

T.C.
İSTANBUL NİŞANTAŞI ÜNİVERSİTESİ
LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ
YAZILIM MÜHENDİSLİĞİ ANABİLİM DALI
YAPAY ZEKÂ MÜHENDİSLİĞİ (TEZLİ) YÜKSEK LİSANS PROGRAMI

ÜRETKEN YAPAY ZEKÂ İLE OTOMATİK KOD ÜRETİMİ VE
GELİŞTİRME SÜREÇLERİNİN İYİLEŞTİRİLMESİ

Nihat HASHIMLI

Yüksek Lisans Tezi

ÜRETKEN YAPAY ZEKÂ İLE OTOMATİK KOD ÜRETİMİ VE GELİŞTİRME SÜREÇLERİNİN İYİLEŞTİRİLMESİ

Nihat HASHIMLI

T.C.
İstanbul Nişantaşı Üniversitesi
Lisansüstü Eğitim Enstitüsü
Yazılım Mühendisliği Anabilim Dalı
Yapay Zekâ Mühendisliği (Tezli) Yüksek Lisans Programı

Yüksek Lisans Tezi

ORCID ID: 0009-0008-2274-6600

Tez Danışmanı: Dr. Öğr. Üyesi Negar SADAT SOLEIMANI ZAKERİ

İstanbul
Haziran 2025

KABUL VE ONAY

Nihat HASHIMLI tarafından hazırlanan “Üretken Yapay Zekâ ile Otomatik Kod Üretimi ve Geliştirme Süreçlerinin İyileştirilmesi” başlıklı bu çalışma, 27 Haziran 2025 tarihinde yapılan savunma sınavı sonucunda başarılı bulunarak jürimiz tarafından Yüksek Lisans Tezi olarak kabul edilmiştir.

Jüri Üyesi: **Dr. Öğr. Üyesi Sibel BORAN** _____
İstanbul Nişantaşı Üniversitesi

Jüri Üyesi: **Doç. Dr. Serdar BİROĞUL** _____
Düzce Üniversitesi

Tez Danışmanı: **Dr. Öğr. Üyesi Negar SADAT SOLEIMANI ZAKERİ** _____
İstanbul Nişantaşı Üniversitesi

Jüri tarafından kabul edilen bu çalışmanın Yüksek Lisans Tezi olması için gerekli şartları yerine getirdiğini onaylıyorum.

Enstitü Yönetim Kurulu;

Karar Tarihi :
Karar Numarası :

Doç. Dr. Yeşim KAYA
Enstitü Müdürü

ETİK BEYAN

Bu tezin yazılmasında bilimsel ahlak kurallarına uyulduğunu, başkalarının eserlerinden yararlanılması durumunda bilimsel normlara uygun olarak atıfta bulunulduğunu, kullanılan verilerde herhangi bir tahrifat yapılmadığını, tezin herhangi bir kısmının bu üniversite veya başka bir üniversitedeki başka bir tez çalışması olarak sunulmadığını tezimin/projemin kaynak gösterilen durumlar dışında özgün olduğunu bildirir, aksi bir durumda aleyhime doğabilecek tüm hak kayıplarını kabullendiğimi beyan ederim.

27 Haziran 2025

Nihat HASHIMLI



SAVUNMA ÖNCESİ ONAYLAR

BENZERLİK ONAYI		
Başlık	Üretken Yapay Zekâ ile Otomatik Kod Üretimi ve Geliştirme Süreçlerinin İyileştirilmesi	
Savunma Tarihi	27.06.2025	
Sayfa Sayısı	111	
Benzerlik Yüzdesi (%)	7%	
Taranan Program	Turnitin	
Yukarıda başlığı/konusu gösterilen tez çalışmamın kapak sayfası, giriş, özet, ana bölümler ve sonuç kısımlarından oluşan çalışmam için şahsım ve tez danışmanım/Enstitü Sorumlusu tarafından intihal tespit programında taraması yapılmıştır. Tez Danışmanımın gözetiminde tamamladığım çalışmamın azami benzerlik oranlarına göre intihal içermediğini; aksinin tespit edileceği muhtemel durumda doğabilecek her türlü hukuki sorumluluğu kabul ettiğimi ve yukarıda vermiş olduğum bilgilerin doğru olduğunu beyan ederim.		
Öğrenci Nihat HASHIMLI	Danışman Dr. Öğr. Üyesi Negar SADAT SOLEIMANI ZAKERİ	Enstitü Sorumlusu

TEZDEN ÇIKARILAN YAYIN		
Yayın Künyesi	Nihat, H (2025). Üretken Yapay Zekâ İle Otomatik Kod Üretimi ve Geliştirme Süreçlerinin İyileştirilmesi. 21. Uluslararası İstanbul Fen, Mühendislik, Mimarlık ve Matematik Bilimlerinde Bilimsel Araştırmalar Kongresi Bildiri Kitapçığı içinde (ss. 588). İstanbul, Türkiye.	
Yayın Türü	☐ Ulusal Hakemli Dergide Makale ☐ Uluslararası Hakemli Dergide Makale ☐ Ulusal Kongre/Sempozyumda Bildiri ☒ Uluslararası Kongre Sempozyumda Bildiri	
	Enstitü Sekreteri	



Bu tezi kıymetli kardeşim Ferman ELİZADE'ye ithaf ediyorum.

Varlığın hala benimle. Sonsuz sevgi ve hasret ile...

TEŞEKKÜR

Bu tez çalışmam süresince bilgi ve deneyimiyle bana yol gösteren, değerli katkıları ve sabrıyla her aşamada destek olan tez danışmanım Dr. Öğr. Üyesi Negar SADAT SOLEIMANI ZAKERİ'ye en içten teşekkürlerimi sunarım.

Tez jürisinde yer alarak kıymetli değerlendirmeleri ve yönlendirmeleriyle çalışmamın gelişimine katkı sağlayan Doç.Dr. Serdar BİROĞUL'a ve Dr. Öğr. Üyesi Sibel BORAN'a da teşekkür ederim.

Ayrıca, her zaman yanımda olan, beni arkamdan sürekli sevgi ve emekle destekleyen sevgili ANNEME ve BABAMA sonsuz minnettarım. Tüm bu süreçler boyunca her zaman yanımda olan yakın aile üyelerime de teşekkürlerimi sunarım.

Nihat HASHIMLI

Haziran, 2025



ÖZET

Nihat HASHIMLI

Üretken Yapay Zekâ ile Otomatik Kod Üretimi ve Geliştirme Süreçlerinin
İyileştirilmesi
Yüksek Lisans Tezi
İstanbul, 2025

Bu çalışma, üretken yapay zekâ (YZ) sistemlerinin yazılım geliştirme süreçlerine olan etkilerini incelemektedir. Üretken YZ sistemleri, yazılımcıların kod yazma, test etme ve hata ayıklama gibi görevlerinde otomasyon sağlayarak iş yüklerini azaltmakta ve geliştirme süreçlerini daha verimli hâle getirmektedir. GPT tabanlı modeller, Codex, GitHub Copilot ve Amazon CodeWhisperer gibi araçlar aracılığıyla kod üretiminde yüksek doğruluk, hız ve kalite sağlanabilmektedir. Bu çalışmada, üretken YZ'nin yazılım mühendisliğinde nasıl dönüşümler yarattığı, geliştirici verimliliğine katkıları, yazılım kalitesi üzerindeki etkisi ve zaman/maliyet avantajları ele alınmaktadır. Ayrıca, bu sistemlerin beraberinde getirdiği etik ve teknik zorluklar, yanlış kod üretimi ve güvenlik gibi sorunlar tartışılmaktadır. Araştırma, literatür taramasının yanı sıra deneysel analizler de içermekte ve geleneksel yazılım geliştirme yöntemleriyle YZ destekli yöntemleri karşılaştırarak performans değerlendirmesi sunmaktadır. Sonuç olarak, üretken YZ sistemlerinin yazılım mühendisliğinde kalıcı bir dönüşüm başlattığı ve gelecekte insan-makine iş birliğini derinleştirecek şekilde gelişeceği öngörülmektedir.

