Sinopfbd, 2025; 10(1): 259-288 https://doi.org/10.33484/sinopfbd.1635155

Sinop Üniversitesi Fen Bilimleri Dergisi
Sinop Uni J Nat Sci
E-ISSN: 2564-7873

https://dergipark.org.tr/tr/pub/sinopfbd

Yapay Zeka Uygulamalarının Python Kod Üretimindeki Performans Karşılaştırması

Metin AKBULUT

How to cite: Akbulut, M. (2025). Yapay zeka uygulamalarının Python kod üretimindeki performans karşılaştırması. *Sinop Üniversitesi Fen Bilimleri Dergisi, 10*(1), *259-288.* https://doi.org/10.33484/sinopfbd.1635155

Araştırma Makalesi

Sorumlu Yazar
Metin AKBULUT
metin@idu.edu.tr

Yazara ait ORCID
M.A: 0000-0002-0296-8934

Received: 07.02.2025
Accepted: 10.06.2025

Öz

Yapay zeka tekniklerinin yazılım üretimi alanında kullanılması, yazılım geliştirme sürecinde çok önemli bir araç olarak ortaya çıkmıştır. Çok sayıda çalışma, yazılım geliştirme yaşam döngüsünün çeşitli aşamalarında yapay zekanın potansiyel uygulamalarını incelemiştir. Yazılım mühendisliği alanı, proje yönetimi, kod üretimi, yazılım testi, hata tahmini ve güvenlik açığı tespiti gibi alanlarda yapay zekayı kullanmaktadır. Bu çalışmada, yapay zekâ uygulamalarının kod kalitesi, üretkenlik, yaratıcılık ve esneklik, hata toleransı ve dokümantasyon açısından objektif bir karşılaştırması yapılmıştır. Değerlendirilen YZ uygulamaları arasında OpenAI (ChatGPT, Codex), Google Gemini (Bard), Copilot (GitHub), DeepCode (Snyk Code) ve Microsoft Copilot bulunmaktadır. Belirli Python görevleri için, yapay zekâ uygulaması tarafından üretilen kodlar senaryolar oluşturularak karşılaştırılmış ve her uygulamanın çözüm doğruluğu, hata toleransı, kod ek açıklamaları ve pratik performansı değerlendirilmiştir. Sonuç olarak, OpenAI Codex karmaşık projeler için oldukça etkili bir çözüm olarak belirlenmiş, GitHub Copilot günlük geliştirme için ideal bir seçim olarak ortaya çıkmış, Google Gemini temel çözümler için yeterli olduğunu kanıtlamış, Microsoft Copilot kurumsal entegrasyon ve güvenlik uyumlu çözümler için kapasitesini göstermiş ve DeepCode kod güvenliği ve kalite kontrolüne verdiği önemle öne çıkmıştır.

Anahtar Kelimeler: Yapay zeka, kodlama, hata toleransı, kod kalitesi, Python kod geliştirme

Performance Comparison of Artificial Intelligence Applications in Python Code Generation

İzmir Demokrosi Üniversitesi, İzmir, Türkiye

Abstract

The employment of artificial intelligence techniques in the realm of software production has emerged as a pivotal instrument within the software development process. A multitude of studies have examined the potential applications of artificial intelligence at various stages of the software development lifecycle. The field of software engineering incorporates artificial intelligence in areas such as project management, code generation, software testing, defect prediction, and vulnerability detection. In this study, an objective comparison was made of AI applications in terms of code quality, productivity, creativity and flexibility, fault tolerance, and documentation. The AI applications evaluated include OpenAI (ChatGPT, Codex), Google Gemini (Bard), Copilot (GitHub), DeepCode (Snyk Code), and Microsoft Copilot. For specific Python tasks, the codes generated by the AI application were

compared by creating scenarios, and the solution accuracy, fault tolerance, code annotations, and practical performance of each application were evaluated. Consequently, OpenAI Codex was identified as a highly effective solution for complex projects, GitHub Copilot emerged as an ideal choice for daily development, Google Gemini proved to be adequate for basic solutions, Microsoft Copilot demonstrated its capacity for enterprise integration and security-compliant solutions, and DeepCode was distinguished for its emphasis on code security and quality control.

Keywords: Artificial intelligence, coding, fault tolerance, code quality, Python code development

Giriş

Yazılım endüstrisi, büyük dil modelleri (Large Language Models-LLM) ve bu modellerin kullanan programların kullanılmaya başlanmasıyla birlikte son yıllarda önemli bir dönüşüme tanık olmuştur. Yapay zekâ (YZ) tabanlı yaklaşımlar, proje yönetimi, kod geliştirme ve sistem analizi de dahil olmak üzere yazılım mühendisliğinin çeşitli yönlerini geliştirmek için kullanılmıştır [1-3]. Kod üretimi için kullanılan YZ teknikleri ile yazılım geliştirme sürecini kolaylaştırma, geliştirme süresini azaltma ve genel yazılım kalitesini iyileştirme potansiyeline sahiptir [2, 4].

Bu çalışmada incelenen uygulamalar (örneğin ChatGPT, GitHub Copilot, vb.) doğrudan "yapay zeka modelleri" olarak adlandırılmaktan ziyade, LLM`in temelinde çalışan kod üretim programlarıdır. Bu programlar çoğu aynı temel LLM`leri (örneğin OpenAI GPT-3.5, GPT-4 gibi) kullanmalarına rağmen, farklı çıktı kalitesi sunabilmektedir. Bu durum, uygulamaların sadece kullandıkları temel modelle değil, aynı zamanda eğitildikleri veri çeşitliliği, sistem düzeyinde yapılan ince ayarlar, kullanım arayüzleri ve bağlamsal ön işlemleme kabiliyetleriyle de ilgilidir. Bu nedenle çalışmada, LLM`ler ile bu modelleri kullanan programlar arasındaki ayrım özellikle vurgulanmıştır.

Bazı çalışmalarda, yapay zekanın yazılım mühendisliğine entegrasyonunu araştırılmıştır. Bu çalışmalar, makine öğrenimi, derin öğrenme ve paralel hesaplama gibi çeşitli yapay zekâ yöntemleri yazılım geliştirme yaşam döngüsünün farklı aşamalarında karşılaşılan zorlukların üstesinden gelmek için yararlanılabileceğini vurgulamıştır [2].

Yapay zeka tekniklerinin proje yönetimi, kod geliştirme ve sistem analizi gibi çeşitli alanlarda nasıl uygulandığını gösteren çalışmalar mevcuttur. Örneğin, proje yönetiminde yapay zeka, proje planlaması ve kaynak yönetimi süreçlerini optimize etmek için kullanılabilir. Kod geliştirme alanında, YZ araçları/uygulamaları, kod üretimini otomatikleştirerek ve kod tamamlama önerileri sunarak yazılım geliştirme sürecini hızlandırabilir. Sistem analizi ve bakımında ise yapay zekâ, yazılım sistemlerinin performansını analiz etmek, hataları tespit etmek ve güvenlik açıklarını belirlemek için kullanılabilir.

YZ tekniklerinin yazılım projelerinin başarısını artırma ve geliştirme süreçlerini verimli hale getirme imkanına sahip olduğu öngörülmektedir.

Bu çalışmanın amacı, LLM modellerini kullanan programların Python kod üretimindeki performanslarını karşılaştırmaktır. Çalışmada, OpenAI Codex, GitHub Copilot, Google Gemini ve Microsoft Copilot gibi YZ uygulamaları değerlendirilmiştir. Bu uygulamaların arkasındaki büyük dil

modelleri (GPT, BERT vb.) ve bu modellerin eğitim verileri, performans farklılıklarının önemli bir nedenidir.

Bu amaçla, çalışmada, YZ uygulamalarının yazılım geliştirme süreçlerine entegrasyonunun potansiyelini ve sınırlılıklarını anlamak amacıyla çeşitli YZ modellerinin kod yazma performansı değerlendirilecektir.

Çalışmada, büyük dil modelleri ve bu modelleri kullanan programların performans farklılıkları detaylı bir şekilde incelenmiştir. YZ uygulamaların performansını değerlendirirken, eğitim verilerinin etkisi göz önünde bulundurulmuştur. Çalışmada, çeşitli Python görevleri için YZ uygulamaların üreteceği kodlar senaryolar oluşturularak, karşılaştırılmış ve her aracın çözüm doğruluğu, hata toleransı, kod açıklamaları ve pratik performansı değerlendirilmiştir. Çalışmanın bulguları, yazılım geliştirme süreçlerinde YZ uygulamaların nasıl daha verimli kullanılabileceğine dair öneriler sunmaktadır.

Literatür

YZ'nin yazılım mühendisliğinde uygulandığı kilit alanlardan biri de proje yönetimidir [1]. Yazılım projelerinin başarısı büyük ölçüde etkili proje yönetimine bağlıdır ve proje planlaması, görev tahsisi ve kaynak yönetiminin çeşitli yönlerini optimize etmek için YZ tekniklerinden yararlanılabilir. Yazılım geliştirme alanında, kod üretimini otomatikleştirmek, kod tamamlamayı iyileştirmek ve yazılım testini geliştirmek için YZ yöntemleri araştırılmıştır [2]. Ayrıca, YZ teknikleri yazılım sistemlerinin analizinde ve bakımında da kullanılabilir. Araştırmacılar, hata tahmini, yazılım güvenlik açığı tespiti ve yazılım yeniden düzenleme gibi görevler için YZ tabanlı yaklaşımların kullanılmasını önermişlerdir [4]. Yazılım mühendisliğinde YZ uygulamaları üzerine yapılan bir araştırmaya göre, YZ'nin entegrasyonu pazara sunma süresini kısaltma ve yazılım sistemlerinin verimliliğini artırma potansiyeline sahiptir [1]. Bununla birlikte, bu yaklaşımların çoğu öncelikle araştırma topluluğu içinde kaldığından ve henüz yazılım geliştirme uygulamalarına yaygın bir şekilde entegre edilmediğinden, bu YZ tekniklerinin yazılım mühendisliği endüstrisinde benimsenmesi nispeten sınırlı olmuştur [4].

Yapay zeka uygulamalarının kod üretimini derinlemesine anlamak için, önceden eğitilmiş dil modelleri kullanılarak Python kodu üretimi ele alınmış ve bu uygulamaların kod üretiminde önemli performans iyileştirmeleri sağladığı, belirlenmiştir [5]. Ayrıca, doğal dil açıklamalarından Python kodu üretmek için bir transformer modeli kullanılmış ve mevcut en iyi modellerden daha yüksek performans gösterdiği tespit edilmiştir [6]. Büyük dil modelleri kullanarak kod üretimini inceleyen çalışmalar, bu modellerin yazılım geliştirme süreçlerindeki etkilerini değerlendirmiştir [7].

Son yıllara içerisindeki yapay zeka uygulamaları ve Python kod üretimi ile ilgili güncel çalışmalar yapılmıştır. Crawford ve arkadaşları, YZ tekniklerinin yazılım mühendisliğinde proje yönetimi uygulamalarını nasıl dönüştürdüğünü incelemiş ve YZ uygulamalarının proje planlaması, görev tahsisi ve kaynak yönetimi gibi alanlarda nasıl kullanıldığını detaylı bir şekilde ele almıştır [1]. Perez ve ark., [5], önceden eğitilmiş dil modelleri kullanarak otomatik kod üretimiyle büyük dil modellerinin Python

kod üretimindeki performansını değerlendirmiş ve önemli performans iyileştirmeleri sağlanmıştır [5]. Liu ve arkadaşları, büyük dil modelleri kullanarak kod üretimini inceleyen bir pilot çalışma ile YZ modellerinin yazılım geliştirme süreçlerindeki etkilerini değerlendirmiştir [7]. Zhang ve Shao, OpenAI ve ChatGPT`nin en son uygulamalarının, YZ uygulamalarının metin üretimi ve kodlama desteği sağlama kapasitesini ele almıştır [8]. Yine Zhang ve arkadaşları, GitHub Copilot'un kullanımını ve karşılaşılan zorlukları empirik bir çalışma ile GitHub Copilot'un belirli görevlerdeki doğruluğu ve kullanıcı deneyimini analiz etmiştir [9].

Bu çalışma, literatürdeki yaklaşımlardan önemli ölçüde farklılık göstermektedir. İlk olarak, genellikle yalnızca doğruluk, hız veya kullanıcı deneyimi gibi tek bir açıdan inceleme yapılırken, bu çalışma çoklu kriter yaklaşımını benimseyerek birden fazla kriteri bir araya getirmiş ve genel başarıya olan katkılarını niceliksel olarak ölçmüştür. İkinci olarak, literatürde kullanıcı deneyimi anketleri gibi nitel yöntemler baskınken, bu çalışmada korelasyon matrisi kantitatif yöntemler uygulanmıştır. Üçüncü olarak, her bir kriterin genel radar grafiği çıkarılarak bir değerlendirme yapılmıştır. Buna göre, OpenAI Codex, tüm kriterlerde yüksek puanlar alarak en iyi performansı göstermiştir. Microsoft Copilot, kod doğruluğu ve verimlilikte güçlüdür, ancak diğer kriterlerde biraz daha düşük performans göstermektedir. Google Gemini, genel olarak diğer uygulamalara göre daha düşük performans göstermiştir ve iyileştirme alanları bulunmaktadır. GitHub Copilot, kod doğruluğu ve verimlilikte güçlüdür, ancak diğer kriterlerde daha düşük performans göstermektedir.

