DevOps En İyi Uygulamalar El Kitabı (OBSS)

Amaç: Bu el kitabının amacı, OBSS'nin DevOps süreçlerinde benimsediği standartları ve kalite kriterlerini tanımlayarak yazılım yaşam döngüsünde tutarlılığı ve yüksek kaliteyi sağlamaktır. Aşağıda kod depolama, CI/CD, güvenlik, gözlemlenebilirlik (observability) ve otomasyon-sürümleme konularında en iyi uygulamalar listelenmiştir.

Kod Deposu Standartları

- **Branch Stratejisi ve İsimlendirme:** Tüm yeni özellik geliştirmeleri ve hata düzeltmeleri, `main` (ana) daldan ayrılan ayrı **feature** dallarında gerçekleştirilmelidir[1]. Anlaşılır ve tutarlı bir dal isimlendirme konvansiyonu kullanın (ör. `feature/özellik-adi`, `bugfix/aciklama`, `hotfix/aciklama` gibi)[2]. Bu sayede ekip üyeleri yapılan çalışmanın amacını dal isminden anlayabilir ve dallar arası çakışmalar en aza iner. Ana dal her zaman çalışır durumda ve güncel tutulmalı, feature dalları kısa ömürlü olup iş tamamlanınca ana dala **pull request (PR)** ile birleştirilmelidir.

- **Pull Request Süreci ve Dal Koruma:** Ana (örn. `main` veya `master`) dalına doğrudan kod gönderimi (push) yapılmamalıdır. Tüm değişiklikler en az bir kod incelemesinden geçtikten sonra bir **pull request** aracılığıyla ana dala birleştirilmelidir[3]. Bu amaçla dal koruma (branch protection) kuralları uygulayarak, PR onayı olmadan ve CI derlemesi/testleri başarılı olmadan ana dala merge yapılmasını engelleyin[3]. PR açıldığında otomatik olarak kod sahiplerine/ekibe atanması ve değişikliklerin bir **CI** pipeline'ında derlenip test edilmesi, kod kalitesini güvence altına alır. Tüm PR'lar birleştirilmeden önce ilgili otomatik kontroller (birim testleri, statik analiz vs.) geçirilmelidir.

- **Kod İnceleme (Code Review) Kalite Kriterleri:** Kod değişiklikleri ekip içerisinde mutlaka gözden geçirilmelidir. Mümkünse her PR için en az **iki kişi** kod incelemesi yapmalıdır (araştırmalar iki bağımsız gözün optimum olduğunu göstermektedir)[4]. İnceleme sırasında geri bildirimler açık ve yapıcı bir dille verilmelidir. PR'ların **küçük ve odaklı** tutulması önemlidir; az ve öz değişiklik içeren PR'lar gözden geçirmek ve anlamak açısından daha verimlidir[5]. Kodun ekipte belirlenmiş stil rehberlerine ve standartlarına uygunluğu doğrulanmalı, ayrıca gerekli birim/integrasyon testlerinin eklendiği ve **test kapsamının** yeterli olup olmadığı kontrol edilmelidir[6]. Bu pratikler, kod kalitesini yükseltir ve olası hataları erken aşamada yakalamaya yardımcı olur.

CI/CD Pipeline Standartları

- **Çoklu Ortam ve Aşamalı Dağıtım:** CI/CD boru hattı (pipeline), geliştirme, test/UAT ve üretim gibi ayrı **ortamlara** dağıtımı destekleyecek şekilde çok aşamalı tasarlanmalıdır. Her ortam için ayrı **pipeline stage**'leri kullanın ve her geçişte kalite kontrollerini uygulayın. Örneğin, **CI (Continuous Integration)** aşamasında her kod

gönderiminde projenin derlenmesi, testlerin çalışması ve kod kalite analizlerinin yapılması otomatik hale getirilmelidir. **CD (Continuous Delivery/Deployment)** aşamaları ise önce test/staging ortamlarına, son olarak üretime dağıtım şeklinde kademelendirilmelidir. Her geçişte gerekli onaylar ve koşullu kontroller tanımlanarak (örn. test ortamında testler başarıyla geçerse otomatik üretime aday olması gibi) hataların üretime çıkmadan yakalanması sağlanır[7]. Bu yapı, değişikliklerin kontrollü ve güvenli bir şekilde farklı ortamlarda sınanarak yayınlanmasını temin eder.