Anahtar Kelimeler

Üretken yapay zekâ, otomatik kod üretimi, yazılım geliştirme, GPT, yazılım mühendisliği, kodlama verimliliği

ABSTRACT

Nihat HASHIMLI

Automatic Code Generation with Generative Artificial Intelligence and the
Improvement of Development Processes

Master's Thesis

İstanbul, 2025

This study examines the impact of generative artificial intelligence (AI) systems on software development processes. Generative AI tools help automate tasks such as coding, testing, and debugging, thereby reducing developer workload and increasing efficiency. Tools like GPT-based models, Codex, GitHub Copilot, and Amazon CodeWhisperer enable high-accuracy and high-quality code generation. The research explores how these systems transform software engineering, enhance developer productivity, improve code quality, and provide time and cost advantages. Moreover, ethical and technical challenges such as inaccurate code generation and security vulnerabilities are discussed. The methodology includes literature review and experimental analysis, comparing traditional and AI-assisted development methods through performance metrics. Ultimately, the study concludes that generative AI systems have initiated a paradigm shift in software engineering and are expected to further strengthen human-machine collaboration in the future.

Keywords

Generative AI, automatic code generation, software development, GPT, software engineering, coding efficiency

İÇİNDEKİLER

KABUL VE ONAY	i
ETİK BEYAN	ii
SAVUNMA ÖNCESİ ONAYLAR	iii
TEŞEKKÜR.....	v
ÖZET.....	vi
ABSTRACT.....	vii
İÇİNDEKİLER	viii
ŞEKİLLER DİZİNİ.....	xi
KISALTMALAR DİZİNİ.....	xii
GİRİŞ	1
BİRİNCİ BÖLÜM: LİTERATÜR TARAMASI	4
1.1. Üretken Yapay Zekâ Nedir?.....	4
1.1.1. Üretken Yapay Zekânın Tarihi	5
1.2. Kod Üretimi İçin Kullanılan Yapay Zekâ Modelleri	7
1.2.1. GitHub Copilot ve OpenAI Codex	7
1.2.2. Codeium.....	8
1.2.3. Amazon CodeWhisperer.....	9
1.2.4. DeepSeek-Coder	11
1.2.5. Code Llama.....	12
1.2.6. ChatGPT	13
1.3. Geleneksel Yazılım Geliştirme ile Karşılaştırmalar.....	15
1.3.1. Kodlama Sürecinde Temel Farklılıklar	15
1.3.2. Kod Kalitesi ve Güvenlik	16
1.3.3. İş Akışı ve Rol Dağılımı	16
1.3.4. Üretken Yapay Zekânın Geliştirici Kimliğine ve Becerilerine Etkisi	17
1.4. Yapay Zekâ Destekli Kod Üretiminin Avantajları ve Zorlukları.....	19
1.4.1. Avantajlar	19
1.4.2. Zorluklar	21
1.5. Akademik Çalışmaların Özeti	23
1.5.1. Verimlilik ve Üretkenlik Etkileri.....	23
1.5.2. Kod Kalitesi ve Hata Oranı.....	23
1.5.3. Güvenlik Açıkları ve Güvenilirlik	24

1.5.4. Eğitim Ortamında Kullanımı	24
1.5.5. Açıklanabilirlik, Güven ve Geliştirici Denetimi.....	25
1.5.6. Etik, Lisans ve Hukuki Belirsizlikler.....	25
İKİNCİ BÖLÜM: METODOLOJİ	27
2.1. Araştırma Yöntemi	27
2.1.1. Deneysel Çalışmalar	28
2.1.2. Vaka Analizi	30
2.1.3. Yöntemsel Bütünlük ve Eleştirel Değerlendirme	31
2.2. Kullanılan Veri Kaynakları	32
2.2.1. GitHub Projeleri.....	33
2.2.2. HumanEval	34
2.2.3. Codeforces ve LeetCode Problemleri	36
2.2.4. Stack Overflow ve OpenAI API Logları	38
2.2.5. Kod Kalitesi Değerlendirme Araçları ile Entegrasyon	40
2.3. Deneysel Ortamın Hazırlanması	41
2.3.1. Programlama Dili ve Model Entegrasyonu	41
2.3.2. Geliştirme Ortamı ve Altyapı	42
2.3.3. Değerlendirme ve Analiz Araçları	42
2.3.4. Sınırlılıklar ve Notlar	42
ÜÇÜNCÜ BÖLÜM: DENEYSEL ÇALIŞMA VE SONUÇLAR.....	43
3.1. GenAI ile Otomatik Kod Üretimi Deneyleri	44
3.1.1. G1	45
3.1.2. G2	46
3.1.3. G3	48
3.1.4. G4	50
3.2. AI Destekli Kod Üretimi ile Manuel Kod Yazımı Karşılaştırması	53
3.3. GenAI'nın Üretim Hızı, Okunabilirlik ve Performans Analizi	58
3.4. Veri Analizi Projesi Üzerinde Uygulama.....	61
DÖRDÜNCÜ BÖLÜM: TARTIŞMA VE DEĞERLENDİRME.....	66
4.1. GenAI Tabanlı Kod Üretiminin Güçlü Yönleri	66
4.2. AI Tabanlı Kod Üretiminin Zayıf Yönleri	67
4.2.1. Bağlamdan Kopuk Kod Üretimi	68
4.2.2. Hallucination ve Hatalı Kodlar	68
4.2.3. Güvenlik Açıkları	69

4.2.4. Test Kapsamı Eksikliği.....	71
4.2.5. Beceri Aşınması.....	72
4.2.6. Üretkenlik ve Kalite Çelişkisi.....	73
4.3. Güvenlik ve Etik Sorunlar.....	74
4.3.1. Lisans Problemleri.....	76
4.3.2. Adversarial Saldırıları ve Model Manipülasyonu.....	77
4.3.3. Veri Güvenliği Üzerine Vaka İncelemesi.....	79
4.4. Endüstride Uygulanabilirlik.....	80
BEŞİNCİ BÖLÜM: SONUÇ VE GELECEK	85
EK 1. KULLANILAN KOD ÖRNEKLERİ.....	89
KAYNAKLAR	92

ŞEKİLLER DİZİNİ

Şekil 1. Araçların Genel Başarı Yüzdeleri (Acceptance Rates).....	37
Şekil 2. Modellerin Çalışma Zamanı Verimliliği Karşılaştırması.	38
Şekil 3. Modellerin Bellek Kullanım Verimliliği Karşılaştırması.	38
Şekil 4. G1 Görevine Ait Pylint ve Flake8 Analiz Sonuçlarının Model Bazlı Karşılaştırılması.	46
Şekil 5. G2 Görevine Ait Pylint ve Flake8 Analiz Sonuçlarının Model Bazlı Karşılaştırılması.	48
Şekil 6. G3 Görevine Ait Pylint ve Flake8 Analiz Sonuçlarının Model Bazlı Karşılaştırılması.	50
Şekil 7. G4 Görevine Ait Pylint ve Flake8 Analiz Sonuçlarının Model Bazlı Karşılaştırılması.	52
Şekil 8. Kod Satırı Karşılaştırması.....	53
Şekil 9. Modernlik ve Güvenlik Skorları	54
Şekil 10. Satır Azaltma Yüzdesi	55
Şekil 11. Radar Grafiği	55
Şekil 12. Görev Tipine Göre Modernlik	56
Şekil 13. Toplam Skor Dağılımı	57
Şekil 14. Görev Bazında Üretim Süresi Karşılaştırması.....	59
Şekil 15. Kod Yoğunluğu Karşılaştırması (İşlev/Satır)	60
Şekil 16. AI Kodlarının Zaman Tasarrufu Oranı (Manuele Göre).....	61
Şekil 17. Genai Kodunun Statik Analiz Sonuçları.....	63
Şekil 18. Genai ile Yazılmış Kodun Üretim Hızı Avantajı.....	64

