

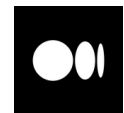
Git / GitHub Giriş



istdatascience.com | info@istdsa.com



tr.linkedin.com/school/istanbul-data-science-academy



medium.com/istanbuldatascienceacademy

GİT != GİTHUB

- Git ve Github, **iki ayrı hedef için** kullanılan iki ayrı araçtır.
- Git ve Github, ekip kodlamaları için ardışık bir düzen içerisinde birbirleriyle arayüz oluşturacak şekilde çalışarak projedeki sorunları ve başarısız olma olasılığını olabildiğince minimumda tutmayı sağlar.

Git KULLANIMI



git



HADI BİR HİKAYE DİNLEYELİM

```
def main():  
    query = "SELECT * FROM customers"  
    table = sql.read(query)  
    customer_records = pd.DataFrame(table)  
    users = customer_records['username']  
    print(users.value_counts())  
  
if __name__ == "__main__":  
    main()
```

AHMET



Ahmet'in bir rapor oluşturmak için
biraz kod yazdığını varsayalım

HADI BİR HİKAYE DİNLEYELİM

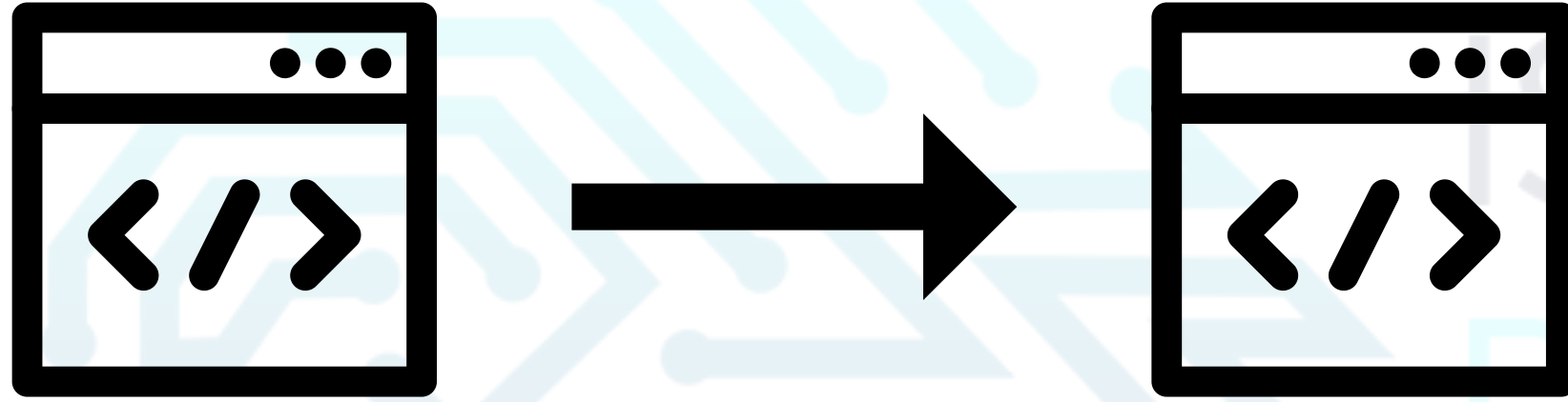
```
def main():  
    query = "SELECT * FROM customers"  
    table = sql.read(query)  
    customer_records = pd.DataFrame(table)  
    users = customer_records['username']  
    print(users.value_counts())  
  
if __name__ == "__main__":  
    main()
```

AHMET



Bravo Ahmet! Önemli
bir şey başardın!

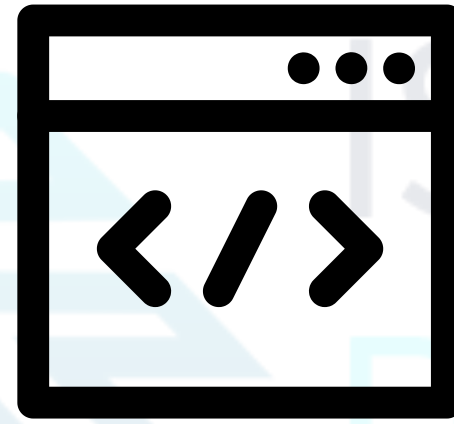
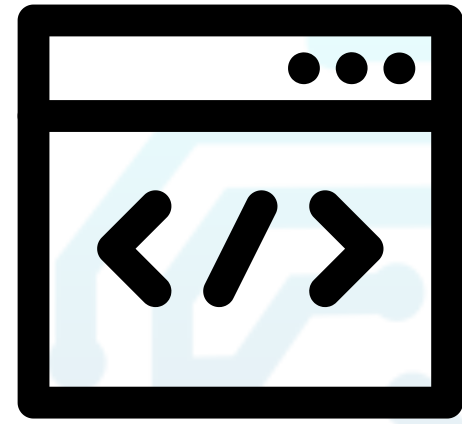
KOD GELİŞTİRME SÜRECİ



Ahmet proje amacına
yönelik biraz kod yazdı

Ancak elde ettiği sonucu
üretime çıkarmadan önce
bazı düzeltmeler yapması
gerektiği farketti

KOD GELİŞTİRME SÜRECİ



AHMET'İN BU
KODU BİRAZ DAHA
DÜZELTMEMEYE
ÇALIŞIYOR

Ahmet proje amacına
yönelik biraz kod yazdı

Ancak elde ettiği sonucu
üretime çıkarmadan önce
bazı düzeltmeler yapması
gerektiği farketti

HADI BİR HİKAYE DİNLEYELİM

```
def get_data(query):  
    """  
    Get's the data from the database  
    and returns it in dataframe format  
    """  
    table_data = sql.read(query)  
    return pd.DataFrame(table_data)  
  
def print_user_report():  
    query = "SELECT * FROM customers"  
    customer_records = get_data(query)  
    users = customer_records['username']  
    print(customer_records.value_counts())  
  
if __name__ == "__main__":  
    print_user_report()
```

Ancak Ahmet bu düzeltmeleri yaparken yanlışlıkla kodun yapısını bozarak artık çalışmamasına neden oluyor!

AHMET



Ahmet kodunu düzeltmek,
yeniden kullanılabilir ve daha
anlaşılır bir hale getirmek istiyor

KOD GELİŞTİRME SÜRECİ



Ahmet proje amacına yönelik biraz kod yazdı

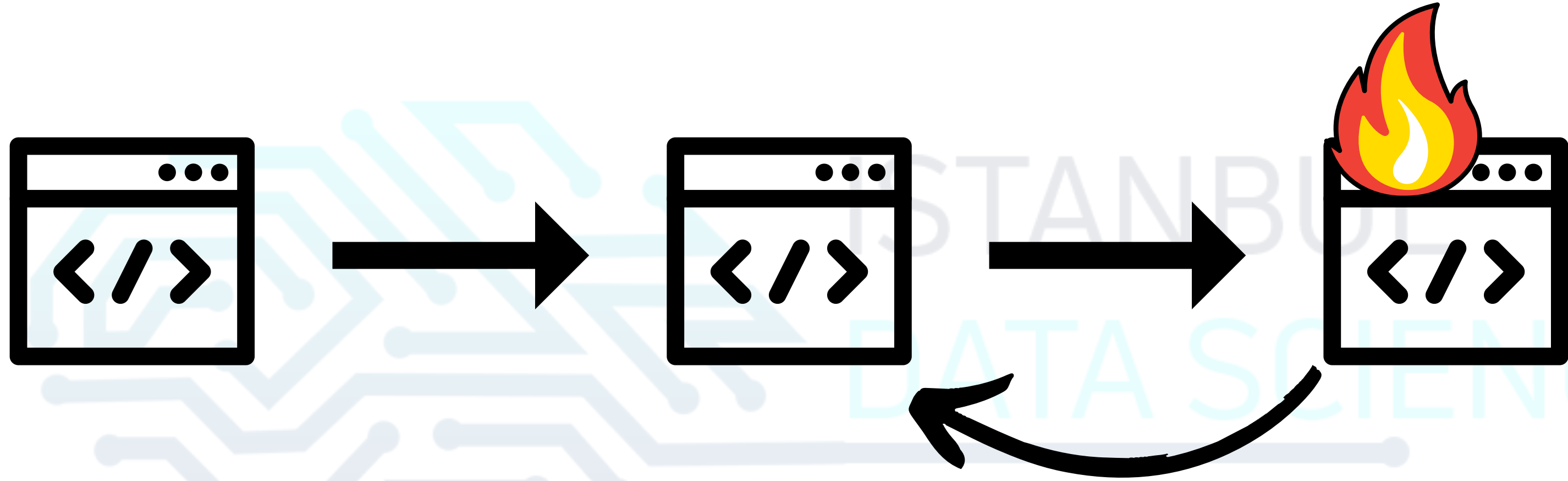
Ancak elde ettiği sonucu üretime çıkarmadan önce bazı düzeltmeler yapması gerektiği farketti

Ama geliştirme yapmaya çalışırken kod bloğunu bozdu

KOD ARTIK ÇALIŞMIYOR, NE YAPMAK LAZIM?

Önceden çalışan bir kod bloğu artık çalışmıyorsa, bu durumu düzeltmek için elimizde iki tane seçeneğimiz var,

1. Kodu nasıl düzelteceğimizi düşünmek
2. Kodun en son çalıştığı zamana sıfırlamak



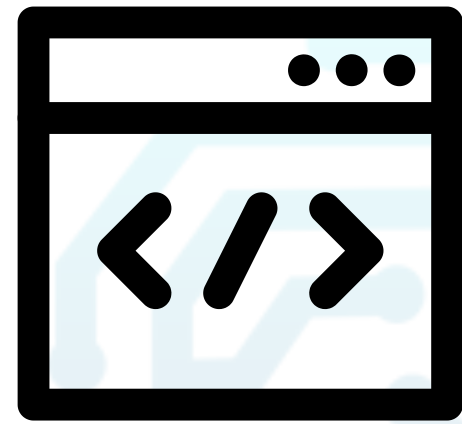
Kodu nasıl düzelteceğimizi düşünmeye hazırlıklı
değilsek, bunu yapmak çok zaman alabilir.
Kendimizi hazırlamak için "Git" kullanalım.

PEKİ GİT BİZİM İÇİN NE YAPAR?

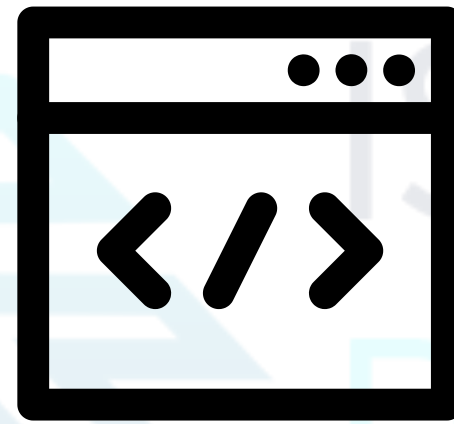
- Bir checkpoint (versiyon kontrol sistemi) oluşturmamızı sağlar.
- Her dosyanın zaman içindeki değişimini takip eder.
- Bu değişikliklerin geçmişini tutar ve istediğimiz zaman kodumuzun istediğimiz sürümüne geri dönmemizi sağlar.
- Aynı kodun birden çok sürümünün aynı anda var olmasına izin verir.