- **Manuel Onay ve Yayın Kontrolü:** Üretim gibi kritik ortamlar için dağıtım adımlarında **manuel onay (approval)** mekanizmaları kullanılmalıdır. Örneğin, GitHub Actions ortam koruma kuralları ya da Azure Pipelines **Environment** onayları ile, ilgili sorumlu kişi onay vermeden üretime geçiş olmayacak şekilde pipeline'ı yapılandırın. Nitekim, bir ortam üzerinde tanımlanan manuel onay kontrolü sayesinde, o ortama dağıtım yapılmadan önce belirlenen yetkili kişi değişiklikleri gözden geçirip onay verebilir; onaylanmadıkça dağıtım başlamaz[8]. Bu yaklaşım özellikle üretim dağıtımlarında ek bir güvenlik katmanı sağlar. Aynı zamanda ortama özel diğer denetimler (örn. mesai saatleri dışında otomatik engelleme, belirli branch'ten deployment izni gibi) de eklenerek istenmeyen zaman ve kaynaklardan dağıtım engellenebilir[9].

- **Geri Alma Stratejileri ve Kesintisiz Dağıtım:** Her dağıtım için bir **rollback (geri alma)** planı hazır bulundurun. Yeni bir sürümde kritik bir sorun çıkarsa, hızlıca önceki kararlı sürüme dönebilmek DevOps'un önemli bir parçasıdır. Örneğin, Azure App Service kullanıyorsanız **deployment slot** özelliği ile kesintisiz dağıtım yapabilirsiniz: Yeni sürümü **staging slot**'unda yayınlayıp doğruladıktan sonra trafiği ana (production) slot ile takas edersiniz. Eğer problem fark edilirse, önceki sürüm barındıran slot'a saniyeler içinde geri dönebilirsiniz[10]. Bu yaklaşım kontrollü yayılım (controlled rollout) sağlar ve olası hatalarda hızlı geri dönüş imkânı sunar. Benzer şekilde **mavi-yeşil dağıtım (blue-green)** veya **kanarya yayımları (canary release)** stratejileri uygulayarak yeni sürümü önce sınırlı kullanıcı kitlesine veya paralel bir ortama yönlendirmek, sorun gözlenmezse kademeli olarak tüm kullanıcılara açmak mümkündür. Böylece hata durumunda etkilenen kapsam sınırlı tutulur ve eski sürüme dönüş kolaylaşır[11]. CI/CD süreçlerinize bu tür geri alma mekanizmalarını entegre etmek, ciddi kesintileri ve geri dönüş sürelerini en aza indirir.

Güvenlik

- **Gizli Anahtar Yönetimi ve Tarama:** Uygulama veya altyapı için gereken tüm **gizli bilgiler (şifreler, API anahtarları, bağlantı dizgileri vb.)** asla kod deposunda düz metin olarak tutulmamalıdır. Bunun yerine Azure ortamlarında **Azure Key Vault** kullanarak bu sırları güvenli bir şekilde saklayın ve CI/CD pipeline'larında gerekli oldukça çekin[7]. Kod depolarında istemeden bile olsa gizli verilerin yer alması riskine karşı **secret scanning** özelliğini aktif edin. GitHub'ın secret scanning

yeteneği, depo içindeki kodu bilinen gizli anahtar kalıplarına karşı tarar ve uyarılar üretir[12]. Git geçmişindeki eski commit'ler de periyodik olarak taranarak geçmişte kalmış sızıntılar dahi tespit edilebilir[13]. Benzer şekilde Azure DevOps ortamları için de Microsoft Defender gibi araçlarla kod ve derleme çıktılarında gizli anahtar taraması yapılabilir[14]. Bu taramalar, yanlışlıkla repoya eklenen kritik bilgilerin kötüye kullanılmasını önlemede hayati öneme sahiptir.

- **En Az Yetki (Least Privilege) İlkesi:** Tüm erişimler ve izinler, **asgari yetki** prensibine uygun olarak verilmelidir. Geliştiriciler, servis hesapları ve pipeline aracılığıyla kullanılan servis bağlantıları, görevlerini yerine getirmek için sadece gereken minimum izinlerle sınırlandırılmalıdır[15][16]. Örneğin, CI/CD işlemleri için kullanılan Azure servis prensiplerine (service principal) doğrudan **Owner** rolü atamak yerine, sadece gerekli kaynaklarda sınırlı işlemleri yapabilecek özel roller tanımlanmalıdır (özellikle üretim ortamında). Microsoft, üretim ortamlarında servis hesaplarına özel kısıtlayıcı roller atanmasını, böylece bu hesapların kritik kaynakları silememesini önermektedir[17]. Azure DevOps gibi platformlar da varsayılan olarak "least privilege" modelini benimser; tüm kullanıcı ve grup izinlerini gözden geçirerek her ortam ve projede bu ilkenin uygulandığından emin olun[18]. Sonuç olarak, ne kullanıcılar ne de otomasyon hesapları ihtiyaç fazlası yetkilerle donatılmamalıdır – bu hem olası kötü niyetli erişimleri sınırlar hem de hatalı işlemlerin zararını azaltır.