YZ'nin yazılım mühendisliğindeki bu uygulamaları, Python gibi esnek ve güçlü bir programlama dilinin kullanımını daha da önemli hale getirirken, yaygın kullanımı da dikkate alınması gereken bir durumdur. Bu kapsamda, Python, YZ tabanlı çözümlerin geliştirilmesinde yaygın olarak tercih edilen bir dil olarak öne çıkmaktadır.

Python Dili

Yüksek seviyeli, genel amaçlı bir programlama dili olan Python, yazılım geliştirme dünyasında baskın bir güç olarak kendini sağlam bir şekilde kanıtlamıştır. 1991 yılında Guido van Rossum tarafından geliştirilen Python, sürekli olarak popülerlik kazanmış ve bilimsel hesaplamadan web geliştirmeye ve ötesine kadar çok çeşitli uygulamalar için tercih edilen bir dil haline gelmiştir [10].

Python'un evrimi, dili daha topluluk odaklı hale getirmek ve şeffaf bir geliştirme sürecine sahip olmak için Python 2.0`ın yayınlandığı 2000`lerin başına kadar izlenebilir [11]. Python 2.0 ve sonraki sürümleri kullanımda kalmaya devam ederken, 2008 yılında piyasaya sürülen Python 3, dile önemli değişiklikler ve iyileştirmeler getirdiği için büyük ilgi ve beklentinin odağı olmuştur [11].

Python'un çok yönlülüğü ve erişilebilirliği, onu birçok programcı ve araştırmacı için tercih edilen bir seçenek haline getirmiştir. Python'un düzenli söz dizimi ve güçlü araçları sayesinde, kullanıcıların çok çeşitli görevleri çözmelerini sağlayarak onu basit matematik düşüncesine çok yakın hale getirmiştir [12]. Dilin çeşitli dillerde geniş bir kod tabanına erişme yeteneği, sözdizimsel basitliği ile birleştiğinde, onu

heterojen metodolojik geçmişe sahip çok sayıda alandan bilim insanları arasında fikirleri uygulamak ve paylaşmak için ideal bir araç haline getirmiştir [13].

Python'un bu esnekliği ve geniş kütüphane desteği, onu YZ modellerinin geliştirilmesi ve test edilmesi için ideal bir dil haline getirmiştir. Bu çalışmada, Python kodlama yetenekleri açısından çeşitli YZ uygulamaları karşılaştırılacaktır.

Python yazan YZ karşılaştırmak için çeşitli kriterler bulunur. Bu kriterler, kod kalitesi, verimlilik, yaratıcılık ve esneklik, hatalara dayanıklılık ve belgelendirmedir. Bu kriterler Tablo 1`de karşılaştırma kriterleri için hangi beklentileri olduğu gösterilmiştir.

Tablo 1. *Yazılım Geliştirme Uygulamalarının Karşılaştırma Kriterleri ve Beklentileri*

Karşılaştırma Kriterleri	**Beklenti**
Kod Kalitesi	İyi yapılandırılmış, okunabilir ve PEP 8 standartları [14].
Verimlilik	Üretilen kod, çözüm süresi ve işlem performansı açsısından verimliliği [15].
Yaratıcılık ve Esneklik	Farklı yollarla benzer problemleri çözebilme kapasitesi [16].
Hatalara Dayanıklılık	Yanlış girdilere veya hatalı senaryolara karşı sağlamlığı [17].
Belgelendirme	Kodla birlikte açıklamalar veya belgeler sağlanması [18].

Bu kriterler ışığında, OpenAI (ChatGPT, Codex), Google Gemini, Copilot (GitHub), Microsoft Copilot ve DeepCode (Snyk Code) gibi popüler YZ uygulamaların Python kodlama yetenekleri karşılaştırılacaktır. Bu uygulamalar:

OpenAI (ChatGPT, Codex): OpenAI ve ChatGPT, hesaplama uygulamalarının doğruluğunu ve hızını artırmak için muazzam bir potansiyele sahiptir ve böylece mühendislerin daha verimli ve güvenilir sistemler tasarlamasını sağlar [8]. Güçlü doğal dil işleme yetenekleriyle detaylı ve mantıklı Python kodları yazabilir. Open AI'ye göre, insan benzeri metin üretimi ve kodlama desteği sağlayarak, doğal dil işleme ve programlama görevlerini kolaylaştıran YZ sistemleridir [19].

Google Gemini: Google Gemini, metin, görüntü, ses ve video gibi farklı türlerdeki bilgileri anlayabilen ve üretebilen, çok yönlü bir YZ aracı olarak tanımlanabilir [20].

Copilot (GitHub): GitHub Copilot, yazılım geliştiricilere kod tamamlama ve öneriler sunarak geliştirme sürecini hızlandıran bir YZ destekli araçtır [21].

DeepCode (Snyk Code): DeepCode (şimdi Snyk Code), YZ destekli bir kod analiz aracıdır; geliştiricilerin yazılım güvenliğini artırmak ve kod kalitesini iyileştirmek için kod tabanında güvenlik açıklarını ve hataları hızlı bir şekilde tespit etmesine olanak tanır [22]. Kod kalitesi ve güvenliğine odaklanır.

Microsoft Copilot: Azure platformuyla entegre, büyük ölçekli kurumsal çözümler için tasarlanmış bir yapay zekâ aracıdır. Kullanıcı taleplerine göre özelleştirilebilir ve Microsoft'un güvenlik standartlarına uygun çalışır. Microsoft Office, Azure DevOps ve Visual Studio gibi yazılımlarla entegre olup, yazılım geliştirme sürecinde öneriler ve iyileştirmeler sunar. Büyük projelerde kod tamamlama ve optimizasyon konusunda etkilidir, ancak bireysel geliştiriciler için karmaşık olabilir [23].

Tablo 2. *LLM Tabanlı Kod Üretim Uygulamalar ve Temel Özellikleri*

Araç / Hizmet	Temel LLM / Teknoloji	Geliştirici / Şirket	Eğitim Verisi ve Özelleştirme	Öne Çıkan Özellikler
ChatGPT (OpenAI)	GPT-3.5 / GPT-4	OpenAI	Genel amaçlı çok geniş metin ve kod veri setleri	Doğal dil arayüzü, kod ve açıklama üretiminde yüksek başarı
Codex (OpenAI)	GPT-3.5-tabanlı Codex	OpenAI	GitHub kodları ve diğer programlama veri kümeleriyle eğitildi	Kod üretimi ve yorumlama odaklı, IDE entegrasyonu destekli
Google Gemini	Gemini 1.5	Google	Google'ın genel dil ve kod veri setleri (belirtilmemiş detaylı kaynak)	Gelişmiş çoklu görev yeteneği, kod tamamlama ve açıklama desteği
GitHub Copilot	Codex (GPT-3.5-tabanlı)	GitHub (Microsoft)	GitHub üzerindeki açık kaynak kodlar ile özelleştirilmiş	VSCode entegrasyonu, hızlı otomatik tamamlama
Microsoft Copilot	GPT-4 / GPT-4 Turbo (entegre model)	Microsoft + OpenAI	Office ve yazılım geliştirme verileriyle desteklenmiş	Microsoft 365 uygulamaları ile entegre, açıklamalı kod yazımı
DeepCode (Snyk Code)	Snyk'e özgü özel LLM	Snyk	Kod güvenliği odaklı verilerle eğitilmiş	Güvenlik odaklı analiz, zayıf noktaların tespiti ve öneri üretimi

Kaynak: *Yazar tarafından oluşturulmuştur* [20-24].

Tablo 2`de aynı temel dil modeline (örneğin GPT-3.5 veya GPT-4) dayanan uygulamaların farklı çıktı kalitesi üretilmesinde kullanılan eğitim verisinin kapsamı, bağlamsal farkındalık düzeyi, kullanım senaryosu (örneğin entegre IDE mi yoksa sohbet arayüzü mü), ve geliştirici firma tarafından yapılan sistemsel özelleştirmelerle ilgisi ortaya konulmuştur. Bu nedenle bu çalışmada sadece uygulamaların nihai performansları değil, altında yatan teknolojik yapı ve amaçlanan kullanım biçimleri de dikkate alınmıştır. Bu karşılaştırma, YZ uygulamalarının yazılım geliştirme süreçlerine entegrasyonunun potansiyelini ve sınırlılıklarını anlamak açısından büyük önem taşımaktadır.

Büyük Dil Modelleri ve Uygulamalar

Bu çalışmada kıyaslanan uygulamaların büyük kısmı, büyük dil modelleri (LLM) olarak bilinen YZ modellerine dayanmaktadır. Büyük dil modelleri, geniş veri kümeleri üzerinde eğitilmiş ve doğal dil işleme (NLP) görevlerinde yüksek performans gösteren derin öğrenme modelleridir. Bu modeller, metin üretimi, dil anlama, çeviri ve daha birçok NLP görevinde kullanılmaktadır.

Çalışmada değerlendirilen uygulamaların arkasındaki esas modeller şunlardır:

- GPT (Generative Pre-trained Transformer): OpenAI tarafından geliştirilen GPT modelleri, büyük dil modelleri arasında en bilinenlerden biridir. GPT-3 ve GPT-4 gibi sürümleri, geniş veri kümeleri üzerinde eğitilmiş ve metin üretiminde yüksek doğruluk ve yaratıcılık göstermektedir [24, 25]. Bu modeller, OpenAI Codex ve GitHub Copilot gibi uygulamaların temelini oluşturmaktadır [26].
- BERT (Bidirectional Encoder Representations from Transformers): Google tarafından geliştirilen BERT modeli, çift yönlü dil anlayışı ile öne çıkar. BERT, metin içindeki kelimelerin bağlamını daha

iyi anlamak için çift yönlü eğitim kullanır ve bu sayede dil anlama görevlerinde yüksek performans gösterir. Google Gemini gibi uygulamalar, BERT ve türevlerini kullanmaktadır [27, 28].

- T5 (Text-to-Text Transfer Transformer): Google tarafından geliştirilen T5 modeli, tüm NLP görevlerini bir metin-çift metin çerçevesinde ele alır. Bu model, metin üretimi, çeviri, özetleme ve diğer birçok görevde kullanılabilir. Google Gemini gibi uygulamalar, T5 modelinden de faydalanmaktadır [28, 29].

Bu modellerin performansı, büyük ölçüde maruz kaldıkları eğitim verilerine bağlıdır. Eğitim verileri, modellerin dil anlama ve üretim yeteneklerini doğrudan etkiler. Örneğin, GPT-3.5 modeli, geniş ve çeşitli veri kümeleri üzerinde eğitilerek farklı uygulamalarda çeşitli başarılar elde etmiştir [24-26]. Bu çalışmada, her bir uygulamanın arkasındaki modelden bağımsız performansa etkisi incelenmiştir. Öyle olmasına rağmen arkasındaki dil modelinin performansı da sınamış oluyoruz. Aynı büyük dil modeli, farklı modeller kullanılarak farklı başarılar elde edebilir [26, 29, 30]. Bu çalışmada, OpenAI Codex, GitHub Copilot, Google Gemini ve Microsoft Copilot gibi uygulamalar değerlendirilmiştir. Bu uygulamaların arkasındaki büyük dil modelleri (GPT, BERT vb.) ve bu modellerin eğitim verileri, performans farklılıklarının önemli bir nedenidir.

Yöntem

YZ programlarının Python ile kod yazımında Tablo 3`te gösterilen temel kod senaryoları (if, for, while, array) tanımlanmış ve bu kodların YZ uygulamaların çözülmesi için Şekil 1`de gösterilen test süreci uygulanmıştır.

