Git & GitHub

Kullanım Rehberi

Başlangıçtan İleri Seviyeye — Kısaltmalar, Açıklamalar,
İpuçları ve GitHub Akışlarıyla

Versiyon kontrolü • Dal yönetimi • İşbirliği • CI/CD • GitHub Actions

Claude by Anthropic · 2025

Rehber İçeriği

1. Git Nedir? Temel Kavramlar

Git'in Kısa Tarihi

Git, 2005 yılında Linux çekirdeği geliştiricisi **Linus Torvalds** tarafından yaratılmış dağıtık bir versiyon kontrol sistemidir. Adı İngiliz argosu olan *git* (aptal, ahmak) sözcüğünden gelir — Torvalds'ın kendi mizahi beyanına göre. Bugün dünyada en yaygın kullanılan VCS (Version Control System) konumundadır.

GitHub ise 2008'de kurulan ve 2018'de Microsoft tarafından satın alınan bir bulut tabanlı Git barındırma platformudur. Git ≠ GitHub: Git bir araç, GitHub ise bu aracı kullanan bir platformdur.

Temel Kavramlar ve Tanımlar

Komut	Açılımı	Ne yapar?
`Repository (repo)`	*Depo / Proje dizini*	Git tarafından takip edilen tüm dosyaları ve geçmişi barındıran klasör
`Commit`	*Taahhüt / Anlık görüntü*	Değişikliklerin kalıcı olarak kaydedildiği, zaman damgalı snapshot
`Branch`	*Dal*	Ana geçmişten ayrılan bağımsız geliştirme hattı
`Merge`	*Birleştirme*	İki dalın değişikliklerini tek dala birleştirme
`Rebase`	*Yeniden temellendirme*	Dalın başlangıç noktasını başka bir commit'e taşıma
`Remote`	*Uzak depo*	İnternette (GitHub, GitLab vb.) barındırılan repo kopyası
`Clone`	*Klon*	Uzak reponun yerel makineye tam kopyası
`Fork`	*Çatal*	Başkasının reposunun kendi hesabınıza kopyalanması (GitHub)
`Pull Request (PR)`	*Çekme isteği*	Değişikliklerinizi ana repoya birleştirme teklifi (GitHub)
`Issue`	*Sorun / Görev*	Hata raporu, özellik isteği veya görev takip birimi (GitHub)
`HEAD`	*Baş / Mevcut konum*	Şu anda üzerinde çalıştığınız commit'i gösteren işaretçi
`Index / Stage`	*Sahne / Hazırlık alanı*	Commit edilmeden önce değişikliklerin bekletildiği ara bölge
`Working Tree`	*Çalışma ağacı*	Disk üzerinde gerçekten gördüğünüz dosyalar
`Stash`	*Gizli raf*	Henüz commit edilmemiş değişiklikleri geçici saklama alanı
`Tag`	*Etiket*	Belirli bir commit'e verilen kalıcı isim (genellikle sürüm için)
`Hook`	*Kanca*	Git olaylarında (commit, push vb.) otomatik çalışan script
`Blob`	*Binary Large OBject*	Git'in dosya içeriklerini sakladığı nesne türü
`Tree`	*Ağaç nesnesi*	Git'in dizin yapısını sakladığı nesne türü

Git'in Üç Bölgesi

Git, değişiklikleri üç ayrı bölgede yönetir. Bunları anlamak, birçok komutun neden var olduğunu açıklar:

1. Working Directory (Çalışma Dizini) Dosyaları düzenlediğiniz gerçek disk alanı. git add ile sahneye taşınır. **2. Staging Area / Index (Sahne)** Commit'e hazır değişikliklerin bekletildiği ara katman. git commit ile kaydedilir. **3. Repository / .git Dizini** Tüm commit geçmişinin ve meta verilerin kalıcı olarak saklandığı yer.

2. Kurulum ve İlk Yapılandırma

Git Kurulumu

Komut	Açılımı	Ne yapar?
`git --version`	*version = sürüm bilgisi*	Kurulu Git sürümünü gösterir
`brew install git`	*brew = Homebrew paket yöneticisi*	macOS'ta Homebrew ile Git kurar
`xcode-select --install`	*xcode-select = Xcode araçları*	macOS Komut Satırı Araçları (Git dahil) kurar

git config — Yapılandırma

git config = *Git CONFIGuration* — Git'in kimlik ve davranış ayarları. Üç kapsam vardır: **--system** (tüm kullanıcılar), **--global** (mevcut kullanıcı, ~/.gitconfig), **--local** (sadece bu repo, .git/config).

Komut	Açılımı	Ne yapar?
`git config --global user.name 'Ad Soyad'`	*--global user.name*	Tüm repolar için kullanıcı adı ayarlar
`git config --global user.email 'e@mail.com'`	*--global user.email*	Tüm repolar için e-posta ayarlar
`git config --global core.editor 'code --wait'`	*core.editor = varsayılan editör*	VS Code'u varsayılan editör yapar
`git config --global core.editor 'nano'`	*nano = basit terminal editörü*	nano'yu varsayılan editör yapar
`git config --global init.defaultBranch main`	*init.defaultBranch = dal adı*	Yeni repolarda varsayılan dal adını 'main' yapar
`git config --global pull.rebase false`	*pull.rebase = merge ile çek*	git pull davranışını merge olarak ayarlar
`git config --global pull.rebase true`	*pull.rebase = rebase ile çek*	git pull davranışını rebase olarak ayarlar
`git config --global core.autocrlf input`	*autocrlf = satır sonu dönüştür*	macOS/Linux için satır sonu ayarı
`git config --global alias.st status`	*alias = takma ad*	'git st' yazınca 'git status' çalışır
`git config --global alias.lg 'log --oneline --graph --all'`	*alias lg*	Görsel log için kısa komut tanımlar
`git config --global alias.undo 'reset HEAD~1 --mixed'`	*alias undo*	'git undo' ile son commit'i geri alır
`git config --list`	*--list = listele*	Tüm aktif Git ayarlarını listeler
`git config --list --show-origin`	*--show-origin = kaynak göster*	Her ayarın hangi dosyadan geldiğini gösterir
`git config user.name`	*tek değeri oku*	Belirli bir ayarın değerini gösterir
`git config --global --unset user.email`	*--unset = sil*	Belirli bir ayarı kaldırır
`cat ~/.gitconfig`	*~/.gitconfig dosyasını oku*	Global Git yapılandırma dosyasını görüntüler

 git config --global core.editor 'code --wait' ile VS Code'u editör yaparsanız, commit mesajlarını arayüzde yazabilirsiniz.

3. Repo Oluşturma ve Klonlama

git init — Yeni Repo Başlatma

git init = *Initialize* — Mevcut klasörde .git dizinini oluşturarak Git takibini başlatır.