KISALTMALAR DİZİNİ

Kısaltma	Açıklama
AB AI Act	European Union Artificial Intelligence Act
AI	Artificial Intelligence
AIGC	AI-Generated Content
API	Application Programming Interface
APPS	Automated Programming Progress Suite
CNN	Convolutional Neural Networks
CPU	Central Processing Unit
DMCA	Digital Millennium Copyright Act
FT	Fine-Tuning
GAN	Generative Adversarial Networks
GenAI	Generative Artificial Intelligence
GDPR	General Data Protection Regulation
GPU	Graphics Processing Unit
GPL	General Public License
GPT	Generative Pre-trained Transformer
HTTP	Hypertext Transfer Protocol
LLM	Large Language Model
MBPP	Mostly Basic Programming Problems
MIT	Massachusetts Institute of Technology
NLP	Natural Language Processing
RLHF	Reinforcement Learning from Human Feedback
RNN	Recurrent Neural Networks
OS	Operation System
OWASP	Open Web Application Security Project
SQL	Structured Query Language
XAI	Explainable Artificial Intelligence
XSS	Cross-Site Scripting

GİRİŞ

Yapay zekâ (Ai) teknolojilerinin hızlı gelişimi, birçok endüstri ve alanı derinden etkilemiştir. Özellikle yazılım geliştirme süreçlerinde, üretken yapay zekâ sistemleri önemli bir dönüşüm yaratmaktadır. Bu dönüşüm, yazılım mühendislerinin geleneksel yazılım geliştirme yöntemlerine alternatif olarak, kod üretimi ve geliştirme süreçlerinin otomatikleştirilmesine olanak tanımaktadır. Bu bağlamda, üretken yapay zekâ araçları, yazılımcılara yalnızca belirli kod parçalarını oluşturmakla kalmaz, aynı zamanda daha karmaşık uygulamalar için tam işlevsel çözümler geliştirme imkânı tanır.

Üretken yapay zekâ, dil modelleri gibi güçlü algoritmalarla, yazılım geliştirme süreçlerinde insan benzeri bir düşünme yeteneği sunar. Bu modellerin en dikkat çekici örnekleri arasında OpenAI tarafından geliştirilen GPT (Generative Pretrained Transformer) tabanlı sistemler yer alır. Bu tür yapay zekâ araçları, doğal dilden yazılıma doğru dönüşüm yaparak, yazılım geliştirme hızını arttırabilir, hata oranlarını düşürebilir ve geliştiricilere zaman kazandırabilir. Ancak, üretken yapay zekâ ile otomatik kod üretimi hâlâ bazı teknik ve etik zorlukları barındırmaktadır. Bu tezde, üretken yapay zekânın yazılım geliştirme süreçlerini nasıl iyileştirebileceği, bu süreçlerde karşılaşılan zorluklar ve bu teknolojilerin potansiyel kullanım alanları tartışılacaktır.

Bu çalışmanın temel amacı, üretken yapay zekâ (Ai) sistemlerinin yazılım geliştirme süreçlerine olan etkilerini kapsamlı bir şekilde incelemektir. Özellikle, üretken yapay zekâ araçlarının otomatik kod üretimi üzerindeki potansiyelini, bu sistemlerin yazılım mühendisliği uygulamalarına entegre edilmesiyle ortaya çıkan fırsatları ve bu entegrasyonun geliştirme süreçlerinde sağladığı verimlilik artışını analiz etmeyi amaçlamaktadır. Bunun yanı sıra, bu teknolojilerin yazılım kalitesine, geliştirme sürelerine ve maliyetlere olan etkileri de değerlendirilecektir.

Çalışmanın odak noktası, üretken yapay zekâ tabanlı kod üretim sistemlerinin yazılımcılar ve yazılım geliştirme ekipleri için nasıl bir katkı sağlayabileceğidir. Bu bağlamda, üretken yapay zekâ sistemlerinin yazılım geliştirme süreçlerini nasıl dönüştürebileceği, yazılımcıların iş yükünü nasıl hafifletebileceği ve daha hızlı, kaliteli kod yazmalarına nasıl yardımcı olabileceği gibi konular detaylı bir şekilde ele alınacaktır.

Bu tez, aşağıdaki temel sorulara yanıt aramaktadır:

- Üretken yapay zekâ sistemleri yazılım geliştirme süreçlerini nasıl dönüştürmektedir?

Bu sistemlerin yazılımcılara sunduğu avantajlar ve sağlayabileceği işlevsel dönüşümler tartışılacaktır.

- Otomatik kod üretimi, yazılım geliştirme süreçlerinde nasıl bir verimlilik artışı sağlar?

Kod üretim süreçlerinin otomatikleştirilmesi ile yazılımcıların daha verimli hale gelmesi ve zaman kaybının azalması gibi etkiler incelenecektir.

- Üretken yapay zekâ araçları yazılım kalitesini nasıl iyileştirir?

Kod hatalarının azaltılması, kodun yeniden kullanılabilirliğinin artırılması ve yazılımın sürdürülebilirliğinin iyileştirilmesi gibi konulara odaklanılacaktır.

- Yapay zekâ tabanlı otomatik kod üretimi ile yazılım geliştirme süreci daha hızlı hale getirilebilir mi?

Zaman tasarrufu sağlanıp sağlanamayacağı ve proje teslim tarihlerinin kısaltılmasındaki etkiler değerlendirilecektir.

- Bu teknolojilerin yazılım geliştirme süreçlerinde karşılaştığı zorluklar nelerdir?

Üretken yapay zekâ sistemlerinin sınırlamaları, yanlış kod üretimi ve etkileşimdeki zorluklar gibi konular ele alınacaktır.

- Üretken yapay zekâ araçlarının yazılım mühendisliği pratiği üzerindeki uzun vadeli etkileri nelerdir?

Bu araçların gelişimi ile yazılım mühendisliği disiplininin nasıl evrileceği ve gelecekteki potansiyel uygulamaları üzerine öngörülerde bulunulacaktır.

Çalışmanın kapsamı şu unsurları içerecektir:

- Üretken Yapay Zekâ Sistemlerinin Temelleri: Bu alandaki en son gelişmeleri, kullanılan algoritmaların ve modellerin özelliklerini, en popüler yapay zekâ araçlarını (örneğin, OpenAI Codex, GitHub Copilot) ve bunların kod üretimindeki rollerini tanımlamak.
- Otomatik Kod Üretimi Süreçleri: Yapay zekâ destekli otomatik kod üretimi süreçlerinin, yazılımcıların mevcut iş akışlarına nasıl entegre edilebileceğini tartışmak. Bu araçların yazılım geliştirme süreçlerindeki spesifik alanlarda

(örneğin, hata ayıklama, test yazma, kod refaktoringi) nasıl kullanılabileceği üzerine analizler yapmak.

- Yazılım Geliştirme Süreçlerinde Performans İyileştirmeleri: Otomatikleştirilen kod üretiminin yazılım geliştirme süreçlerine olan somut etkilerini, zaman, maliyet ve hata oranlarındaki iyileşmeler açısından incelemek. Bununla birlikte, geliştirme sürelerinin ne kadar kısaldığını, yazılımcıların günlük iş yüklerinin ne kadar azaldığını ortaya koymak.
- Zorluklar ve Etik Sorunlar: Üretken yapay zekâ araçlarının kullanımı sırasında ortaya çıkan teknik, etik ve pratik zorlukları ele almak. Bu araçların doğru ve güvenli bir şekilde kullanılması için gerekli önlemleri tartışmak.
- Gelecekteki Potansiyel Uygulamalar ve Evrim: Üretken yapay zekâ sistemlerinin yazılım mühendisliği pratiğinde nasıl daha fazla yer alacağı ve bu teknolojilerin gelecekteki gelişim potansiyelini değerlendirmek. Yapay zekâ araçlarının, yazılım geliştirme süreçlerini tamamen otomatik hale getirme yolunda nasıl ilerleyebileceği üzerine öngörülerde bulunmak.