Tablo 3. *Senaryolar (if, for, while, array)*

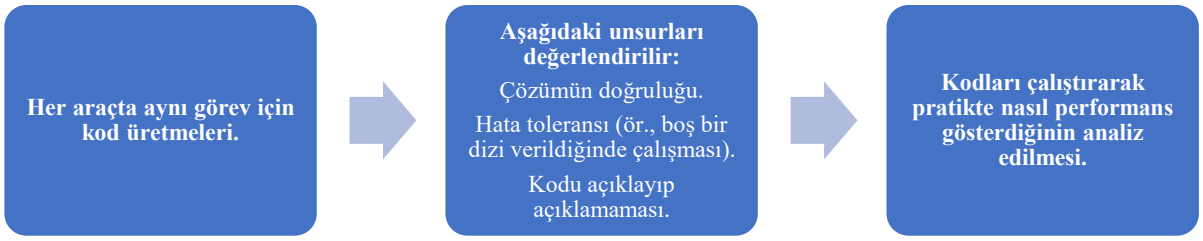
1. Dizi Elemanlarının Toplamını Hesaplama (for ve while)	**2. If-Else ile Sınıflandırma**	**3. Dizi İçerisinde En Büyük ve Küçük Elemanları Bulma**

# Example task for comparison 	
numbers = [1, 2, 3, 4, 5]
 	
# With for loop 	
total = 0
for num in numbers:
  total += num
 	
# With while loop 	
i = 0
total = 0
while i < len(numbers):
  total += numbers[i]
  i += 1
 	
print("Total:", total)
	# Determine if a number is positive, negative, or zero
	def classify_number(num):
	  if num > 0:
	  return "Positive"
	  elif num < 0:
	  return "Negative"
	  else:
	  return "Zero"
		# Find max and min in a list
		numbers = [10, 5, 3, 15, 20]
		max_num = max(numbers)
		min_num = min(numbers)
		
		print(f"Max: {max_num}, Min: {min_num}")

Tablo 3`te gösterilen temel kod senaryoları belirlenmesinde, yazılım mühendisliği literatüründe yaygın olarak karşılaşılan kod üretim görevleri esas alınmıştır. Özellikle Perez ve ark. [5], Liu ve ark. [7] gibi çalışmalar, kod üretimi değerlendirmelerinde işlevsel olarak anlamlı, derlenebilir, test edilebilir ve kullanıcı gereksinimlerine dayalı görevlerin kullanılmasını önermektedir [5, 7].

Bu bağlamda, deneysel senaryolar, hem YZ destekli uygulamaların farklı zorluk seviyelerindeki görevlerdeki başarımını gözlemleyebilmek, hem de kod tamamlama, algoritma üretimi, hata düzeltme ve açıklama üretme gibi farklı kodlama yeteneklerini karşılaştırabilmek için çeşitli kod üretim alt görevlerini temsil edecek şekilde seçilmiştir. Senaryo seçiminde ayrıca GitHub Copilot, OpenAI Codex ve benzeri uygulamalar değerlendirildiği ve Zhang ve Shao [8], Zhang ve ark. [31] gibi çalışmalar referans alınarak benzer temsili görev tipleri tercih edilmiştir [8, 31].

Dolayısıyla, deneysel senaryolar hem önceki ampirik çalışmalara dayalı hem de YZ uygulamalarının pratik yazılım geliştirme süreçlerine katkısını ölçmeyi hedefleyen görev türlerinden seçilmiştir. Böylece sonuçların hem karşılaştırılabilirliği hem de geçerliliği sağlanmaya çalışılmıştır.



Şekil 1.*Test Süreci*

Şekil 1`de her YZ aracının aynı görev için kod üretmesini ve ardından bu kodların doğruluk, hata toleransı, açıklama ve pratik performans açısından değerlendirilmesini içeren üç aşamalı test süreci gösterilmiştir. Bu test yaklaşımı ile OpenAI (ChatGPT, Codex), Google Gemini, Copilot (GitHub) test edilecektir. Ancak Code Llama (Meta), DeepSeek, Amazon CodeWhisperer ve Replit Ghostwriter gibi diğer bazı gelişmiş araçlar çalışmaya dahil edilmemiştir. Bunun temel nedenleri arasında, bu araçların çalışmanın yürütüldüğü dönemde istikrarlı sürümlerinin sınırlı olması, doğrudan erişiminin kısıtlı olması ya da değerlendirme kriterlerine uygun sistematik test ortamlarının yeterince olgunlaşmamış olması yer almaktadır. Amazon CodeWhisperer gibi araçlar ise yalnızca AWS altyapısı üzerinden erişilebilmekte ve özel entegrasyon gerektirmektedir [32-35]. Bu araçların sistematik ve eşit koşullarda değerlendirilebilmesine olanak tanımaması nedeniyle, çalışmanın derinlemesine ve karşılaştırmalı analiz hedefi doğrultusunda kapsam dışında bırakılmaları uygun görülmüştür. Nitekim literatürde de benzer çalışmalarda araç seçiminin yaygınlık, erişilebilirlik, kullanım kolaylığı ve geliştirici topluluklarındaki bilinirlik gibi kriterlere göre yapıldığı görülmektedir [36, 37]. Ayrıca, kapsamın kontrol altında tutulması, derinlemesine analiz yapılabilmesi açısından da araç sayısı sınırlı tutulmuştur. Bu durum, çok sayıda aracı yüzeysel olarak incelemek yerine, az sayıda aracı çok boyutlu analiz etmek şeklinde bilinçli bir tercihi yansıtmaktadır. Bu yaklaşım, yazılım mühendisliği odaklı ampirik çalışmalarda önerilen yöntemsel bir tercihtir [31].

YZ yazılımları kod üretim için oluşturduğu beklentiler farklıdır. OpenAI Codex genellikle hem doğru hem de kapsamlı çözümler sunarken, açıklamalar eklemektedir. GitHub Copilot geliştirme sırasında

öneriler bulunarak, daha kısa ve doğrudan çözümler sunmaktadır. Google Gemini kod çözümünde basit yaklaşımlar tercih ederek sonuca ulaşmaktadır. DeepCode yazılan kodun güvenliğini ve optimizasyonunu analizine odaklanmıştır.

Kod örneklerini bu uygulamalara ürettirerek belirli senaryolar için hangi aracın daha iyi olduğunu net bir şekilde görebilmek için bu dört yapay zekayı karşılaştırmak için aşağıdaki gibi detaylı bir analiz yapıldı. Tablo 4`te yapaya zeka yazılımlarının her birinin güçlü yönleri, kullanım alanları ve zayıf noktaları belirtilmiştir.

Tablo 4. *Güçlü Yönleri, Kullanım Alanları ve Zayıf Noktaları*

	OpenAI Codex	**GitHub Copilot**	**Google Gemini**	**DeepCode (Snyk Code)**	**Microsoft Copilot**
Özellikler	**Geliştirme:** Doğal dilden kod üretebilir. Kompleks ve özgün kodlar yazma kapasitesi oldukça yüksek. **Esneklik:** Kullanıcı tarafından verilen açıklamalar doğrultusunda özelleştirilmiş kod üretebilir. **Desteklediği Diller:** Python başta olmak üzere birçok programlama dilini destekler.	**Geliştirme:** Kod tamamlama ve öneri odaklı bir araçtır. **Destek:** Kullanıcı yazmaya başladığında otomatik öneriler sunar. **Kapsam:** Daha çok kullanılan örüntüleri tamamlar.	**Geliştirme:** Genel YZ yanıtları arasında kodlama taleplerini de yerine getirebilir. **Kapsam:** Daha çok temel algoritmalar ve basit çözümler sunar.	**Odak:** Kod analizi, kalite ve güvenlik. **Amaç:** Yazılan kodun hatalarını bulma, güvenlik açıklarını analiz etme ve iyileştirme önerileri sunma.	**Kurumsal Odak:** Azure platformuyla entegrasyonlu, büyük ölçekli kurumsal çözümler için uygundur. **Özelleştirilebilir:** Kullanıcı taleplerine ve belirli projelere göre API düzeyinde özelleştirme yapılabilir. **Güvenlik ve Uyumluluk:** Microsoft'un güvenlik standartlarına uygun çalışır ve veri gizliliğine önem verir.
Güçlü Yönler	Zor problemler için yaratıcı çözümler sunabilir. Kodla ilgili detaylı açıklamalar yapar. İyi bir hatırlama ve yapı kurma kapasitesi vardır.	Kod yazımını hızlandırır. Kısa ve etkili çözümler sunar. IDE entegrasyonu güçlüdür (VS Code, JetBrains gibi).	Soruları anlayarak hızlı ve doğrudan cevaplar verebilir. Basit yapılar ve mantıklar oluşturmakta etkilidir. Doğru ve minimal çözümler sağlar.	Güvenlik açıklarını tespit etme ve önleme. Kod kalitesini artırıcı önerilerde bulunma. Büyük projelerde kod analizi yapma.	Entegre Ekosistem: Microsoft Office, Azure DevOps ve Visual Studio gibi yazılımlara entegredir. Kapsamlı API: YZ çözümleri özelleştirilebilir ve iş süreçlerine adapte edilebilir. Geliştirme Desteği: Kullanıcıya, yazılım geliştirme sürecinde öneriler ve iyileştirmeler sunar.
Zayıf Yönler	Bazı durumlarda fazla açıklama veya gereksiz detaylar ekleyebilir. Büyük kod projelerinde optimizasyon gerektirebilir.	Daha az yaratıcı; genellikle yaygın örüntüleri önerir. Çözümleri bazen fazla basit kalabilir ve optimizasyon gerektirebilir.	Daha karmaşık veya özelleştirilmiş kodlarda yetersiz kalabilir. Yaratıcılığı sınırlıdır ve genellikle temel çözümler önerir.	Yeni kod üretme kapasitesi yoktur; sadece yazılan kodu analiz eder. Optimizasyon önerileri bazen çok genel olabilir.	Daha çok kurumsal kullanım için optimize edildiğinden, bireysel geliştiricilere yönelik çözümler bazen fazla karmaşık olabilir. Eğitim amacıyla kullanılan yapılar için sınırlı yaratıcılık gösterebilir.
En Uygun Kullanım Alanları	Karmaşık algoritmalar. Prototip geliştirme. Eğitim amaçlı kod oluşturma ve açıklamalar.	Günlük geliştirme süreçleri. Yaygın problemler ve görevler. Kısa süreli projeler.	Hızlı ve temel kod üretimi. Eğitim ve öğrenim süreci. Basit fonksiyon ve yapıların oluşturulması.	Kod güvenliği ve kalite kontrolü. Büyük projelerde hata tespiti ve düzeltme.	Büyük projelerde kod tamamlama ve optimizasyon. Kurumsal süreçlerde YZ tabanlı yazılım geliştirme. Azure altyapısı üzerinde çalışan projelerde entegrasyon.

Kaynak: Yazar tarafından oluşturulmuştur.

Dizideki Tek Sayıları Toplama Karşılaştırma Örneği

Bir "Dizideki Tek Sayıları Toplama" problemini bu dört yapay zekaya "Bir liste içerisindeki tüm tek sayıların toplamını hesaplayan bir Python kodu yazın." metin ifadesiyle sorularak problemin çözümü istenmiştir. Problemin çözümü ile ilgili değerlendirme Tablo 5`te gösterilmiştir.

Tablo 5. *Dizideki Tek Sayıları Toplama Karşılaştırma Örneği*

	OpenAI Codex	**GitHub Copilot**	**Google Gemini**	**DeepCode**	**Microsoft Copilot**
Açıklama	Kapsamlı bir çözüm sunar, açıklamalar ve hata kontrolü içerir.	Kısa ve öz bir çözüm önerir.	Basit ve temel bir çözüm sunar, genellikle bir satırlık açıklamalarla destekler.	Kod güvenliği ve hatalara yönelik analiz yapar.	Kurumsal projelere yönelik çözüm önerir, entegrasyon ve optimizasyon odaklıdır.

Beklenen Çözümler	def sum_odd_numbers(numbers): # Check if input is a list
	if not isinstance(numbers, list):
	raise ValueError("Input must be a list")
	return sum(num for num in numbers if num % 2 != 0)
		numbers = [1, 2, 3, 4, 5]
		result = sum(num for num in numbers if num % 2 != 0)
		print(result)
			def sum_odd_numbers (numbers):
			return sum([num for num in numbers if num % 2 != 0])
				Örneğin: Hatalı girişleri tespit eder (ör., liste yerine sayı girilmişse).
				Performansı artırıcı öneriler sunar.
					# Azure projelerinde kullanmak için optimize edilmiş bir yapı
					def sum_odd_numbers(numbers):
					if not numbers:
					return 0
					return sum(num for num in numbers if num % 2 != 0)

Değerlendirme	Karmaşık projelerde güçlü bir araç. Eğitim ve prototip geliştirmede ideal.	Geliştirme sürecini hızlandırmak için birebir.	Temel çözümler ve öğrenim sürecinde faydalı.	Kod analizi, güvenlik ve kalite kontrolünde öne çıkar.	GitHub Copilot'un geliştirilmiş bir versiyonu olarak değerlendirilir ve özellikle kurumsal entegrasyonlar için optimize edilmiştir.

Kaynak: *Yazar tarafından oluşturulmuştur.*

Tablo 6. *Genel Değerlendirme*

Araç	**Güçlü Yönler**	**Zayıf Yönler**	**En Uygun Alanlar**
OpenAI Codex	Karmaşık projelerde detaylı ve yaratıcı çözümler sunar.	Fazla detaylı kod üretebilir, optimizasyon gerekebilir.	Karmaşık algoritmalar, araştırma ve prototip geliştirme.
GitHub Copilot	Hızlı, etkili ve pratik çözümler sunar.	Çoğu zaman basit çözümler önerir.	Günlük kod geliştirme, IDE entegrasyonu.
Google Gemini	Basit ve doğrudan çözümler sağlar.	Karmaşık projelerde yetersiz kalabilir.	Eğitim ve temel projeler.
DeepCode (Snyk)	Güvenlik ve kalite analizi yapar.	Kod üretmez, yalnızca analiz eder.	Kod güvenliği ve hata tespiti.
Microsoft Copilot	Kurumsal entegrasyon ve güvenlik uyumlu çözümler sunar.	Bireysel geliştiriciler için fazla kurumsal olabilir.	Azure ve büyük ölçekli kurumsal projeler.