Komut	Açılımı	Ne yapar?
`git init`	*init = initialize (başlat)*	Mevcut dizini Git reposu yapar (.git klasörü oluşturur)
`git init proje-adi`	*init + klasör adı*	Yeni klasör oluşturup içini repo yapar
`git init --bare repo.git`	*--bare = çıplak repo*	Çalışma ağacı olmayan sunucu tipi repo oluşturur
`ls -la .git/`	*.git içeriğini gör*	HEAD, config, objects, refs dizinlerini görürsünüz
`cat .git/HEAD`	*HEAD dosyasını oku*	Hangi dalda olduğunuzu gösterir
`cat .git/config`	*repo yapılandırmasını oku*	Bu repoya özgü Git ayarlarını gösterir

git clone — Repo Klonlama

git clone = *Clone* — Uzak reponun tam kopyasını yerel makineye indirir. Tüm geçmiş, dal ve etiketlerle birlikte.

Komut	Açılımı	Ne yapar?
`git clone https://github.com/u/repo.git`	*HTTPS protokolü*	HTTPS üzerinden klonlar (parola gerekebilir)
`git clone git@github.com:u/repo.git`	*SSH protokolü*	SSH anahtarıyla klonlar (önerilen)
`git clone URL proje-klasoru`	*hedef klasör adı*	Belirtilen klasör adıyla klonlar
`git clone --depth 1 URL`	*--depth = derinlik*	Sadece son commit ile yüzeysel klonlar (hızlı)
`git clone --depth 1 --branch v2.0 URL`	*--branch = belirli dal/etiket*	Belirli dal veya etiketi klonlar
`git clone --single-branch --branch main URL`	*--single-branch*	Sadece main dalını klonlar
`git clone --recurse-submodules URL`	*--recurse-submodules*	Alt modülleriyle birlikte klonlar
`git clone --mirror URL`	*--mirror = ayna*	Uzak reponun tam aynasını oluşturur

 SSH ile klonlamak için önce SSH anahtarı oluşturun: ssh-keygen -t ed25519 -C 'email@example.com' Ardından ~/.ssh/id_ed25519.pub içeriğini GitHub > Settings > SSH Keys'e ekleyin.

4. Dosya Takibi ve Commit

git status — Durum Kontrolü

git status = Working tree, stage ve takip durumunu gösterir.

Komut	Açılımı	Ne yapar?
`git status`	*status = durum*	Değiştirilen, sahnelenmiş ve izlenmeyen dosyaları gösterir
`git status -s`	*-s = short (kısa)*	Tek satırlık kısa durum formatı (M=modified, A=added, ?=untracked)
`git status -u`	*-u = untracked*	İzlenmeyen dosyaları da gösterir
`git status --branch`	*--branch = dal bilgisi*	Durum + uzak dal karşılaştırmasını gösterir

git add — Sahneye Alma

git add = *Add to index* — Dosyaları Staging Area'ya (sahneye) alır. Commit sadece sahnedeki değişiklikleri kaydeder.

Komut	Açılımı	Ne yapar?
`git add dosya.txt`	*add = ekle*	Tek dosyayı sahneye alır
`git add .`	*. = mevcut dizindeki her şey*	Tüm değiştirilmiş ve yeni dosyaları sahneye alır
`git add -A`	*-A = All (tümü)*	Silinen dosyalar dahil her şeyi sahneye alır
`git add *.txt`	*glob kalıbı*	Kalıpla eşleşen tüm dosyaları ekler
`git add src/`	*klasör adı*	Belirli klasörün tüm değişikliklerini ekler
`git add -p`	*-p = patch (parça parça)*	Değişiklikleri tek tek (hunk) inceleyerek seçer
`git add -i`	*-i = interactive*	Etkileşimli sahne menüsünü açar
`git add -u`	*-u = update tracked*	Sadece izlenen dosyaların güncellemelerini ekler
`git rm --cached dosya.txt`	*--cached = sadece index'ten sil*	Dosyayı sahneden çıkarır, diskten silmez
`git restore --staged d.txt`	*restore --staged = sahneyi geri al*	Dosyayı sahneden çıkarır (git add'i geri alır)

git diff — Fark Görüntüleme

Komut	Açılımı	Ne yapar?
`git diff`	*diff = difference (fark)*	Sahneye alınmamış değişiklikleri gösterir
`git diff --staged`	*--staged = sahnelenmiş*	Sahneye alınmış değişiklikleri gösterir
`git diff HEAD`	*HEAD = son commit*	Son commit ile mevcut durumu karşılaştırır
`git diff main feature`	*iki dal karşılaştır*	İki dal arasındaki farkları gösterir
`git diff HEAD~2 HEAD`	*HEAD~2 = 2 commit öncesi*	Son 2 commit'teki değişiklikleri gösterir
`git diff --stat`	*--stat = istatistik*	Detay yerine değişiklik özetini gösterir
`git diff --name-only`	*--name-only = sadece ad*	Değişen dosya adlarını listeler
`git diff dosya.txt`	*tek dosya farkı*	Sadece belirtilen dosyanın farkını gösterir

git commit — Değişiklikleri Kaydetme

git commit = *Commit* — Sahnedeki değişiklikleri kalıcı olarak Git geçmişine kaydeder. Her commit benzersiz bir SHA-1 hash alır.

Komut	Açılımı	Ne yapar?
`git commit -m 'mesaj'`	*-m = message (mesaj)*	Tek satır mesajla commit oluşturur
`git commit`	*editör açılır*	Varsayılan editörde çok satırlı mesaj yazar
`git commit -am 'mesaj'`	*-a = all, -m = message*	İzlenen dosyaları add+commit tek adımda yapar
`git commit --amend`	*--amend = düzelt/değiştir*	Son commit mesajını veya içeriğini değiştirir
`git commit --amend --no-edit`	*--no-edit = mesajı değiştirme*	Mesajı koruyarak son commit'e dosya ekler
`git commit --allow-empty -m 'trigger CI'`	*--allow-empty*	Değişiklik olmadan boş commit oluşturur
`git commit -S -m 'mesaj'`	*-S = GPG Sign (imzala)*	GPG anahtarıyla imzalı commit oluşturur

