SAKARYA ÜNİVERSİTESİ
BİLGİSAYAR VE BİLİŞİM BİLİMLERİ FAKÜLTESİ

ISE 465 - BULUT BİLİŞİM
PROJE RAPORU

PROJE: NIMBUS AI

Öğrenci: Omar Huseynov
No: B221200559

1. GİRİŞ VE PROJE AMACI

Günümüz bulut bilişim dünyasında, sistemlerin sadece çalışır durumda olması yeterli değildir; aynı zamanda hatalara karşı dayanıklı (fault-tolerant) ve kendi kendini iyileştirebilir (self-healing) olması gerekmektedir. Bu proje kapsamında geliştirilen 'Nimbus AI', Infrastructure as a Service (IaaS) ve Platform as a Service (PaaS) katmanlarını birleştiren modern bir bulut izleme çözümüdür.

Projenin temel amacı, OpenStack altyapısı üzerinde çalışan, Docker konteyner teknolojisi ile izole edilmiş ve Python Flask ile geliştirilmiş, yüksek erişilebilirliğe sahip bir web uygulaması dağıtmaktır. Proje, özellikle 'Chaos Engineering' prensiplerini uygulayarak, sistem çökmelerine karşı alınan otomatik önlemleri simüle etmektedir.

2. KULLANILAN TEKNOLOJİLER VE MİMARİ

Proje mimarisi dört ana katmandan oluşmaktadır:

- **Bulut Altyapısı (IaaS):** OpenStack MicroStack (Beta). Yerel geliştirme ortamında özel bir bulut (Private Cloud) simülasyonu sağlamıştır.
- **Sanallaştırma Katmanı:** Ubuntu 22.04 LTS Instance. Compute (Nova) ve Network (Neutron) servisleri üzerinden yönetilmiştir.
- **Konteyner Orkestrasyonu:** Docker Engine. Uygulamanın bağımlılıkları izole edilmiş ve 'Restart Policy' ile dayanıklılık sağlanmıştır.
- **Yazılım Dili:** Python 3.9 (Flask Framework). Backend mantığı ve Frontend (HTML5/JS) servisi tek bir süreçte birleştirilmiştir.

3. YAZILIM GELİŞTİRME VE KOD ANALİZİ

Uygulama, 'Monolitik Mikroservis' mimarisi ile geliştirilmiştir. Bu yaklaşım, tek bir dosya içinde (Single-File App) tüm servis mantığını barındırarak dağıtım karmaşıklığını azaltır. Aşağıda projenin kritik kod blokları ve çalışma mantıkları açıklanmıştır.

3.1. Backend Konfigürasyonu ve Güvenlik

Flask uygulaması başlatılırken, oturum güvenliği için kriptografik olarak güvenli rastgele bir anahtar (Secret Key) üretilmektedir. Ayrıca, uygulamanın çalıştığı konteynerin kimliği (Hostname) işletim sistemi seviyesinde çekilerek arayüzde gösterilmektedir.

Kod: Flask Başlatma ve Güvenlik Ayarları

```
import psutil
import os
import threading
from flask import Flask, jsonify

app = Flask(__name__)
# Güvenlik için rastgele secret key üretimi
app.secret_key = os.urandom(24)
start_time = time.time()
stress_mode = False

# Konteyner ID'sini işletim sisteminden çekme
container_id = os.uname()[1]
```

3.2. Frontend Motoru

Geleneksel Flask uygulamalarında kullanılan 'Jinja2 Template Inheritance' yapısı, tek dosya mimarisinde dosya sistemi hatalarına yol açabilmektedir. Bu sorunu çözmek için 'Atomic Design' prensibi uygulanmıştır. HTML bileşenleri (Navbar, Footer, Kartlar) statik dosyalar yerine, Python fonksiyonları tarafından dinamik olarak üretilmektedir.

Kod: Dinamik HTML Üretim Fonksiyonu

```
# ATOMIC DESIGN PATTERN
# HTML şablonları statik dosyalar yerine, dinamik Python fonksiyonları
# olarak tanımlanmıştır. Bu sayede 'Template Inheritance' hataları
önlenmiştir.

def get_navbar(active_page):
    links = [('home', '/'), ('status', '/status')]
    # ... (Dinamik menü oluşturma döngüsü)
    return f"""
    <nav class="navbar">
        <div class="container">
            <a href="/" class="brand">NIMBUS AI</a>
            </div>
    </nav>
    """
```

3.3. Etkileşimli Dashboard ve WebSocket Simülasyonu

Kullanıcı arayüzü, sunucu ile asenkron (AJAX/Fetch API) haberleşerek sayfa yenilenmesine gerek kalmadan verileri günceller. JavaScript motoru, CPU ve RAM kullanımını anlık olarak görselleştirir ve eşik değerler aşıldığında (örn: %80 CPU) kullanıcıyı görsel olarak uyarır.