- **CI/CD Runner İzolasyonu ve Güvenliği:** Kendi barındırdığınız (self-hosted) build ajanları veya GitHub Actions **runner**'ları kullanıyorsanız, bunların güvenliğine özel önem gösterin. Ortak paylaşımlı runners yerine, **ephemeral (geçici)** runner yapılarını tercih edin; her iş için sıfırdan oluşturulup iş bitince silinen geçici runner'lar, ardışık işler arasında veri kalıntısını önleyerek gizli bilgilerinizin sızma riskini büyük ölçüde azaltır[19]. Runner makinelerini farklı güvenlik gereksinimlerine göre **gruplandırın** ve izole edin – örneğin üretim dağıtımlarını gerçekleştiren runner'lar ile test ortamı runner'larını ayrı gruplarda tutun[20]. GitHub Runner Grupları kullanarak belirli runner grubunu sadece belirli depolar veya ortamlarda kullanılabilir yapmak mümkündür; böylece güven düzeyine göre ayrıştırma sağlanır[20]. Ayrıca, herkese açık depoların şirket içi özel runner'lara erişimine izin vermeyin (public projeler, sadece GitHub'ın kendi hosted runner'larını kullanmalı)[21]. CI süreçlerinde kullanılan erişim token'larının ve izinlerin de minimum düzeyde olmasını sağlayın – GitHub Actions için varsayılan `GITHUB_TOKEN`'ın sadece gerekli okuma/yazma yetkileriyle kısıtlanması önerilir[22]. Bu önlemler, CI/CD altyapınızın dış tehditlere veya hatalı kod çalıştırmalarına karşı daha dayanıklı olmasını temin eder.

Observability (İzlenebilirlik)

- **Uygulama Telemetrisi (Application Insights):** Canlı sistemlerin izlenebilir olması, sorunların hızlı tespiti ve performans iyileştirmesi için kritik önem taşır. Azure Application Insights gibi APM (Application Performance Monitoring) araçlarını uygulamalarınıza entegre ederek isteklere ait **telemetri verilerini** toplayın.

Application Insights, uygulamanızın çalışma sırasında ürettiği istek/yanıt süreleri, hata oranları, bellek ve CPU kullanımı gibi metriklerin yanı sıra istisna detayları, dağıtılmış izleme (distributed tracing) verileri ve bağımlılık çağrıları hakkında detaylı bilgi sunar[23]. Örneğin, bir hatanın oluştuğu anda stack trace bilgisini, hatadan hemen önce çağrılan dış servisleri, kullanıcı oturum bilgilerini yakalayarak sorunların kök neden analizini kolaylaştırır[23]. Uygulama telemetrisi sayesinde üretim ortamında karşılaşılan bir sorunu, kod düzeyinde tespit etmek ve yeniden oluşturmak çok daha hızlı hale gelir.

- **Merkezi Log Toplama (Log Analytics):** Farklı uygulama ve servislerden gelen logları merkezi bir yerde toplamak ve analiz etmek, karmaşık sistemlerde bütünsel bir görünürlük sağlar. Azure Monitor'un bir parçası olan **Log Analytics** bu amaçla kullanılabilir. Uygulamalarınızın loglarını ve telemetri verilerini bir **Log Analytics çalışma alanına** yönlendirerek hepsini tek bir havuzda biriktirin. Log Analytics, güçlü sorgu dili **Kusto Query Language (KQL)** desteğiyle büyük ölçekli log verileri üzerinde zengin sorgular çalıştırmanıza imkân tanır[24]. Örneğin, belirli bir zaman aralığında hata kodu "500" dönen istek sayısını bulabilir, veya farklı mikroservisler arasında bir işlem kimliğine göre ilişkilendirme yaparak uçtan uca iz sürebilirsiniz. Toplanan log verilerini düzenli olarak analiz ederek anormal durumları proaktif şekilde fark edebilirsiniz (ör. beklenmedik hatalarda ani artışlar, yanıt sürelerinde uzamalar gibi belirtiler)[24]. Log verilerinin merkezi ve sorgulanabilir olması, hem operasyonel izleme hem de güvenlik denetimleri açısından büyük avantaj sağlar.