KOD GELİŞTİRME SÜRECİ

CHECKPOINT (KONTROL NOKTASI)

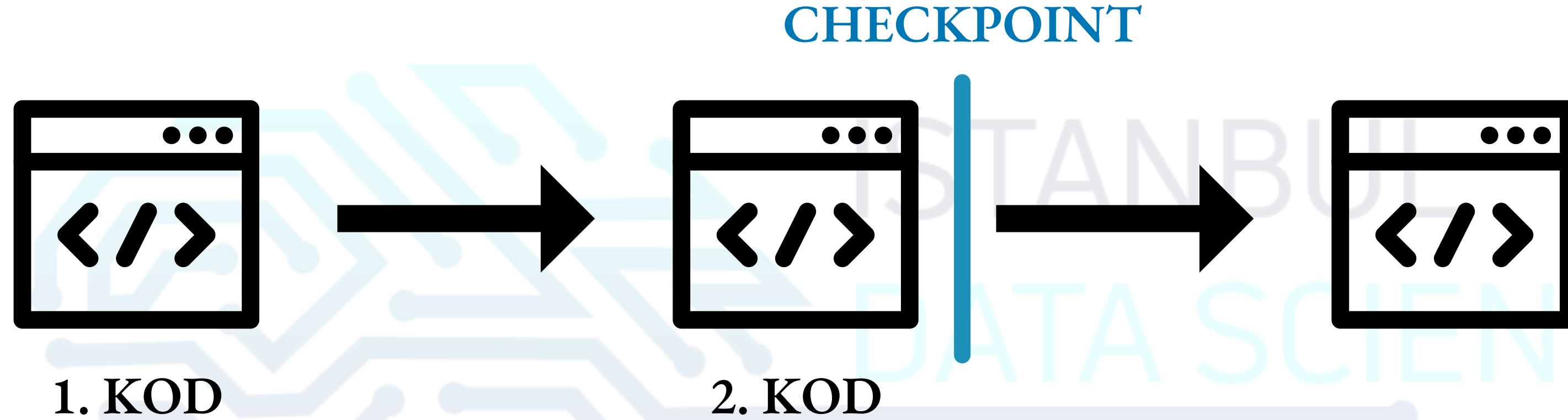


1. KOD

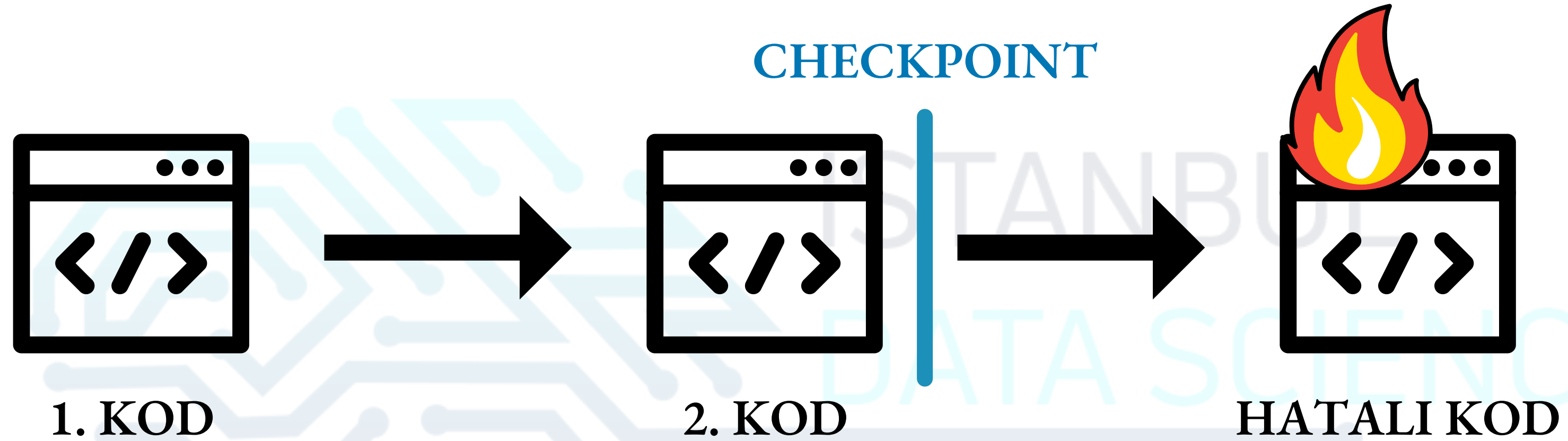


2. KOD

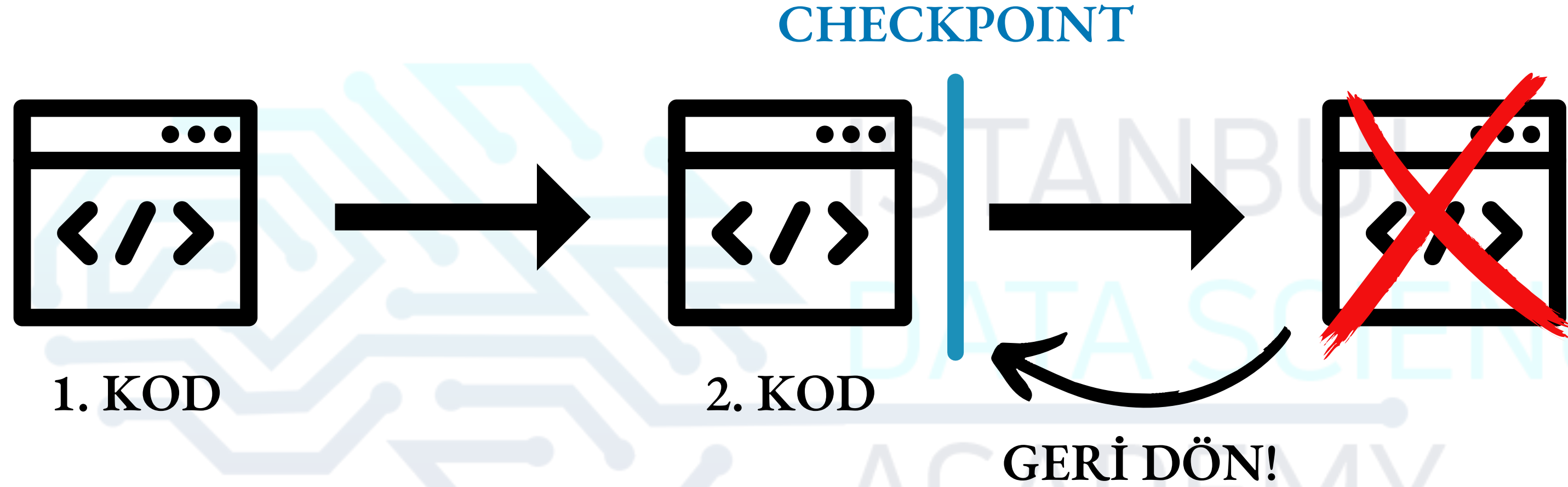
KOD GELİŞTİRME SÜRECİ



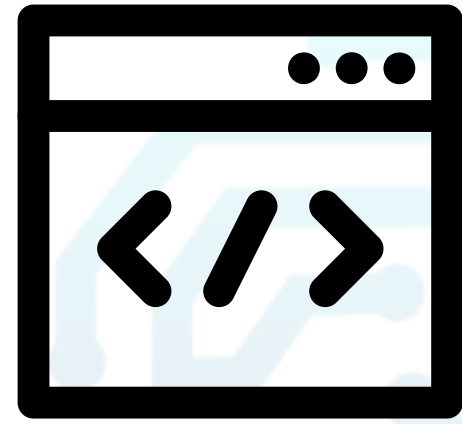
KOD GELİŞTİRME SÜRECİ



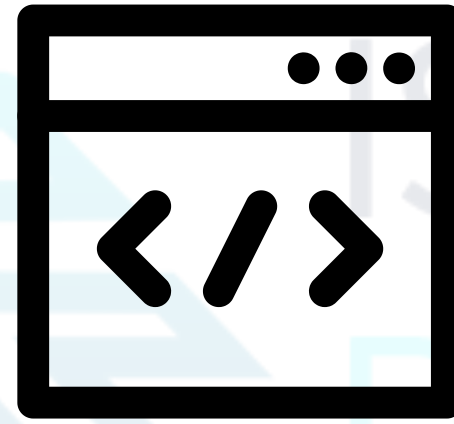
KOD GELİŞTİRME SÜRECİ



KOD GELİŞTİRME SÜRECİ



1. KOD



2. KOD



GİT VERSİYON KONTROL SİSTEMİ

Bu versiyon kontrol sistemi, "**commits**" adı verilen yapılardan oluşur. Bir "**commit**" esasen Git'in projedeki tüm dosyaların neye benzediğini bildiği bir zaman bilgisidir. Her dosyadaki kodu hatırlar ve böylece her zaman o versiyona geri dönebiliriz.

Git'e "**versiyon kontrol sistemi**" denmesinin nedeni budur. Ancak bir "**commit**" te bulunmadan önce Git'in temel 3 aşamasını anlamamız gerekiyor.

**WORKING
DIRECTORY**

**STAGING
AREA**

REPOSITORY

Git, versiyon kontrol sistemi üç bağımsız "aşamadan" oluşur. Hadi her aşamayı teker teker inceleyelim.

WORKING DIRECTORY

Working Directory (çalışma dizini) alanı bizim bilgisayarımızdaki “normal” dosya sistemimizdir. Bu, kodu şu anda bilgisayarımızda olduğu gibi takip eder. Eğer bazı değişiklikler yaptıysak, çalışma dizini tüm bu değişiklikleri görür ancak herhangi bir şekilde geçmişini takip etmez.

STAGING AREA

Staging Area, bir commit için nasıl hazırlandığımızdır. Bir commit'te bulunmadan önce, Git'e belirli dosyaların commit işlemi için “aşamalı” olmasını istediğimizi söyleriz (Git dışı terimlerle, Git'e, checkpointten bu yana değişip değişmediğini görmek için bu dosyayı kontrol etmesini istediğimizi söylüyoruz).

REPOSITORY

Repository, tüm checkpointlerimizin yaşadığı yerdir. Her commit'te bulunduğumuzda, tüm bu değişiklikler repoya gönderilir. Repository, bir projenin her versiyonuyla ilgili tüm bilgilerin sahibidir. Kod repositoryye commit edilene kadar hiçbir checkpoint oluşturulmamıştır.