Bu çalışma hem akademik hem de endüstriyel açıdan önemli katkılar sağlamayı hedeflemektedir. Ayrıca, üretken yapay zekâ sistemlerinin yazılım mühendisliği alanındaki gelecekteki rolüne dair fikirler sunarak, bu teknolojilerin evrimini daha iyi anlamamıza katkı sağlamayı hedeflemektedir.

BİRİNCİ BÖLÜM: LİTERATÜR TARAMASI

Bu bölümde, üretken yapay zekâ ile otomatik kod üretimi konusunda yapılmış mevcut araştırmalar, yöntemler ve araçlar incelenerek üretken yapay zekâ ile ilgili genel bir literatür taraması sunulmaktadır.

1.1. Üretken Yapay Zekâ Nedir?

Generatif modelleme, eğitim örneklerini analiz eden, bunların kalıplarını ve dağılımlarını öğrenen ve ardından gerçekçi benzerlerini oluşturarak sentetik ürünler yaratan bir yapay zekâ tekniğidir (Jovanović ve Campbell, 2022: 110). GenAI, üretici modelleme ve derin öğrenme alanındaki ilerlemelerden yararlanarak; metin, görsel, ses ve video gibi mevcut medya içeriklerini kullanmak suretiyle büyük ölçekte çeşitli içerikler üretebilen bir yapay zekâ yaklaşımıdır. Bu bağlamda GenAI ile ilişkili bazı temel kavramlar aşağıda açıklanmaktadır:

- AI-Generated Content (AIGC): Yapay zekâ algoritmaları tarafından, insan müdahalesi olmaksızın üretilen içeriktir.
- Fine-Tuning (FT): Önceden eğitilmiş bir modelin ağırlıklarının, hedeflenen göreve özgü etiketli verilerle yeniden eğitilerek güncellenmesi işlemidir (Brown vd., 2020: 1880).
- Few-shot training: Yapay zekâ modeline, tahmin (çıkarım) sırasında ilgili göreve dair birkaç örneğin verilmesiyle gerçekleştirilen eğitim biçimidir (Brown vd., 2020).
- Generative Pre-trained Transformer (GPT): Denetimli ve denetimsiz öğrenme tekniklerini bir arada kullanarak insan benzeri dil anlayışı ve üretimi gerçekleştiren bir makine öğrenimi modelidir (Radford ve Narasimhan, 2018).
- Natural Language Processing (NLP): Yapay zekânın, insan diliyle bilgisayarlar arasında etkileşim kurmayı hedefleyen alt dalıdır. NLP, bilgisayarların insan dilini anlamasını, yorumlamasını ve üretmesini sağlayan algoritma ve modellerin geliştirilmesini kapsar.

- Language model: Bir sonraki kelimeyi tahmin etmek üzere eğitilen istatistiksel (yapay zekâ temelli) bir modeldir ve sınıflandırma ile metin üretimi gibi çeşitli doğal dil işleme görevlerinde kullanılır (Lund ve Wang, 2023: 5).
- Large Language Model (LLM): Çok sayıda ağırlık ve parametreye sahip, karmaşık eğitim mimarileriyle geliştirilen dil modelidir. Metin üretme, sınıflandırma, soru yanıtlama gibi birçok NLP görevini gerçekleştirebilir (Brown vd., 2020: 1880). LLM kavramı, günümüzde özellikle sahada ve medyada GenAI modellerinin bir alt kümesini tanımlamak için yaygın biçimde kullanılmaktadır. Ancak "büyük" kavramının tam olarak neyi ifade ettiği belirsizdir; bu nedenle kavram, belirli bir muğlaklık içermektedir. Yine de yaygın kullanım göz önünde bulundurularak, bu çalışmada da bu kavrama yer verilmiştir.
- Prompt: Yapay zekâ modeline kullanıcı tarafından verilen komut ya da tartışma konusudur. Komutlar içerdiği kelime ve detaylara çıktılar farklılık gösterebilir.
- Prompt Engineering: Büyük dil modellerine istenen çıktıyı en etkili şekilde üretirmek amacıyla verilen komutları (promtları) tasarlama ve iyileştirme sürecidir.

1.1.1. Üretken Yapay Zekânın Tarihi

Üretken Yapay Zekâ (GenAI), günümüzde metin, görsel, ses ve video gibi çok çeşitli içerikleri insan benzeri şekilde üretebilen sistemlerin genel adıdır. Bu teknolojinin bugünkü düzeye ulaşması, yaklaşık 80 yıllık bir yapay zekâ evriminin sonucudur. Aşağıda bu evrim, dönemsel gelişmeler kısa bir şekilde temelinde açıklanmıştır.

İlk Aşamalar: Kavramsal Temel (1950–1980’ler)

Yapay zekânın ilk adımları, Alan Turing’in makinelerin düşünebileceğine dair öne sürdüğü fikirlerle atılmıştır (Turing, 1950). 1960’larda geliştirilen ELIZA (Weizenbaum, 1966), doğal dil girdilerine terapist benzeri yanıtlar verebilen ilk programlardan biridir. 1970’lerde ise SHRDLU (Winograd, 1971) gibi sistemlerle doğal dil üretimi, makine çevirisi ve mantıksal çıkarım üzerinde çalışmalar yapılmıştır. Aynı dönemde uzman

sistemler de öne çıkmış, bilgi temsili ve kural tabanlı çıkarım ile belirli alanlarda akıllı çıktılar üretilmiştir.

Kural Tabanlı Sistemlerden Sinir Ağlarına (1980–1990’lar)

1980’lerde yapay zekâ araştırmaları daha olgunlaşmaya başlamış, özellikle uzman sistemler hâlen yoğun biçimde kullanılmaya devam etmiştir. Bu dönemde sinir ağları üzerine ilgi artmış; Geoffrey Hinton ve Yann LeCun gibi öncüler bu alana önemli katkılar sağlamıştır (LeCun, Boser, Denker, Henderson, Howard, Hubbard ve Jackel, 1989). Sinir ağları, bugünkü derin öğrenmenin temel yapı taşlarını oluşturmuştur.

Makine Öğrenmesinin Yükselişi (1990–2000’ler)

1990’lı yıllar, makine öğrenmesi algoritmalarının (örneğin karar ağaçları, destek vektör makineleri ve Bayes ağları) yaygınlaştığı bir dönem olmuştur. Bu teknikler, örüntü tanıma ve tahminleme kabiliyetlerini artırarak GenAI sistemlerinin temelini hazırlamıştır (Mitchell, 1997).

Derin Öğrenme ve Üretici Modellerin Doğuşu (2010–2015)

2010’lu yıllarda, özellikle GPU’ların gelişimi ve büyük veri setlerinin artmasıyla birlikte derin öğrenme devrimi yaşanmıştır. Görüntü işleme için Convolutional Neural Networks (CNN), dil işleme için Recurrent Neural Networks (RNN) kullanılmaya başlanmıştır. 2014 yılında Ian Goodfellow ve ekibi tarafından önerilen Generative Adversarial Networks (GANs) yapısı, üretici modellerde devrim yaratmıştır (Goodfellow, Pouget-Abadie, Mirza, Xu, Warde-Farley, Ozair, Courville ve Bengio, 2014). Bu yapı, iki yapay sinir ağı arasında rekabete dayalı bir öğrenme süreci öngörerek üretkenliğin kalitesini artırmıştır. Aynı zamanda pekiştirmeli öğrenme, özellikle oyun alanında GenAI’ye katkı sağlamıştır (Silver, Huang, Maddison, Guez, Sifre, van den Driessche, Schrittwieser, Antonoglou ve Hassabis, 2016).