 Commit mesajı yazım kuralı: • İlk satır: 50 karakter özet (büyük harf, nokta yok) • Boş satır • Ayrıntılı açıklama (72 karakter sütun) Örnek: 'feat: kullanıcı giriş ekranı eklendi'

git log — Geçmiş İnceleme

Komut	Açılımı	Ne yapar?
`git log`	*log = günlük*	Tüm commit geçmişini gösterir
`git log --oneline`	*--oneline = tek satır*	Her commit'i hash + mesajla tek satırda gösterir
`git log --oneline --graph`	*--graph = dal grafiği*	Dal birleşmelerini ASCII ağaçla gösterir
`git log --oneline --graph --all`	*--all = tüm dallar*	Tüm dalları görsel olarak gösterir
`git log --oneline --graph --all --decorate`	*--decorate = dal adları*	Dal ve etiket adlarını da gösterir
`git log -5`	*-n = son n commit*	Son 5 commit'i gösterir
`git log -p`	*-p = patch*	Her commit'in fark içeriğini gösterir
`git log --stat`	*--stat = istatistik*	Her commit'te değişen dosyaları özetler
`git log --author='Ali'`	*--author = yazar filtresi*	Belirli yazara ait commit'leri filtreler
`git log --since='2024-01-01'`	*--since = tarihten beri*	Belirli tarihten sonraki commit'leri gösterir
`git log --until='2024-12-31'`	*--until = tarihe kadar*	Belirli tarihten önceki commit'leri gösterir
`git log --grep='hata'`	*--grep = mesajda ara*	Commit mesajında arama yapar
`git log -S 'fonksiyon_adi'`	*-S = pickaxe arama*	Belirli string ekleyen/silen commit'leri bulur
`git log --follow dosya.txt`	*--follow = dosyayı takip et*	Yeniden adlandırma dahil dosya geçmişini gösterir
`git log main..feature`	*iki dal arası*	feature'da olup main'de olmayan commit'leri gösterir
`git shortlog -sn`	*shortlog = kısa özet*	Yazarlara göre commit sayısını özetler
`git show HASH`	*show = göster*	Belirli commit'in içeriğini gösterir
`git show HEAD~2:dosya.txt`	*HEAD~2 + dosya yolu*	2 commit öncesindeki dosyanın içeriğini gösterir
`git blame dosya.txt`	*blame = sorumluyu bul*	Her satırın hangi commit'te kim tarafından yazıldığını gösterir
`git blame -L 10,20 dosya.txt`	*-L = line range*	Sadece 10-20. satırların blame bilgisini gösterir

 git log --oneline --graph --all --decorate komutunu ~/.gitconfig'e alias olarak ekleyin: git config --global alias.lg 'log --oneline --graph --all --decorate'

5. Dal (Branch) Yönetimi

Dal Nedir?

Dal (branch), commit geçmişinden bağımsız olarak ayrılan bir geliştirme hattıdır. Git'te dallar son derece hafiftir — aslında sadece bir commit'e işaret eden **41 baytlık bir dosyadır**. Bu yüzden dal oluşturmak ve geçiş yapmak anlık gerçekleşir.

> Yaygın Dal Stratejileri: **Git Flow:** main • develop • feature/* • release/* • hotfix/* **GitHub Flow:** main • feature/* (doğrudan PR ile birleşir) **Trunk Based:** main (kısa ömürlü feature dalları, feature flag'ler)

git branch — Dal Komutları

Komut	Açılımı	Ne yapar?
`git branch`	*branch = dal*	Yerel dalları listeler (* aktif dalı işaretler)
`git branch -a`	*-a = all (tümü)*	Yerel ve uzak tüm dalları listeler
`git branch -r`	*-r = remote*	Sadece uzak dalları listeler
`git branch -v`	*-v = verbose*	Her dalın son commit özeti ile listeler
`git branch -vv`	*-vv = daha ayrıntılı*	Uzak dal takip bilgisiyle listeler
`git branch yeni-dal`	*branch + ad*	Mevcut commit'ten yeni dal oluşturur (geçmez)
`git branch yeni-dal main`	*branch + ad + başlangıç*	main'den başlayan yeni dal oluşturur
`git branch -d dal`	*-d = delete*	Birleştirilmiş dalı güvenli şekilde siler
`git branch -D dal`	*-D = force delete*	Birleştirilmemiş olsa bile dalı zorla siler
`git branch -m eski yeni`	*-m = move/rename*	Dalı yeniden adlandırır
`git branch -M eski yeni`	*-M = force rename*	Hedef dal varsa bile yeniden adlandırır
`git branch --merged`	*--merged = birleşmiş*	Mevcut dala zaten birleşmiş dalları listeler
`git branch --no-merged`	*--no-merged = birleşmemiş*	Henüz birleşmemiş dalları listeler
`git push origin --delete dal`	*uzak dalı sil*	GitHub'daki uzak dalı siler

git checkout ve git switch — Dal Geçişi

git switch Git 2.23 ile geldi ve dal geçişi için önerilen modern komuttur. **git checkout** hâlâ geçerlidir ve dal geçişi dışında da kullanılır.

Komut	Açılımı	Ne yapar?
`git switch dal-adi`	*switch = değiştir*	Belirtilen dala geçer (modern)
`git switch -c yeni-dal`	*-c = create*	Dal oluşturup geçer (modern)
`git switch -c yeni main`	*-c + başlangıç noktası*	main'den yeni dal oluşturup geçer
`git switch -`	*- = önceki dal*	Bir önceki dala geri döner
`git checkout dal-adi`	*checkout = geçiş yap*	Belirtilen dala geçer (klasik)
`git checkout -b yeni-dal`	*-b = branch*	Oluştur ve geç (klasik)
`git checkout -b dal origin/d`	*-b + uzak kaynak*	Uzak dalı takip ederek yerel dal oluşturur
`git checkout HASH`	*commit hash ile geçiş*	Detached HEAD modunda commit'e gider
`git checkout HEAD~3`	*HEAD~3 = 3 önceki commit*	3 commit öncesine gider (detached HEAD)
`git checkout -- dosya.txt`	*-- = dosya yolu işareti*	Dosyayı son commit haline geri döndürür

 'Detached HEAD' durumunda (checkout ile commit'e gidince) yaptığınız değişiklikler için mutlaka git switch -c yeni-dal yazın, aksi hâlde kaybolur.

6. Birleştirme: Merge ve Rebase

git merge — Dalları Birleştirme

git merge, hedef dalın değişikliklerini mevcut dala birleştirir. İki türü vardır: **Fast-forward** (çatallanma yoksa doğrudan ilerler) ve **3-way merge** (iki dalda ayrı değişiklik varsa yeni merge commit oluşturur).

Komut	Açılımı	Ne yapar?
`git merge feature`	*merge = birleştir*	feature dalını mevcut dala birleştirir
`git merge --no-ff feature`	*--no-ff = no fast-forward*	Her zaman merge commit oluşturur (geçmişi korur)
`git merge --ff-only feature`	*--ff-only = sadece fast-forward*	Sadece FF mümkünse birleştirir, değilse hata verir
`git merge --squash feature`	*--squash = sıkıştır*	Tüm commit'leri tek commit'e sıkıştırır
`git merge --abort`	*--abort = vazgeç*	Çakışma durumunda birleştirmeyi iptal eder
`git merge --continue`	*--continue = devam et*	Çakışma çözüldükten sonra birleştirmeye devam eder
`git merge -X ours feature`	*-X ours = bizimkini tercih et*	Çakışmalarda mevcut dalın içeriğini seçer
`git merge -X theirs feature`	*-X theirs = diğerini tercih et*	Çakışmalarda birleştirilen dalın içeriğini seçer

 Çakışma (conflict) oluştğunda Git dosyalara işaretler ekler: <<<<<<< HEAD (mevcut daldaki içerik) ======= (gelen daldaki içerik) >>>>>>> feature Bu işaretleri manuel düzenleyip git add + git merge --continue yapın.

git rebase — Yeniden Temellendirme

git rebase, dalınızın başlangıç noktasını (base) başka bir commit'e taşır. Doğrusal bir geçmiş oluşturur.
Altın kural: Push edilmiş commit'leri asla rebase etmeyin!