WORKING
DIRECTORY

STAGING
AREA

REPOSITORY

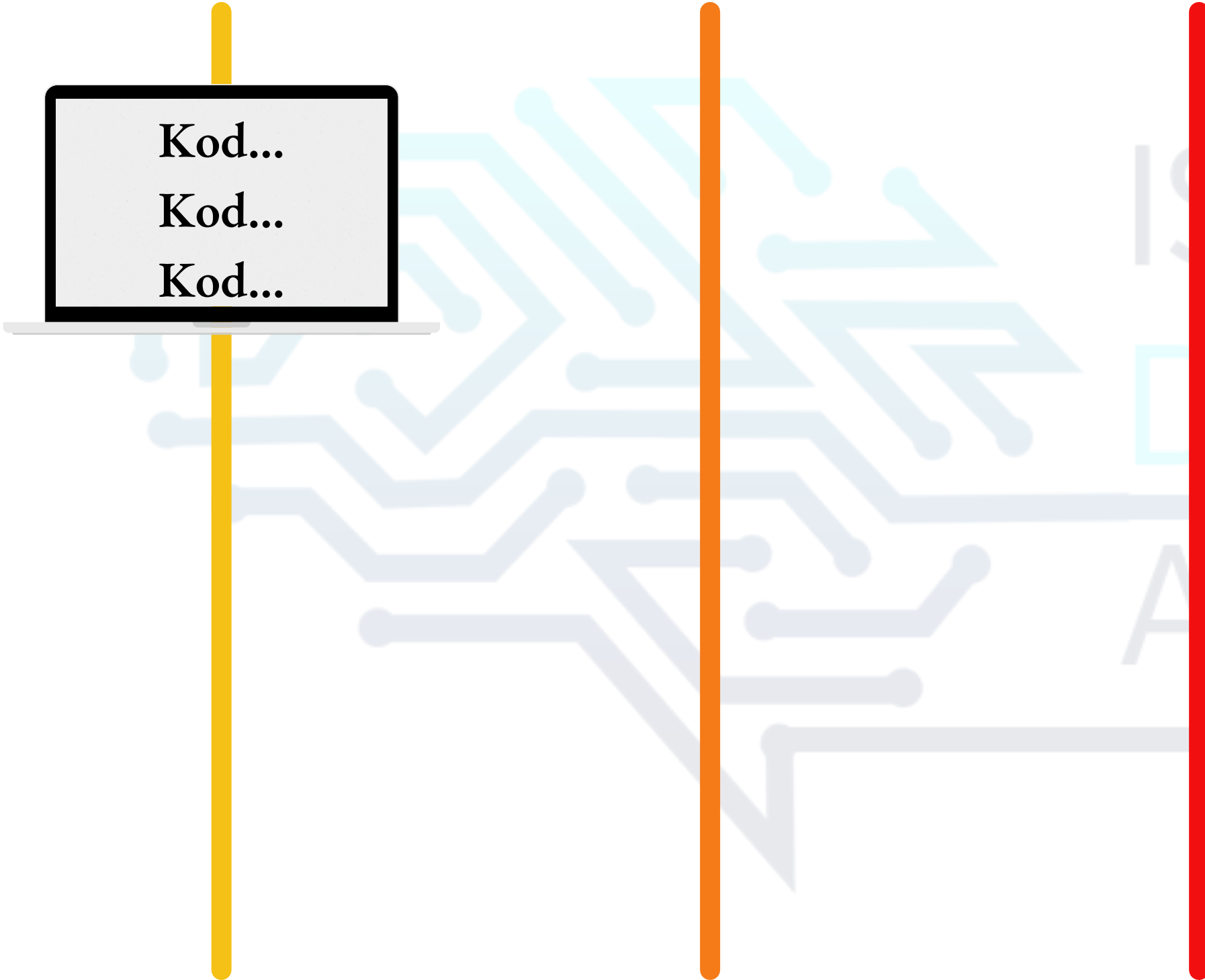


GİT GENEL İŞLEM AKIŞI

WORKING DIRECTORY

STAGING AREA

REPOSITORY



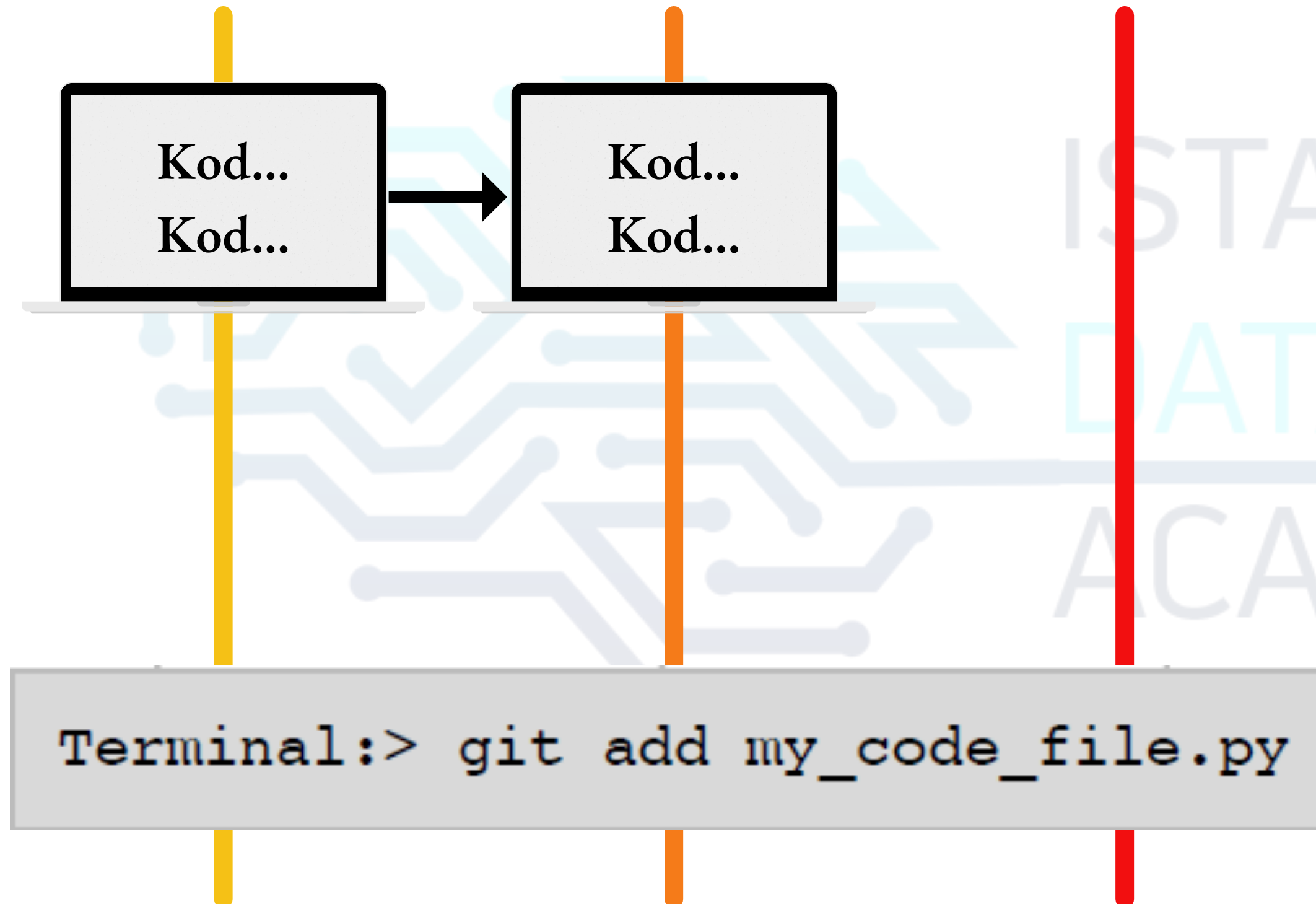
Kod...
Kod...
Kod...

Kodum üzerinde kendi
bilgisayarımdaki çalışma
dizinimde çalışıyorum.

WORKING
DIRECTORY

STAGING
AREA

REPOSITORY

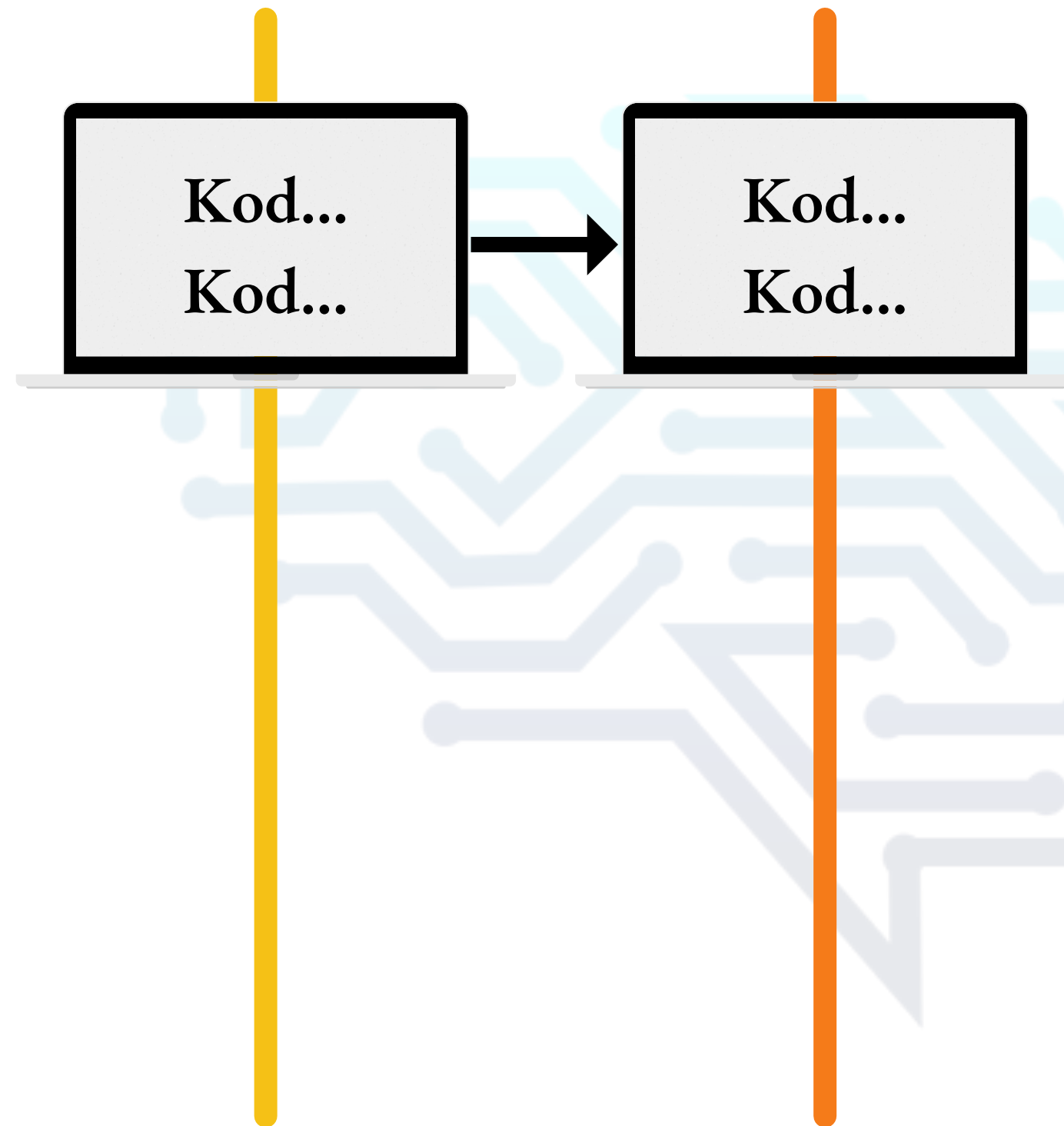


Şimdi bir checkpoint
oluşturmak için
hazırlanalım. Git'e
yazdığım kodu Staging
Area'ya eklemesini
söylemem gerekiyor.