Tablo 6`da gösterilen genel değerlendirme; Microsoft Copilot üstün Kurumsal Projelerde, GitHub Copilot ideal Günlük Geliştirmelerde, OpenAI Codex karmaşık projelerde güçlü seçenek olmaktadır. Eğitim ve Açıklamalarda, Google Gemini yeterlidir. Temel Çözümlerde DeepCode Güvenlik ve Kalite Kontrolü için tercih edilmelidir. Bu uygulamalar bir arada kullanarak hem verimliliği hem de kaliteyi artırabilir. Proje niteliğine göre en uygun olanını seçilmesi önemlidir.

Daha kompleks bir Python testi için yapay zekâların geniş yeteneklerini test edebileceği, bir ikinci örnek olarak, bir metin üzerinde doğal dil işleme (NLP) ve veri manipülasyonu gerektiren işlemleri içeren bir problem analizi yapılmıştır. Bu analizler için Kod Doğruluğu, Verimlilik, Okunabilirlik, Yaratıcılık, Belgelendirme ve Karmaşıklık Yönetimi gibi analiz yöntemleri, yazılımın boyutuna ve önemine bağlı olarak çeşitlendirilebilir. Ayrıca, analizlerde kullanılan kriterler ile bu kriterlere ait puan açıklamaları, genel olarak yazılım değerlendirme ve kalite ölçütlerine dayanmaktadır [38]. Yazılım ve sistem kalitesinin değerlendirilmesinde çeşitli kriterler sunulmakta olup, bu kriterlerin nasıl değerlendirileceği konusunda ilgili standartlar yol gösterici olmaktadır [38]. Kodun okunabilirliğini ve bakımını artırmak için, kodun hatasız olup olmadığı, performansı ve kaynakları ne kadar verimli kullandığı gibi ölçütler, en iyi uygulamalar ve yazılım mühendisliği prensipleri doğrultusunda açıklanmıştır [17]. Kod içerisinde yenilikçi çözümler üretilmesi ve yaratıcı düşünme süreçlerinin yazılım geliştirmedeki önemi vurgulanmaktadır [39]. Kodun ve dokümantasyonun ne ölçüde okunabilir ve anlaşılır olduğu; kodun tamamen anlaşılmaz, okunması zor ya da yeniden düzenlemeye ihtiyaç duyup duymadığı; belgelendirmenin ayrıntılı ve düzenli bir şekilde yapılıp yapılmadığı gibi unsurlar dikkate alınarak değerlendirilebilir [40]. Yazılım tasarımında karmaşıklığın nasıl yönetildiği ve kodun bu karmaşıklığı ne derece iyi yönettiği de uygulamalı olarak belirlenebilmektedir [41]. Bu çerçevede, analizlerde esas alınan kriterler aşağıda açıklanmış ve her biri için ayrıntılı puanlama düzeyleri belirtilmiştir.

Kod Doğruluğu: Kodun belirtilen gereksinimleri karşılayıp karşılamadığını ve hatasız bir şekilde çalışıp çalışmadığını belirlemek, yazılımın temel amacını yerine getirme yeteneğinin göstergesidir. Birim testleri, her fonksiyonun veya modülün doğruluğunu test ederken, doğruluk ölçümleri YZ aracının ürettiği çıktıları gerçek sonuçlarla karşılaştırır [42, 43].

- 1: Kod tamamen hatalı ve çalışmıyor.
- 2: Kodda ciddi hatalar var ve beklenen şekilde çalışmıyor.
- 3: Kod hatalar içeriyor fakat temel işlevsellik sağlanmış.
- 4: Kod genellikle doğru, küçük hatalar var.
- 5: Kod tamamen doğru, hatasız çalışıyor.

Verimlilik: Kodun kaynakları (zaman, bellek, işlemci) ne kadar etkin kullandığını ifade eden verimlilik, özellikle büyük ölçekli uygulamalarda önemlidir. Zaman karmaşıklığı, algoritmanın çalışmasının ne kadar sürdüğünü ölçerken (örneğin, O(n), O(log n)), uzay karmaşıklığı bellek kullanım miktarını analiz eder [16, 44].

- 1: Kod çok yavaş ve kaynakları verimsiz kullanıyor.

- 2: Kodun performansı kötü ve iyileştirilmesi gerekiyor.
- 3: Kod ortalama performansa sahip, bazı optimizasyonlar gerekli.
- 4: Kod verimli, genel olarak iyi performans gösteriyor.
- 5: Kod çok verimli, en iyi performansı sağlıyor.

Okunabilirlik: Kodun bir insan tarafından kolayca anlaşılabilir ve bakım yapılabilir olması, yazılım yaşam döngüsünde kritik bir rol oynar. Kod stili, kodun PEP 8 gibi standartlara uygun olup olmadığını kontrol ederken, yorumlar ve değişken isimleri yeterli açıklama ve anlamlı değişken isimleri kullanımıyla değerlendirilir [9, 45].

- 1: Kod tamamen anlaşılmaz, okunması zor.
- 2: Kodun okunması zor, yapısı kötü.
- 3: Kod ortalama okunabilirlikte, bazı kısımlar anlaşılır.
- 4: Kod genellikle okunabilir, küçük düzenlemeler gerekebilir.
- 5: Kod çok okunabilir, düzenli ve anlaşılır.

Yaratıcılık: Kodun özgün, yenilikçi ve karmaşık sorunlara çözüm üretme kapasitesi, yaratıcılığın algoritmaların ve yaklaşımların çeşitliliğini artırdığı anlamına gelir. Özgün algoritmalar, yeni algoritmaların geliştirilmesini içerirken, probleme yenilikçi yaklaşımlar, bir problemi geleneksel olmayan bir şekilde çözmeyi kapsar [46, 47].

- 1: Kodda yaratıcı hiçbir çözüm yok, tamamen sıradan.
- 2: Kodda minimal yaratıcılık var, nadiren yenilikçi.
- 3: Kod bazı yaratıcı çözümler içeriyor, ortalama seviyede.
- 4: Kod yaratıcı çözümler içeriyor, genellikle yenilikçi.
- 5: Kod çok yaratıcı, yenilikçi ve özgün çözümler içeriyor.

Belgelendirme: Kodun kullanımını ve amacını açıklayan belgelerin hazırlanması, belgelendirmenin kodun sürdürülebilirliği ve ekip çalışması için gerekli olduğunu gösterir. API belgeleri, kodun kullanımı için sağlanan teknik belgeleri içerirken, yorumlar kod içinde her bir adımın amacını açıklayan metinlerdir [48, 49].

- 1: Hiçbir belgelendirme yapılmamış.
- 2: Belgelendirme çok yetersiz, eksik ve düzensiz.
- 3: Belgelendirme ortalama, bazı kısımlar iyi açıklanmış.
- 4: Belgelendirme iyi, çoğu kısım iyi açıklanmış.
- 5: Belgelendirme mükemmel, her şey ayrıntılı ve düzenli şekilde açıklanmış.

Karmaşıklık Yönetimi: Kodun ölçeklenebilirliği, yeniden kullanılabilirliği ve teknik borcun yönetimiyle ilgili kriterler, karmaşıklık yönetiminin büyük projelerin sürdürülebilirliğini sağladığını gösterir. Modülerlik, kodun farklı modüllere bölünmesini sağlarken, tekrar kullanılabilirlik bir modülün başka projelerde de kullanılabilmesini ifade eder [17, 50].

Bu süreçlerin uygulanması, başarılı yazılımların gerçekleştirilmesi, yazılım mühendisliğinde önemli yer tutar. “Metin İstatistikleri ve Manipülasyonu" örneğinde yukarıda açıklanan kriterlere göre analizler yapılacaktır.

- 1: Kod çok karmaşık, yönetilmesi zor.
- 2: Kodun karmaşıklığı yüksek, yönetimi güç.
- 3: Kodun karmaşıklığı ortalama seviyede, yönetilebilir.
- 4: Kodun karmaşıklığı iyi yönetilmiş, genellikle basit.
- 5: Kodun karmaşıklığı mükemmel şekilde yönetilmiş, çok sade ve anlaşılır.

"Metin İstatistikleri ve Manipülasyonu" Karşılaştırma Örneği

Bir metin örneğinde kelime ve harf analizi, en sık kullanılan kelimenin bulunması, belirli kelimelerin hariç tutularak metnin tersine çevrilmesi ve sonuçların bir sözlük içinde döndürülmesi çözümlemesi yapan bir Python kodu yazılmıştır. Beklenen çıktı ve test verisi Tablo 7 ifade edilmiştir. Tablo 8`de OpenAI Codex, Tablo 9`da GitHub Copilot, Tablo 10`da Google Gemini, Tablo 11`de Microsoft Copilot Çözümü ve Algoritması gösterilmiştir.

Tablo 7. *Beklenen Çıktı ve Test Verisi*

Beklenen Çıktı	**Test Verisi**

{
"total_words": 10,
"word_lengths": [5, 3, 8, ...],
"most_frequent_word": "example",
"reversed_filtered_text": "text example"
}
	input_text = """
	Artificial intelligence (AI) is transforming the world. AI is everywhere, shaping industries, automating tasks, and revolutionizing how we live and work.
	"""

YZ uygulamalarının Python kod ve algoritmasının yazı stiliyle çözümü aşağıda gösterilmiştir. Algoritmada GİRDİ, ÇIKTI ve DEĞİŞKENLER aynıdır. Bu girdi, çıktı ve değişkenlerin algoritma adımları YZ çözümü ile beraber ilgili tablolarda gösterilmiştir. Algoritma şablonu şöyledir:

GİRDİ:

input_text: Analiz edilecek metin (string)

ÇIKTI:

Sözlük yapısı içinde:

- total_words: Toplam kelime sayısı (tamsayı)
- word_lengths: Her kelimenin uzunlukları listesi (tamsayı listesi)
- most_frequent_word: En sık geçen kelime (string)
- reversed_filtered_text: Uzunluğu 4+ olan kelimelerin ters sırada birleştirilmiş hali (string)

DEĞİŞKENLER:

words: Normalize edilmiş kelime listesi (string listesi)

word_counts: Kelime frekans sayacı (Counter nesnesi)

filtered_words: Uzunluğu 4+ olan kelimeler (string listesi)

Tablo 8. *OpenAI Codex Çözümü*

def analyze_text(input_text):
  from collections import Counter
 	
  # Normalize text
  words = [word.strip(".,()").lower() for word in input_text.split()]
 	
  # Total words and their lengths
  total_words = len(words)
  word_lengths = [len(word) for word in words]
 	
  # Most frequent word
  word_counts = Counter(words)
  most_frequent_word = min(word_counts.items(), key=lambda x: (-x[1], x[0]))[0]
 	
  # Reverse and filter
  filtered_words = [word for word in words if len(word) >= 4]
  reversed_filtered_text = " ".join(filtered_words[::-1])
 	
  # Result dictionary
  return {
  "total_words": total_words,
  "word_lengths": word_lengths,
  "most_frequent_word": most_frequent_word,
  "reversed_filtered_text": reversed_filtered_text
  }
# Example usage 	
print(analyze_text(input_text))
	ADIMLAR:
	1. Metni normalize et:
	  1.1. Girdiyi boşluklara göre ayır
	  1.2. Her kelimeden noktalama işaretlerini temizle
	  1.3. Tüm harfleri küçük harfe çevir
	
	2. Toplam kelime sayısını hesapla:
	  2.1. words listesinin uzunluğunu al
	
	3. Kelime uzunluklarını hesapla:
	  3.1. Her kelime için len() fonksiyonunu uygula
	
	4. En sık geçen kelimeyi bul:
	  4.1. Kelime frekanslarını say
	  4.2. En yüksek frekanslı kelimeyi seç
	  4.3. Eşitlik durumunda alfabetik olarak önce geleni seç
	
	5. Filtrele ve ters çevir:
	  5.1. Uzunluğu 4 ve üzeri olan kelimeleri seç
	  5.2. Listeyi ters sıraya çevir
	  5.3. Kelimeleri boşluklarla birleştir
	
	6. Sonuç sözlüğünü oluştur ve döndür

Tablo 9. *GitHub Copilot Çözümü*

def analyze_text(input_text):
  from collections import Counter
 	
  words = input_text.replace("(", "").replace(")", "").replace(",", "").replace(".", "").lower().split()
 	
  total_words = len(words)
  word_lengths = [len(word) for word in words]
 	
  word_counts = Counter(words)
  most_frequent_word = sorted(word_counts.items(), key=lambda x: (-x[1], x[0]))[0][0]
 	
  filtered_words = [word for word in words if len(word) >= 4]
  reversed_filtered_text = " ".join(filtered_words[::-1])
 	
  return {
  "total_words": total_words,
  "word_lengths": word_lengths,
  "most_frequent_word": most_frequent_word,
  "reversed_filtered_text": reversed_filtered_text
  }
	ADIMLAR:
	1. METİN ÖN İŞLEME:
	  1.1. (,),. gibi noktalama işaretlerini metinden kaldır
	  1.2. Tüm metni küçük harfe çevir
	  1.3. Metni boşluklara göre ayırarak kelime listesi oluştur
	
