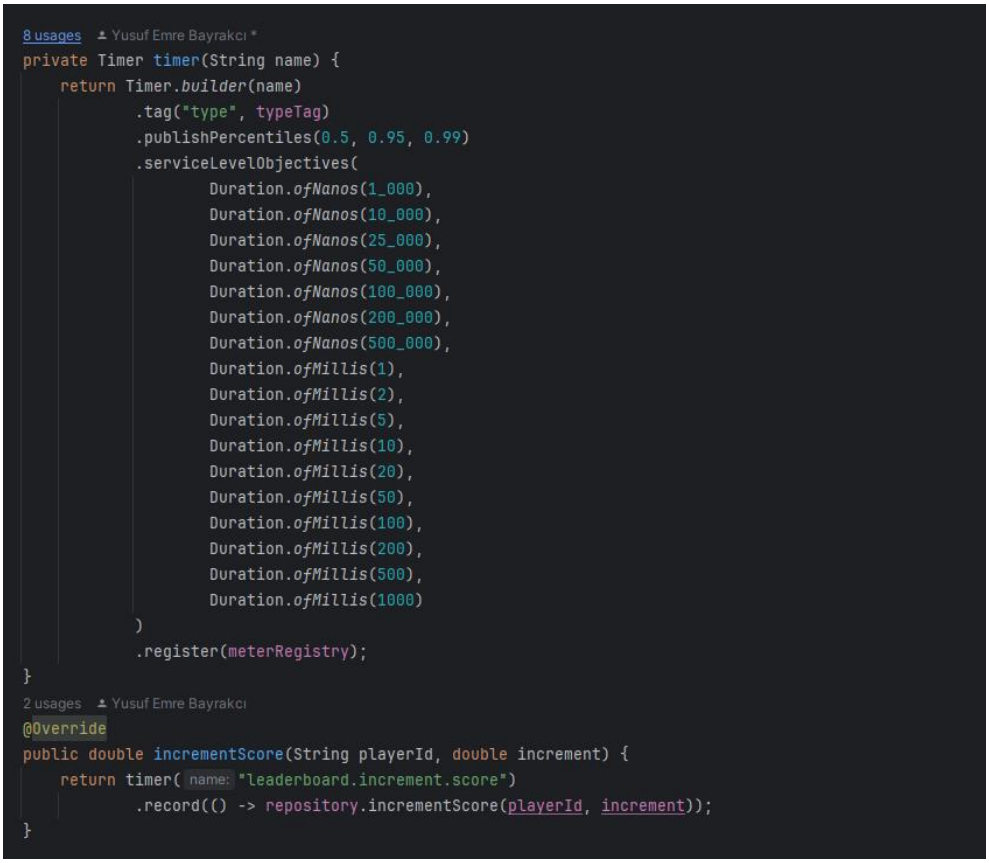
Gözlemleme ve İzleme Altyapısı: Projede yalnızca fonksiyonel gereksinimlerin karşılanması değil, aynı zamanda sistemin performansının sürekli gözlemlenebilmesi de hedeflenmiştir. Bunun için Micrometer, Prometheus ve Grafana teknolojileri bir arada kullanılmıştır. Docker Compose aracılığıyla bu bileşenler entegre edilmiş, böylece sistemin gözlemlenebilirliği yüksek bir seviyeye çıkarılmıştır.



Şekil 3.9. Gözlemleme altyapısı şeması

Micrometer ile Fonksiyon Bazlı Ölçüm: Projede performans ölçümlerinin sağlıklı yapılabilmesi için Micrometer kütüphanesi kullanılmıştır. Buradaki amaç, sistemin çekirdek işlevlerinin ne kadar sürede tamamlandığını gözlemlemek olmuştur. Bu nedenle ölçümler yalnızca repository katmanında yapılmıştır; HTTP isteklerinin iletilmesi, JSON serileştirme veya ağ gecikmeleri gibi dış etkenler ölçüme dahil edilmemiştir. Böylece elde edilen veriler, yalnızca iş mantığının performansını yansıtmaktadır.

Bu işlevi yerine getirmek için projeye InstrumentedLeaderboardRepository sınıfı eklenmiştir. Bu sınıf, Decorator deseni ile mevcut repository implementasyonlarının (Java In-Memory, Caffeine veya Redis) üzerine oturtulmuştur. Yani hangi repository türü seçilirse seçilsin, yapılan işlemler aynı yöntemle ölçülmüştür [8].



Şekil 3.10. InstrumentedLeaderboardRepository Micrometer kullanımı görüntüsü

Sınıfın içinde ölçümler için bir Timer hazırlanır. Timer, ölçülecek işlemin adını (örneğin leaderboard.increment.score), repository türünü belirten etiketi (typeTag) ve kullanılacak histogram aralıklarını alarak oluşturulur. Bu yapılandırma, ölçümlerin hem ayrıntılı hem de karşılaştırılabilir olmasını sağlamaktadır [8].

Fonksiyonların ölçülmesi, timer.record yapısı ile gerçekleştirilir. Örneğin, bir oyuncunun skorunu artırma işlemi incrementScore fonksiyonu çağrıldığında Ölçüm süreci şu şekilde işlemektedir: İlk olarak Timer başlatılmakta, ardından repository üzerinde skor artırma işlemi gerçekleştirilmektedir. İşlem tamamlandığında Timer kaydı durdurulmakta ve elde edilen süre Micrometer tarafından ilgili histogram aralıklarına yerleştirilmektedir.

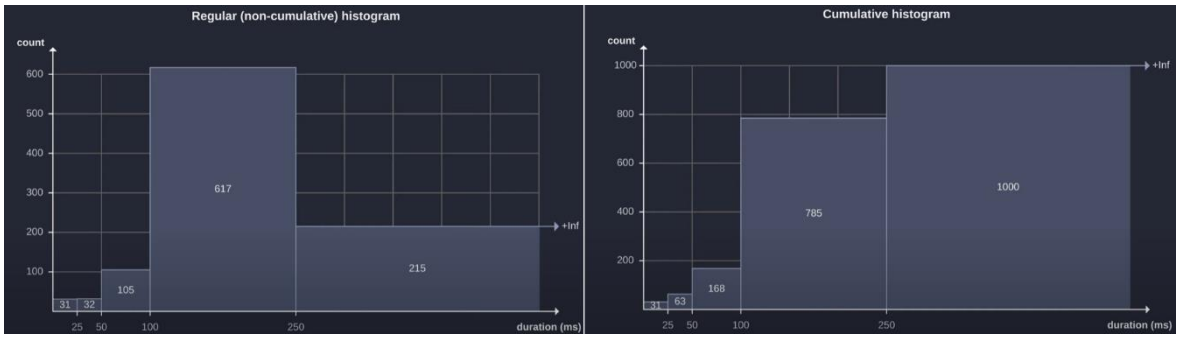
Benzer şekilde diğer fonksyionlar da aynı mekanizma ile ölçülmektedir.

Micrometer, ölçülen tüm süreleri arkada toplamakta ve Prometheus'un anlayabileceği formatta sunmaktadır. Prometheus özel olarak belirlenmiş bir saniyelik aralıklarla bu metrikleri sorgulamakta, böylece her fonksiyonun performansı neredeyse gerçek zamanlı olarak gözlemlenebilmektedir. Daha sonra Grafana üzerinden hazırlanan panellerde bu

veriler grafikler halinde görselleştirilmiş, repository çözümleri arasında detaylı karşılaştırmalar yapılabilmiştir.