WORKING
DIRECTORY

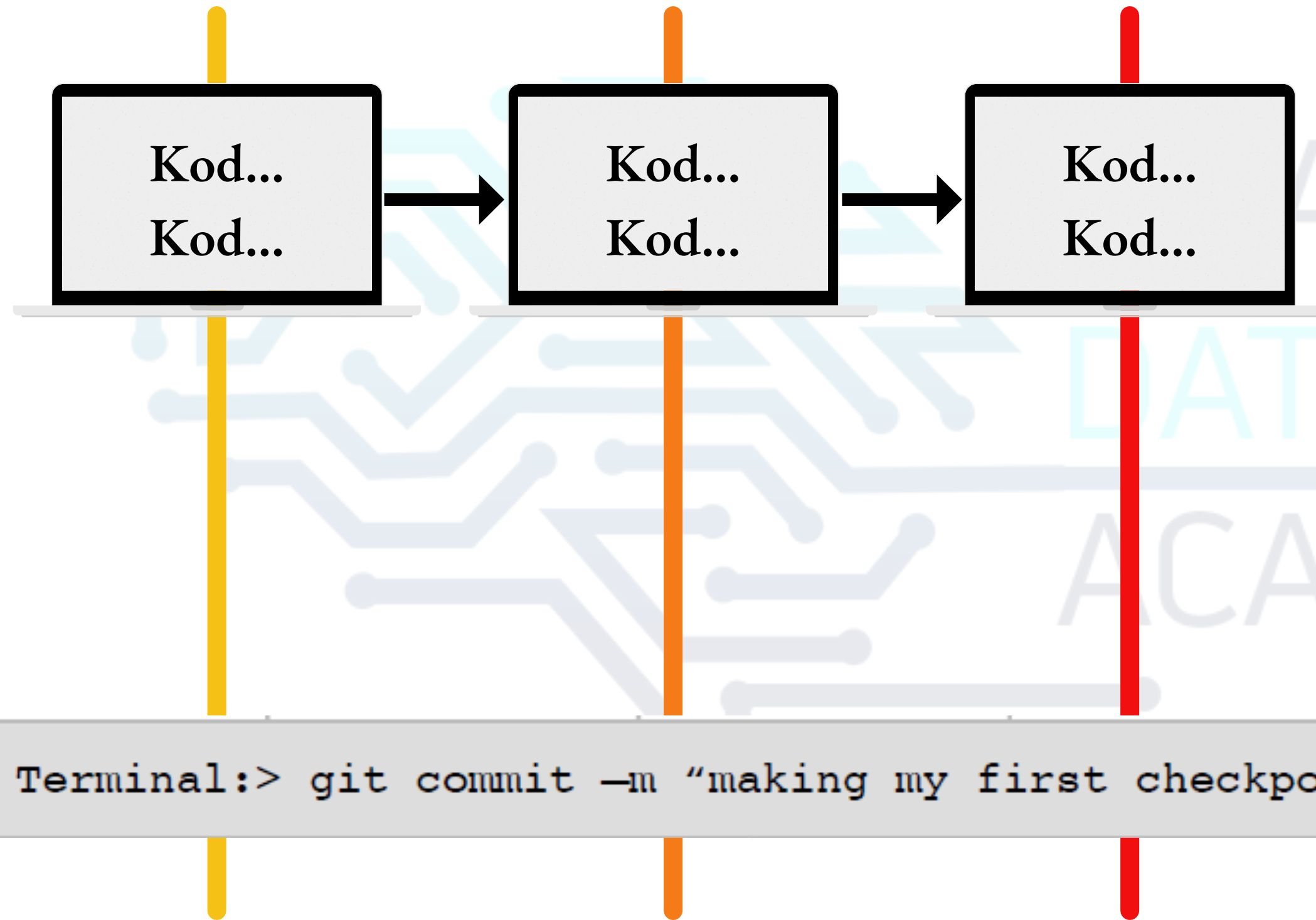
STAGING
AREA

REPOSITORY



Bu noktada, kodum için henüz bir checkpoint oluşturmadım, sadece bunun için hazırladım. Hazırladıktan sonra ise, artık bir checkpoint oluşturabilirim.

WORKING DIRECTORY STAGING AREA REPOSITORY



Aslında şimdi bir commit işleminde bulunduk. Staging Area'daki her şey Repository'e eklenir ve kodun bu versiyonunu temsil etmesi için ilgili checkpointe özel bir isim verilir.

Buna "commit mesajı" denir ve bunu bölüme yazdığınız mesaj oldukça önemlidir. Git'in her bir checkpointe verdiği özel isim sadece "d9rgy6431abql" anlam ifade etmeyen yapılardan oluşan bir dizi karakterdir. Belirli bir kod bloğuna geri dönmeniz gerekirse, yaptığınız commit işleminin amacını açıkça tanımlayan bir mesajınız olması gerekir.

```
Terminal:> git commit -m "making my first checkpoint"
```

COMMIT MESAJLARINA İLİŞKİN BİR KILAVUZ

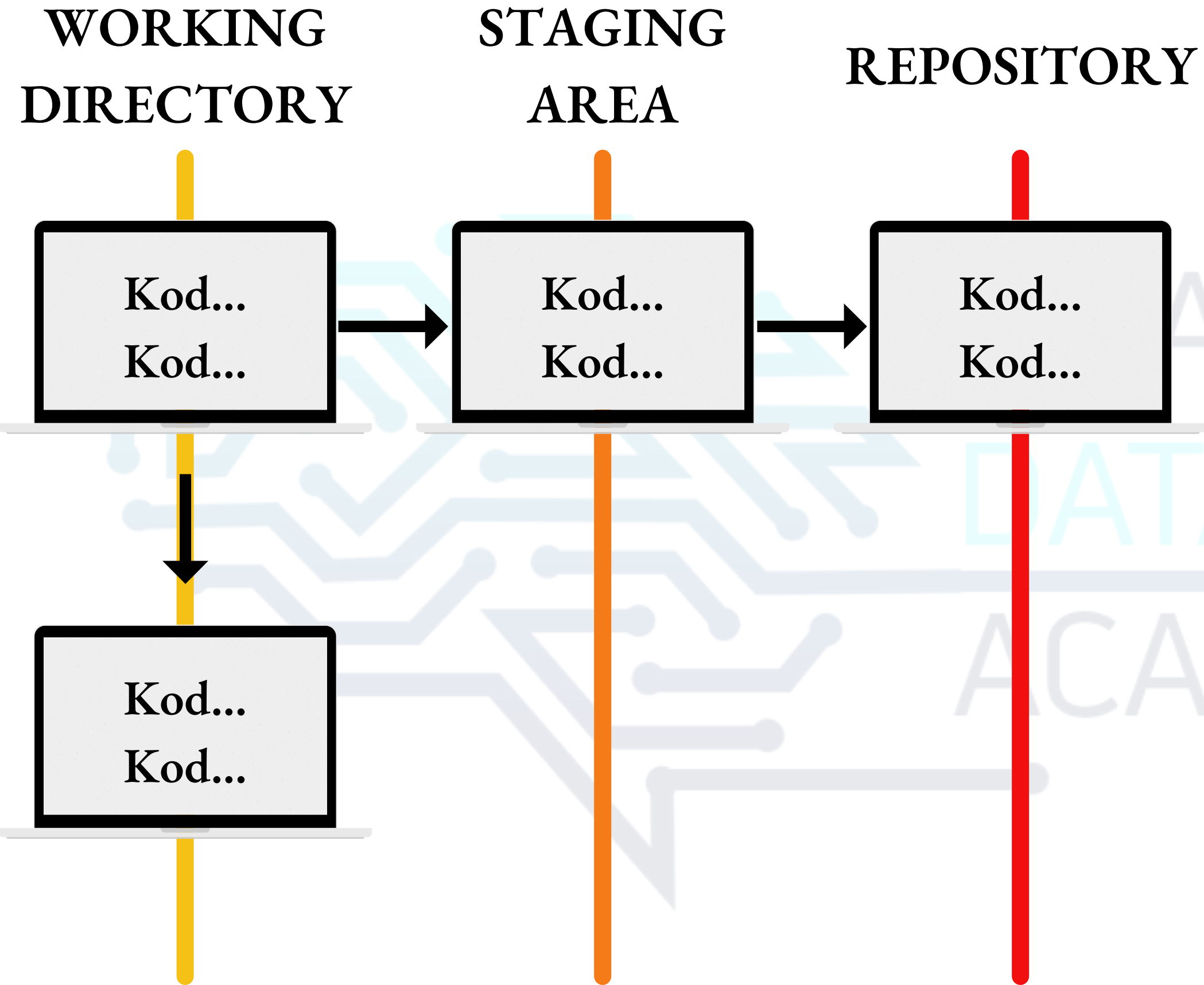
Commit mesajlarının yapısı şu şekilde tasarlanmalıdır,

- Son işlemde bu yana nelerin değiştiği açıkça belirtilmeli
- Hangi hataların giderildiği ve hangi yeni kodun eklendiği bildirilmeli