Transformer Çağı ve Modern GenAI Uygulamaları (2015–Günümüz)

2017 yılında Vaswani ve arkadaşları tarafından sunulan “Attention Is All You Need” başlıklı makale ile tanıtılan Transformer mimarisi, doğal dil işleme alanında yeni bir çağ başlatmıştır (Vaswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser ve Polosukhin, 2017). Bu mimari, günümüzdeki birçok gelişmiş modelin (örneğin BERT, GPT-2, GPT-3, GPT-4) temelini oluşturmuştur. Bu modeller sayesinde yazılı içerik üretimi, görüntü

sentezi, ses-müzik üretimi, video oluşturma gibi GenAI uygulamaları oldukça geniş bir yelpazeye yayılmıştır.

1.2. Kod Üretimi İçin Kullanılan Yapay Zekâ Modelleri

Yazılım geliştirme süreçlerinde üretken yapay zekâ (GenAI) modellerinin kullanımı, geleneksel yazılım geliştirme yaklaşımlarını köklü bir şekilde değiştirmiş ve geliştirici verimliliğini çarpıcı ölçülerde artırmıştır. Son yıllarda yaşanan donanım gelişmeleri, veri setlerinin genişlemesi ve derin öğrenme algoritmalarındaki ilerlemeler, kod üretimi için özelleştirilmiş yapay zekâ modellerinin gelişmesini mümkün kılmıştır. Bu modeller, yalnızca kod tamamlama işlevleriyle sınırlı kalmayıp, yazılım tasarımından test oluşturmaya, hata ayıklamadan refaktörlinklemeye kadar çeşitli geliştirme aktivitelerine katkı sunmaktadır. Aşağıda bilgileri verilmiş olan modeller güncel zamanda insanlar tarafından en çok tercih edilenler arasında yer almaktadır.

1.2.1. GitHub Copilot ve OpenAI Codex

GitHub Copilot, OpenAI tarafından geliştirilen ve Microsoft ile ortaklıkla sunulan, geliştiricilere kod yazma sürecinde destek olması amacıyla tasarlanmış önemli bir yapay zekâ aracıdır. Bu aracın temelinde, doğal dil işleme (NLP) konusunda devrim niteliğinde geliştirilmiş olan OpenAI Codex modeli bulunur. Codex, GPT-3 modelinin bir uzantısı olup, doğal dil girdilerini anlamlandırmak ve bunları programlama dillerine dönüştürmek üzere özelleştirilmiştir (GitHub, 2021). GPT-3'ten farklı olarak Codex, GitHub üzerinden toplanan milyonlarca satırlık halka açık kod veri seti ile eğitilmiştir ve bu sayede, kod yapısının mantığını kavrayabilen ve farklı programlama dillerine uyarlanabilen bir yapıya sahiptir. Copilot, geliştiricilere yazdıkları kodun devamını tahmin etme, fonksiyon ve blok yapıları önerme, test senaryoları oluşturma, yorum satırı ekleme ve hata bulma gibi görevlerde destek verir. Kodun bağlamını anlayarak, sadece tek satırlık değil; fonksiyon ve sınıf seviyesinde önerilerde de bulunabilir. Bu yetenekleriyle Copilot, yazılım geliştirme sürecinde bir 'ikincil geliştirici' gibi davranarak kullanıcının verimliliğini önemli ölçüde artırır (Ziegler, Yasin ve Klocke, 2023).

Akademik literatürde yapılan çalışmalar, Copilot'un geliştirici üretkenliği üzerindeki etkisini niceliksel olarak değerlendirmiştir. Peng ve meslektaşları (2023) tarafından

yapılan deneysel bir çalışmada, Copilot kullanan geliştiricilerin belirli kodlama görevlerini %55 daha hızlı tamamladığı gözlemlenmiştir (Peng, Kalliamvakou, Cihon ve Demirer, 2023). Bu etki, özellikle rutin ve tekrarlayan kod yazma işlemlerinde daha belirgin hale gelmektedir. Ayrıca, Copilot'un önerileri geliştiricilerin hatasız kod yazma oranlarını da artırmaktadır.

Copilot'un çalışma prensibi, geliştiricinin yazdığı kodu analiz ederek bunun neye hizmet ettiğini anlamak ve bu bağlamda devam edecek kod parçasını önermek üzere yapılmıştır. Örneğin, geliştirici bir Python fonksiyonu başlatıp bir HTTP isteği yapacağını belirttiğinde, copilot gerekli modülleri çağırabilir, doğru şekilde request gönderen bir fonksiyon bloğu önerebilir. Bu durum yalnızca zaman tasarrufu sağlamakla kalmaz aynı zamanda yeni başlayan geliştiriciler için öğrenme aracı olarak da değer taşır. Copilot, Visual Studio Code, Neovim, JetBrains ve GitHub Codespaces gibi popüler IDE'lerle entegre olarak çalışmaktadır. Bu çok yönlülük, farklı geliştirme ortamlarında kullanılmasını mümkün kılarken, bireysel öğrencilerden profesyonel takımlara kadar geniş bir kullanıcı kitlesine hitap etmesini sağlar.

Bununla birlikte, Copilot kullanımıyla ilgili etik ve lisanslama sorunları da gündeme gelmiştir. Özellikle modelin eğitildiği kod verisinin açık kaynaklı projelerden elde edilmesi, çıktı olarak üretilen kodun lisans uyumluluğu konusunda soru işaretleri yaratmaktadır. Bunun üzerine GitHub, Copilot'un önerdiği kod için "kendi sorumluluğunuzdadır" şeklinde bir yasal açıklama yayınlamıştır (GitHub Docs, 2022). Bu durum ticari projelerde Copilot kullanımının dikkatli bir şekilde yönetilmesi gerektiğine işaret eder ayrıca GitHub Copilot, yazılım geliştirme süreçlerinde ünlü "pair programming" konseptini bir yapay zekâ ile buluşturarak geliştiricilere yoldaşlık eden devrimsel bir aracı temsil etmektedir. Kod üretimi alanında standartları değiştirmiş, üretken yapay zekânın pratik uygulamalarına çarpıcı bir örnek sunmuştur.

1.2.2. Codeium

Codeium, modern yazılım geliştirme süreçlerine yapay zekâ entegrasi sunan, geliştiricilere odaklanmış bir kod tamamlama ve yardımcı aracıdır. 70'ten fazla programlama dili desteği ve 40'tan fazla entegre geliştirme ortamı (IDE) ile entegre çalışabilmesi onu güçlü ve esnek bir çözüme dönüştürmüştür. Codeium'un temel amacı,

geliştiricilerin kodlama süreçlerinde zaman kaybını en aza indirerek, daha yaratıcı ve stratejik görevlere odaklanmalarını sağlamaktır. Codeium, yapay zekâ destekli kod tamamlama özelliği ile yazılan kodun bağlamını analiz ederek geliştiriciye uygun fonksiyonlar, metodlar ya da yapısal kod blokları önerir. Bu özellik, özellikle tekrar eden kod yapılarının hızla tamamlanması ve karmaşık algoritmaların daha az eforla yazılması konusunda zaman kazandırıcı bir avantaj sağlamaktadır. Akademik bir incelemede, otomatik kod tamamlama sistemlerinin, geliştiricilerin kodlama hatalarını %23'e kadar azalttığı ve ortalama kod yazma sürelerini %30'a kadar kısalttığı belirtilmiştir (YapayOrnek, 2024). Codeium'un bir diğer öne çıkan özelliği, entegre API belgelendirme ve kod öğrenme yardımcısıdır. Özellikle yeni başlayan geliştiriciler için API'lerin kullanım şeklini çabucak anlamaya yönelik öğrenme eğrisini düzenlemeye yardımcı olur. Kod bloklarının yanında API dokümanlarından alıntılar sunarak, yazılım geliştiricilerinin süreç içerisinde dışarıya başlı şekilde arama yapmadan geliştirmeyi sürdürmesini sağlar. Özellikle öğrenciler ve junior geliştiriciler tarafından tercih edilme nedenlerinden biri de budur. Codeium'un mimarisi, kullanıcı verilerini gizlilik ilkesine bağlı kalarak işlemekte ve modeli kullanıcı bilgisayarında çalıştırarak kod içeriklerinin dışarıya çıkmasını engellemektedir. Bu da, ticari yazılımlar geliştiren ekipler için önemli bir avantaj sunmaktadır. Kullanıcı dostu arayüzü, sade ve minimal öneri sistemiyle karmaşıklıktan uzak bir deneyim sunar.