Histogram aralıkları gecikme ve verim ölçümü: Projede gecikme ölçümlerinde histogram kovaları yaklaşımı tercih edilmiştir. Bunun temel nedeni, her isteğin gecikmelerini tek tek ölçmenin sistem üzerinde büyük bir yük oluşturmasıdır. Bunun yerine, ölçülen değerler önceden tanımlanmış aralıklara (bucket) yerleştirilmiş, böylece sistemin performans dağılımları verimli ve doğru bir şekilde gözlemlenebilmiştir.

Prometheus'un histogram aralıkları yığılımlı (kümülatif) mantıkla çalışmaktadır. Bu, her aralığın yalnızca kendi aralığını değil, o aralığa kadar olan tüm değerleri kapsadığı anlamına gelmektedir. Örneğin ≤100 ms aralığındaki sayı, 100 ms veya daha kısa sürede tamamlanan tüm isteklerin toplamını içermektedir.



Şekil 3.11. Kümülatif ve kümülatif olmayan histogramların karşılaştırılması görüntüsü [22]

Şekilde görüldüğü üzere, soldaki normal histogram yalnızca ilgili aralıktaki istek sayısını gösterirken, sağdaki kümülatif histogram her aralığın altındaki toplam istek sayısını göstermektedir. Bu yaklaşım sayesinde, belirli eşiklerin altında kalan istek oranları kolayca hesaplanabilmektedir.

Bu kümülatif mantık, yüzdelik gecikme değerlerinin hesaplanmasında doğrudan kullanılmaktadır. Prometheus'un histogram_quantile fonksiyonu, aralık dağılımlarını kullanarak p50 (median), p90 ve p99 gibi yüzdelik değerleri hesaplayabilmektedir. Örneğin p50 gecikmesi, tüm isteklerin %50'sinin daha kısa sürede tamamlandığı noktayı gösterirken p90 ve p99 değerleri, sistemde nadiren görülen uzun gecikmeleri tespit etmeye yardımcı olmaktadır. Bu yöntem sayesinde yalnızca ortalama gecikmeleri değil, gecikme dağılımının uç noktaları da analiz edilebilmiştir.

Projede kullanılan histogram aralıkları, ölçümlerin hem hassas hem de anlamlı olmasını sağlayacak şekilde seçilmiştir. Çok kısa sürede tamamlanan işlemler için mikrosaniye düzeyinde dar aralıklar tanımlanmış; daha uzun sürebilen işlemler için milisaniye seviyesinde geniş aralıklar belirlenmiştir.

Çizelge 3.3. Projede kullanılan histogram kovaları aralıkları

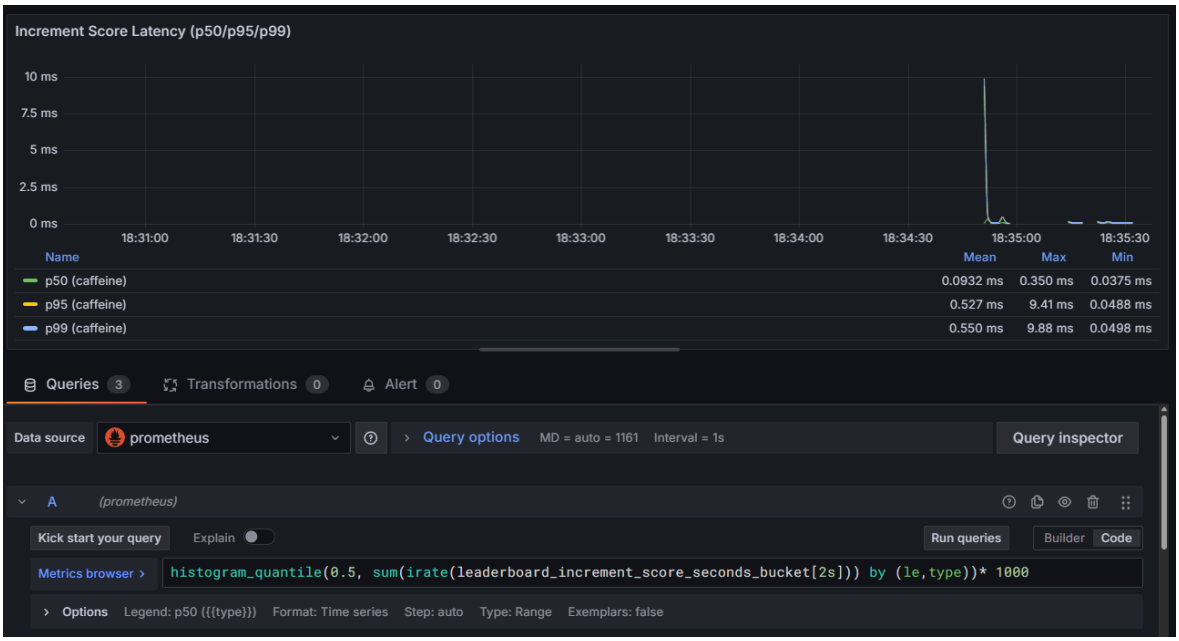
Aralık Değeri	**Açıklama**
1 µs	Mikro düzeyde en hızlı işlemler
10 µs	Mikro saniye ölçeğinde işlemler
25 µs	Düşük gecikmeli işlemler
50 µs	Alt milisaniye seviyesinde işlemler
100 µs	Alt milisaniye seviyesinde
200 µs	Daha uzun mikro işlemler
500 µs	Milisaniyeye yaklaşan işlemler
1 ms	Milisaniye düzeyinde
2 ms	Küçük gecikmeli işlemler
5 ms	Orta düzey gecikmeli işlemler
10 ms	Yüksek çağrı yoğunluğu altında
20 ms	Orta-yüksek gecikmeli işlemler
50 ms	Yük altında belirgin gecikmeler
100 ms	Daha ağır yük altında
200 ms	Uzun süren işlemler için
500 ms	Yüksek gecikmeli nadir işlemler
1000 ms	En uzun gözlemlenen işlemler

Bu optimizasyon sayesinde sistemin hem hızlı hem de nadiren görülen yavaş işlemleri yakalanabilmiş; test sonuçları çok daha doğru ve karşılaştırılabilir hale gelmiştir.

PromQL ile Gecikme ve Verim Ölçümleri: Projede toplanan metrikler Prometheus'un sorgu dili PromQL kullanılarak analiz edilmiştir. Böylece sistemin yalnızca ortalama davranışı değil, yüzdelik gecikmeleri ve işlem verimi de hesaplanabilmiştir.