Komut	Açılımı	Ne yapar?
`git rebase main`	*rebase = yeniden temellendir*	Mevcut dalı main'in üzerine taşır
`git rebase --onto B A feature`	*--onto = hedef belirt*	feature dalını A yerine B'nin üstüne taşır
`git rebase --abort`	*--abort = iptal et*	Rebase'i iptal edip önceki duruma döner
`git rebase --continue`	*--continue = devam et*	Çakışma çözüldükten sonra rebase'e devam eder
`git rebase --skip`	*--skip = atla*	Mevcut commit'i atlayarak rebase'e devam eder
`git rebase -i HEAD~3`	*-i = interactive, HEAD~3 = 3 geri*	Son 3 commit'i etkileşimli düzenler
`git rebase -i HEAD~5`	*-i ile 5 commit düzenle*	Editörde pick/squash/reword/drop seçenekleri sunulur

```
Etkileşimli Rebase (-i) Komutları: pick = commit'i olduğu gibi al reword = commit mesajını değiştir edit = commit'i düzenle (git commit --amend) squash = önceki commit ile birleştir (mesajları birleştir) fixup = önceki commit ile birleştir (mesajı sil) drop = commit'i tamamen sil
```

⚠ **Rebase, commit hash'lerini DEĞİŞTİRİR. Başkaları da üzerinde çalışıyorsa push edilmiş commit'leri rebase etmek tehlikelidir!**

git cherry-pick — Seçici Commit Alma

Komut	Açılımı	Ne yapar?
`git cherry-pick HASH`	*cherry-pick = kiraz topla*	Başka daldaki tek commit'i mevcut dala ekler
`git cherry-pick A..B`	*A..B = commit aralığı*	A'dan B'ye kadar commit'leri (A hariç) ekler
`git cherry-pick A^..B`	*A^..B = A dahil aralık*	A dahil B'ye kadar commit'leri ekler
`git cherry-pick --no-commit H`	*--no-commit = commit etme*	Değişiklikleri uygular ama commit etmez
`git cherry-pick --abort`	*--abort = iptal*	Cherry-pick'i iptal eder
`git cherry-pick --continue`	*--continue = devam*	Çakışma çözülünce devam eder

7. Uzak Repo (Remote) ve Senkronizasyon

git remote — Uzak Repo Yönetimi

Remote, yerel reponun bağlı olduğu uzak (GitHub/GitLab vb.) kopyadır. Varsayılan uzak adı **origin** olur.

Komut	Açılımı	Ne yapar?
`git remote`	*remote = uzak*	Bağlı uzak repoları listeler
`git remote -v`	*-v = verbose*	Uzak repoların URL'lerini gösterir
`git remote add origin URL`	*add = ekle, origin = varsayılan ad*	Uzak repo bağlantısı ekler
`git remote add upstream URL`	*upstream = kaynak repo*	Fork'ta kaynak repoya bağlantı ekler
`git remote set-url origin URL`	*set-url = adresi güncelle*	Mevcut uzak repo adresini değiştirir
`git remote rename origin yeni`	*rename = yeniden adlandır*	Uzak repo adını değiştirir
`git remote remove origin`	*remove = kaldır*	Uzak repo bağlantısını siler
`git remote show origin`	*show = göster*	Uzak repo hakkında detaylı bilgi gösterir
`git remote prune origin`	*prune = buda*	Silinmiş uzak dalların izlerini temizler

git fetch, pull, push — Senkronizasyon

Komut	Açılımı	Ne yapar?
`git fetch`	*fetch = getir*	Uzak değişiklikleri indirir ama birleştirmez
`git fetch origin`	*fetch + uzak ad*	origin'deki tüm dal güncellemelerini indirir
`git fetch --all`	*--all = tüm uzaklar*	Tüm uzak repoları günceller
`git fetch --prune`	*--prune = buda*	Silinmiş uzak dalları yereldekilere yansıtır
`git fetch origin main:main`	*uzak:yerel dal eşleme*	Uzak main'i yerel main'e indirir
`git pull`	*pull = çek*	fetch + merge işlemini tek adımda yapar
`git pull origin main`	*pull + uzak + dal*	origin/main'i çekip mevcut dala birleştirir
`git pull --rebase`	*--rebase = rebase ile çek*	fetch + rebase (temiz doğrusal geçmiş)
`git pull --rebase origin main`	*rebase pull + kaynak*	origin/main'i rebase ile entegre eder
`git pull --ff-only`	*--ff-only = sadece FF*	Sadece fast-forward mümkünse çeker
`git push`	*push = it*	Mevcut dalı takip ettiği uzak dala gönderir
`git push origin main`	*push + uzak + dal*	Yerel main'i origin/main'e gönderir
`git push -u origin main`	*-u = set-upstream*	İlk push; takip bağlantısı kurar
`git push origin local:remote`	*yerel:uzak dal eşleme*	Farklı adlı dala push eder
`git push --force-with-lease`	*force-with-lease = güvenli zorla*	Başkası push etmediyse zorla push eder
`git push --force`	*--force = zorla it*	Uzak geçmişin üzerine yazar (TEHLİKELİ!)
`git push origin --delete dal`	*--delete = uzak dalı sil*	GitHub'daki uzak dalı siler
`git push origin --tags`	*--tags = etiketleri gönder*	Tüm yerel etiketleri uzak repoya gönderir
`git push origin v1.0.0`	*tek etiket gönder*	Belirli etiketi uzak repoya gönderir

⚠ **git push --force yerine daima git push --force-with-lease kullanın! --force, takım arkadaşlarının commit'lerini silebilir.**

8. Geri Alma ve Tarih Yönetimi

git restore — Dosya Geri Alma

git restore Git 2.23 ile geldi; dosyaları geri almak için önerilen modern komuttur.