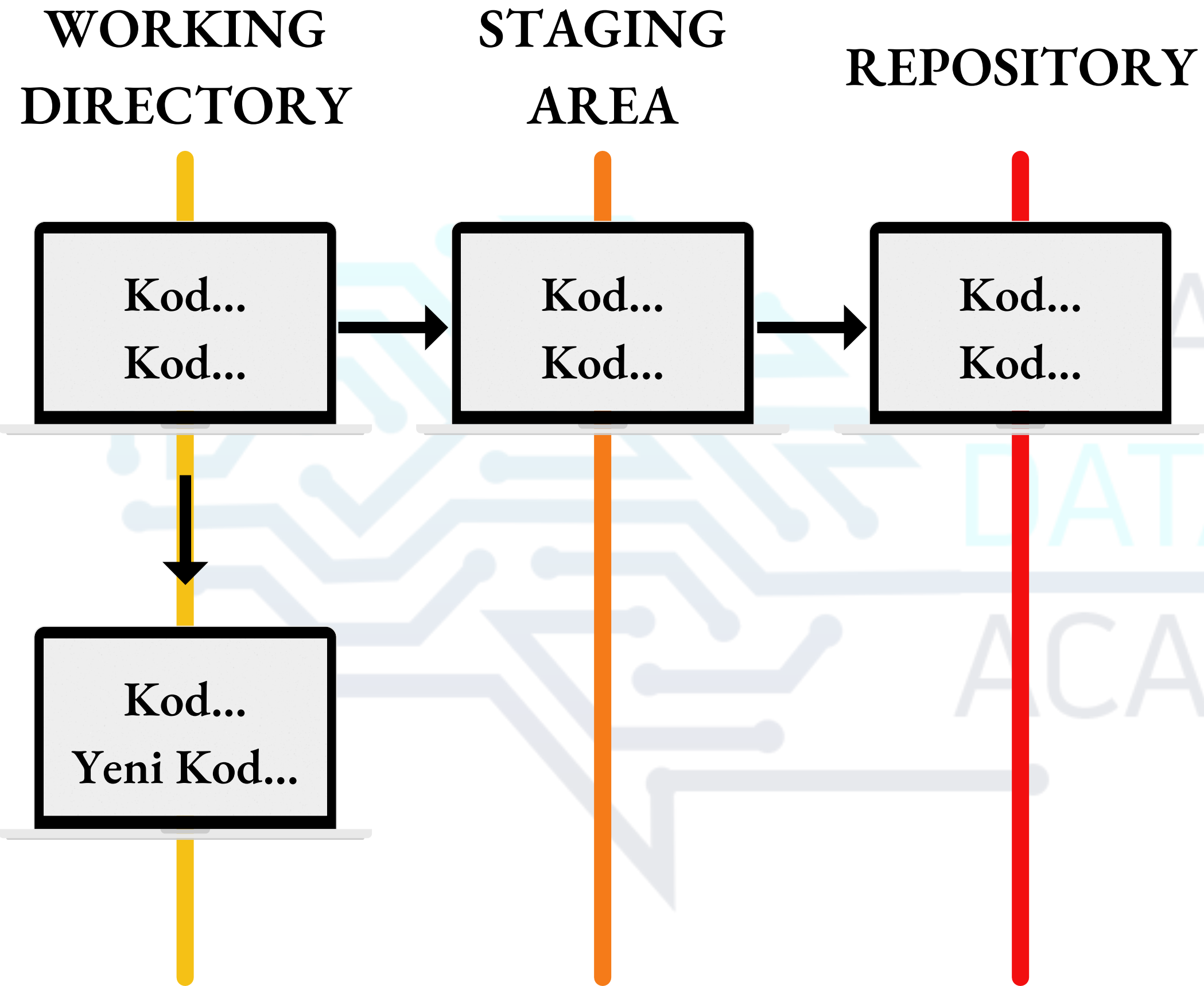
İnsanlar iyi commit mesajları hakkında uzun uzadıya fikir yürütürler.

İşte konuyla ilgili bazı güzel örnekler,

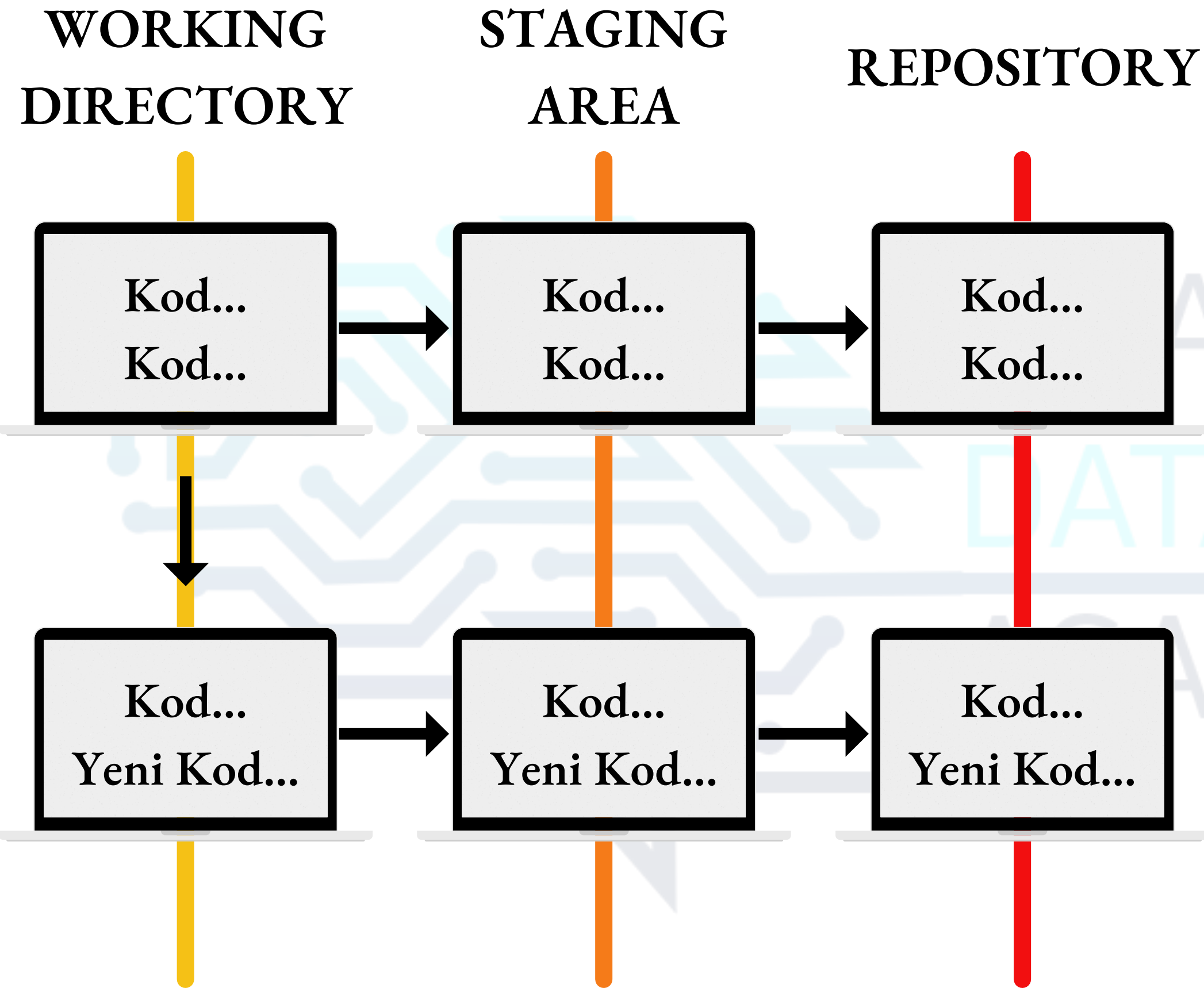
- <https://hackernoon.com/what-makes-a-good-commitmessage-995d23687ad>
- <https://chris.beams.io/posts/git-commit/>



Bu noktada repository ve bilgisayarımızdaki çalışma dizinimiz eşleşiyor.



Kendi bilgisayarımızda
güncellediğimiz kod
parçası artık hazır.



Süreci tekrar gözden geçirirsek, repository de yaşayan yeni bir commit ekleyeceğimizi görebiliriz.

Peki ya orijinal haline geri dönmek istersem?

Git'in içerisinde **Git Log** adında bir komut var. Git log önceki tüm commit'lerin bir listesini ve bunlarla ilgili bilgileri görüntüler.

Commit ID

```
ataoz@Ata-Laptop MINGW64 ~/dsegitim1 (master)
$ git log
commit 326c8c9994bc5d681af4e930f89b3f7a3aa274ea (HEAD -> master)
Author: Ata Özarslan <74244365+ataozarslan@users.noreply.github.com>
Date: Sat Mar 19 14:33:24 2022 +0300

    ds1.txt dosyası güncellendi

commit d3b58c59f876c0f9328d105721faf78c9ca63c3a
Author: Ata Özarslan <74244365+ataozarslan@users.noreply.github.com>
Date: Sat Mar 19 14:31:06 2022 +0300

    ds1.txt dosyası oluşturuldu
```