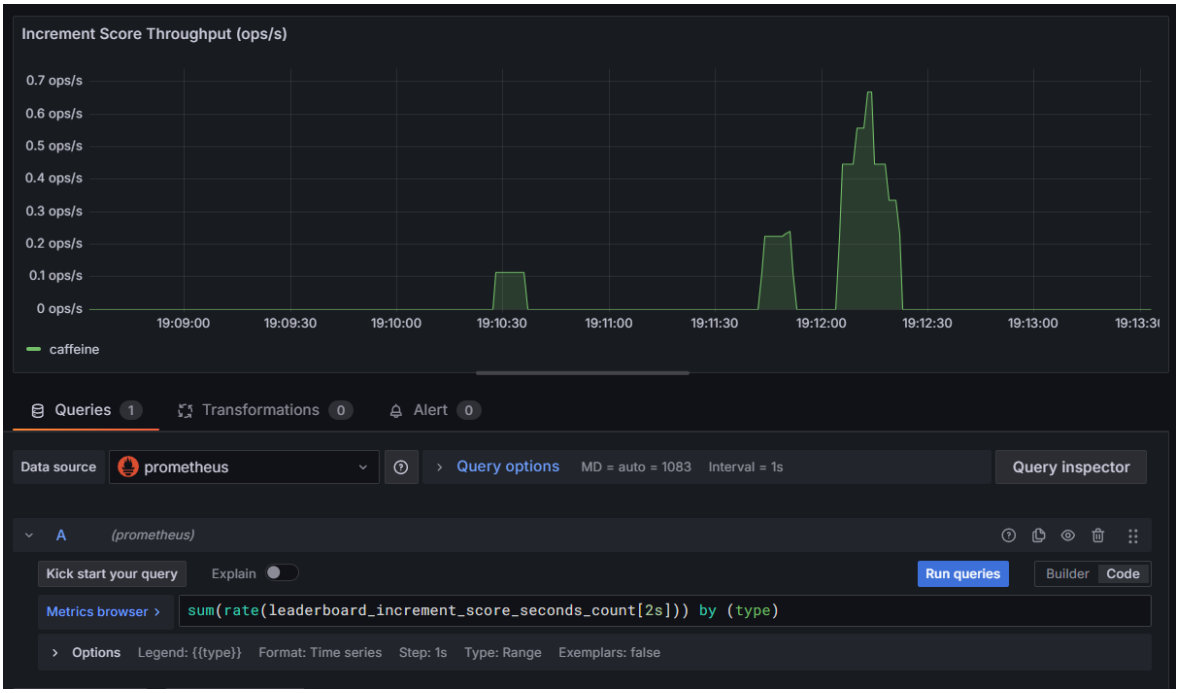
Gecikme ölçümleri için şekil 3.12'de görüldüğü üzere histogram_quantile fonksiyonu kullanılmıştır. Bu fonksiyon, Prometheus'un kümülatif histogram aralık verilerini temel alarak yüzdelik değerleri hesaplamaktadır. Projede p50 (median), p95 ve p99 değerleri elde edilmiştir.

Ölçümlerde irate fonksiyonu da kullanılmıştır. Prometheus her 1 saniyede bir metrik topladığı için, doğru sonuçlar alınabilmesi amacıyla 2 saniyelik pencere seçilmiş, böylece iki nokta üzerinden anlık artış hızı hesaplanmıştır. Ayrıca sorgular repository türüne göre ayrıştırılarak Java Caffeine ve Redis çözümleri ayrı ayrı gözlemlenmiştir.



Şekil 3.12. Grafana panelinde gecikme (p50, p95, p99) ölçümleri örnek görüntüsü

Verim ölçümleri için ise rate fonksiyonu kullanılmıştır. Bu fonksiyon belirli bir zaman aralığında kaç işlem yapıldığını hesaplamaktadır. Projede saniye başına yapılan skor artırma işlemleri hesaplanarak repository çözümlerinin verim değerleri elde edilmiştir. Böylece hangi çözümün yüksek yük altında daha fazla işlem gerçekleştirebildiği net bir şekilde ortaya konmuştur.



Şekil 3.13. Grafana panelinde verim (throughput) ölçümleri örnek görüntüsü

Grafana Panelleri: Projede Prometheus tarafından toplanan metrikler Grafana üzerinden görselleştirilmiştir. Böylece sistemin performansı yalnızca sayısal değerler üzerinden değil, kullanıcıya anlaşılır grafikler halinde sunulabilmiştir.

Grafana panelleri, repository türlerine göre ayrıştırılmış metrikleri gösterecek şekilde yapılandırılmıştır. Bu sayede Java bellek içi, Caffeine ve Redis çözümleri aynı paneller üzerinde yan yana incelenebilmiştir. Kullanıcı, repository implementasyonlarının farklı yük koşullarındaki davranışlarını hem gecikme hem de verim açısından doğrudan gözlemleyebilmiştir.

Gecikme değerleri için hazırlanan panelde, histogram aralıkları üzerinden hesaplanan p50, p95 ve p99 latency değerleri çizdirilmiştir. Şekil 3.12'de görüldüğü üzere, bu görselleştirme sayesinde sistemin hem tipik gecikmeleri hem de nadir görülen uzun gecikmeleri net bir biçimde analiz edilebilmiştir. Grafana'nın sağladığı ek bir özellik olarak, bu yüzdelik değerler için ortalama (mean), en düşük (min) ve en yüksek (max) değerler de tutulabilmektedir. Böylece sistemin zaman içindeki dalgalanmaları, yalnızca yüzdelikler üzerinden değil, istatistiksel özetler aracılığıyla da daha net anlaşılabilmektedir.

Verim ölçümleri için hazırlanan panellerde ise saniye başına gerçekleştirilen skor artırma işlemleri gösterilmiştir. Şekil 3.13'te görüldüğü üzere, verim değerleri sayesinde farklı hafıza çözümlerinin yoğun yük altındaki üretkenliği karşılaştırılmıştır.

Test Ortamı: Projede gerçekleştirilen performans testleri, belirli donanım ve yazılım koşulları altında yürütülmüştür. Bu koşulların açıkça belirtilmesi, elde edilen sonuçların güvenilirliği ve tekrarlanabilirliği açısından önem taşımaktadır.

Testler, HP ZBook Fury G16 model bir iş istasyonunda gerçekleştirilmiştir. Cihaz, Intel Core i7-13850HX işlemciye, 64 GB kapasiteye sahip 5600 MT/sn hızında belleğe sahiptir ve Windows 10 Enterprise işletim sistemi üzerinde çalışmaktadır. Bu güçlü donanım, yüksek eşzamanlılık altında sistemin performansının sağlıklı biçimde ölçülmesine olanak tanımıştır.

Yazılım ortamında Java 21 sürümü kullanılmıştır. Redis, 8.0.2 sürümü ile Docker 4.43.1 üzerinden konteyner olarak çalıştırılmıştır. Caffeine kütüphanesi 3.2.2 sürümüyle doğrudan uygulamanın belleğinde kullanılmış; Java çözümü ise standart ConcurrentHashMap tabanlı implementasyon ile uygulanmıştır.

Docker Compose altyapısı, Redis'in yanı sıra Prometheus ve Grafana servislerini de aynı ağ üzerinde çalıştırmak için yapılandırılmıştır. Bu sayede Micrometer tarafından üretilen metrikler Prometheus aracılığıyla düzenli aralıklarla toplanmış, Grafana panelleri üzerinden görselleştirilmiştir.