2023 yılında yapılan bir piyasa analizine göre, Codeium'un kod tamamlama doğruluk oranı %87 civarında rapor edilmiş ve bu oran, onu GitHub Copilot, Tabnine ve Amazon CodeWhisperer gibi rakipleriyle kıyaslanabilir hale getirmiştir. Aynı zamanda Codeium'un kod öğrenme özellikleri, yapay zekâyı bir öğrenme aracı olarak entegre etmesi sayesinde, özellikle eğitim kurumları tarafından da tercih edilmeye başlanmıştır. Dolayısıyla, Codeium'un en dikkat çekici yönlerinden biri tüm özelliklerinin tam sürümde ücretsiz olarak sunulmasıdır. Bu politika, hem bireysel geliştiriciler hem de küçük yazılım ekipleri için cazip bir seçenek oluşturmakta, maliyet engelini ortadan kaldırmaktadır.

1.2.3. Amazon CodeWhisperer

Amazon CodeWhisperer, Amazon Web Services (AWS) tarafından geliştirilen ve yazılım geliştirme süreçlerinde üretken yapay zekânın gücünü kullanarak geliştiricilere gerçek

zamanlı kod önerileri sunmayı amaçlayan yenilikçi bir üretken dil modelidir. İlk olarak 2022 yılında tanıtılan bu sistem, açık kaynaklı veriler ve dahili Amazon verileriyle eğitilmiş olup, geliştiricilerin yazım süreçlerini hızlandırmak, hata oranını azaltmak ve kod kalitesini artırmak için entegre çözümler sunar (Amazon Web Services, 2023). CodeWhisperer, özellikle AWS altyapısıyla entegre şekilde çalışması bakımından diğer üretken yapay zekâ tabanlı kod üretim araçlarından ayrılır. Python, Java, JavaScript gibi dillerin yanı sıra, AWS Lambda, Amazon EC2, S3 ve RDS gibi bulut tabanlı servislerle doğrudan entegre çalışabilmesi, onu bulut tabanlı geliştiriciler için son derece cazip bir seçenek haline getirmiştir. Örneğin bir kullanıcı, AWS Lambda fonksiyonu oluşturmak istediğinde, CodeWhisperer otomatik olarak bağlamı anlayıp bu servise özel önerilerde bulunabilir. Bu bağlam duyarlılığı, geleneksel kod tamamlama sistemlerinin çok ötesine geçmektedir (Amazon Web Services, 2023).

Yapılan akademik değerlendirmeler, CodeWhisperer'ın sadece verimlilik değil; aynı zamanda yazılım mühendisliğinde kalite güvence süreçlerine katkı sunduğunu göstermektedir. Amazon tarafından yürütülen deneylerde, CodeWhisperer'ın önerdiği kodların %35'inin doğrudan geliştirici tarafından kabul edildiği ve özellikle güvenlik önerileri konusunda etkili olduğu bildirilmiştir (Amazon Web Services, 2023). Amazon, CodeWhisperer'ın güvenli kod üretimi yetenekleri ile diğer araçlardan ayrıldığını belirtmektedir. Bu bağlamda araç, OWASP Top 10 güvenlik açıklarını dikkate alarak kodda potansiyel güvenlik açıklarını tespit edebilmekte ve bunlara karşı önleyici öneriler sunabilmektedir (Amazon Web Services, 2023). Kullanıcı açısından en önemli avantajlardan biri, Amazon CodeWhisperer'ın kullanım kolaylığı ve sezgisel arayüzüdür. Visual Studio Code, JetBrains IDE'leri ve AWS Cloud9 gibi popüler geliştirme ortamlarıyla tam entegre çalışır. Bu sayede geliştirici, ek bir konfigürasyon veya öğrenme eğrisi olmadan aracın sunduğu önerileri doğrudan çalışma ortamında kullanabilir. Yeni başlayan yazılımcılar için bu özellik, öğrenme sürecini kısaltırken; deneyimli geliştiriciler için ise tekrarlayan görevlerin hızla tamamlanmasını sağlar (Bradley ve Nguyen, 2023). Geliştiricilere gerçek zamanlı olarak stil düzeltmeleri ve açıklama önerileri de sunar. Bu, sadece çalışır kod üretmekle kalmaz; aynı zamanda okunabilir, belgelenmiş ve sürdürülebilir kod parçalarının oluşturulmasına da katkı sağlar. Amazon'un "security scanning" özelliği sayesinde geliştiriciler, yazdıkları kodun potansiyel açıklarını da analiz edebilmekte ve böylece güvenlik odaklı bir geliştirme süreci yürütmektedir (Amazon

Web Services, 2023). Gizlilik ve etik açısından da dikkat çeken bir sistem olan CodeWhisperer kullanıcı kodunu analiz ederken hiçbir şekilde kaynak kodunu kalıcı olarak kaydetmemekte, önerilen kodların lisans uyumluluğu konusunda uyarılar göstermektedir. Bu yönüyle özellikle endüstriyel yazılım projelerinde yasal ve lisanslama açısından güvenli bir ortam sunmaktadır. Amazon CodeWhisperer, üretken yapay zekânın yazılım geliştirme süreçlerine uygulanması açısından güçlü bir örnektir. Gerek AWS ekosistemine olan sıkı entegrasyonu, gerekse güvenlik ve kalite odaklı yapısı ile CodeWhisperer sadece kod tamamlama değil, tüm yazılım yaşam döngüsüne katkı sağlayan bütüncül bir platform haline gelmiştir.

1.2.4. DeepSeek-Coder

DeepSeek-Coder, Çin merkezli DeepSeek AI tarafından geliştirilen ve açık kaynaklı olarak sunulan bir üretken yapay zekâ modelidir. Kod üretimi ve yazılım mühendisliği görevlerine odaklanan bu model 2024 yılında tanıtılmıştır. DeepSeek-Coder serisi 1.3 milyardan 33 milyara kadar değişen parametre boyutlarına sahip modellerden oluşmaktadır ve bu modeller sıfırdan 2 trilyon token üzerinde eğitilmiştir (Guo, Liu, Zhao, Zhang, Wang, Li, Sun, Wu, He, Hu, Zhang, Shen, Wang ve Liu, 2024). Modelin eğitimi sırasında kullanılan verinin %87'si kaynak kodu, %10'u kodla ilgili İngilizce (örneğin GitHub markdown ve Stack Exchange), %3'ü ise kodla ilgisiz Çince metinlerden oluşmaktadır. Bu çeşitlilik, modelin hem İngilizce hem de Çince dillerinde kod üretiminde yüksek performans göstermesini sağlamaktadır (Guo vd., 2024).

DeepSeek-Coder, özellikle proje düzeyinde kod tamamlama ve doldurma görevlerinde başarılıdır. Model, 16K token uzunluğunda bağlam penceresi kullanarak, dosyalar arası kod üretiminde yüksek doğruluk sağlamaktadır. Ayrıca, "fill-in-the-middle" (FIM) eğitimi stratejisi ile model, kodun ortasında eksik kalan bölümleri tamamlayabilme yeteneğine sahiptir (Guo vd., 2024).