- **Alarm ve Uyarı Mekanizmaları:** İzleme verilerini toplamak kadar, **alerte edilebilir** olmak da önemlidir. Azure Monitor üzerinden **metrik** ve **log tabanlı uyarılar** tanımlayarak anormal durumlarda ekiplerin haberdar olmasını sağlayın. Örneğin, CPU kullanımı %80'i aşarsa ya da uygulama hata oranı belirli bir eşiğin üstüne çıkarsa otomatik olarak uyarı üretilecek şekilde kural oluşturulabilir. Tanımlanan uyarılar, **Action Group**'lar aracılığıyla ilgili kişilere email, SMS veya Teams bildirimi gönderebilir ya da otomatik tetikleyici mekanizmaları (Azure Functions, Logic Apps vb.) çalıştırabilir. Bu sayede sistemde bir sorun oluştuğunda henüz kullanıcılar fark etmeden operasyon ekibine haber verilir ve müdahale süresi kısalır[25][26]. İleri seviye senaryolarda, **dinamik eşik değerleri** kullanarak alarmların normal sezonsal dalgalanmalara uyum sağlamasını ve sadece gerçekten anormal durumlarda tetiklenmesini de sağlayabilirsiniz[26]. Hatta Azure Monitor uyarılarını otomatik ölçeklendirme ile entegre ederek, belirli bir metriğin kritik eşiğe ulaşması durumunda sistemin kendini ölçeklendirmesini veya bir kurtarma senaryosunun devreye alınmasını sağlayabilirsiniz[27]. Sonuç olarak, doğru yapılandırılmış bir alarm sistemi, izleme verilerinizi eyleme dönüştürerek sistemlerinizi **öz-düzeltir (self-healing)** hale getirebilir[27].

Otomasyon ve Sürümleme

- **Altyapının Kod ile Yönetimi (IaC):** Altyapı ve konfigürasyonlarınızı kod olarak tanımlamak, tutarlılık ve izlenebilirlik açısından en iyi uygulamadır. **Infrastructure**

as Code (IaC) prensibini benimseyerek, sunucular, veritabanları, ağ ayarları, Azure hizmetleri gibi kaynakları kod ile oluşturup yönetebilirsiniz. Azure ekosisteminde bunun için **ARM şablonları** veya daha modern bir dil olan **Bicep** kullanılabilir. Bu tanımlayıcı şablon dosyalarında tüm kaynaklarınızın istenen durumu ve bağımlılıkları belirtilir; parametreler ve modüller kullanarak tekrar kullanılabilir hale getirilir[28]. IaC yaklaşımları sayesinde altyapı değişiklikleri sürüm kontrolüne tabi olur ve belirli bir yapılandırmanın **geri alınabilir** olması, farklı ortamlarda aynı şekilde yeniden kurulabilmesi sağlanır. Örneğin, bir test ortamını kod ile tarif edip gerekince tek komutla oluşturabilir, testler bittiğinde silebilirsiniz. Benzer şekilde üretim ortamının bir kopyasını hızla ayağa kaldırıp deneme yapıp kapatabilirsiniz. Kodla tanımlanmış altyapı, **tekrar üretilebilirlik** sağlar – aynı tanımı kullanarak her seferinde aynı yapı oluşur[29]. Bu da hataları azaltır ve "ortam uyuşmazlığı" problemlerini ortadan kaldırır. Ayrıca CI/CD pipeline'larına entegre edilen IaC adımları ile kodunuzla birlikte altyapı değişiklikleri de otomatik olarak yönetilebilir hale gelir. Kısaca, manuel işlem yerine otomasyon skriptleri ve şablonları kullanmak, hem zamandan kazandırır hem de insan hatalarını en aza indirir.