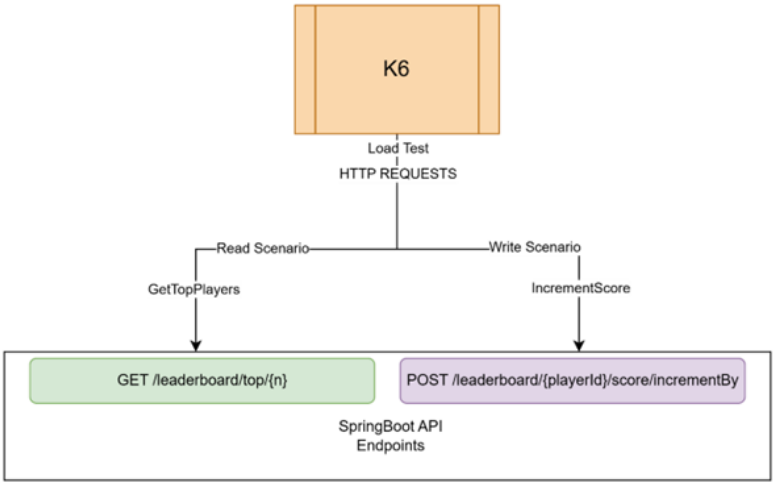
Test Senaryosu: Performans testlerinin temel amacı, lider tablosu uygulamasının yoğun yük altındaki davranışlarını gözlemlemek ve özellikle yazma işlemlerinde kayıp olmadan sistemin ne kadar okuma isteğini karşılayabildiğini belirlemek olmuştur. Çünkü lider tablolarında skor güncellemeleri kritik öneme sahiptir; bir yazma kaybı sistemin güvenilirliğini doğrudan etkilemektedir. Buna karşılık, okuma isteklerinde yaşanabilecek gecikmeler kısmen tolere edilebilir; kullanıcıya lider tablosunun birkaç milisaniye geç gösterilmesi kabul edilebilir bir durumdur.

Testler, k6 yük testi aracı kullanılarak gerçekleştirilmiştir. Her test başlangıcında 10 000 oyuncu oluşturulmuş, ardından farklı yük senaryoları uygulanmıştır. Senaryolar üç aşamadan oluşmaktadır:

1. Senaryo: 500 yazma, değişken okuma yükü: Bu aşamada saniyede 500 skor güncelleme isteği gönderilmiş, eşzamanlı olarak sistemin kaldırabileceği maksimum okuma yükü ölçülmüştür. Amaç, kayıpsız 500 yazma işlemi yapılırken sistemin ne kadar okuma isteğini işleyebildiğini belirlemektir.

2. Senaryo: 1000 yazma, değişken okuma yükü: Bu aşamada yazma yükü iki katına çıkarılarak saniyede 1000 skor güncelleme isteği gönderilmiştir. Sistem üzerindeki yük artışı ile birlikte okuma kapasitesinde yaşanabilecek değişimler gözlemlenmiş, yine yazma kaybı olmadan işlenebilecek okuma miktarı belirlenmiştir.

3. Senaryo: Tam yük testi: Bu aşamada sistem hem yüksek okuma hem de yüksek yazma yükü altında test edilmiştir. Amaç, sistemin sınır koşullarındaki davranışlarını ve gecikme dağılımlarını analiz etmektir. Bu amaçla 2000 yazma ve 10 000 okuma isteği gönderilmiştir. Bu senaryo, repository çözümlerinin (Java In-Memory, Caffeine, Redis) dayanıklılık ve ölçeklenebilirlik sınırlarını ortaya koymuştur.



Şekil 3.14. K6 yük testi senaryolarının genel akış şeması

Şekilde görüldüğü üzere, k6 aracı istemciyi simüle ederek Spring Boot uygulamasına HTTP istekleri göndermektedir. Testlerde iki temel senaryo uygulanmıştır:

Okuma senaryosu, lider tablosunun en iyi oyuncularını listeleyen uç noktaya yapılan isteklerden oluşmaktadır. Bu amaçla GET /leaderboard/top/{n} uç noktası kullanılmıştır. Bu senaryoda belirli sayıda oyuncunun sıralaması istenmiş, sistemin yüksek okuma trafiği altında ne kadar hızlı ve doğru yanıt verebildiği gözlemlenmiştir [8].

Yazma senaryosu ise, oyuncuların skorlarının artırılmasına dayanmaktadır. Bunun için POST /leaderboard/{playerId}/score/incrementBy uç noktası tercih edilmiştir. Bu uç nokta üzerinden belirli oyuncuların skorları artırılmış, sistemin yüksek yazma trafiği altında ne kadar verimli çalıştığı ve verilerin doğru şekilde güncellenip güncellenmediği ölçülmüştür [8].

Senaryolar sırasında belirli sayıda eşzamanlı sanal kullanıcı simüle edilmiştir. Bu kullanıcıların bir bölümü yalnızca okuma işlemleri yaparken, diğer bölümü yazma işlemleri gerçekleştirmiştir. Böylece sistem hem yoğun sorgu altında hem de sürekli güncellemelerle birlikte test edilmiştir.

Ölçülen Metrikler: Gecikme ölçümleri sistemin yanıt verme hızını değerlendirmek amacıyla tercih edilmiştir. Burada yalnızca ortalama gecikme değil, yüzdelik değerler (p50, p95, p99) dikkate alınmıştır. Ortalama değer sistemin genel performansını gösterirken, p95 ve p99 değerleri nadiren yaşanan ancak kullanıcı deneyimini olumsuz etkileyebilecek uzun gecikmeleri ortaya çıkarmıştır. Böylece sistemin sadece tipik değil, uç senaryolardaki davranışı da gözlemlenebilmiştir.

Verim (throughput) ölçümleri, sistemin yoğun yük altındaki üretkenliğini değerlendirmek için kullanılmıştır. Lider tablosu gibi sürekli güncellenen bir yapıda yalnızca hızlı olmak yeterli değildir; aynı zamanda saniye başına yüksek sayıda işlem işlenebilmesi gerekir. Bu nedenle verim sistemin yazma kaybı olmadan ne kadar okuma yükünü kaldırabildiğini belirlemede temel bir gösterge olmuştur.

Bu iki metrik birlikte ele alındığında, sistemin hem hız hem de kapasite açısından bütüncül bir değerlendirmesi yapılabilmiştir. Gecikme ölçümleri, yanıt sürelerini uç değerler dahil olmak üzere ortaya koyarken; verim değerleri, aynı koşullarda ne kadar fazla işlemin kayıpsız gerçekleştirilebildiğini göstermiştir. Böylece farklı hafıza çözümlerinin güçlü ve zayıf yönleri somut verilerle kıyaslanabilir hale gelmiştir.

Test Sonuçları: Gerçekleştirilen performans testleri sonucunda elde edilen gecikme (latency) ve verim (throughput) değerleri Çizelge 3.4, Çizelge 3.5 ve Çizelge 3.6'da sunulmuştur. Çizelge 3.4'te ölçümler, 10 000 oyuncunun bulunduğu bir ortamda saniyede 500 yazma işlemi kayıpsız olarak gerçekleştirilirken yapılmıştır. Çizelge 3.5'te ölçümler, 10 000 oyuncunun bulunduğu bir ortamda saniyede 1000 yazma işlemi kayıpsız olarak gerçekleştirilirken yapılmıştır. Çizelge 3.6'da ise ölçümler, 10 000 oyuncunun bulunduğu bir ortamda saniyede 2000 yazma ve 10000 okuma isteği gönderildiğinde maksimum gerçekleştirilebilen işlem baz alınarak yapılmıştır.