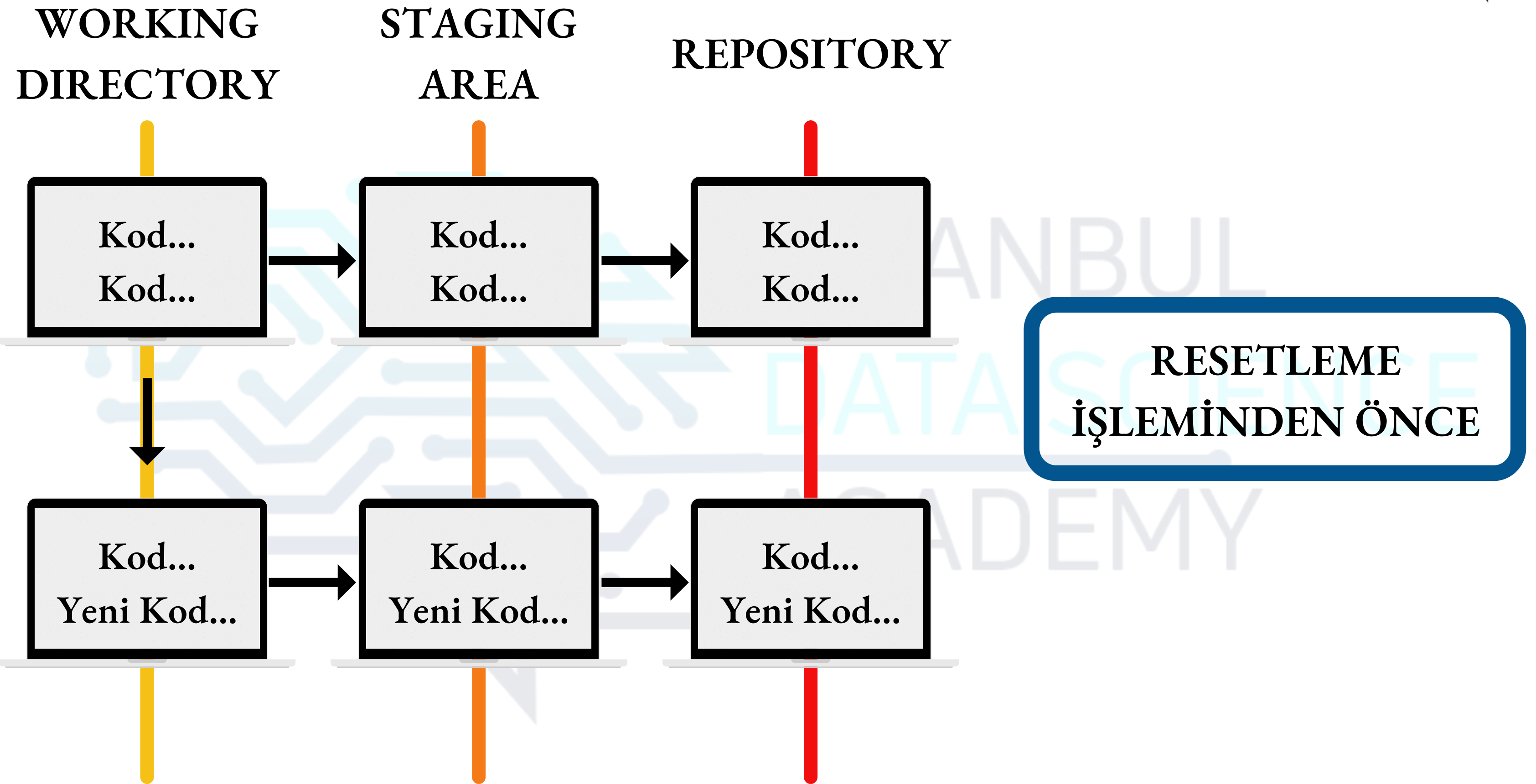
Commit Mesajı

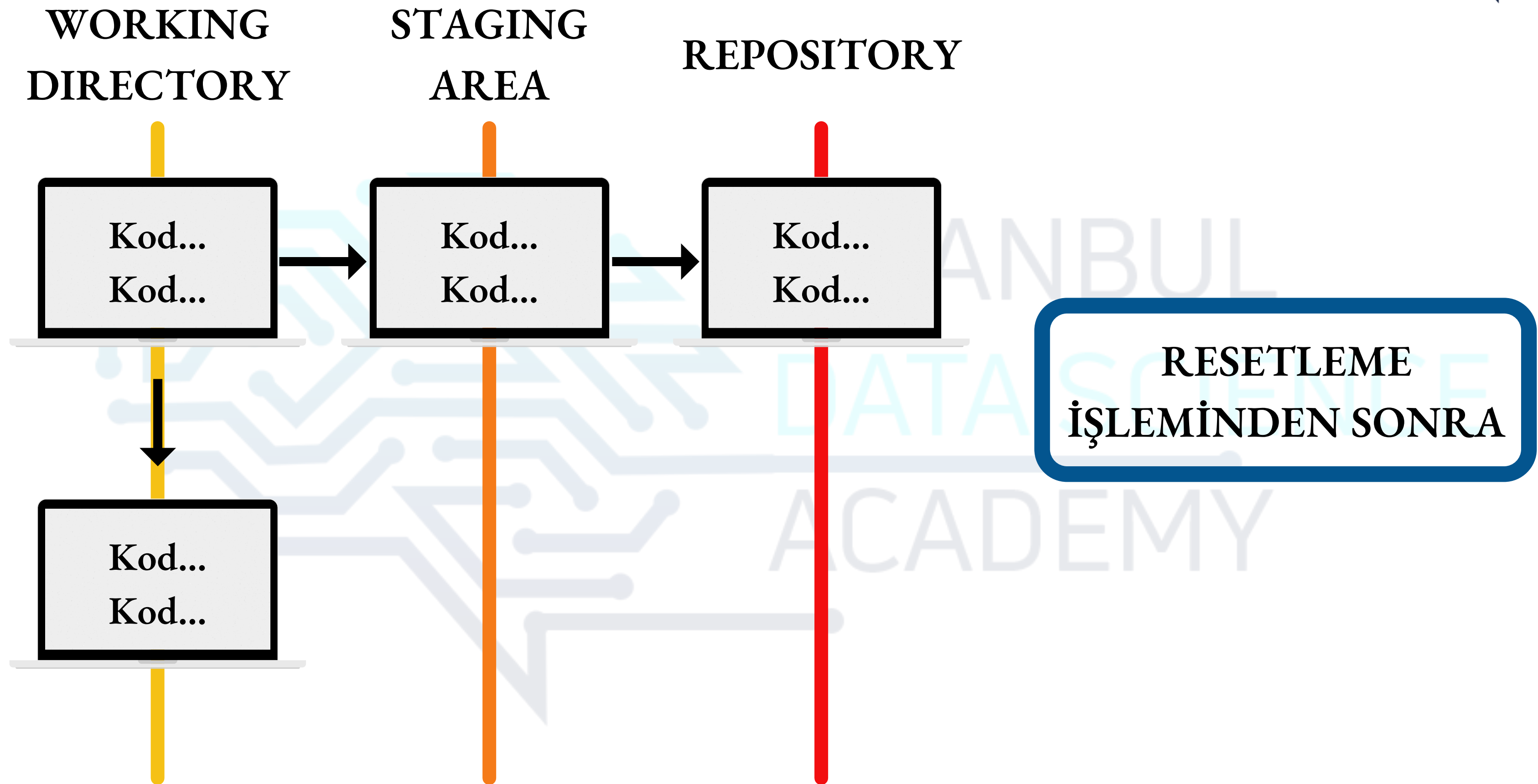
Bu nedenle, yazdığımız kodun bir önceki versiyonuna dönmek istersem, **doğru commit kimliğini** bulabilir ve ardından Git'e bu duruma sıfırlamasını söyleyebilirim.

```
Terminal:> git reset 128fc8819d8433971302cc94f5f3db7af08d4c51
```



Commit ID



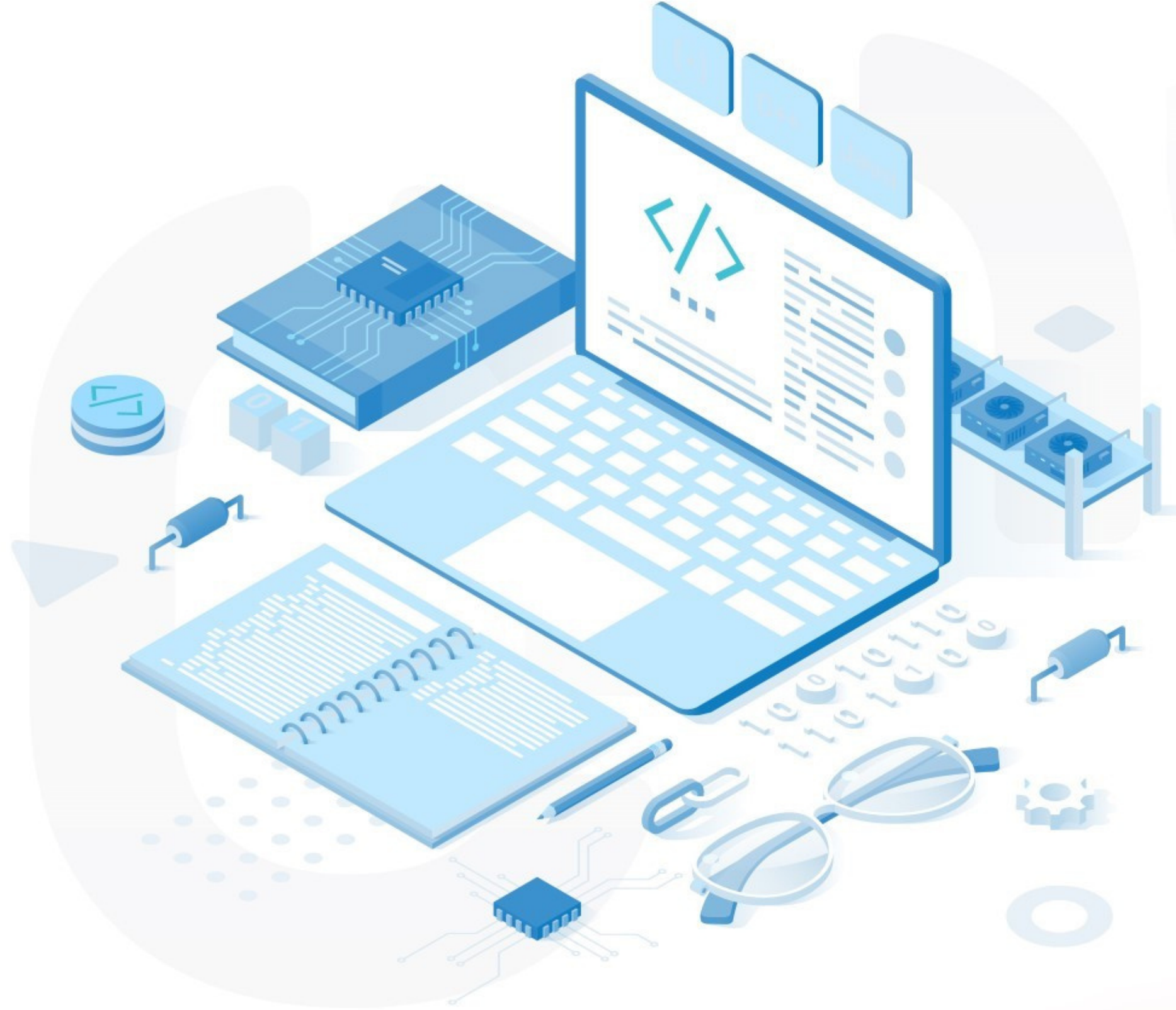


GİT: KISA BİR TEKRAR

Git bize zaman içinde **kodumuzu takip etme yeteneği** verir.

- Bize kodumuzun **geçmişini** ve **önceki bir duruma sıfırlama yeteneği** verir.
- Checkpoint'ler oluşturmak için esnek bir arayüz sağlar. (**commit**)

GİTHUB KULLANIMI



GitHub, Git'in ne yaptığına bakar ve **"Pek çok insanın birbirleri arasında kod paylaşımı yapmasını istersem ne olur?"** diye sorar.

- Kod paylaşımına online bir bileşen ekler.
- Git'in 3 temel aşamasına yeni bir aşama ekler.
- Oluşturduğunuz repoların bilgisayarınızdan başka bir yerde çevrimiçi olarak depolanmasını sağlar.