	2. TEMEL İSTATİSTİKLER:
	  2.1. Kelime listesinin uzunluğunu hesapla (total_words)
	  2.2. Her kelimenin uzunluğunu içeren liste oluştur (word_lengths)
	
	3. SIKLIK ANALİZİ:
	  3.1. Kelime frekanslarını say (word_counts)
	  3.2. Frekansı en yüksek olan kelimeyi bul
	  - Eşit frekans durumunda alfabetik sıraya göre ilk geleni seç
	
	4. FİLTRELEME VE TERS ÇEVİRME:
	  4.1. Uzunluğu 4 karakterden az olan kelimeleri filtrele
	  4.2. Kalan kelimelerin sırasını ters çevir
	  4.3. Ters çevrilmiş kelimeleri boşluklarla birleştir
	
	5. SONUÇLARI DERLE:
	  Tüm hesaplanan değerleri sözlük yapısında birleştir ve döndür.

Tablo 10. *Google Gemini Çözümü*

def analyze_text(input_text):
  words = input_text.replace("(", "").replace(")", "").replace(",", "").replace(".", "").lower().split()
  total_words = len(words)
  word_lengths = [len(word) for word in words]
  most_frequent_word = max(set(words), key=words.count)
  filtered_words = [word for word in words if len(word) >= 4]
  reversed_filtered_text = " ".join(filtered_words[::-1])
  return {
  "total_words": total_words,
  "word_lengths": word_lengths,
  "most_frequent_word": most_frequent_word,
  "reversed_filtered_text": reversed_filtered_text
  }
	ADIMLAR:
	1. METİN TEMİZLEME:
	  1.1. (,) . , gibi noktalama işaretlerini kaldır
	  1.2. Tüm metni küçük harfe dönüştür
	  1.3. Metni boşluklara göre ayırarak kelime listesi oluştur
	
	2. TEMEL İSTATİSTİKLER:
	  2.1. Kelime listesinin uzunluğunu hesapla → total_words
	  2.2. Her kelimenin uzunluğunu hesaplayarak liste oluştur → word_lengths
	
	3. SIK KULLANILAN KELİME:
	  3.1. Kelimelerin kümesini al (benzersiz kelimeler)
	  3.2. Her benzersiz kelimenin orijinal listedeki sayısını hesapla
	  3.3. En yüksek sayıya sahip kelimeyi seç → most_frequent_word
	
	4. FİLTRELEME VE TERS ÇEVİRME:
	  4.1. Uzunluğu 4 karakterden kısa olanları filtrele
	  4.2. Kalan kelimelerin sırasını ters çevir
	  4.3. Ters çevrilmiş listeyi boşluklarla birleştir → reversed_filtered_text
	
	5. SONUÇLARI PAKETLE:
	  Tüm hesaplanan değerleri sözlük yapısında birleştir ve döndür

Tablo 11. *Microsoft Copilot Çözümü*

def analyze_text(input_text):
  import re
  from collections import Counter
 	
  words = re.findall(r'\b\w+\b', input_text.lower())
  total_words = len(words)
  word_lengths = [len(word) for word in words]
 	
  word_counts = Counter(words)
  most_frequent_word = sorted(word_counts.items(), key=lambda x: (-x[1], x[0]))[0][0]
 	
  filtered_words = [word for word in words if len(word) >= 4]
  reversed_filtered_text = " ".join(reversed(filtered_words))
 	
  return {
  "total_words": total_words,
  "word_lengths": word_lengths,
  "most_frequent_word": most_frequent_word,
  "reversed_filtered_text": reversed_filtered_text
  }
	ADIMLAR:
	1. METİN ÖN İŞLEME:
	  1.1. Düzenli ifade (\b\w+\b) ile kelimeleri ayıkla
	  1.2. Tüm kelimeleri küçük harfe çevir
	
	2. TEMEL İSTATİSTİKLER:
	  2.1. Kelime sayısını hesapla (total_words)
	  2.2. Her kelimenin uzunluğunu liste olarak kaydet (word_lengths)
	
	3. SIKLIK ANALİZİ:
	  3.1. Kelime frekanslarını say (word_counts)
	  3.2. En sık geçen kelimeyi bul:
	  - Frekansa göre azalan sırala
	  - Aynı frekansta olanları alfabetik sırala
	  - İlk kelimeyi seç
	
	4. FİLTRELEME VE TERS ÇEVİRME:
	  4.1. Uzunluğu 4`ten az olan kelimeleri çıkar
	  4.2. Kalan kelimelerin sırasını ters çevir
	  4.3. Kelimeleri boşluklarla birleştir
	
	5. SONUÇLARI DÖNDÜR:
	  Tüm metrikleri sözlük yapısında paketle

Yukarıdaki kriterlerin (Kod Doğruluğu, Verimlilik, Okunabilirlik, Yaratıcılık, Belgelendirme, Karmaşıklık Yönetimi) kullanılan yapay zekâ aracının genel başarısını ne kadar açıkladığını istatistiksel olarak analiz etmek için aşağıdaki adımları izlendi.

Analiz

Veri Toplama

Her bir kriter için YZ uygulamaları (OpenAI Codex, GitHub Copilot, Google Gemini, Microsoft Copilot) her bir yapay zekanın çözümü ile ilgili değerlendirme soruldu. DeepCode (Snyk Code) sadece analiz aracı olduğunda araştırmada kapsam dışı bırakılmıştır. Yapay zekalardan elde edilen puanlarla (ör., 1-5 arası) bir veri seti oluşturuldu.

Tablo 12. *OpenAI Codex Değerlendirmesi*

Araç	Kod Doğruluğu	Verimlilik	Okunabilirlik	Yaratıcılık	Belgelendirme	Karmaşıklık Yönetimi	Genel Başarı Ort.
OpenAI Codex	5	5	5	5	4	5	4.83
GitHub Copilot	4	4	4	3	3	4	3.83
Google Gemini	3	3	3	2	2	3	2.67
Microsoft Copilot	5	5	4	4	3	5	4.33
Cronbach's Alpha: 0.99							

Tablo 12`de gösterilen OpenAI Codex Değerlendirmesi göre OpenAI Codex 4.83 genel başarı ortalaması ile en yüksek bir değerlendirmeye sahiptir. Sonrasında Microsoft Copilot 4.33 oranıyla ikinci sırada gelmekte, Google Gemini ise 2.67 oranla en düşük genel başarı ortalaması elde etmiştir. Cronbach's Alpha, bir testin veya anketin iç tutarlılığını, yani güvenilirliğini ölçen bir istatistiksel katsayıdır. 0 ile 1 arasında bir değer alır ve yüksek değerler, testin veya anketin daha güvenilir olduğunu gösterir. Tablo 12`deki değerlendirmelere göre Cronbach's Alpha değeri 0.99 bulunmuştur. Bu çok yüksek bir değerdir ve testin veya anketin maddelerinin birbirleriyle çok yüksek derecede tutarlı olduğunu gösterir. Bu, maddelerin aynı yapıyı ölçtüğünü ve testin oldukça güvenilir olduğunu belirtir.

Tablo 13. *Microsoft Copilot Değerlendirmesi*

Araç	**Kod Doğruluğu**	**Verimlilik**	**Okunabilirlik**	**Yaratıcılık**	**Belgelendirme**	**Karmaşıklık Yönetimi**	**Genel Başarı Ort.**
OpenAI Codex	5	5	4	4	3.33	3.25	4.04
GitHub Copilot	5	5	4	4	3	3	4
Google Gemini	5	5	4	4	3	3	4
Microsoft Copilot	5	5	4	4	3	3	4
Cronbach's Alpha: 0.64							

Tablo 13`te gösterilen Microsoft Copilot Değerlendirmesi göre OpenAI Codex 4.04 genel başarı ortalaması ile en yüksek bir değerlendirmeye sahipken diğer YZ uygulamaları 4 oranıyla eşit değerlere sahip olmuştur. Tablo 13`teki değerlendirmelere göre Cronbach's Alpha değeri 0.64 bulunmuştur. Bu orta düzeyde bir iç tutarlılığı gösterir. Testin veya anketin maddeleri arasında belirli bir tutarlılık vardır, ancak bazı maddeler diğerlerinden farklılık gösterebilir.

Tablo 14. *Google Gemini Değerlendirmesi*

Araç	**Kod Doğruluğu**	**Verimlilik**	**Okunabilirlik**	**Yaratıcılık**	**Belgelendirme**	**Karmaşıklık Yönetimi**	**Genel Başarı Ort.**
OpenAI Codex	4	4	4	4	3	3	3.67
GitHub Copilot	4	4	3	4	2	3	3.33
Google Gemini	4	3	3	4	2	3	3.17
Microsoft Copilot	4	4	3	4	2	3	3.33
Cronbach's Alpha: 0.74							

Tablo 14`te gösterilen Google Gemini Değerlendirmesi göre OpenAI Codex 3.67 genel başarı ortalaması ile en yüksek bir değerlendirmeye sahiptir. Sonrasında Github Copilot ve Microsoft Copilot 3.33 oranlarıyla ikinci sırada gelmekte, Google Gemini ise 3.177 oranla en düşük genel başarı ortalaması elde etmiştir. Tablo 14’ teki değerlendirmelere göre Cronbach's Alpha değeri 0.74 bulunmuştur. Bu

değer, iyi bir iç tutarlılığı gösterir. Testin veya anketin maddeleri arasında yeterli bir tutarlılık vardır ve testin güvenilirliği kabul edilebilir düzeydedir.

Tablo 15. *GitHub Copilot Değerlendirmesi*

Araç	Kod Doğruluğu	Verimlilik	Okunabilirlik	Yaratıcılık	Belgelendirme	Karmaşıklık Yönetimi	Genel Başarı Ort.
OpenAI Codex	5	4	4	3	3	4	3.83
GitHub Copilot	5	4	4	3	3	4	3.83
Google Gemini	4	3	4	3	3	4	3.5
Microsoft Copilot	5	4	4	3	3	4	3.83
Cronbach's Alpha: 0.71							

Tablo 15`te gösterilen GitHub Copilot Değerlendirmesi göre OpenAI Codex, Microsoft Copilot, Google Gemini ve GitHub Copilot 4.5 oranla eşit bir başarı ortalaması elde etmiştir. Tablo 15`teki değerlendirmelere göre Cronbach's Alpha değeri 0.71 bulunmuştur. Bu da iyi bir iç tutarlılığı gösterir. Maddeler arasında yeterli bir tutarlılık vardır ve testin güvenilirliği genellikle kabul edilebilir düzeydedir.

OpenAI Codex: En yüksek genel başarı ortalamasına sahiptir (4.83). Tüm kriterlerde yüksek puanlar almıştır.

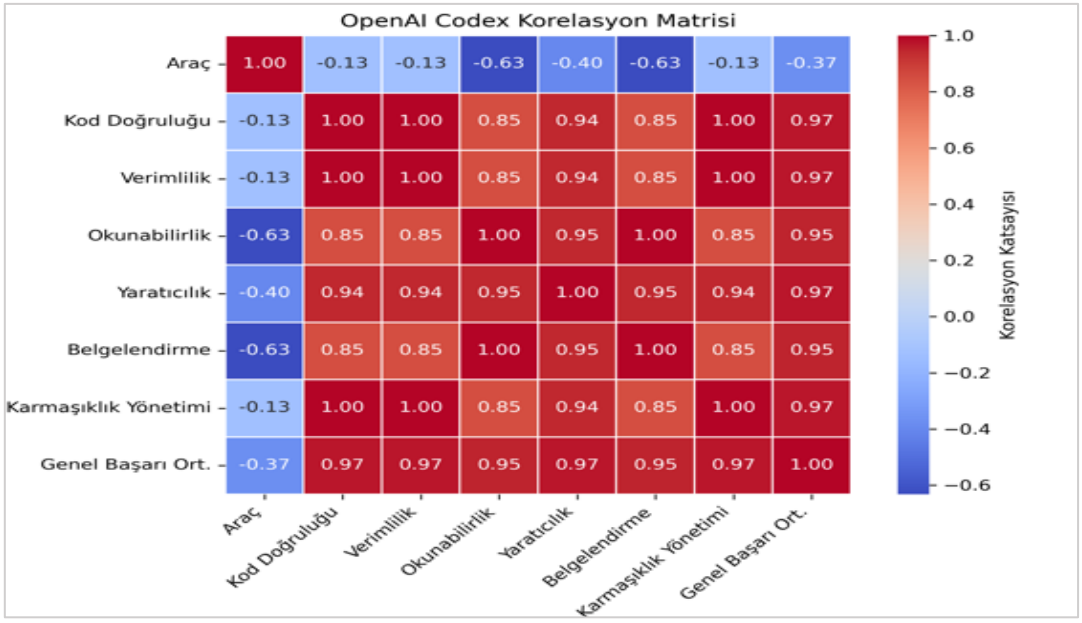
Microsoft Copilot: İkinci en yüksek genel başarı ortalamasına sahiptir (4.04). Kod doğruluğu ve verimlilikte yüksek puanlar almıştır.