Testler, bir sunucu ortamında değil, kişisel bir iş istasyonunda gerçekleştirilmiştir. Bu nedenle donanımın arka planda çalıştırdığı diğer işlemler gecikme değerlerinde küçük

dalgalanmalara yol açabilmiştir. Bu etkiyi azaltmak ve sonuçların güvenilirliğini artırmak amacıyla her senaryo birden fazla kez tekrarlanmış ve elde edilen değerler raporlanmıştır.

Çizelge 3.4. Saniyede 500 yazma ve kayıpsız okuma gecikme ve verim değerleri

SENARYO	SKOR ARTTIRMA (YAZMA SENARYOSU)					İLK 100 OYUNCU SORGULAMA (OKUMA SENARYOSU)				
YÖNTEM	%	Ortalama (ms)	En yüksek (ms)	En düşük (ms)	Verim (işlem/s)	%	Ortalama (ms)	En yüksek (ms)	En düşük (ms)	Verim (işlem/s)
JAVA	p50	0,000709	0,00569	0,0005	500	p50	3,51	3,98	3,33	7300
	p95	0,00294	0,00991	0,00095		p95	6,88	122	4,86	
	p99	0,00452	0,0441	0,00099		p99	9,86	198	4,98	
CAFFEINE	p50	0,00624	0,0332	0,00486	500	p50	3,53	3,72	3,51	6250
	p95	0,0197	0,0692	0,015		p95	8,39	142	4,87	
	p99	0,0305	0,17	0,023		p99	12,7	188	4,99	
REDIS	p50	1,04	1,33	0,661	500	p50	18,1	58,5	2,18	4000
	p95	2,62	4,53	1,88		p95	54,3	151	4,89	
	p99	6,25	37,5	2,5		p99	86,8	194	8,42	

Çizelge 3.4'e göre Java bellek içi çözümü, en düşük gecikme sürelerini sunarak öne çıkmaktadır. Skor artırma işlemlerinde p50 değeri yalnızca 0,000709 ms olarak ölçülmüş, p95 ve p99 değerleri de mikro saniye seviyelerinde kalmıştır. Bu sonuç, Java bellek içi yaklaşımının çalışma prensibi sayesinde yazma işlemlerinde son derece hızlı olduğunu göstermektedir. Okuma tarafında p50 değeri 3,51 ms, p95 değeri 6,88 ms ve p99 değeri 9,86 ms olarak kaydedilmiş, sistem saniyede yaklaşık 7 300 sorguyu kayıpsız olarak işleyebilmiştir. Bu veriler, Java çözümünün yüksek verimlilikte çalışabildiğini ortaya koymaktadır. Ancak bu çözümde veriler yalnızca bellekte tutulduğundan, sistem yeniden başlatıldığında tüm verilerin kaybolacağı göz önünde bulundurulmalıdır.

Caffeine çözümü, Java'ya kıyasla daha yüksek gecikme sürelerine sahip olsa da, yine de düşük seviyelerde kalmayı başarmıştır. Skor artırma işlemlerinde p50 değeri 0,00624 ms, p95 değeri 0,0197 ms ve p99 değeri 0,0305 ms olarak ölçülmüştür. Okuma tarafında ise p50 değeri 3,53 ms, p95 değeri 8,39 ms ve p99 değeri 12,7 ms olarak kaydedilmiş, verim değeri saniyede 6 250 sorgu olarak gerçekleşmiştir. Caffeine, özellikle TTL ve önbellek

yönetiminden kaynaklanan ek maliyetlere rağmen dengeli bir performans sunmuş; yüksek okuma yükü altında dahi yanıt süreleri kabul edilebilir seviyelerde kalmıştır.

Redis çözümü, bellek dışı erişim ve ağ tabanlı çalışması nedeniyle daha yüksek gecikme süreleri göstermiştir. Özellikle bu projede Redis, Docker üzerinde ayrı bir konteyner olarak çalıştırılmıştır. Bu durum, Java ve Caffeine çözümlerinde olduğu gibi uygulamanın belleğine doğrudan erişmek yerine, ağ üzerinden haberleşmeyi zorunlu kılmıştır. Bu nedenle her okuma ve yazma işleminde ek bir ağ gecikmesi oluşmuş, gecikme değerleri diğer çözümlere kıyasla daha yüksek gerçekleşmiştir.

Skor artırma işlemlerinde p50 değeri 1,04 ms, p95 değeri 2,62 ms ve p99 değeri 6,25 ms olarak ölçülmüştür. Okuma tarafında ise p50 değeri 18,1 ms, p95 değeri 54,3 ms ve p99 değeri 86,8 ms olarak kaydedilmiş, verim değeri saniyede 4 000 sorgu olarak gerçekleşmiştir. Bu değerler Java ve Caffeine'e kıyasla daha yüksek olmakla birlikte, Redis'in veri kaybetmeme özelliği (persistence) ve dağıtık kullanım desteği göz önünde bulundurulduğunda performans kaybının kabul edilebilir olduğu söylenebilir. Ayrıca Redis üzerinde uygulanan pipelining optimizasyonunun, özellikle yüksek yazma yüklerinde verim değerlerini olumlu yönde etkilediği gözlemlenmiştir.

Çizelge 3.5. Saniyede 1000 yazma ve kayıpsız okuma gecikme ve verim değerleri

SENARYO	SKOR ARTIRMA (YAZMA SENARYOSU)					İLK 100 OYUNCU SORGULAMA (OKUMA SENARYOSU)				
YÖNTEM	%	Ortalama (ms)	En yüksek (ms)	En düşük (ms)	Verim (işlem/s)	%	Ortalama (ms)	En yüksek (ms)	En düşük (ms)	Verim (işlem/s)
JAVA	p50	0,000583	0,00516	0,000500	1000	p50	3,33	3,50	1,74	6500
	p95	0,00122	0,00959	0,000950		p95	4,85	4,95	4,57	
	p99	0,00196	0,00998	0,000990		p99	5,62	32,7	4,94	
CAFFEINE	p50	0,00582	0,00693	0,00552	1000	p50	3,51	3,56	3,49	5500
	p95	0,0148	0,0231	0,00978		p95	4,89	4,97	4,86	
	p99	0,0240	0,0530	0,0198		p99	6,01	8,80	4,98	
REDIS	p50	0,988	1,47	0,685	1000	p50	12,6	77,6	2,12	3700
	p95	3,33	18,7	1,67		p95	43,5	166	4,83	
	p99	7,11	43,9	3,63		p99	66,9	196	7,16	

Çizelge 3.5’te sistemin daha yüksek yazma yükü altındaki performansını gözlemlemek amacıyla saniyede 1000 skor artırma işlemi gerçekleştirilmiş ve bu sırada okuma yükü artırılarak sistemin sınırları ölçülmüştür. Bu senaryoda, yazma kaybı yaşanmadan kaç okuma isteğinin karşılanabildiği belirlenmiştir.

Java çözümü, önceki senaryoda olduğu gibi en düşük gecikme değerlerini korumuştur. Skor artırma tarafında p50 değeri 0,000583 ms gibi son derece düşük bir seviyede kalmış, p95 ve p99 değerleri de mikro saniye düzeyinde tutulabilmiştir. Okuma tarafında p50 değeri 3,33 ms olarak ölçülmüş, p95 ve p99 değerleri sırasıyla 4,85 ms ve 5,62 ms seviyelerinde gerçekleşmiştir. Sistemin saniyede yaklaşık 6500 okuma isteğini karşılayabilmesi, Java çözümünün hem hız hem de kapasite açısından oldukça başarılı olduğunu göstermektedir. Ancak bu yaklaşımda verilerin yalnızca bellekte saklanıyor olması, sistem yeniden başlatıldığında tüm bilgilerin kaybolacağı gerçeğini değiştirmemektedir.