Komut	Açılımı	Ne yapar?
`git restore dosya.txt`	*restore = geri yükle*	Çalışma dizinindeki değişikliği geri alır
`git restore .`	*. = tüm dosyalar*	Tüm izlenen dosyaları son commit'e döndürür
`git restore --staged dosya.txt`	*--staged = sahneden al*	Dosyayı sahneden çıkarır (add'i geri alır)
`git restore --source HEAD~2 d.txt`	*--source = kaynak commit*	Dosyayı 2 commit önceki haline döndürür
`git restore --source HASH d.txt`	*belirli commit'ten geri yükle*	Dosyayı istenen commit haline döndürür

git reset — Commit Geri Alma

git reset, HEAD işaretçisini belirli bir commit'e taşır. Üç modu vardır:

```
--soft → Commit geri alınır, değişiklikler sahneye döner (index korunur) --mixed →
Commit + sahne geri alınır, değişiklikler çalışma dizininde kalır (varsayılan) --hard →
Commit + sahne + çalışma dizini tamamen sıfırlanır (değişiklikler kaybolur!)
```

Komut	Açılımı	Ne yapar?
`git reset --soft HEAD~1`	*--soft HEAD~1 = 1 commit yumuşak geri*	Son commit'i geri alır, değişiklikler sahnede kalır
`git reset --mixed HEAD~1`	*--mixed = varsayılan mod*	Son commit'i ve sahneyi geri alır (değişiklikler dizinde)
`git reset --hard HEAD~1`	*--hard = sert sıfırlama*	Son commit + tüm değişiklikler tamamen silinir
`git reset --hard HEAD`	*HEAD = mevcut commit*	Sahne ve çalışma dizinini temizler (commit korunur)
`git reset --hard origin/main`	*uzak dala sıfırla*	Yerel main'i origin/main ile tam eşler
`git reset HEAD~3 --soft`	*3 commit birleştirmek için*	Son 3 commit'i birleştirmeye hazırlar
`git reset HASH`	*belirli commit'e git*	HEAD'i belirli commit'e taşır

⚠ git reset --hard GERİ ALINAMAZ! Kaybolacak değişiklikleriniz varsa önce git stash veya ayrı dal oluşturun.

git revert — Güvenli Geri Alma

git revert, belirli bir commit'in etkisini geri alan YENİ bir commit oluşturur. Push edilmiş commit'leri geri almanın güvenli yoludur.

Komut	Açılımı	Ne yapar?
`git revert HASH`	*revert = geri döndür*	Commit'i geri alan yeni bir commit oluşturur
`git revert HEAD`	*HEAD = son commit*	Son commit'i güvenle geri alır
`git revert HEAD~2..HEAD`	*aralık revert*	Son 2 commit'i tek tek geri alır
`git revert --no-commit HASH`	*--no-commit = commit etme*	Değişiklikleri uygular ama commit etmez
`git revert -m 1 MERGE_HASH`	*-m 1 = ana dal tarafını koru*	Merge commit'i geri alır (ana dalı korur)
`git revert --abort`	*--abort = iptal*	Revert işlemini iptal eder

Genel kural: Yerel commit → git reset | Push edilmiş commit → git revert

git stash — Geçici Saklama

git stash = *Stash* — Commit edilmemiş değişiklikleri geçici bir rafa kaldırır. Dal değiştirmeniz gerektiğinde çok kullanışlıdır.

Komut	Açılımı	Ne yapar?
`git stash`	*stash = sakla*	İzlenen dosyalardaki değişiklikleri saklar
`git stash push -m 'mesaj'`	*push -m = mesajlı sakla*	Açıklama ekleyerek saklar
`git stash -u`	*-u = untracked (izlenmeyenler)*	İzlenmeyen dosyaları da saklar
`git stash -a`	*-a = all (gitignore dahil)*	Tüm dosyaları (gitignore edilenler dahil) saklar
`git stash list`	*list = listele*	Saklanan tüm değişiklikleri gösterir
`git stash show`	*show = göster*	En son stash'in içeriğini gösterir
`git stash show -p`	*-p = patch*	En son stash'in detaylı içeriğini gösterir
`git stash pop`	*pop = al ve sil*	Son stash'i uygular ve listeden çıkarır
`git stash apply`	*apply = uygula*	Son stash'i listeden silmeden uygular
`git stash apply stash@{2}`	*stash@{n} = belirli stash*	Belirli stash'i uygular
`git stash drop stash@{0}`	*drop = bırak/sil*	Belirli stash'i listeden siler
`git stash clear`	*clear = temizle*	Tüm stash listesini siler
`git stash branch yeni-dal`	*branch = yeni dala al*	Stash'i yeni dal oluşturarak uygular

9. Etiket (Tag) ve Sürüm Yönetimi

git tag — Sürüm Etiketleme

Tag iki türdür: **Lightweight tag** (sadece isim, commit işaretçisi) ve **Annotated tag** (isim + mesaj + imzalayan kişi + tarih — önerilen).

Komut	Açılımı	Ne yapar?
`git tag`	*tag = etiket*	Tüm etiketleri listeler
`git tag -l 'v1.*'`	*-l = list + kalıp*	Kalıpla eşleşen etiketleri listeler
`git tag v1.0.0`	*lightweight tag*	Mevcut commit'e basit etiket ekler
`git tag -a v1.0.0 -m 'mesaj'`	*-a = annotated, -m = message*	Açıklamalı (önerilen) etiket ekler
`git tag -a v1.0.0 HASH`	*belirli commit'e etiket*	Geçmişteki bir commit'i etiketler
`git tag -s v1.0.0 -m 'mesaj'`	*-s = sign (GPG imzala)*	GPG ile imzalı etiket oluşturur
`git show v1.0.0`	*etiket bilgisi göster*	Etiketin detaylarını ve commit içeriğini gösterir
`git push origin v1.0.0`	*tek etiketi gönder*	Belirli etiketi GitHub'a gönderir
`git push origin --tags`	*--tags = tüm etiketler*	Tüm yerel etiketleri gönderir
`git push origin :refs/tags/v1`	*etiket silme (push ile)*	Uzak etiketi siler
`git tag -d v1.0.0`	*-d = delete*	Yerel etiketi siler
`git checkout v1.0.0`	*etikete geçiş*	Detached HEAD modunda etikete gider
`git checkout -b v1-branch v1.0.0`	*etiket üzerine dal aç*	Etiket üzerinden yeni dal oluşturur

 Semantik sürümleme (SemVer) kullanın: v1.2.3 → MAJÖr.MİNÖR.YAMA Breaking change → MAJÖR | Yeni özellik → MİNÖR | Hata düzeltme → YAMA

10. Gelişmiş Git Teknikleri

git bisect — Hata Bulma

git bisect = *Binary Search + bisect (ikiye böl)* — Hangi commit'te bir hatanın girdiğini ikili arama ile bulur. O(log n) karmaşıklığında.