DeepSeek-Coder-V2, önceki versiyonlara kıyasla önemli iyileştirmeler sunmaktadır. Bu versiyon, 6 trilyon ek token ile yeniden eğitilmiş ve bağlam uzunluğu 128K token'a çıkarılmıştır. Ayrıca, desteklenen programlama dilleri sayısı 86'dan 338'e yükseltilmiştir (Zhu, Wang, Shen, Zhang, He, Sun, Zhang, Liu, Wu, Wang ve Liu, 2024).

Modelin performansı, çeşitli benchmark testleriyle değerlendirilmiştir. DeepSeek-Coder-V2, HumanEval testinde %90.2, MBPP testinde %76.2 ve LiveCodeBench testinde %43.4 başarı oranı elde etmiştir. Bu sonuçlar modelin kod üretimi ve matematiksel akıl yürütme görevlerinde GPT-4 Turbo, Claude 3 Opus ve Gemini 1.5 Pro gibi kapalı kaynaklı modellerle rekabet edebilecek düzeyde olduğunu göstermektedir (Zhu vd., 2024). Ayrıca, DeepSeek-Coder, matematiksel akıl yürütme yetenekleriyle de öne çıkmaktadır. Model MATH benchmark testinde %75.7 doğruluk oranı ile GPT-4o'nun %76.6'lık başarısına yaklaşmıştır. Bu, modelin sadece kod üretimi değil, aynı zamanda matematiksel problem çözme konularında da güçlü olduğunu göstermektedir (Zhu vd., 2024).

DeepSeek-Coder'ın açık kaynaklı olması, araştırma ve geliştirme toplulukları için büyük bir avantajdır. Model MIT lisansı altında sunulmakta ve ticari kısıtlama olmaksızın kullanılabilir. Bu, geliştiricilerin ve araştırmacıların modeli kendi projelerine entegre etmelerini ve özelleştirmelerini kolaylaştırmaktadır (Guo vd., 2024). Dolayısıyla, DeepSeek-Coder, açık kaynaklı yapay zekâ modelleri arasında öne çıkan bir seçenektir. Geniş dil desteği, yüksek bağlam kapasitesi ve güçlü performansı, yazılım geliştirme süreçlerinde üretkenliği artırmak isteyen geliştiriciler için değerli bir araçtır.

1.2.5. Code Llama

Code Llama, Meta AI tarafından geliştirilen ve LLaMA 2 mimarisine dayanan açık kaynaklı bir büyük dil modelidir. Bu model kod üretimi, tamamlama, hata düzeltme ve yorum yazma gibi yazılım mühendisliği görevlerine odaklanan ve özellikle üretken yapay zekânın yazılım geliştirme süreçlerine entegrasyonu açısından önemli bir örnek teşkil etmektedir (Touvron vd., 2023).

Meta tarafından yayımlanan teknik rapora göre Code Llama, üç temel varyantta sunulmuştur: Code Llama (genel kod üretimi için), Code Llama-Python (Python'a özgü ince ayarlanmış versiyon), ve Code Llama-Instruct (doğal dil komutlarına yanıt verebilen sürüm). Bu modellerin 7B, 13B ve 34B parametrelili versiyonları mevcuttur (Touvron vd., 2023). Eğitim sürecinde 500 milyar token'a sahip kod odaklı bir veri kümesi kullanılmıştır. Bu veri kümesi Python, C++, Java gibi programlama dillerinin yanı sıra Markdown belgeleri ve açıklayıcı metinleri de içermektedir. Böylece model, hem kod

yazma hem de kodu açıklama görevlerinde yüksek başarı göstermektedir (Touvron vd., 2023), (AI Papers Academy, 2023). Code Llama, “fill-in-the-middle” (FIM) özelliğiyle kodun yalnızca baştan sona değil, ortasındaki eksik bölümleri de tamamlayabilir. Bu özellik özellikle 7B ve 13B varyantlarında optimize edilmiştir ve çoklu dosya ortamlarında kod bağlamını kavrayarak verimli çıktı üretimini destekler (Touvron vd., 2023).

Benchmark sonuçlarına göre Code Llama-Python 7B modeli, HumanEval testinde %67.6, MBPP testinde ise %65 düzeyinde başarı göstermiştir. Bu, modeli GPT-3.5 seviyesiyle karşılaştırılabilir hale getirmektedir (Touvron vd., 2023). Ek olarak, LLM'lerin sürdürülebilir yazılım geliştirme üzerindeki etkilerini araştıran bir çalışmada, Code Llama'nın enerji verimliliği analiz edilmiştir. Sonuçlar, bazı durumlarda modelin ürettiği kodların enerji tüketimi açısından insan yazımı kodlara yakın performans gösterdiğini ortaya koymuştur (Islam vd., 2025). Code Llama'nın "open-weight" olarak sunulması, akademik topluluk ve ticari geliştiriciler için önemli bir avantaj sağlar. MIT benzeri açık lisanslarla kullanıma sunulan model, şeffaflık ve yeniden üretilebilirlik ilkelerini destekler (Touvron vd., 2023). Genel olarak Code Llama, üretken yapay zekânın yazılım geliştiricilere sunduğu imkanları genişleten, teknik olarak güçlü ve açık yapısıyla topluluk temelli araştırma ve uygulamaları mümkün kılan bir modeldir.

1.2.6. ChatGPT

ChatGPT, OpenAI tarafından geliştirilen ve çeşitli doğal dil işleme (NLP) ve yazılım geliştirme görevlerinde yaygın olarak kullanılan, üretken yapay zekâ tabanlı bir büyük dil modelidir (LLM). İlk versiyonu 2022 yılında yayınlanan bu sistem, GPT-3.5 ve daha sonra GPT-4 tabanlı versiyonlarla güçlendirilmiştir. GPT-4 tabanlı ChatGPT, Mart 2023'te tanıtılmış ve geliştiricilere kod yazma, hata ayıklama, yorum ekleme ve test senaryosu oluşturma gibi çok sayıda yazılım odaklı işlev sunmaya başlamıştır (OpenAI, 2023).

ChatGPT, doğrudan doğal dil girdilerini alarak çeşitli programlama dillerinde çıktı üretebilen bir sistemdir. Python, JavaScript, C++, Java ve SQL gibi yaygın diller başta olmak üzere, yüzlerce dili desteklemektedir. Model, bağlam duyarlılığı ve başlangıçtan sona kod blokları oluşturabilme yeteneği ile geliştirici iş akışlarına çok yönlü katkı sunar

(Sobania, Briesch ve Rothlauf, 2023). OpenAI, ChatGPT'yi geliştirirken Reinforcement Learning from Human Feedback (RLHF) adlı eğitim yöntemini kullanmıştır. Bu yöntem, modelin insan benzeri çıktılar üretebilmesi için denetimli ve pekistirmeli öğrenmeyi birleştirir. Kod yazımı için bu, bağlam doğruluğu yüksek ve hatasız çıktılar anlamına gelir (OpenAI, 2023). 2023 yılında yapılan bir değerlendirmede, ChatGPT'nin yazılım mühendisliği görevlerinde kullanıldığında, geliştiricilerin verimliliğini %35 ila %55 oranında artırdığı gözlemlenmiştir. Özellikle junior geliştiriciler için öğrenme sürecini kolaylaştırdığı ve kodlama hatalarını azaltmaya yardımcı olduğu belirtilmiştir (Sobania, Briesch ve Rothlauf, 2023). Ayrıca, Stack Overflow verileriyle kıyaslandığında, ChatGPT tarafından verilen yanıtların %52'sinin geliştiriciler tarafından doğru ve kullanılabilir bulunduğu bildirilmektedir (Denny, de Abreu ve German, 2023).