Caffeine çözümü, bu senaryoda da Java’ya yakın performans sergilemiştir. Skor artırma işlemlerinde p50 değeri 0,00582 ms olarak kaydedilmiş, p95 ve p99 değerleri sırasıyla 0,0148 ms ve 0,0240 ms seviyelerinde gerçekleşmiştir. Okuma tarafında ise p50 değeri 3,51 ms, p95 değeri 4,89 ms ve p99 değeri 6,01 ms olarak ölçülmüş, sistem saniyede yaklaşık 5500 sorguyu işleyebilmiştir. Bu sonuçlar, Caffeine’in bellek yönetiminde getirdiği ek maliyetlere rağmen tutarlı ve kabul edilebilir gecikme süreleri sunduğunu ortaya koymaktadır.

Redis çözümü, Docker üzerinde ayrı bir konteynerde çalışması nedeniyle ağ gecikmelerinden etkilenmeye devam etmiştir. Skor artırma işlemlerinde p50 değeri 0,988 ms, p95 değeri 3,33 ms ve p99 değeri 7,11 ms olarak gerçekleşmiştir. Okuma tarafında p50 değeri 12,6 ms, p95 değeri 43,5 ms ve p99 değeri 66,9 ms düzeylerinde kaydedilmiştir. Verim açısından sistem saniyede yaklaşık 3700 sorgu işleyebilmiştir. Bu değerler Java ve Caffeine’in gerisinde kalsa da Redis’in sunduğu kalıcılık ve dağıtık mimari desteği göz önüne alındığında, daha uzun gecikmelerin karşılığında daha yüksek güvenilirlik sağlandığı söylenebilir.

Genel olarak bakıldığında, saniyede 1000 yazma işlemi yükü altında Java çözümü yine en yüksek verimi sunmuş; Caffeine yakın değerlerle tutarlı performans göstermiştir. Redis ise ağ tabanlı çalışmasının getirdiği ek maliyetlere rağmen kayıpsız yazma sağlayarak güvenilirliğini korumuştur.

Çizelge 3.6. Saniyede 2000 yazma ve 10000 okuma isteği gecikme ve verim değerleri

SENARYO	SKOR ARTIRMA (YAZMA SENARYOSU)					İLK 100 OYUNCU SORGULAMA (OKUMA SENARYOSU)				
YÖNTEM	%	Ortalama (ms)	En yüksek (ms)	En düşük (ms)	Verim (işlem/s)	%	Ortalama (ms)	En yüksek (ms)	En düşük (ms)	Verim (işlem/s)
JAVA	p50	0,000510	0,000596	0,000501	1500	p50	3,53	3,54	3,52	7700
	p95	0,00141	0,00727	0,000952		p95	4,91	4,93	4,89	
	p99	0,00230	0,00952	0,000992		p99	7,49	9,68	6,04	
CAFFEINE	p50	0,00658	0,0345	0,00481	1350	p50	3,54	3,82	3,53	6600
	p95	0,0186	0,0954	0,00951		p95	16,7	714	4,90	
	p99	0,0412	0,950	0,00990		p99	24,1	943	7,28	
REDIS	p50	1,38	1,52	1,26	700	p50	69,8	80,8	58,9	3600
	p95	3,77	4,54	3,19		p95	161	177	141	
	p99	5,10	9,15	4,65		p99	3,53	3,54	3,52	

Çizelge 3.6'da ise sistem saniyede 2000 skor artırma işlemi ve 10 000 okuma isteği ile sınanmıştır. Amaç, oldukça yüksek bir yük altında yazma kaybı yaşanmadan sistemin hem gecikme hem de verim açısından nasıl davrandığını gözlemlemektir

Java çözümü, bu yoğun senaryoda dahi güçlü bir performans sergilemiştir. Skor artırma tarafında p50 değeri 0,000510 ms seviyesinde kalmış, p95 ve p99 değerleri sırasıyla 0,00141 ms ve 0,00230 ms olarak kaydedilmiştir. Bu değerler, sistemin yazma yükü artmasına rağmen neredeyse aynı hızda işlem yapmaya devam edebildiğini göstermektedir. Okuma tarafında p50 değeri 3,53 ms, p95 değeri 4,91 ms ve p99 değeri 7,49 ms olarak ölçülmüş, sistem saniyede yaklaşık 7700 okuma isteğini kayıpsız olarak işleyebilmiştir. Bu sonuç, Java yaklaşımının bellek içi çalışmasının getirdiği hız avantajını koruduğunu ortaya koymaktadır.

Caffeine çözümü, Java'ya kıyasla biraz daha yüksek gecikmeler göstermiştir. Skor artırma işlemlerinde p50 değeri 0,00658 ms, p95 değeri 0,0186 ms ve p99 değeri 0,0412 ms olarak kaydedilmiştir. Okuma tarafında ise p50 değeri 3,54 ms, p95 değeri 16,7 ms ve p99 değeri 24,1 ms olarak gerçekleşmiştir. Saniyede yaklaşık 6600 okuma isteği kayıpsız işlenebilmiş, ancak p95 ve p99 değerlerindeki artış, yük arttığında Caffeine'in önbellek yönetiminin ek

maliyetlerini ortaya koymuştur. Buna rağmen sistem tutarlı şekilde yanıt vermeye devam etmiş ve yazma işlemlerinde kayıp yaşanmamıştır.

Redis çözümü, bu yoğun senaryoda gecikme değerlerinde belirgin bir artış göstermiştir. Docker üzerinde ayrı bir konteynerde çalıştırılmasının getirdiği ağ gecikmeleri, bu yük altında daha belirgin hale gelmiştir. Skor artırma tarafında p50 değeri 1,38 ms, p95 değeri 3,77 ms ve p99 değeri 5,10 ms olarak kaydedilmiştir. Okuma tarafında ise p50 değeri 69,8 ms, p95 değeri 161 ms ve p99 değeri 208 ms seviyelerine ulaşmıştır. Sistemin saniyede yaklaşık 3600 okuma isteğini karşılayabilmesi, Redis'in yoğun yük altında Java ve Caffeine'in gerisinde kaldığını göstermektedir. Bununla birlikte Redis, veri kaybetmeme özelliği ve dağıtık kullanım avantajları sayesinde bu performans düşüşünü işlevsellik açısından tolere edilebilir kılmıştır.

Genel olarak bu senaryoda, Java bellek içi çözümü en düşük gecikmeleri ve en yüksek verim değerlerini sunmuş; Caffeine yakın performans göstermiş ancak yüksek yüzdeliklerde gecikme artışı yaşamıştır. Redis ise ağ tabanlı çalışmasının getirdiği dezavantajlar nedeniyle daha yüksek gecikme sürelerine sahip olmuş, fakat güvenilirliği sayesinde dengeli bir seçenek olmaya devam etmiştir.