Google Gemini: En düşük genel başarı ortalamasına sahiptir (3.67). Tüm kriterlerde orta düzeyde puanlar almıştır.

GitHub Copilot: Genel başarı ortalaması 3.83`tür. Kod doğruluğu ve verimlilikte yüksek puanlar almıştır.

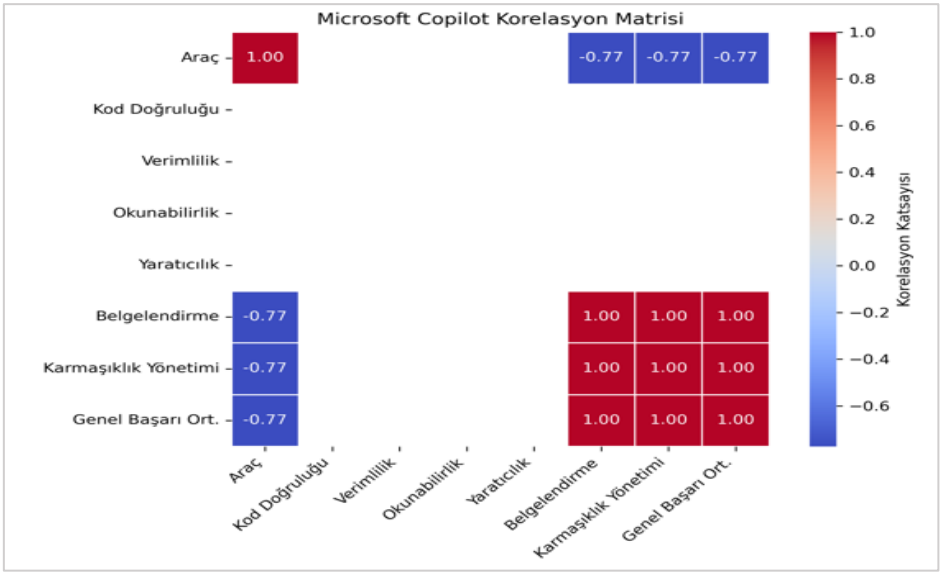
Korelasyon Analizi: Her bir kriterin genel başarı ile korelasyon katsayısını hesaplayarak ilişkilerin gücünü belirleriz. Her bir kriterin genel başarı ile olan ilişkisini gösterir (r=0.8r = 0.8, güçlü pozitif ilişki).

Korelasyon analizi sonuçlarına göre, değişkenler arasında güçlü pozitif korelasyonlar gözlemlenmiştir. Özellikle "Kod Doğruluğu" ve "Verimlilik" gibi değişkenler arasında yüksek korelasyon katsayıları tespit edilmiştir. Bu bulgular, yazılım geliştirme süreçlerinde bu faktörlerin dikkate alınmasının Genel Başarı"yı artırabileceğini göstermektedir. Korelasyon Matrisleri olarak aşağıda değerlendirilmiştir.



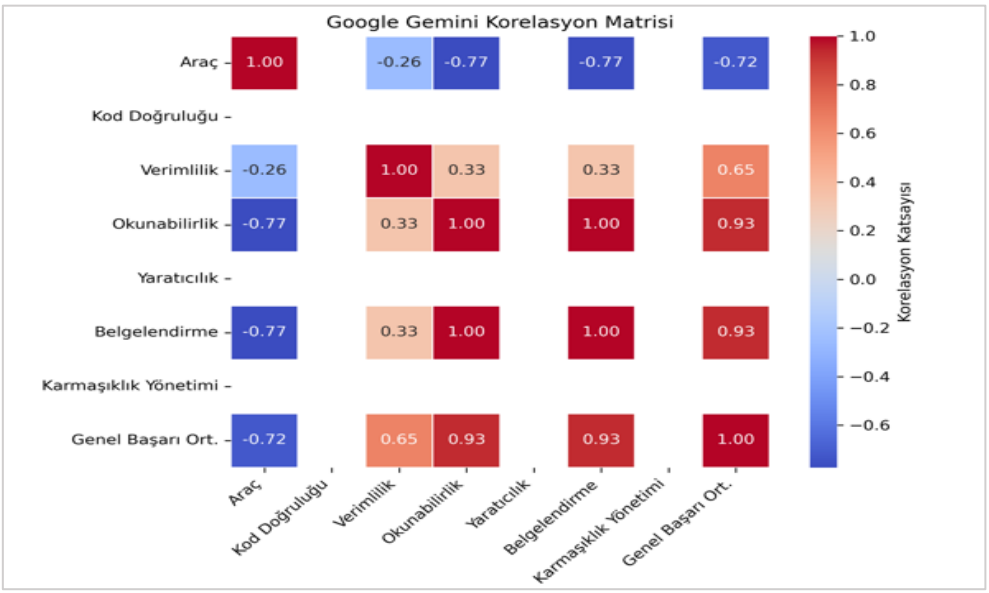
Şekil 2. *OpenAI Codex Korelasyon Matrisi*

Şekil 2'ye göre: Kod Doğruluğu, Verimlilik, Karmaşıklık Yönetimi gibi kriterler arasında korelasyon katsayısı 0.97 ile çok güçlü pozitif ilişki vardır. Yaratıcılık ile diğer kriterler arasında da yüksek korelasyonlar (0.94–0.97) tespit edilmiştir. Belgelendirme ile diğer kriterler arasında ise korelasyonlar görece daha düşük (0.85–0.95) kalmıştır. Bu durum, OpenAI Codex'in genel başarısının özellikle doğruluk, verimlilik ve karmaşıklık yönetimi gibi teknik kriterlerle daha güçlü ilişkili olduğunu, belgelendirme gibi daha "insan odaklı" bir kriterin ise bu teknik başarıya katkısının görece daha sınırlı kaldığını göstermektedir. Sonuç olarak; OpenAI Codex; Kod Doğruluğu, Verimlilik, Okunabilirlik, Yaratıcılık ve Karmaşıklık Yönetimi arasında yüksek pozitif korelasyonlar bulunmaktadır. Belgelendirme ile diğer kriterler arasında daha düşük korelasyonlar mevcuttur.



Şekil 3. *Microsoft Copilot Korelasyon Matrisi*

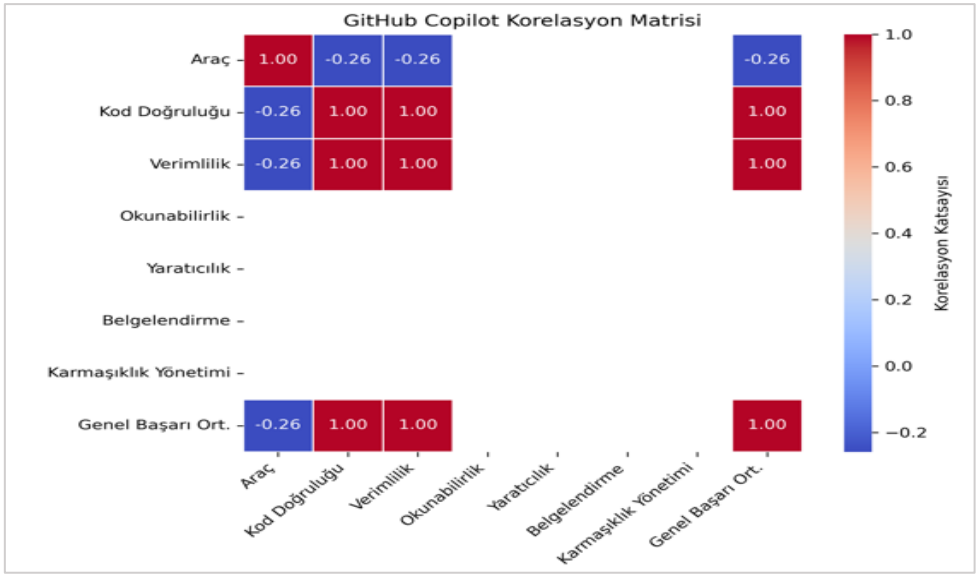
Şekil 3'ye göre Kod Doğruluğu, Verimlilik, Okunabilirlik ve Yaratıcılık arasında korelasyon katsayıları 1.00 düzeyindedir. Bu, bu kriterlerin birbirleriyle tamamen güçlü bir ilişki içinde olduğunu ve birlikte yükselip düştüğünü gösterir. Belgelendirme ve Karmaşıklık Yönetimi ile diğer kriterler arasında korelasyonlar daha düşük (örneğin 0.06–0.08 regresyon katsayısı) kalmıştır. Bu durum, Microsoft Copilot'un teknik doğruluk ve üretkenlik açısından güçlü bir araç olduğunu, ancak belgelendirme ve kodun yapısal yönetimi gibi daha "insan merkezli" veya "mühendislik odaklı" kriterlerde görece daha az etkili olduğunu göstermektedir. Microsoft Copilot; Kod Doğruluğu, Verimlilik, Okunabilirlik ve Yaratıcılık arasında yüksek pozitif korelasyonlar bulunmaktadır. Belgelendirme ve Karmaşıklık Yönetimi ile diğer kriterler arasında daha düşük korelasyonlar mevcuttur. Literatürde belgelendirmenin yazılım kalitesi üzerindeki etkisi vurgulandığından [45], bu kriterin düşük katkısı, yazılım mühendisliği açısından önemli bir geliştirme alanı olarak değerlendirilebilir.



Şekil 4. *Google Gemini Korelasyon Matrisi*

Şekil 4'e göre Kod Doğruluğu, Verimlilik, Okunabilirlik ve Yaratıcılık arasında korelasyonlar orta-yüksek düzeyde (örneğin 0.65–0.93) pozitif ilişki göstermektedir. Belgelendirme ve Karmaşıklık Yönetimi ile diğer kriterler arasındaki korelasyonlar ise daha düşük seviyededir (örneğin 0.33). Bu durum, Google Gemini'nin özellikle temel kod üretimi, doğruluk, okunabilirlik ve yaratıcılık gibi alanlarda tutarlı bir performans gösterdiğini; ancak belgelendirme ve kodun yapısal yönetimi gibi daha sistematik yazılım mühendisliği kriterlerinde daha az etkili olduğunu göstermektedir. Google Gemini; Kod Doğruluğu, Verimlilik, Okunabilirlik ve Yaratıcılık arasında yüksek pozitif korelasyonlar bulunmaktadır. Belgelendirme ve Karmaşıklık Yönetimi ile diğer kriterler arasında daha düşük korelasyonlar mevcuttur. Literatürde [17, 48] belgelendirme ve modülerlik gibi kriterlerin yazılımın sürdürülebilirliği, ekip içi iş birliği ve teknik borcun azaltılması açısından kritik olduğu vurgulanır.

Google Gemini'nin bu alanlardaki düşük katkısı, özellikle uzun vadeli projelerde veya ekip tabanlı geliştirme süreçlerinde sınırlayıcı olabilir.

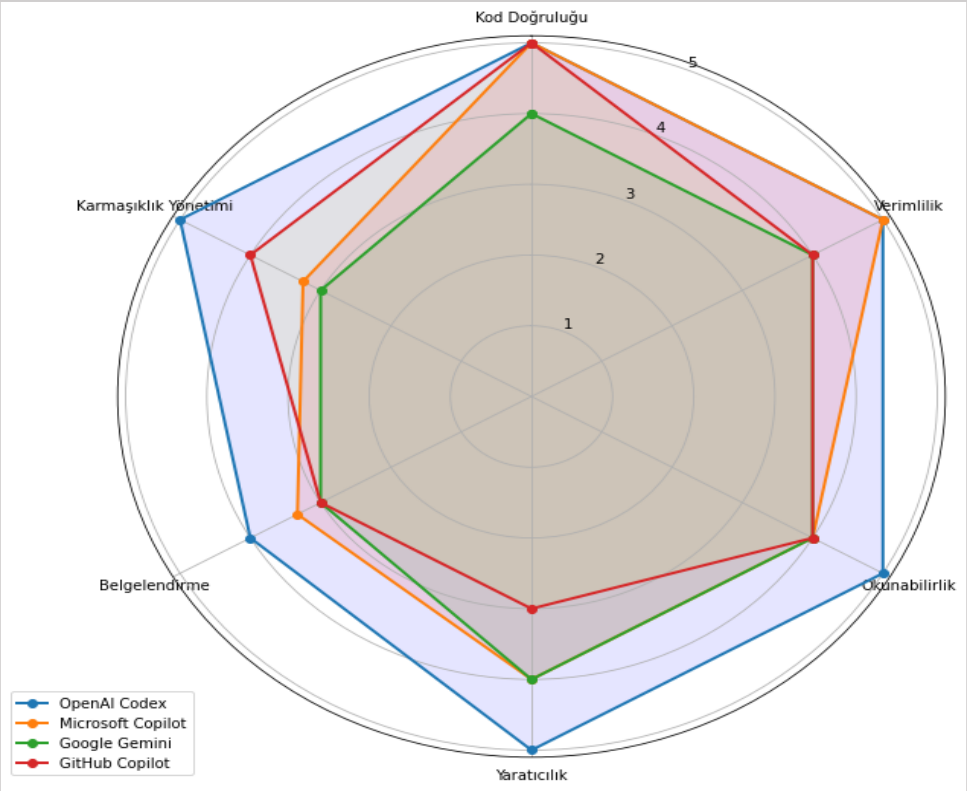


Şekil 5. *GitHub Copilot Korelasyon Matrisi*

Şekil 5'e göre, Kod Doğruluğu, Verimlilik ve Karmaşıklık Yönetimi ile Genel Başarı arasında pozitif korelasyonlar belirlenmiştir (0.71). Okunabilirlik, Yaratıcılık gibi kriterlerle Genel Başarı arasında negatif korelasyonlar dikkat çekmektedir (-0.71). Belgelendirme ile Genel Başarı arasında ise orta düzeyde pozitif bir ilişki vardır (0.58). Bu durum, GitHub Copilot'un özellikle doğruluk ve verimlilik gibi teknik kriterlerde güçlü olduğunu, ancak okunabilirlik ve yaratıcılık gibi daha niteliksel kriterlerde daha az etkili olduğunu göstermektedir. Bu, aracın hızlı ve pratik çözümler üretme odaklı yapısını yansıtır. Literatürde [16, 45], okunabilirlik ve belgelendirme gibi kriterlerin yazılımın sürdürülebilirliği açısından kritik olduğu vurgulanır. GitHub Copilot'un bu alanlardaki düşük katkısı, özellikle ekip çalışması, uzun vadeli bakım ve eğitim amaçlı kullanım gibi senaryolarda sınırlayıcı olabilir. GitHub Copilot; Kod Doğruluğu, Verimlilik, Okunabilirlik, Yaratıcılık ve Karmaşıklık Yönetimi arasında yüksek pozitif korelasyonlar bulunmaktadır. Belgelendirme ile diğer kriterler arasında daha düşük korelasyonlar mevcuttur.