Komut	Açılımı	Ne yapar?
`git bisect start`	*start = başlat*	Bisect oturumunu başlatır
`git bisect bad`	*bad = kötü (mevcut commit hatalı)*	Mevcut commit'i hatalı olarak işaretler
`git bisect good v1.0.0`	*good = iyi (bu commit iyi)*	Bilinen iyi commit'i belirtir
`git bisect good`	*iyi olduğunu onayla*	Git'in getirdiği commit iyiyse yazılır
`git bisect bad`	*kötü olduğunu onayla*	Git'in getirdiği commit hatalıysa yazılır
`git bisect reset`	*reset = sıfırla*	Bisect biter, HEAD önceki konuma döner
`git bisect log`	*log = oturum günlüğü*	Bisect oturumunun geçmişini gösterir
`git bisect run pytest test.py`	*run = otomasyon*	Testi otomatik çalıştırarak bisect yapar

git submodule — Alt Modüller

Komut	Açılımı	Ne yapar?
`git submodule add URL klasor`	*submodule add = alt modül ekle*	Başka repoyu proje içine alt modül olarak ekler
`git submodule init`	*init = başlat*	Alt modül yapılandırmasını başlatır
`git submodule update`	*update = güncelle*	Alt modülleri kayıtlı commit'e günceller
`git submodule update --init --recursive`	*--recursive*	İç içe alt modülleri de başlatır
`git submodule foreach git pull`	*foreach = hepsi için*	Tüm alt modüllerde komut çalıştırır
`git submodule status`	*status = durum*	Alt modüllerin durumunu listeler
`git submodule deinit klasor`	*deinit = kaldır*	Alt modülü devre dışı bırakır
`git rm klasor`	*alt modül dosyasını sil*	Alt modülü projeden kaldırır

git worktree — Çoklu Çalışma Ağacı

Komut	Açılımı	Ne yapar?
`git worktree add ../yol dal`	*worktree add = yeni ağaç*	Farklı dizinde başka dal için çalışma ağacı açar
`git worktree list`	*worktree list = listele*	Tüm bağlı çalışma ağaçlarını listeler
`git worktree remove ../yol`	*worktree remove = kaldır*	Çalışma ağacını kaldırır
`git worktree prune`	*prune = buda*	Geçersiz çalışma ağacı kayıtlarını temizler

 git worktree ile aynı repo üzerinde birden fazla dalda eş zamanlı çalışabilirsiniz. stash'e gerek kalmaz.

git reflog — Tüm Hareketlerin Günlüğü

git reflog = *Reference Log* — HEAD'in tüm hareketlerini kaydeder. Yanlışlıkla silinen commit'leri kurtarmak için son çaredir.

Komut	Açılımı	Ne yapar?
`git reflog`	*reflog = referans günlüğü*	HEAD'in tüm geçmiş konumlarını gösterir
`git reflog show main`	*show + dal adı*	main dalının hareket geçmişini gösterir
`git checkout HEAD@{3}`	*HEAD@{n} = n hareket öncesi*	Belirli reflog konumuna gider
`git reset --hard HEAD@{2}`	*reflog + hard reset*	HEAD'i 2 hareket öncesine sıfırlar
`git branch kurtarilan HEAD@{5}`	*reflog + dal oluştur*	Kayıp commit'ten yeni dal oluşturarak kurtarır

Diğer Gelişmiş Komutlar

Komut	Açılımı	Ne yapar?
`git clean -fd`	*clean = temizle, -f force -d dir*	İzlenmeyen dosya ve klasörleri siler
`git clean -n`	*-n = dry run*	Silinecekleri gösterir, silmez
`git archive --format=zip HEAD > proje.zip`	*archive = arşivle*	Repoyu zip dosyası olarak dışa aktarır
`git shortlog -sn`	*shortlog = kısa özet*	Yazarlara göre commit sayısını özetler
`git count-objects -v`	*count-objects = nesne say*	Repo'nun veritabanı boyutunu gösterir
`git gc`	*gc = Garbage Collect*	Gereksiz nesneleri temizler, repo'yu sıkıştırır
`git fsck`	*fsck = File System CheckK*	Repo bütünlüğünü kontrol eder
`git rev-parse HEAD`	*rev-parse = referansı çözümle*	Mevcut commit'in tam hash'ini gösterir
`git ls-files`	*ls-files = dosyaları listele*	Git'in takip ettiği tüm dosyaları listeler
`git grep 'kelime'`	*git grep = izlenen dosyalarda ara*	Sadece Git'in takip ettiği dosyalarda arar
`git notes add -m 'not' HASH`	*notes = commit'e not ekle*	Commit'e ek not ekler (commit'i değiştirmez)

11. GitHub — Temel Kullanım

GitHub CLI (gh) — Komut Satırından GitHub

GitHub CLI, GitHub işlemlerini terminal üzerinden yapmanızı sağlar. Kurulum: **brew install gh → gh auth login**

Komut	Açılımı	Ne yapar?
`gh auth login`	*auth login = giriş yap*	GitHub hesabınızla kimlik doğrular
`gh auth status`	*auth status = oturum durumu*	Mevcut GitHub oturumunu gösterir
`gh repo create`	*repo create = repo oluştur*	Etkileşimli olarak yeni GitHub reposu oluşturur
`gh repo create ad --public`	*--public = herkese açık*	Herkese açık repo oluşturur
`gh repo create ad --private`	*--private = gizli*	Gizli repo oluşturur
`gh repo clone kullanici/repo`	*repo clone = klonla*	GitHub reposunu klonlar
`gh repo fork kullanici/repo`	*repo fork = fork et*	Repoyu kendi hesabınıza fork eder
`gh repo view`	*repo view = görüntüle*	Mevcut repoyu tarayıcıda açar
`gh repo list`	*repo list = repolarım*	Kendi repolarınızı listeler
`gh repo delete kullanici/repo`	*repo delete = sil*	Repoyu siler (onay gerektirir)
`gh browse`	*browse = tarayıcıda aç*	Mevcut repoyu GitHub'da tarayıcıda açar
`gh browse --settings`	*--settings = ayarlar*	Repo ayarlarını tarayıcıda açar

GitHub Issues — Sorun Takibi

Komut	Açılımı	Ne yapar?
`gh issue list`	*issue list = sorunları listele*	Açık issue'ları listeler
`gh issue list --state closed`	*--state closed = kapalılar*	Kapalı issue'ları listeler
`gh issue create`	*issue create = sorun aç*	Etkileşimli olarak yeni issue oluşturur
`gh issue create -t 'Başlık' -b 'Açıklama'`	*-t = title, -b = body*	Başlık ve gövdeyle issue oluşturur
`gh issue view 42`	*issue view = issue görüntüle*	42 numaralı issue'yu gösterir
`gh issue close 42`	*issue close = kapat*	Issue'yu kapatır
`gh issue reopen 42`	*issue reopen = yeniden aç*	Kapalı issue'yu yeniden açar
`gh issue comment 42 -b 'metin'`	*issue comment = yorum ekle*	Issue'ya yorum ekler
`gh issue edit 42 --title 'Yeni'`	*issue edit = düzenle*	Issue başlığını veya içeriğini düzenler
`gh issue pin 42`	*issue pin = sabitle*	Issue'yu reponun en üstüne sabitler
`gh issue develop 42`	*issue develop = dal oluştur*	Issue için bağlantılı dal oluşturur