Genel Değerlendirme: Test sonuçları üç farklı hafıza çözümlerinin yoğun yük altında nasıl davrandığını ortaya koymuştur. Ancak hangi çözümün en uygun olduğu sorusunun yanıtı yalnızca ölçülen hız değerlerine bağlı değildir. Her bir yaklaşım, belirli avantajları ve sınırlamalarıyla farklı kullanım senaryolarına hitap etmektedir. Sistem tasarımı, yalnızca performans değil, aynı zamanda dayanıklılık, genişletilebilirlik, kalıcılık ve bakım kolaylığı gibi pek çok faktöre bağlıdır.

Java çözümü, yapılan tüm testlerde en düşük gecikme ve en yüksek verim değerlerini sunmuştur. Bellek içi doğrudan veri erişimi, ek bir katman veya bağımlılık olmadan çalıştığı için milisaniyenin altında yanıt sürelerine olanak tanımıştır. Ancak bu hız, veri kalıcılığı pahasına sağlanmaktadır. Uygulama yeniden başlatıldığında tüm veriler kaybolur ve sistemin çok kullanıcılı veya dağıtık bir mimaride çalışması gerekirse ciddi ölçeklenebilirlik problemleri ortaya çıkabilir. Bu nedenle Java çözümü, daha çok geçici verilerin işlendiği, test ortamlarında referans performans elde etmek için kullanılan veya küçük ölçekli uygulamalarda tercih edilebilecek bir çözümdür.

Caffeine, Java ile benzer bir bellek içi yaklaşım sunsa da sunduğu TTL, boyut limiti ve otomatik temizlik gibi ek özelliklerle daha gelişmiş bir yapı oluşturur. Performans açısından Java'dan biraz daha geride kalmasına rağmen, önbellek yönetimi sayesinde daha kontrollü bir bellek kullanımı sağlar. Veri kalıcılığı olmamakla birlikte, Caffeine özellikle okuma ağırlıklı sistemlerde, yüksek performanslı ancak kalıcılık gerektirmeyen senaryolarda etkili bir çözüm sunar. Ayrıca yazma gecikmeleri düşük kaldığı sürece, yüksek eşzamanlılık altında kararlı bir performans sergileyebilir.

Redis ise diğer iki çözüme göre daha yüksek gecikme değerlerine sahip olmasına rağmen, veri güvenliği ve ölçeklenebilirlik açısından açık ara daha avantajlıdır. Redis'in ağ tabanlı bir çözüm olması, özellikle Docker üzerinde ayrı bir konteyner olarak çalıştırıldığında, okuma ve yazma işlemlerinde belirgin gecikmelere yol açmıştır. Ancak bunun karşılığında Redis, verileri bellekte tutarken aynı zamanda kalıcı olarak disk üzerinde saklayabilmekte, birden fazla istemcinin erişimine uygun, dağıtık bir yapı sunabilmektedir. Bu yönüyle Redis, özellikle gerçek zamanlı sistemlerde veya kritik verilerin yönetildiği ortamlarda tercih edilmelidir.

Bu bulgular ışığında, hangi çözümün tercih edilmesi gerektiği aşağıdaki karar tablosunda özetlenmiştir:

Çizelge 3.7. Hafıza çözümlerinin karar tablosu

Kriter	Java	Caffeine	Redis
Hız (Latency, Throughput)	Çok yüksek (en hızlı çözüm)	Yüksek (Java'ya yakın)	Orta (Docker tabanlı çalışmadan dolayı daha yüksek gecikme)
Kalıcılık	Yok	Yok	Var (disk üzerinde veri saklama desteği)
Ölçeklenebilirlik	Sınırlı (tek makine bellek tabanlı)	Orta (önbellek tabanlı, tek makine odaklı , çoklu iplik desteği)	Yüksek (dağıtık mimarilere uygun)
Kullanım Alanı	Test ortamları, geçici veriler	Okuma ağırlıklı, düşük gecikme gereken uygulamalar	Kritik sistemler, kalıcılık ve güvenilirlik gerektiren uygulamalar

Çizelge 3.7'de de görüldüğü üzere, Java çözümü en yüksek performansı sunmasına rağmen kalıcılık eksikliği nedeniyle üretim ortamlarında sınırlı kullanım alanına sahiptir. Caffeine, dengeli performansıyla okuma yoğunluklu sistemler için uygundur. Redis ise yüksek gecikmelere rağmen veri güvenliği ve ölçeklenebilirliği sayesinde özellikle kritik ve dağıtık sistemlerde tercih edilmelidir.

4. SONUÇ

ASELSAN'da gerçekleştirilen bu staj süresince, kurumsal yazılım geliştirme süreçlerine doğrudan dahil olunmuş ve modern teknolojilerin entegrasyonu üzerine yoğun bir deneyim kazanılmıştır. Geliştirilen liderlik tablosu uygulaması, üç farklı veri yönetim yaklaşımının (Java, Caffeine, Redis) performans ve kullanım senaryoları açısından karşılaştırılmasına olanak tanımıştır.

Testler sonucunda, Java bellek içi yöntemi en yüksek hız ve en düşük gecikmeyi sağlamış; ancak kalıcılık ve ölçeklenebilirlik eksiklikleri nedeniyle sınırlı bir kullanım alanı olduğu görülmüştür. Caffeine Cache, TTL ve bellek yönetimi avantajlarıyla dengeli bir performans sunmuş; okuma ağırlıklı senaryolarda güvenilir bir alternatif olmuştur. Redis ise görece yüksek gecikmelere rağmen, sunduğu kalıcılık, sıralama desteği ve dağıtık mimari uyumluluğu ile kritik ve yüksek erişilebilirlik gerektiren sistemler için uygun bir çözüm olarak öne çıkmıştır.

Bu staj sürecinde yalnızca teknik açıdan değil, aynı zamanda ekip çalışması, problem çözme, zaman yönetimi ve kurumsal iletişim konularında da değerli deneyimler edinilmiştir. Micrometer, Prometheus ve Grafana gibi gözlemleme araçlarının entegrasyonu, sistem performansının sürekli izlenmesinin önemini açıkça ortaya koymuştur. Ayrıca k6 aracıyla gerçekleştirilen yük testleri, gerçekçi senaryolar altında sistem davranışını analiz etme becerisi kazandırmıştır.

Staj sırasında Docker ve konteyner teknolojilerinin, modern yazılım geliştirme süreçlerinde ölçeklenebilirlik ve sürdürülebilirlik sağlamak açısından ne derece kritik olduğu deneyimlenmiştir. Bu süreç, bilgisayar mühendisliği alanında yalnızca yazılım geliştirme değil, aynı zamanda sistem mimarisi ve DevOps uygulamaları hakkında da bilgi kazandırmıştır.

Bununla birlikte, işyeri ile imzalanmış sadakat ve gizlilik sözleşmelerim gereği, proje dışında kurumsal süreçlere dair detaylar bu raporda paylaşılmamıştır. Ancak gizlilik sınırları içinde kalarak, yürütülen teknik projenin kapsamı ve bu projeden elde edilen kazanımlar detaylı biçimde aktarılmıştır.

Sonuç olarak, bu staj hem akademik bilgilerin uygulamaya aktarılması hem de profesyonel mühendislik pratiğinin tecrübe edilmesi bakımından son derece faydalı olmuştur. Elde edilen teknik becerilerin ve profesyonel deneyimlerin, ilerideki iş hayatında büyük katkı sağlayacağı düşünülmektedir.