- **Sürüm Numaralandırma ve Git Etiketleme:** Yazılım projelerinde versiyon takibi için tutarlı bir sürümleme stratejisi uygulanmalıdır. **Anlamsal Sürümleme (Semantic Versioning)** yaygın bir yaklaşımdır: değişikliklerin etkisine göre major.minor.patch formatında numaralar artırılır (örn. `v1.2.0` -> `v1.2.1` küçük bir düzeltme, `v1.3.0` yeni bir özellik, `v2.0.0` geriye dönük uyumluluğu bozacak büyük değişiklik gibi). Bu versiyon numaralarını Git deposunda **etiket (tag)** olarak kullanmak, kod depo yönetiminde en iyi uygulamalardandır. Örneğin yayınladığınız her üretim sürümünü `vX.Y.Z` şeklinde Git tag ile işaretlerseniz, gelecekte belirli bir sürümün tam olarak hangi kod durumuna karşılık geldiğini kolayca bulabilirsiniz[30]. Git etiketleri depo içinde kalıcı işaretlerdir; belirli bir commit'i etiketlemek, o commit'in önemli bir sürüm olduğunu ifade eder. Semantic versioning ile birleştiğinde, etiket isimleri değişikliklerin büyüklüğünü ve önemini de yansıtır hale gelir[31][32]. Birçok araç ve platform (GitHub dahil) bu etiketleri "sürüm" olarak tanır ve sürüm notları, yayınlar oluştururken kullanmanıza imkân tanır[32]. Ayrıca mümkünse **otomatik sürüm artırma** ve **değişiklik günlüğü (changelog)** oluşturma süreçleri CI/CD'ye entegre edilmelidir. Örneğin, ana dala merge edilince patch versiyonunu artırıp yeni tag oluşturan bir iş akışı kurulabilir. Sonuç olarak, açık ve otomatikleştirilmiş bir sürümleme politikası, dağıtımlarınızın izlenebilirliğini artırır ve hem geliştiriciler hem de paydaşlar için netlik sağlar.

Kaynaklar: Kurumunuzun DevOps standartlarını belirlerken yararlanabileceğiniz ek kaynaklar: Microsoft tarafından sunulan Azure DevOps rehber dokümanları[3][9], GitHub resmi dokümantasyonu[8][12] ve sektörde kabul görmüş DevOps makaleleri ve en iyi uygulama önerileri[10][30]. Bu kaynaklarda anlatılan prensip ve araçlar, OBSS özelinde belirlenen standartlarla uyumlu olup, güncel DevOps yaklaşımlarına dayanmaktadır. DevOps süreçlerinizi bu en iyi uygulamalara göre sürekli gözden geçirip iyileştirmek, uzun vadede daha güvenilir, hızlı ve kaliteli yazılım teslimatı yapmanızı sağlayacaktır.

[1] [2] [3] [4] Git branching guidance - Azure Repos | Microsoft Learn

https://learn.microsoft.com/en-us/azure/devops/repos/git/git-branching-guidance?view=azure-devops

[5] [6] Kod inceleme (code review) için en iyi uygulamalar nelerdir? · Soru Cevap

https://qna.com.tr/soru-cevap/kod-inceleme-code-review-icin-en-iyi-uygulamalar-nelerdir

[7] [10] [11] [23] [24] [27] [28] [29] Developing Azure Compute Solutions: App Services, Functions, and Containerized Apps - Exam-Labs

https://www.exam-labs.com/blog/developing-azure-compute-solutions-app-services-functions-and-containerized-apps

[8] [9] Pipeline deployment approvals - Azure Pipelines | Microsoft Learn

https://learn.microsoft.com/en-us/azure/devops/pipelines/process/approvals?view=azure-devops

[12] [13] [14] Best Practices | AKS DevSecOps Workshop

https://azure.github.io/AKS-DevSecOps-Workshop/modules/Module3/intro.html

[15] Azure DevOps Security Best Practices - Blog - GitProtect.io

https://gitprotect.io/blog/azure-devops-security-best-practices/

[16] DevOps security considerations overview - Cloud Adoption Framework

https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/ready/considerations/security-considerations-overview

[17] [18] End-to-end governance in Azure when using CI/CD - Azure DevOps | Microsoft Learn

https://learn.microsoft.com/en-us/devops/operate/governance-cicd

[19] [20] [21] Best practices working with self-hosted GitHub Action runners at scale on AWS | AWS DevOps & Developer Productivity Blog

https://aws.amazon.com/blogs/devops/best-practices-working-with-self-hosted-github-action-runners-at-scale-on-aws/

[22] Securing GitHub Actions Workflows

https://wellarchitected.github.com/library/application-security/recommendations/actions-security/

[25] Best practices for Azure Monitor alerts - Microsoft Learn

https://learn.microsoft.com/en-us/azure/azure-monitor/alerts/best-practices-alerts

[26] 7 Best Practices for Azure Monitor Alerts - Economize Cloud

https://www.economize.cloud/blog/azure-monitor-alerts/

[30] [31] [32] Managing Releases with Semantic Versioning and Git Tags

https://www.gitkraken.com/gitkon/semantic-versioning-git-tags