GİTHUB: KODLAMA TAKIMLARINI YÖNETMEK

Repository

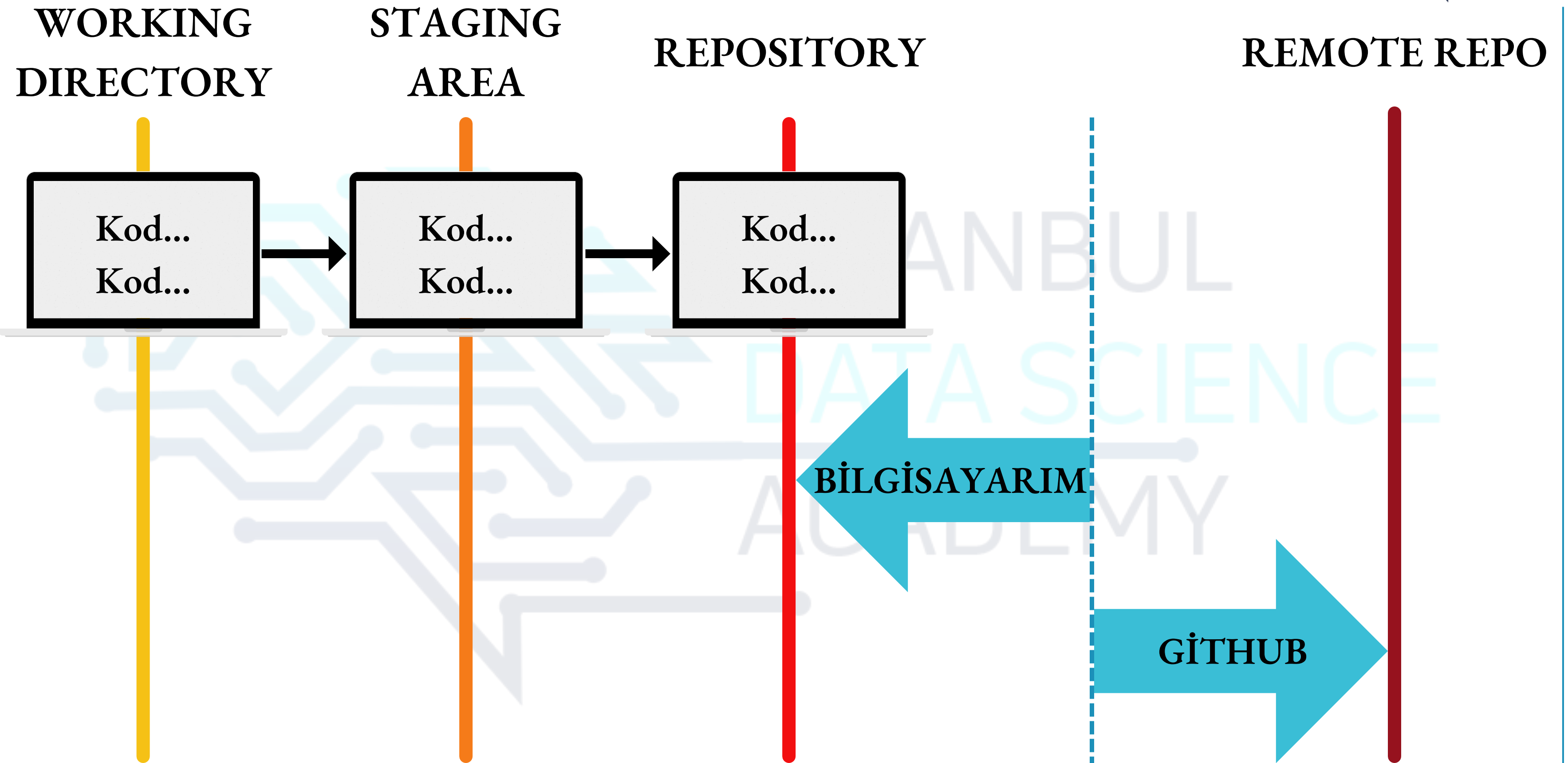
Dosyalar

Commit Sayısı

Yazarlar

The screenshot shows a GitHub repository page for 'Zekeryabesiroglu / DSFeb22'. The repository is private and has 5 watchers, 27 forks, and 14 stars. The main branch is 'main' with 2 branches and 0 tags. The repository contains a list of files and folders: SQL(Oracle), prework, project01, project02, project03, README.md, git-cheat-sheet-education.pdf, and pandas-revisited.ipynb. The README.md file is selected, showing the title 'DSFeb22' and the description 'Her derste uygulama kod çalışması yapılacaktır.' The right sidebar shows the 'About' section with no description, website, or topics provided. It also shows 14 stars, 5 watching, and 27 forks. The 'Releases' section shows no releases published. The 'Packages' section shows no packages published. The 'Contributors' section shows 5 contributors.

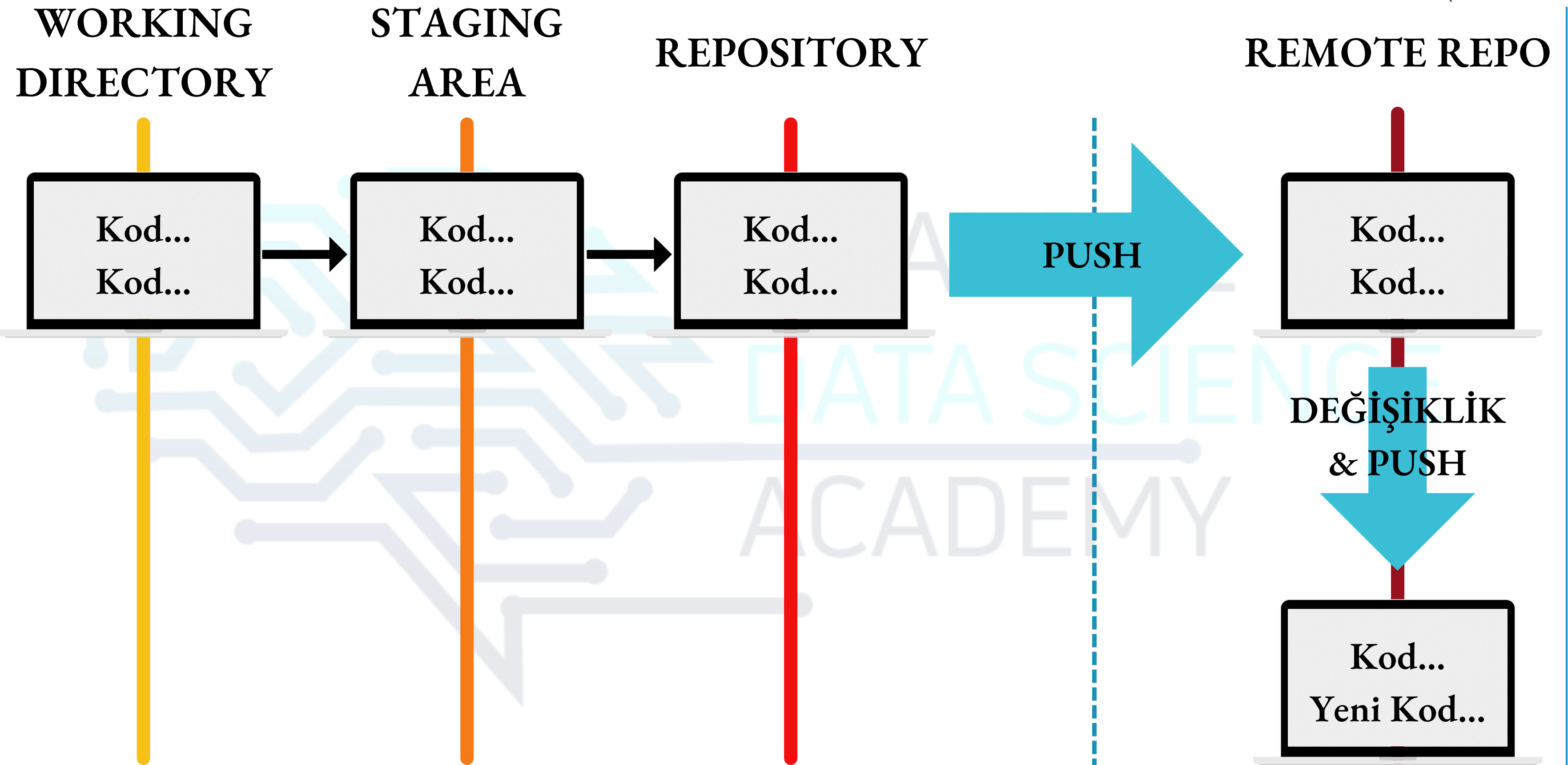
File/Folder	Commit Message	Commit Time
SQL(Oracle)	Add files via upload	3 months ago
prework	Update README.md	3 months ago
project01	Delete metis_git_cheatsheet.pdf	2 months ago
project02	Add files via upload	last month
project03	Add files via upload	yesterday
README.md	Update README.md	3 months ago
git-cheat-sheet-education.pdf	Add files via upload	2 months ago
pandas-revisited.ipynb	Add files via upload	3 months ago