Kod: JavaScript Veri Çekme ve Grafik Güncelleme

```
// FRONTEND METRIC FETCHING
// Sunucudan gerçek zamanlı verileri çeker ve grafikleri günceller.

function updateDashboard() {
    fetch('/api/stats')
        .then(r => r.json())
        .then(data => {
            // CPU Barını güncelle ve renk değiştir
            const cpuFill = document.getElementById('cpu-fill');
            cpuFill.style.width = data.cpu + '%';

            // Eşik değeri aşılırsa uyarı rengi (Kırmızı)
            if(data.cpu > 80) cpuFill.style.backgroundColor = '#ef4444';
            else cpuFill.style.backgroundColor = '#6366f1';
        });
}
```

4. DAYANIKLILIK VE SELF-HEALING MEKANİZMASI

Projenin en önemli özelliği, sistem hatalarına karşı otonom tepki verebilmesidir. Bu yetenek iki aşamada kurgulanmıştır: Hatayı Simüle Etme (Chaos) ve İyileştirme (Healing).

4.1. Kill Switch (Hata Simülasyonu)

Sistemin çökme durumunu test etmek için özel bir API endpoint geliştirilmiştir. Bu endpoint çağrıldığında, uygulama 'sys.exit()' yerine 'os._exit(1)' komutunu çalıştırır. Bu komut, Python yorumlayıcısını anında ve zorla kapatarak gerçek bir 'Crash' (Çökme) senaryosu oluşturur.

Kod: Sistemi Çökerten Python Fonksiyonu

```
# CHAOS ENGINEERING (KILL SWITCH)
# Bu fonksiyon, sistemin dayanıklılığını test etmek için
# Python sürecini kasıtlı olarak sonlandırır.

@app.route('/api/kill')
def api_kill():
```

```
    def kill_process():
        time.sleep(1)
        # sys.exit() yerine os._exit(1) kullanıldı.
        # Böylece Python Interpreter anında kapanır ve
        # Docker Healthcheck mekanizması tetiklenir.
        os._exit(1)

    threading.Thread(target=kill_process).start()
    return "Kill Signal Received", 200
```

4.2. Docker Healthcheck ve Recovery

Docker, 'HEALTHCHECK' talimatı ile uygulamanın yaşayıp yaşamadığını her 30 saniyede bir kontrol eder. Eğer uygulama (Kill Switch nedeniyle) cevap veremez hale gelirse, Docker Daemon durumu 'Unhealthy' olarak işaretler ve '--restart always' politikası gereği konteyneri sıfırdan başlatır.

Kod: Dockerfile Sağlık Kontrolü ve Kullanıcı Ayarları

```
# ENTERPRISE GRADE DOCKERFILE
FROM python:3.9-slim

# Güvenlik: Root olmayan kullanıcı
RUN useradd -m nimbus
USER nimbus

# Otomasyon: Sağlık Kontrolü (Self-Healing Trigger)
HEALTHCHECK --interval=30s --timeout=5s --retries=3 \
  CMD curl --fail http://localhost:5000/api/stats || exit 1

# Restart Policy: Container Orchestrator (Docker/K8s) tarafından yönetilir.
CMD ["python", "app.py"]
```

5. DAĞITIM SÜRECİNDE KARŞILAŞILAN ZORLUKLAR

5.1. Dinamik IP Adresi Sorunu

OpenStack MicroStack servisi, host makine her yeniden başlatıldığında DHCP üzerinden farklı bir IP adresi almaktaydı. Bu durum, Dashboard erişimini ve SSH bağlantılarını koparıyordu.

Çözüm: Ubuntu 'netplan' konfigürasyonu düzenlenerek sanal ağ arayüzüne (enp0s3) statik bir IP (192.168.1.200) atandı. Böylece servis sürekliliği sağlandı.

5.2. Internal Server Error (500) Hataları

Uygulama geliştirme aşamasında, şablon motorunun dosya yollarını bulamaması nedeniyle sık sık 500 hatası alındı. Bu, özellikle 'render_template' fonksiyonunun harici .html dosyaları araması nedeniyle oluştu.

Çözüm: Tüm HTML kodları 'render_template_string' fonksiyonu ile doğrudan Python kodu içine gömüldü. Bu yöntem, uygulamanın tek bir dosya olarak taşınmasını ve hatasız çalışmasını garanti altına aldı.

6. SONUÇ VE KAZANIMLAR

Bu proje çalışması ile birlikte, teorik bulut bilişim kavramları pratik bir zemine oturtulmuştur. Özellikle IaaS (OpenStack) ve PaaS (Docker) katmanlarının entegrasyonu, modern yazılım dağıtım süreçlerinin (CI/CD) temelini oluşturmuştur.

Elde edilen 'Nimbus AI' platformu, sadece bir izleme aracı değil, aynı zamanda bulut tabanlı sistemlerin dayanıklılığını test eden bir laboratuvar ortamı niteliğindedir. Proje, kurumsal standartlarda kodlama, güvenlik sıkılaştırması (root olmayan kullanıcı) ve otonom yönetim (Self-Healing) prensiplerini başarıyla uygulamıştır.

7. KAYNAKÇA

1. OpenStack Foundation. (2025). MicroStack Installation & Architecture Guide.
2. Docker Inc. (2025). Dockerfile Best Practices for Enterprise Applications.
3. Google Site Reliability Engineering. (2016). 'Embracing Risk & Service Level Objectives'. O'Reilly Media.
4. Pallets Projects. (2024). Flask Documentation (v3.0.x): Application Patterns.