GitHub Releases — Sürüm Yayınlama

Komut	Açılımı	Ne yapar?
`gh release create v1.0.0`	*release create = sürüm oluştur*	Etiket üzerinden GitHub Release oluşturur
`gh release create v1.0.0 --title 'v1.0.0' --notes 'Değişiklikler'`	*--notes*	Notlarla sürüm oluşturur
`gh release create v1.0.0 dosya.zip`	*dosya ekleme*	Sürüme dosya (asset) ekler
`gh release list`	*release list = sürümleri listele*	Tüm sürümleri listeler
`gh release view v1.0.0`	*release view = sürüm görüntüle*	Sürüm detaylarını gösterir
`gh release download v1.0.0`	*release download = indir*	Sürüm dosyalarını indirir
`gh release delete v1.0.0`	*release delete = sil*	Sürümü siler
`gh release upload v1.0.0 dosya.zip`	*release upload = dosya ekle*	Mevcut sürüme yeni dosya ekler

12. GitHub — İşbirliği ve Pull Request

Pull Request (PR) Akışı

Pull Request, kendi dalınızdaki değişiklikleri başka bir dala birleştirme teklifinizdir.

```
Standart PR Akışı: 1. git checkout -b feature/yeni-ozellik → Yeni dal oluştur 2. (değişiklikler yapılır...) 3. git add . && git commit -m 'feat: açıklama' → Commit et 4. git push -u origin feature/yeni-ozellik → GitHub'a gönder 5. gh pr create → PR oluştur 6. (code review, düzeltmeler...) 7. gh pr merge 42 --squash → PR'ı birleştir 8. git switch main && git pull → Yerel main'i güncelle 9. git branch -d feature/yeni-ozellik → Yerel dalı sil
```

gh pr — Pull Request Komutları

Komut	Açılımı	Ne yapar?
`gh pr create`	*pr create = PR oluştur*	Etkileşimli PR oluşturur
`gh pr create -t 'Başlık' -b 'Gövde'`	*-t title, -b body*	Başlık ve açıklamayla PR oluşturur
`gh pr create --base main --head feature`	*--base / --head*	Kaynak ve hedef dalı belirterek PR açar
`gh pr create --draft`	*--draft = taslak*	İncelemeye hazır olmayan taslak PR oluşturur
`gh pr create --reviewer ali,mehmet`	*--reviewer = inceleyenler*	İnceleyecek kişileri belirterek PR açar
`gh pr list`	*pr list = PR listele*	Açık PR'ları listeler
`gh pr list --state all`	*--state all = tümü*	Tüm durumlardan PR'ları listeler
`gh pr view 42`	*pr view = görüntüle*	42 numaralı PR'ı gösterir
`gh pr view 42 --web`	*--web = tarayıcıda aç*	PR'ı tarayıcıda açar
`gh pr checkout 42`	*pr checkout = dala geç*	PR dalını yerel makineye indirir ve geçer
`gh pr review 42 --approve`	*pr review --approve = onayla*	PR'ı onaylar
`gh pr review 42 --request-changes`	*--request-changes = değişiklik iste*	PR'a değişiklik talep eder
`gh pr review 42 --comment -b 'metin'`	*--comment = yorum yap*	PR'a yorum ekler
`gh pr merge 42`	*pr merge = birleştir*	PR'ı birleştirir
`gh pr merge 42 --squash`	*--squash = sıkıştırarak birleştir*	Tüm commit'leri tek commit'te birleştirir
`gh pr merge 42 --rebase`	*--rebase = rebase ile birleştir*	Rebase yaparak birleştirir
`gh pr merge 42 --merge`	*--merge = merge commit ile*	Merge commit oluşturarak birleştirir
`gh pr close 42`	*pr close = kapat*	PR'ı birleştirmeden kapatır
`gh pr status`	*pr status = durumum*	Sizinle ilgili PR'ları özetler
`gh pr diff 42`	*pr diff = fark göster*	PR'ın tüm değişikliklerini gösterir

Fork Tabanlı İş Akışı

Açık kaynak projelere katkıda bulunmak için fork akışı kullanılır.

Komut	Açılımı	Ne yapar?
`gh repo fork kullanici/repo --clone`	*fork + klonla*	Forklayıp yerel makineye klonlar
`git remote add upstream https://github.com/kullanici/repo.git`	*upstream = kaynak*	Orijinal repoya bağlantı ekler
`git fetch upstream`	*upstream'i getir*	Orijinal repodaki güncellemeleri indirir
`git merge upstream/main`	*upstream/main'i birleştir*	Orijinal değişiklikleri fork'unuza entegre eder
`git rebase upstream/main`	*upstream üzerine rebase*	Fork'unuzu orijinal repo üzerine yeniden temellendirir
`git push origin main`	*kendi fork'unuzu güncelle*	Senkronize edilmiş fork'u GitHub'a gönderir

 Düzenli olarak 'git fetch upstream && git rebase upstream/main' yaparak fork'unuzu güncel tutun.

13. GitHub Actions — CI/CD

GitHub Actions Nedir?

GitHub Actions, GitHub'a entegre CI/CD (Continuous Integration / Continuous Deployment) platformudur. **Workflow** dosyaları **.github/workflows/** klasöründe YAML formatında saklanır ve tetikleyici olaylara (push, PR, zamanlama vb.) göre çalışır.

> Temel Kavramlar: **Workflow** → Otomasyon sürecinin tamamı (.yml dosyası) **Job** → Workflow içindeki bağımsız çalışma birimi **Step** → Job içindeki tek bir adım (komut veya action) **Action** → Yeniden kullanılabilir adım paketi (marketplace'ten) **Runner** → Job'ların çalıştığı sanal makine (ubuntu-latest vb.)