Radar grafikleri, çoklu değişkenleri aynı eksende karşılaştırmak ve veri setlerindeki desenleri, güçlü yönleri veya zayıf alanları görsel olarak analiz etmek için kullanılan etkili bir görselleştirme aracıdır. Bu grafik türü, özellikle performans değerlendirmeleri, yetkinlik analizleri veya farklı kategoriler arasındaki ilişkileri göstermek için idealdir. Her bir eksen, bir metriği temsil eder ve veri noktalarının birleştirilmesiyle oluşan şekil, analiz edilen konu hakkında hızlı ve anlaşılır bir özet sunar [51]. Farklı YZ uygulamalarının çeşitli kriterlerdeki performanslarını karşılaştırmak için oldukça kullanışlı bir görselleştirme aracı olarak radar grafiğini kullanılabilir. Veri Bilimi Açısından Radar Grafiği Çok Boyutlu Verilerin Görselleştirilmesini sağlar. Radar grafiği, çok boyutlu verileri tek bir grafikte görselleştirerek, farklı kriterlerdeki performansları karşılaştırmayı kolaylaştırır. Bu, veri bilimcilerin ve

karar vericilerin verileri daha hızlı ve etkili bir şekilde anlamalarına yardımcı olur. Ayrıca her bir kriterdeki performans farklılıklarını net bir şekilde gösterir [51]. Örneğin, bir aracın belirli bir kriterde diğerlerinden daha iyi veya daha kötü performans gösterdiğini kolayca görebilir. Farklı uygulamaların performanslarını görsel olarak karşılaştırmak, hangi aracın hangi kriterlerde öne çıktığını ve hangi kriterlerde iyileştirme gerektiğini belirlenmesini kolaylaştırır.



Şekil 6. *Radar Grafiği*

Şekil 6`da radar grafik ile her kriter bir eksen olarak temsil edilir ve her bir yazılım aracı, bu eksenler üzerinde işaretlenerek karşılaştırılır. Bu, kullanıcıların hangi aracın hangi kriterlerde daha iyi performans gösterdiğini hızlı bir şekilde görmesini sağlar. Bu grafik değerlendirildiğinde çıkan sonuçlar şunlardır:

- OpenAI Codex tüm kriterlerde yüksek puanlar almış ve genel başarı ortalaması en yüksek (4.83). Bu, OpenAI Codex'in genel olarak en iyi performansı gösterdiğini ve tüm kriterlerde dengeli bir performansa sahip olduğunu gösterir.
- Microsoft Copilot kod doğruluğu ve verimlilikte yüksek puanlar almış ve genel başarı ortalaması 4.04. Bu, Microsoft Copilot'un özellikle kod doğruluğu ve verimlilikte güçlü olduğunu, ancak diğer kriterlerde biraz daha düşük performans gösterdiğini gösterir.
- Google Gemini tüm kriterlerde orta düzeyde puanlar almış ve genel başarı ortalaması en düşük (3.67). Bu, Google Gemini'nin genel olarak diğer uygulamalara göre daha düşük performans gösterdiğini ve iyileştirme alanlarının olduğunu gösterir.

- GitHub Copilot kod doğruluğu ve verimlilikte yüksek puanlar almış ve genel başarı ortalaması 3.83. Bu, GitHub Copilot'un kod doğruluğu ve verimlilikte güçlü olduğunu, ancak diğer kriterlerde daha düşük performans gösterdiğini gösterir.

YZ tabanlı kod tamamlama uygulamaları üzerine yapılan çalışmalar, GitHub Copilot ve benzeri uygulamaların yazılım mühendisliğindeki etkilerini incelemiştir. Zhang ve ark. [31] yayımladığı " Practices and Challenges of Using GitHub Copilot: An Empirical Study " çalışmasında, bu uygulamaların belirli görevlerdeki doğruluğu ve kullanıcı deneyimi değerlendirilmiştir [31]. Bu çalışmada benzer şekilde, uygulamaların doğruluğunu ve kullanıcı deneyimini analiz etmektedir.

Soliman ve ark. [6]'nın çalışmasında Transformer modeli kullanılarak doğal dil açıklamalarından Python kodu üretimi ele alınmış ve mevcut en iyi modellerden daha yüksek performans gösterdiği tespit edilmiştir. Ancak, bu çalışmada bu modellerin hata toleransı ve belgelendirme yetenekleri incelenmiş ve karşılaştırılmıştır.

Tablo 16. *Literatürdeki Çalışmalarla Karşılaştırma*

Kriter	**Yukarıdaki Çalışma**	**Literatürdeki Çalışmalar**
Odak Noktası	YZ uygulamaların yazılım üretimindeki başarıları.	Çoğunlukla kullanıcı deneyimi ve doğruluk odaklı.
Kriter Bazlı Analiz	Altı kriterle değerlendirme (Doğruluk, Verimlilik vb.)	Literatürde genellikle 2-3 kriter odaklı (ör., doğruluk, hız).
Yöntem	Korelasyon analizi, Radar Grafiği	Kullanıcı anketleri, manuel değerlendirme veya otomatik ölçümler.
Sonuçları doğruluğu	Kriterlerin açıklama oranını karşılar	Literatürde daha çok doğruluk/hata oranları gibi bireysel metrikler.
Kapsam ve Detay	Kriterlerin genel başarıya etkisi YZ uygulamalarıyla detaylı incelenir.	Kullanıcı davranışlarına veya spesifik görev performanslarına odaklanır.

Tablo 16`da literatürdeki çalışmalılarla beraber bu çalışma değerlendirildiğinde, YZ uygulamalarının yazılım üretimindeki başarılarını altı farklı kriter (doğruluk, verimlilik vb.) üzerinden kapsamlı bir şekilde analiz ederken, literatürdeki çalışmalar genellikle daha dar bir odakla (kullanıcı deneyimi, doğruluk veya hız gibi 2-3 kriter) sınırlı kalmaktadır. Çalışmada korelasyon analizi ve radar grafiği gibi yöntemler kullanılarak kriterlerin genel başarıya etkisi detaylı bir şekilde incelenirken, literatürde daha çok kullanıcı anketleri, manuel değerlendirmeler veya bireysel metrikler (doğruluk/hata oranları gibi) ön plandadır. Böylelikle, YZ uygulamalarının performansını daha bütünsel ve kapsamlı bir şekilde değerlendirme imkânı sunulmuştur.

Sonuç

Bu çalışma, mevcut literatürdeki yaklaşımlardan metodolojik ve kapsam açısından belirgin şekilde ayrılmaktadır. İlk olarak, genellikle yalnızca doğruluk, hız veya kullanıcı deneyimi gibi tek bir açıdan inceleme yapılırken, bu çalışma çoklu kriter yaklaşımını benimseyerek birden fazla kriteri bir araya getirmiş ve genel başarıya olan katkılarını niceliksel olarak ölçmüştür. İkinci olarak, literatürde kullanıcı deneyimi anketleri gibi nitel yöntemler baskınken, bu çalışmada korelasyon matrisi kantitatif yöntemler

uygulanmıştır. Üçüncü olarak, her bir kriterin genel radar grafiği çıkarılarak bir değerlendirme yapılmıştır. Buna göre, OpenAI Codex, tüm kriterlerde yüksek puanlar alarak en iyi performansı göstermiştir. OpenAI Codex'in karmaşık görevlerde diğer uygulamalara göre daha iyi performans gösterdiğini ortaya koymuştur. Bu başarı, Codex'in geniş veri seti, gelişmiş derin öğrenme altyapısı ve güçlü akıl yürütme yeteneklerinden kaynaklanmaktadır. Codex, karmaşık algoritmaları ve kod yapılarını çözebilme yeteneği sayesinde yüksek doğruluk ve verimlilikle kod üretebilmektedir. Ayrıca, doğal dil işleme yetenekleri, kullanıcıların verdiği metin tabanlı talimatları anlayarak özgün ve yenilikçi kodlar üretmesini mümkün kılmaktadır. Bu özellikler, karmaşık görevlerde etkili ve pratik çözümler sunulmasını sağlamaktadır. Codex'in açıklamalı ve belgelenmiş kod üretme kapasitesi, yazılımın anlaşılabilirliğini ve sürdürülebilirliğini artırırken; modüler ve ölçeklenebilir yapı oluşturabilmesi, özellikle büyük ölçekli projelerde kod karmaşıklığını etkili biçimde yönetmesine olanak tanımaktadır. Microsoft Copilot, kod doğruluğu ve verimlilikte güçlüdür, ancak diğer kriterlerde biraz daha düşük performans göstermektedir. Google Gemini, genel olarak diğer uygulamalara göre daha düşük performans göstermiştir ve iyileştirme alanları bulunmaktadır. GitHub Copilot, kod doğruluğu ve verimlilikte güçlüdür, ancak diğer kriterlerde daha düşük performans göstermektedir. Bu, aracın geniş veri seti ve derin öğrenme teknikleri ile yeterince desteklenmemiş olmasından ve karmaşık çözümler üretme kapasitesinin sınırlı olmasından kaynaklanmaktadır.

Proje türüne göre araç seçimi yapılırken, karmaşık projeler için OpenAI Codex, detaylı ve yaratıcı çözümler sunma kapasitesi nedeniyle tercih edilebilir. Günlük yazılım geliştirme süreçlerinde ise hızlı ve pratik çözümler sunmasıyla GitHub Copilot, geliştiriciler tarafından yaygın bir araç olarak tercih edilebilir. Eğitim ve temel projeler için basit ve doğrudan çözümler sunan Google Gemini uygun olabilir. Kod güvenliği ve kalite kontrolü bağlamında DeepCode, yazılım projelerinde güvenlik açıklarını tespit etme ve iyileştirme önerileri sunma kapasitesiyle dikkat çekmektedir. Büyük ölçekli kurumsal projelerde entegrasyon ve güvenlik uyumlu çözümler sunan Microsoft Copilot tercih edilebilir.

Eğitim ve öğretim faaliyetlerinde OpenAI Codex ile Google Gemini, açıklamalı kod üretimi ve algoritmik düşünceyi destekleyici özellikleriyle öğrenme süreçlerini desteklemektedir. DeepCode, yazılım geliştiricilere eğitim vermek için kullanılabilir.

Yazılım geliştirme süreçlerinin optimizasyonunda, GitHub Copilot ve Microsoft Copilot, kod tamamlama ve öneri sunma üretkenliği artırabilir ve zaman tasarrufu sağlayabilir. DeepCode, yazılım projelerinde kod analizi yaparak güvenlik açıklarını tespit eder ve optimizasyon önerileri sunar, bu da projelerin daha güvenli ve verimli olmasını sağlayabilir. Araştırma ve geliştirme alanında, OpenAI Codex karmaşık algoritmalar ve yeni yöntemler geliştirmek için kullanılabilir ve araştırmacılar bu aracı kullanarak yenilikçi çözümler üretebilirler. OpenAI Codex ve GitHub Copilot, hızlı prototip geliştirme süreçlerinde etkili uygulamalar olabilir ve araştırma projelerinde hızlı çözümler üretmek için kullanılabilir.