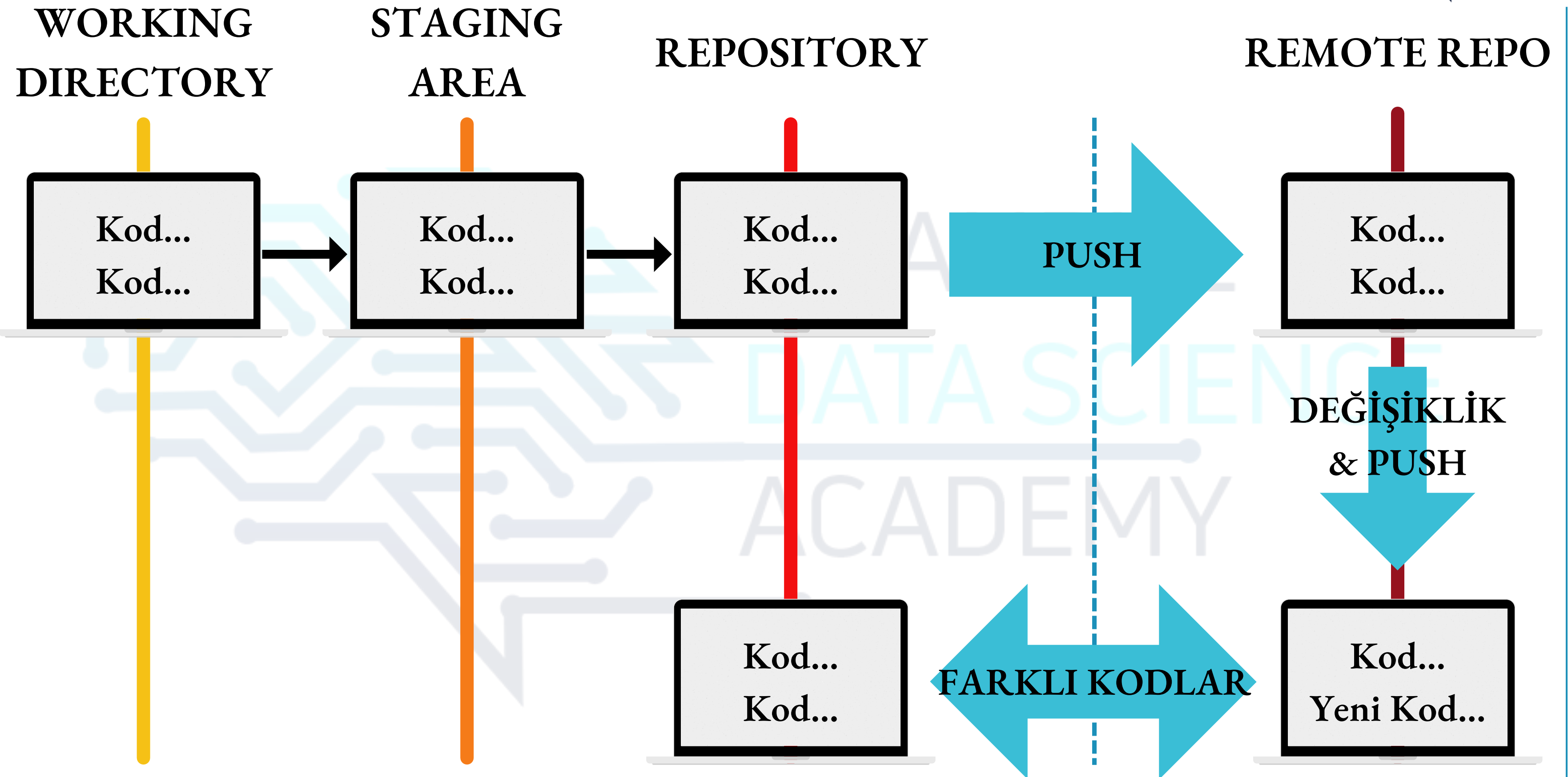


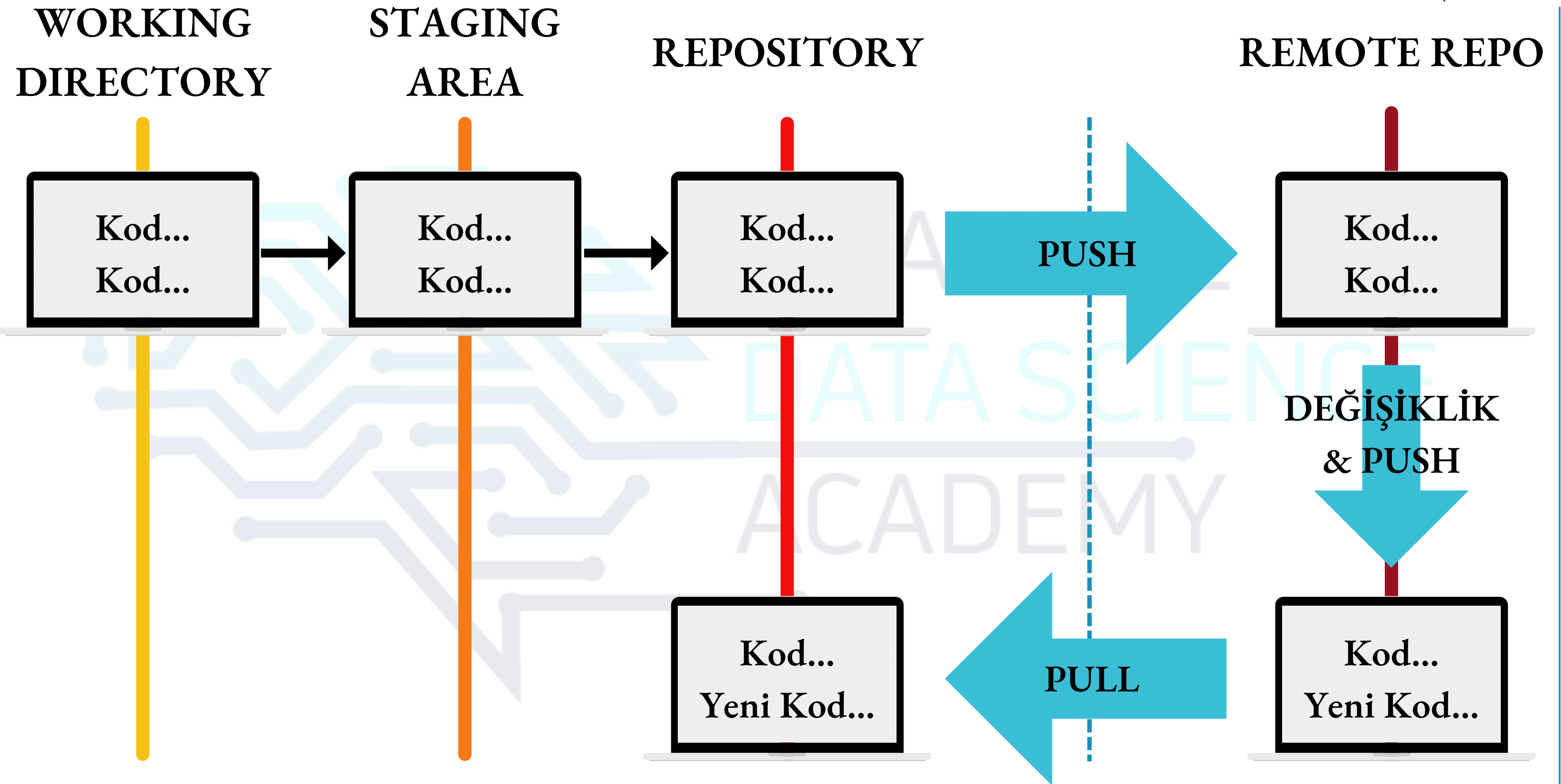
GİTHUB: REMOTE VS LOCAL

Repomuzun mevcut versiyonunu, bilgisayarımız ile "remote" bir versiyonu arasında taşımak için **push** ve **pull** işlemlerini kullanırız.

- **Push**, "Remotedaki versiyonu benim kodum gibi göster" der ve özellikle son kayıtlarda değişen dosyaları günceller.
- **Pull** ise, "Repo'nun remotedaki versiyonunda meydana gelen değişiklikleri getir ve kodumun bunlarla eşleşmesini sağla" der.







GİTHUB: REMOTE VS LOCAL

- GitHub, yalnızca Git'ten onları **izlemesini istediğiniz son zamandan beri değişen** dosyaları değiştirir.
- Bu yüzden Git'ten belirli bir dosyayı izlemesini hiç istemediysem (bir commit işlemi yaparak), kodumu **push** etsem bile remote versiyonuna gitmeyecek.

**ŞİMDİ HADI KENDİ BİLGİSAYARIMIZDA
TEST ADINDA BİR REPOSITORY
OLUŞTURUP BUNU REMOTEDAKİ GİTHUB
HESABIMIZLA BAĞLAYALIM :)**

- Tabiki aradaki bağlantıyı sağlamadan önce, ilk olarak GitHub'da kendimize bir hesap oluşturmamız (eğer hesabınız yoksa) ve sonrasında yeni bir repository oluşturmamız gerekiyor.
- İlk olarak terminalde daha önce başlattığımız repoya geçin. **Git Init**'le daha önce yaptığımız klasörde olmanız gerekiyor. (veya bunun altındaki herhangi bir dizin)

GİTHUB'I LOCAL REPOMUZA BAĞLAMAK

Artık hem GitHub'da remote bir repomuz hem de kendi localimizde tuttuğumuz test dosyalarımız var. Şimdi hadi bunları birbirine bağlayalım.

```
Terminal:> git remote add origin URL_FROM_GITHUB
```

Reponun local olmayan versiyonunu söylemek üzere olduğunu söyler.

GİTHUB'I LOCAL REPOMUZA BAĞLAMAK

Artık hem GitHub'da remote bir repomuz hem de kendi localimizde tuttuğumuz test dosyalarımız var. Şimdi hadi bunları birbirine bağlayalım.

```
Terminal:> git remote add origin URL_FROM_GITHUB
```

Arayüz oluşturmak için yeni bir remote alan eklediğimizi söyler.

GİTHUB'I LOCAL REPOMUZA BAĞLAMAK

Artık hem GitHub'da remote bir repomuz hem de kendi localimizde tuttuğumuz test dosyalarımız var. Şimdi hadi bunları birbirine bağlayalım.

```
Terminal:> git remote add origin URL_FROM_GITHUB
```

Remote alana "origin" diyeceğimizi söyler.
(literatürde bu şekilde kullanılır)

GİTHUB'I LOCAL REPOMUZA BAĞLAMAK

Artık hem GitHub'da remote bir repomuz hem de kendi localimizde tuttuğumuz test dosyalarımız var. Şimdi hadi bunları birbirine bağlayalım.

```
Terminal:> git remote add origin URL_FROM_GITHUB
```

Remote reponun nerede
yaşadığını söyler.

GİTHUB'I LOCAL REPOMUZA BAĞLAMAK

Aşağıdaki komut satırını kullanarak remote konumlarımızı test etmemiz gerekiyor,

```
Terminal:> git remote -v
```

Sonrasında ise aşağıdaki gibi bir çıktı görmeyi bekliyoruz,

```
origin https://github.com/ZWMiller/test (fetch)  
origin https://github.com/ZWMiller/test (push)
```

GİTHUB'I LOCAL REPOMUZA BAĞLAMAK

Artık yazdığımız kodu GitHub'a taşıyabiliriz,

```
Terminal:> git push origin master
```

Commit geçmişimizi GitHub'a
taşımak istediğimizi söyler.

GİTHUB'I LOCAL REPOMUZA BAĞLAMAK

Artık yazdığımız kodu GitHub'a taşıyabiliriz,

```
Terminal:> git push origin master
```

Az önce yarattığımız "origin" adlı
remote dizini kullanmasını söyler.

GİTHUB'I LOCAL REPOMUZA BAĞLAMAK

Artık yazdığımız kodu GitHub'a taşıyabiliriz,

```
Terminal:> git push origin master
```

Bu değişiklikleri "master" branchine koymasını söyler. Branchlerden sonra bahsedeceğiz ama "master" branchi en önemlileri.

- Git ve GitHub, iyi kodları sürdürülebilir kılmak için oldukça güçlü araçlardır.
- Commit'ler aracılığıyla kontrol noktaları oluşturmamıza izin verirler ve kodumuzu bozduğumuz zamanlarda geriye dönme gücü verirler.
- GitHub, kodumuzu başkalarıyla paylaşmamıza ve birden çok kişinin kodda değişiklik yapmasına olanak tanır.

- Akıllı bir programcı, tüm projeleri için bir GitHub deposu oluşturur ve kodlarını burada tutar. Bu şekilde eğer laptoplarına kahve dökerlerse kodları yedeklenmiş olur.
- Sık sık committe bulunmak iyi bir fikirdir. Bunu genellikle kodumuzdaki her tamamlanan "noktadan" sonra yapmak gerekir.
- Artık birlikte çalışan birkaç fonksiyonu yazmayı mı bitirdiniz? Commit.
- Koddaki yorumlar mı güncellendi? Commit.