Örnek Workflow: Test ve Dağıtım

Aşağıda push ve PR'da test çalıştıran temel bir workflow örneği verilmiştir:

```
# .github/workflows/ci.yml name: CI Pipeline on: push: branches: [ main, develop ] pull_request: branches: [ main ] jobs: test: runs-on: ubuntu-latest steps: - uses: actions/checkout@v4 - uses: actions/setup-python@v5 with: python-version: '3.12' - run: pip install -r requirements.txt - run: pytest --cov=. --cov-report=xml deploy: needs: test if: github.ref == 'refs/heads/main' runs-on: ubuntu-latest steps: - uses: actions/checkout@v4 - run: ./deploy.sh
```

gh workflow — Actions Yönetimi

Komut	Açılımı	Ne yapar?
`gh workflow list`	*workflow list = listele*	Repodaki tüm workflow'ları listeler
`gh workflow view ci.yml`	*workflow view = görüntüle*	Workflow detaylarını gösterir
`gh workflow run ci.yml`	*workflow run = çalıştır*	Workflow'u manuel tetikler
`gh run list`	*run list = çalışmaları listele*	Tüm workflow çalışmalarını listeler
`gh run view 123456`	*run view = çalışma detayı*	Belirli çalışmanın detaylarını gösterir
`gh run view 123456 --log`	*--log = günlük*	Çalışma günlüklerini gösterir
`gh run watch`	*run watch = canlı izle*	Mevcut çalışmayı canlı izler
`gh run rerun 123456`	*run rerun = yeniden çalıştır*	Başarısız çalışmayı yeniden tetikler
`gh run cancel 123456`	*run cancel = iptal et*	Devam eden çalışmayı iptal eder
`gh run download 123456`	*run download = artefakt indir*	Çalışmanın artefaktlarını indirir
`gh secret set API_KEY`	*secret set = gizli değer ekle*	Repo gizli değişkeni (secret) tanımlar
`gh secret list`	*secret list = gizli değerleri listele*	Tanımlı secret'ları listeler
`gh variable set ENV=prod`	*variable set = değişken ekle*	Repo değişkeni tanımlar

14. .gitignore ve Git Konfigürasyonu

.gitignore — Dosyaları Hariç Tutma

.gitignore, Git'in takip etmemesi gereken dosya ve klasörleri belirtir. Repo kökünde ve alt klasörlerde bulunabilir.

```
# Python için örnek .gitignore __pycache__/ # Derleme önbelleği (klasör) *.pyc # Python bytecode dosyaları *.pyo # Optimize edilmiş bytecode .env # Ortam değişkenleri (ASLA push etmeyin!) .venv/ # Sanal ortam dist/ # Derleme çıktısı build/ # Build klasörü *.egg-info/ # Paket meta verisi .pytest_cache/ # Pytest önbelleği .DS_Store # macOS meta verisi *.log # Log dosyaları !important.log # ! = bu dosyayı hariç tutma (takip et)
```

Komut	Açılımı	Ne yapar?
`touch .gitignore`	*.gitignore oluştur*	Repo kökünde .gitignore dosyası oluşturur
`curl -s https://www.toptal.com/developers/gitignore/api/python > .gitignore`	*gitignore.io*	Hazır .gitignore şablonu indirir
`git check-ignore -v dosya.txt`	*check-ignore = neden ignore?*	Dosyanın neden ignore edildiğini gösterir
`git rm -r --cached .`	*--cached = sadece index'ten kaldır*	Hatalı takip edilen dosyaları temizler
`git add . && git commit -m 'fix: .gitignore güncellendi'`	*temizlik sonrası commit*	Index temizliğini commit eder

 Hassas bilgiler (.env, API anahtarları, şifreler) ASLA repo'ya commit edilmemelidir. git history'de kalır ve hard reset ile bile tamamen silinemez.

.gitattributes — Dosya Öznitelikleri

.gitattributes, dosyaların nasıl işleneceğini belirtir: satır sonu dönüşümü, diff davranışı, birleştirme stratejisi vb.

```
# .gitattributes örneği * text=auto # Tüm metin dosyaları: otomatik satır sonu *.sh text eol=lf # Shell dosyaları: LF satır sonu *.bat text eol=crlf # Batch dosyaları: CRLF satır sonu *.png binary # PNG: binary (LF dönüşümü yok) *.jpg binary # JPG: binary *.pdf binary # PDF: binary CHANGELOG.md merge=union # Birleştirmede her iki tarafı al *.cs diff=csharp # C# dosyaları için diff algoritması
```

Yararlı Git Konfigürasyon İpuçları

Komut	Açılımı	Ne yapar?
`git config --global color.ui auto`	*color.ui = renk*	Git çıktılarını renklendirir
`git config --global core.pager 'less -R'`	*core.pager = sayfalayıcı*	Uzun çıktılar için sayfalayıcı ayarlar
`git config --global merge.tool vimdiff`	*merge.tool = birleştirme aracı*	Çakışma çözüm aracını ayarlar
`git config --global diff.tool vimdiff`	*diff.tool = fark aracı*	Fark görüntüleme aracını ayarlar
`git config --global rerere.enabled true`	*rerere = Reuse Recorded Resolution*	Çakışma çözümlerini hatırlar
`git config --global push.default current`	*push.default = current*	Sadece mevcut dalı push eder
`git config --global fetch.prune true`	*fetch.prune = buda*	Her fetch'te silinmiş uzak dalları temizler
`git config --global help.autocorrect 1`	*help.autocorrect = düzelt*	Yanlış yazılan komutu otomatik düzeltir
`git config --global log.date relative`	*log.date = göreli tarih*	Tarihleri '2 saat önce' şeklinde gösterir
`git config --global branch.sort -committerdate`	*branch.sort*	Dalları son commit tarihine göre sıralar

15. Hızlı Başvuru Tablosu

Kategori	Sık Kullanılan Komutlar
Yapılandırma	`git config --global user.name` • `git config --list` • `git config --global alias.lg`
Repo Başlat	`git init` • `git clone URL` • `git clone --depth 1 URL`
Durum / Fark	`git status -s` • `git diff` • `git diff --staged` • `git diff --stat`
Sahne / Commit	`git add .` • `git add -p` • `git commit -m` • `git commit --amend`
Geçmiş	`git log --oneline --graph --all` • `git blame` • `git show HASH` • `git shortlog -sn`
Dal	`git branch -vv` • `git switch -c` • `git branch -d` • `git branch --merged`
Birleştirme	`git merge --no-ff` • `git rebase -i HEAD~3` • `git cherry-pick HASH`
Uzak / Senkron	`git remote -v` • `git fetch --prune` • `git pull --rebase` • `git push -u origin`
Geri Alma	`git restore .` • `git reset --soft HEAD~1` • `git revert HEAD` • `git reflog`
Stash	`git stash push -m` • `git stash list` • `git stash pop` • `git stash branch`
Etiket	`git tag -a v1.0.0 -m` • `git push origin --tags` • `git tag -d` • `git show v1.0.0`
Gelişmiş	`git bisect start` • `git worktree add` • `git gc` • `git clean -fd`
GitHub CLI	`gh repo create` • `gh pr create` • `gh pr merge --squash` • `gh issue create`
Actions	`gh workflow run` • `gh run list` • `gh run watch` • `gh secret set`
.gitignore	`git check-ignore -v` • `git rm -r --cached .` • `curl gitignore.io`
Yardım	`git help komut` • `git komut --help` • `man git-komut` • `gh help`

Bu rehberi faydalı buldunuz mu? Git komutlarını ezberlemek değil, ne zaman hangi komutu kullanacağınızı anlamak önemlidir. Her gün commit atın, dalları kullanın ve git log --oneline --graph --all ile geçmişinizi izleyin!