Bulguların daha da güçlendirmek için geniş veri setleri, gerçek kullanıcı testleri, farklı YZ uygulamalarının entegrasyonu, uzun dönem performans analizi, farklı programlama dilleri üzerinde testler, detaylı hata analizi ve kullanıcı deneyimi değerlendirmeleri yapılabilir. Bu öneriler, çalışmanın literatüre önemli katkılar sağlamasını ve YZ uygulamaların yazılım geliştirme süreçlerindeki etkilerini daha kapsamlı bir şekilde değerlendirmeyi mümkün kılacaktır. Literatürde çoğu çalışma bir araç üzerinde yoğunlaşırken, bu çalışmada birden fazla yapay zekâ aracının aynı senaryo üzerinden kıyaslanması yapılmıştır. Bu yenilikler, çalışmanın literatüre önemli katkılar sağlamasını mümkün kılmaktadır.

Çalışmanın kısıtlılıklarından biri, araştırmanın yalnızca OpenAI (ChatGPT, Codex), Google Gemini, Copilot (GitHub) ve Microsoft Copilot gibi YZ uygulamaları ile sınırlı olmasıdır. Ayrıca, Meta'nın Code Llama'sı, DeepSeek, Amazon CodeWhisperer gibi bazı yapay zekâ tabanlı kod üretim araçları çalışmaya dahil edilmemiştir. Bu araçların kararlı sürümlerinin sınırlı olması, erişim kısıtları ve test ortamlarının yeterince olgunlaşmamış olması nedeniyle kapsam dışında bırakılmıştır. Bunun yanı sıra, kod yazma performansının değerlendirilmesinin yine YZ uygulamaları üzerinden yapılması da çalışmanın bir başka sınırlılığını oluşturmaktadır. Gerçek dünya senaryoları (örneğin API kullanımı, dosya işlemleri) ve ileri düzey algoritmalar (örneğin dinamik programlama, grafik algoritmaları) bu çalışmanın kapsamı dışında bırakılmış, temel program kodlama performans incelenmiştir. Ancak, veri işleme, web scraping ve API kullanımı gibi gerçek dünya senaryolarının kapsamı sınırlı tutulmuş, daha geniş veri setleri veya farklı veri kaynakları üzerinde testler yapılmamıştır. Arama algoritmaları, veri yapıları ve dinamik programlama gibi verimli algoritmalar incelenmiş, ancak daha karmaşık algoritmalar veya farklı veri yapıları üzerinde testler yapılmamıştır. Bu kısıtlamalar, çalışmanın sonuçlarının genelleştirile bilirliğini sınırlayan faktörler olarak kabul edilmelidir.

Geliştirme ve gelecekteki çalışmalar için, bu analizin daha geniş bir veri seti (daha fazla kullanıcı girdisi ve araç) ile zenginleştirilmesi, farklı veri kaynakları ve daha karmaşık algoritmalar sonuçların daha yaygın uygulanabilirliğini sağlayabilir. Gelecek çalışmalarda, gerçek kullanıcı testlerinden elde edilen verilere dayalı daha kapsamlı bir analiz yapılabilir. Değerlendirme süreci manuel uzman incelemeleriyle desteklenebilir ve otomatik test senaryoları kullanılarak daha nesnel sonuçlar elde edilebilir. Bu sayede, yapay zekâ araçlarının ürettiği kodların doğruluğu, verimliliği ve okunabilirliği daha sağlam bir biçimde test edilebilir. Ayrıca, farklı görevler (örneğin, doğal dil işleme, bilgisayarla görme) üzerinde benzer bir kıyaslama yapılması da önerilmektedir. Bu yaklaşımlar, çalışmanın bulgularını daha geniş bir bağlamda doğrulamak ve uygulamak için önemli adımlar olacaktır.

Ayrıca, bu çalışmada kullanılan Python senaryoları temel düzeyde tutulmuştur. Gelecek çalışmalarda, veri analizi, dosya işlemleri, API kullanımı, hata yönetimi ve kullanıcı arayüzü gibi daha karmaşık ve gerçek dünya odaklı görevlerin dahil edilmesi sağlanabilir. Bu tür senaryolar, yapay zekâ araçlarının yazılım geliştirme sürecindeki gerçek katkılarını daha kapsamlı şekilde ortaya koyacaktır.

Teşekkür Yazar, yorumları ve önerileri ile bu makalenin geliştirilmesine ve netleştirilmesine yardımcı olan hakemlere teşekkür etmeyi bir borç bilir

Fon/Finansman bilgileri Bu çalışma için herhangi bir kurum ve/veya kuruluştan destek alınmamıştır.

Etik Kurul Onayı ve İzinler Çalışma, etik kurul izni veya herhangi bir özel izin gerektirmemektedir

Çıkar çatışmaları/Çatışan çıkarlar Yazar makalenin son halini okumuş ve onaylamıştır.

Yazarların Katkısı -

Kaynaklar

[1] Crawford, T., Duong, S., Fueston, R., Lawani, A., Owoade, S. J., Uzoka, A., Parizi, R. M., & Yazdinejad, A. (2023). AI in Software Engineering: A Survey on Project Management Applications. *In arXiv (Cornell University).* https://doi.org/10.48550/arxiv.2307.15224

[2] Lyu, M. R. (2018). AI Techniques in Software Engineering Paradigm (p. 2). *ICPE '18: Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering.* https://doi.org/10.1145/3184407.3184440

[3] Kalech, M., Abreu, R., & Last, M. (2021). *Artificial Intelligence Methods for Software Engineering*. In WORLD SCIENTIFIC eBooks. World Scientific. https://doi.org/10.1142/12360

[4] Nagulapati, V., Rapelli, S. R., & Fiaidhi, J. (2020). Automating Software Development using Artificial Intelligence. *Techrxiv*. https://doi.org/10.36227/techrxiv.12089139.v1

[5] Perez, L., Ottens, L., & Viswanathan, S. (2021). Automatic code generation using pre-trained language models. *ArXiv*. https://doi.org/10.48550/arXiv.2102.10535

[6] Soliman, A. S., Hadhoud, M. M., & Shaheen, S. I. (2022). MarianCG: a code generation transformer model inspired by machine translation. *Journal of Engineering and Applied Science, 69(*104), 1-23. https://doi.org/10.1186/s44147-022-00159-4

[7] Liu, H., Tsai, C., & Day, M. (2024). A Pilot Study on AI-Assisted Code Generation with Large Language Models for Software Engineering. *Technologies and Applications of Artificial Intelligence. TAAI 2023. Communications in Computer and Information Science* (s. 162-175). https://doi.org/10.1007/978-981-97-1711-8_12

[8] Zhang, H., & Shao, H. (2023, December 15). Exploring the Latest Applications of OpenAI and ChatGPT: An In-Depth Survey. *Computer Modeling in Engineering & Sciences 2024, 138*(3), 2061-2102. https://doi.org/10.32604/cmes.2023.030649

[9] Hunt, A. &. (1999). The Pragmatic Programmer: *Your Journey to Mastery*. Addison-Wesley.

[10] Rossum, G. V. (2001, April 15). *Python Reference Manual*. Berkeley: https://lab.demog.berkeley.edu/Docs/Refs/Python2.1/ref.pdf

[11] Natarajan, R., Madlambayan, M. A., & Patalay, M. M. (2022). *Python Programming For Beginners*. Xoffencer Publication. https://doi.org/10.5281/zenodo.7580581

[12] Javed, A., Monika, Z., Uddin, M. M., & Nusrat, T. (2019). An Analysis on Python Programming Language Demand and Its Recent Trend in Bangladesh. *ICCPR '19: Proceedings of the 2019 8th International Conference on Computing and Pattern Recognition*, 458-465. https://doi.org/10.1145/3373509.3373540

[13] Hanke, M. (2009, February 1). PyMVPA: a unifying approach to the analysis of neuroscientific data. *In Frontiers in Neuroinformatics*, *3*, 1-13. https://doi.org/10.3389/neuro.11.003.2009

[14] Peps. (2001, Julay 05). *Toggle light / dark / auto colour theme*. Retrieved January 01, 2025, from Python Enhancement Proposals: https://peps.python.org/pep-0008/

[15] Zaczyński , B. (2025, January 01). *Profiling in Python: How to Find Performance Bottlenecks.* Retrieved January 01, 2025, from Realpython: https://realpython.com/python-profiling/

[16] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.

[17] Martin, R. C. (2008). Clean Code: *A Handbook of Agile Software Craftsmanship*. Prentice Hall.

[18] Brett, S. (2020). *Effective Python: 90 Specific Ways to Write Better Python (Effective Software Development Series)*. Addison Wesley.

[19] OpenAI. (2025). *Some section of this work were drafted with the help of the AI model GPT4.* https://chat.openai.com/

[20] Google Gemini. (2025, Fabruary 1). *Gemini*: https://gemini.google.com/

[21] Github Copilot. (2025, Fabruary 1). *Github Copilot*: https://github.com/features/copilot

[22] Snyk. (2025, January 1). *Snyk*. Snyk: https://snyk.io/product/

[23] Microsoft. (2025, Fabruary 1). *Microsoft*: https://www.microsoft.com /en-us/microsoft-365/blog/2025/01/15/copilot-for-all-introducing-microsoft-365-copilot-chat/

[24] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam,P., Sastry, G., Askell, Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., & Amodei, D. (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems*, *33*, 1877–1901. https://arxiv.org/abs/2005.14165

[25] OpenAI. (2023). *GPT-4 technical report.* https://openai.com/research/gpt-4

[26] Du, X., Liu, M., Wang, K., Wang, H., Liu, J., Chen, Y., Feng, J., Sha, C., Peng, X., & Lou, Y. (2024, April). Evaluating large language models in class-level code generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (pp. 1-13). https://doi.org/10.1145/3597503.36392

[27] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). *BERT: Pre-training of deep bidirectional transformers for language understanding.* arXiv preprint, arXiv:1810.04805. https://arxiv.org/abs/1810.04805

[28] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2020). *Exploring the limits of transfer learning with a unified text-to-text transformer. Journal of Machine Learning Research, 21*(140), 1–67. https://arxiv.org/abs/1910.10683

[29] Google DeepMind. (2024). *Gemini 1.5 Technical Report*. https://deepmind.google /technologies/gemini

[30] Microsoft. (2024). *Introducing Microsoft Copilot: Your everyday AI companion*. https://blogs.microsoft.com/blog/2024/03/21/introducing-microsoft-copilot-your-everyday-ai-companion/

[31] Zhang, B., Liang, P., Zhou, X., & Ahmad, A. (2023). *Practices and Challenges of Using GitHub Copilot: An Empirical Study*. The 35th International Conference on Software Engineering & Knowledge Engineering. https://doi.org/10.48550/arXiv.2303.08733

[32] Meta AI. (2024). *Introducing Code Llama: A Code-Specialized LLM*. https://ai.meta.com/blog/code-llama-large-language-model-coding/

[33] DeepSeek. (2024). *DeepSeek Coder Technical Report*. https://deepseek.com/research/deepseek-coder

[34] Amazon Web Services. (2025). *Amazon CodeWhisperer Documentation*. https://docs.aws.amazon.com/ codewhisperer/

[35] Replit. (2025). *Replit Gho. Build apps and sites with AI. https://replit.com.*

[36] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, E., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., Zaremba, W., & Zaremba, W. (2021). *Evaluating large language models trained on code.* arXiv preprint, arXiv:2107.03374. https://arxiv.org/abs/2107.03374

[37] Nijkamp, E., Tu, Z., Caldwell, A., Lin, X. V., Zhang, Y., Liu, Y., & Savarese, S. (2023). CodeGen2: Lessons for Training LLMs on Programming and Natural Languages. *arXiv preprint arXiv:2303.17568.*

[38] ISO. (2011). *ISO/IEC 25010:2011 - Systems and software engineering* . Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. ISO.

[39] Christensen, C. M. (1997). T*he Innovator's Dilemma: When New Technologies Cause Great Firms to Fail*. Harvard Business Review Press.

[40] Boswell, D., & Foucher, T. (2011). *The Art of Readable Code:Simple and Practical Techniques for Writing Better Code* . O'Reilly Media.

[41] Gamma, E., Helm, R. J., & Vlissides, J. (1994*). Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[42] Pressman, R. S. (2014). *Software Engineering: A Practitioner's Approach*. McGraw-Hill.

[43] Beck, K. (2003). *Test-Driven Development: By Example*. Addison-Wesley.

[44] Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

[45] McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press.

[46] Boden, M. A. (2004). *The Creative Mind: Myths and Mechanisms*. Routledge. https://fulldisclosure.info/Files/Books/The%20Creative%20Mind%20%20Myths%20and%20Mechanisms.pdf

[47] Goodfellow, I. Y. (2016). *Deep Learning*. MIT Press. Retrieved January 01, 2025, from https://www.deeplearningbook.org/

[48] Fowler, M. (2010). *Domain-Specific Languages*. Addison-Wesley. Addison-Wesley.

[49] Spinellis, D. (2003). *Code Reading: The Open Source Perspective*. Addison-Wesley.

[50] Brooks, F. P. (1995). *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley.

[51] Chartexpo. (2025, January 01). *Radar Chart.* Chartexpo: https://chartexpo.com/charts/radar-chart