KAYNAKLAR

1. İnternet: ASELSAN – Hakkımızda. https://www.aselsan.com/tr/hakkimizda (2025)

2. İnternet: ASELSAN – Yerleşkelerimiz. https://www.aselsan.com/tr/hakkimizda/yerleskelerimiz 2025)

3. İnternet: ASELSAN – Finansal Göstergeler .https://www.aselsan.com/tr/yatirimci-iliskileri/finansal-gostergeler (2025)

4. İnternet: ASELSAN – Faaliyet Alanları. https://www.aselsan.com/tr/faaliyet alanlari(2025)

5. İnternet: ASELSAN – Tarihçesi. https://www.aselsan.com/tr/hakkimizda/tarihce (2025)

6. İnternet: Azure DevOps. https://azure.microsoft.com/tr-tr/products/devops (2025).

7. İnternet: Docker. https://www.docker.com (2025).

8. İnternet: Github – LeaderboardREST Reposiyory. https://github.com/noxennn/ LeaderboardREST

9. İnternet: Gradle. https://gradle.org (2025).

10. İnternet: Grafana. https://grafana.com (2025).

11. İnternet: Java. https://aws.amazon.com/what-is/java/ (2025).

12. İnternet: Kafka. https://kafka.apache.org (2025).

13. İnternet: Katmanlı Mimari Nedir?. https://medium.com/devopsturkiye/ katmanl%C4%B1-mimari-nedir-84d097b674c2 (2021).

14. İnternet: Kubernetes. https://kubernetes.io (2025).

15. İnternet: Kotlin. https://kotlinlang.org (2025).

16. İnternet: Loki. https://grafana.com/oss/loki (2025).

17. İnternet: Prometheus. https://prometheus.io (2025).

18. İnternet: React. https://react.dev (2025).

19. İnternet: Redis. https://redis.io (2025).

20. İnternet: Spring Boot. https://spring.io/projects/spring-boot (2025).

21. İnternet: TypeScript. https://www.typescriptlang.org (2025).

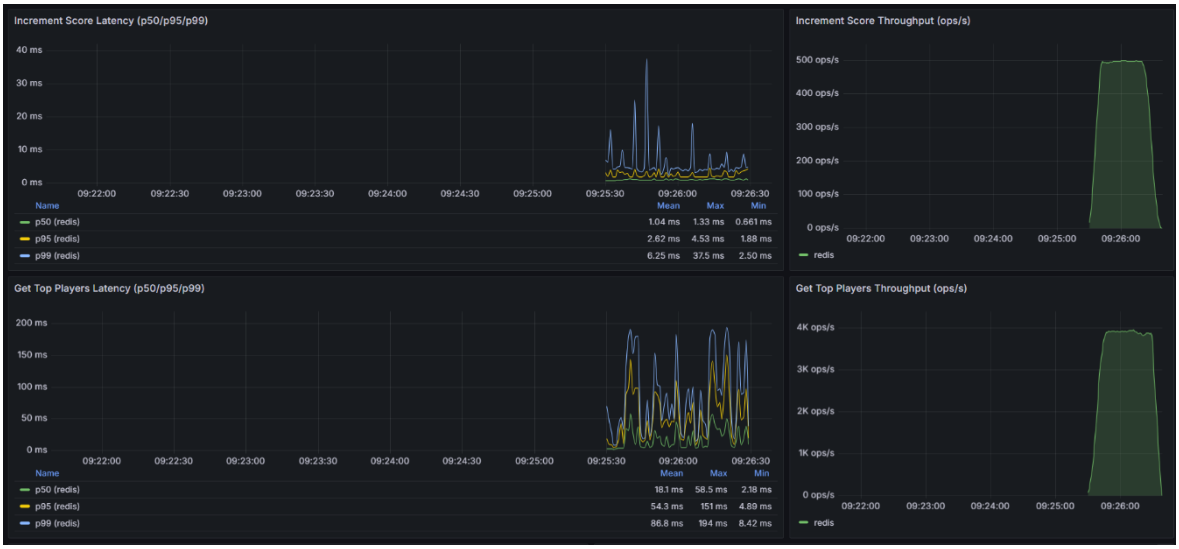
22. İnternet: Youtube – Understanding Prometheus Histograms. https://www.youtube.com/ watch?v=yYbXak-1hew&t=148s (2025)

EKLER

EK-1. 500 Okuma kayıpsız yazma Java Grafana ölçüm görüntüsü

EK-2. 500 Okuma kayıpsız yazma Caffeine Grafana ölçüm görüntüsü

EK-3. 500 Okuma kayıpsız yazma Redis Grafana ölçüm görüntüsü

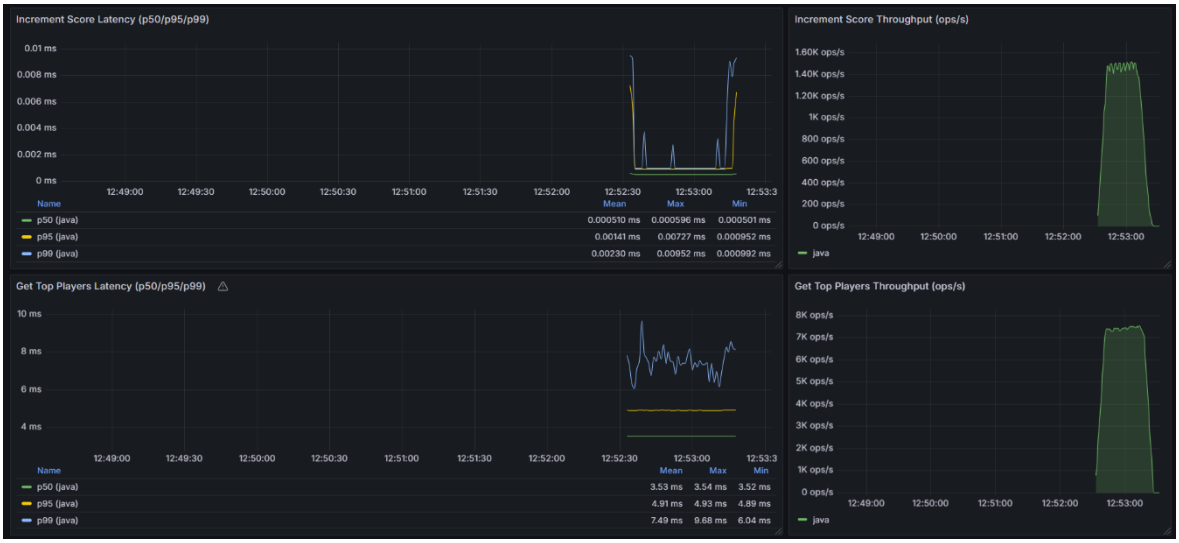


EK-4. 1000 Okuma kayıpsız yazma Java Grafana ölçüm görüntüsü

EK-5. 1000 Okuma kayıpsız yazma Caffeine Grafana ölçüm görüntüsü

EK-6. 1000 Okuma kayıpsız yazma Redis Grafana ölçüm görüntüsü

EK-7. 10000 Okuma 2000 Yazma Java Grafana ölçüm görüntüsü



EK-8. 10000 Okuma 2000 Yazma Caffeine Grafana ölçüm görüntüsü

EK-9. 10000 Okuma 2000 Yazma Redis Grafana ölçüm görüntüsü