Kodlama dışında ChatGPT, belge oluşturma, kod açıklaması yazma, teknik borçları analiz etme ve yazılım mimarisi tasarlama gibi görevlerde de kullanılmaktadır. Bunun yanı sıra, OpenAI tarafından sunulan "function calling" ve "code interpreter" özellikleri ile modelin API tabanlı sistemlerde fonksiyonel çağrılar yapabilmesi ve girdileri analiz edebilmesi sağlanır (OpenAI, 2023). Güncel olarak ChatGPT, OpenAI tarafından sürekli geliştirilmekte olup, GPT-4 Turbo gibi versiyonlarla daha hızlı ve çok yönlü çıktılar sunabilmektedir. Modelin "open-weight" olmaması, yani kapalı kaynaklı olarak dağıtılması, akademik kullanımda bazı sınırlar oluştursa da, API ve ChatGPT Pro abonelik modelleriyle geniş kitlelere erişim sağlamaktadır. GPT-4.1 modeli, OpenAI tarafından 14 Nisan 2025 tarihinde duyurulmuş olan, üretken yapay zekâ alanındaki en güncel ve gelişmiş dil modelidir. GPT-4 serisinin devamı niteliğinde olan bu sürüm, özellikle kapsamlı bağlamları anlama ve karmaşık görevleri yerine getirme konularında önemli iyileştirmeler sunmaktadır. En belirgin yeniliği 1 milyon token'a ulaşan geniş bağlam penceresidir. Bu özellik, önceki sürümlere kıyasla çok daha uzun metinleri tutarlı ve bağlamsal olarak derinlemesine analiz edebilme yeteneğini artırmıştır. Böylece, akademik, teknik ve hukuki metinlerdeki kapsamlı dokümanları işleme ve yorumlama performansı kayda değer ölçüde yükselmiştir. Modelin dikkat çeken diğer önemli yönü, kodlama yetkinliklerinin belirgin şekilde geliştirilmiş olmasıdır. Özellikle SWE-bench ve SWE-Lancer gibi kodlama performans testlerinde GPT-4.1, yazılım mühendisliği alanındaki görevlerde olağanüstü sonuçlar sergileyerek, geliştiricilerin üretkenliğini önemli ölçüde artırabilecek potansiyele sahip olduğunu kanıtlamıştır. Ayrıca, GPT-4.1

görsel veri işleme ve yorumlama konusunda da önemli ilerlemeler kaydetmiştir. MMMU (Multimodal Massive Multitask Understanding) ve MathVista gibi çok modlu benchmark testlerinde üstün sonuçlar göstererek, görsel ve metinsel bilgiyi entegre bir biçimde anlamlandırma yeteneğini artırmıştır.

1.3. Geleneksel Yazılım Geliştirme ile Karşılaştırmalar

Yazılım geliştirme süreçleri, tarihsel olarak belirli aşamalardan oluşur: gereksinim analizi, tasarım, kodlama, test ve bakım. Geleneksel yazılım geliştirme, bu süreçleri insan merkezli olarak ve çoğunlukla manuel yürütür. Üretken Yapay Zekâ (GenAI) tabanlı kod üretimi ise bu sürecin bazı bileşenlerini otomatikleştirerek geliştirici üzerindeki bilişsel yükü azaltmayı amaçlamaktadır (Pandey, Singh, Wei ve Shankar, 2024).

1.3.1. Kodlama Sürecinde Temel Farklılıklar

Geleneksel yazılım geliştirme süreçleri, yazılım mühendisliğinin temel prensiplerine dayanarak, belirli aşamalarla ilerler: gereksinim analizi, tasarım, kodlama, test ve bakım. Bu süreçlerde, geliştiriciler kodu manuel olarak yazar, test eder ve bakımını üstlenirler. Bu yaklaşım, geliştiricilerin yazılımın her yönünü derinlemesine anlamalarını sağlar, ancak zaman alıcı ve hata yapma olasılığı yüksek bir süreçtir (Pandey, Singh, Wei ve Shankar, 2024).

Üretken Yapay Zekâ (GenAI) tabanlı kod üretimi, bu geleneksel süreci daha kolay bir hale dönüştürmektedir. GenAI araçları, doğal dil girdilerini analiz ederek otomatik olarak kod önerileri sunar. Örneğin, GitHub Copilot geliştiricinin yazdığı bir fonksiyon açıklamasına dayanarak ilgili kodu otomatik olarak tamamlayabilir (Ziegler, Yasin ve Klocke, 2024). Bu, özellikle tekrarlayan kod parçalarının yazımında zaman tasarrufu sağlar ve geliştiricilerin daha karmaşık problemlere odaklanmalarına olanak tanır ancak, GenAI araçlarının sunduğu otomasyon, geliştiricilerin kodun işleyişini tam olarak anlamadan kullanmalarına neden olabilir. Bu durum, özellikle karmaşık sistemlerde, beklenmeyen hatalara yol açabilir. Ayrıca, GenAI tarafından üretilen kodun kalitesi ve güvenliği, kullanılan modelin eğitim verilerine ve algoritmalarına bağlıdır (Ziegler, Yasin ve Klocke, 2024).

Genel olarak GenAI tabanlı kod üretimi, geleneksel yazılım geliştirme süreçlerine kıyasla önemli avantajlar sunmaktadır. Ancak, bu araçların etkin ve güvenli bir şekilde kullanılması için geliştiricilerin dikkatli olmaları ve üretilen kodu titizlikle incelemeleri gerekmektedir (Pandey, Singh, Wei ve Shankar, 2024), (Ziegler, Yasin ve Klocke, 2024).

1.3.2. Kod Kalitesi ve Güvenlik

Kod kalitesi ve güvenlik, yazılım geliştirme süreçlerinin en önemli bileşenlerindendir. Geleneksel yöntemlerde geliştiriciler kodu manuel olarak yazar ve test ederler, bu da kodun kalitesini ve güvenliğini doğrudan etkiler. Ancak, insan hatası riski her zaman mevcuttur (Pearce, Ahmad, Tan ve Dolan-Gavitt, 2022). GenAI tabanlı araçlar kod üretim sürecini otomatikleştirerek bu riskleri azaltmayı hedefler. Ancak yapılan araştırmalar, GenAI tarafından üretilen kodların önemli bir kısmının güvenlik açıkları içerdiğini göstermektedir. Örneğin, GitHub Copilot tarafından üretilen kodların %40'ının güvenlik açısından sorunlu olduğu rapor edilmiştir (Pearce, Ahmad, Tan ve Dolan-Gavitt, 2022). Bu durum GenAI araçlarının eğitim verilerinin kalitesi ve çeşitliliği ile doğrudan ilişkilidir. Eğitim verilerinde bulunan güvenlik açıkları üretilen kodlara da yansır. Dolayısıyla, GenAI araçlarının önerdiği kodların güvenliğini değerlendirmek için geliştiricilerin ek testler yapmaları gerekmektedir (Yetiştiren, Özsoy, Ayerdem ve Tüzün, 2023). GenAI araçları kod üretim sürecini hızlandırırsa da, kodun kalitesi ve güvenliği konusunda dikkatli olunması gerekmektedir. Geliştiricilerin, üretilen kodları titizlikle incelemeleri ve gerekli güvenlik önlemlerini almaları önemlidir (Pearce, Ahmad, Tan ve Dolan-Gavitt, 2022), (Yetiştiren, Özsoy, Ayerdem ve Tüzün, 2023).

1.3.3. İş Akışı ve Rol Dağılımı

Yazılım geliştirme süreçlerinde iş akışı ve rol dağılımı projelerin etkinliği ve başarısı için kritik öneme sahiptir. Geleneksel yöntemlerde geliştiriciler kodu manuel olarak yazar, test eder ve bakımını üstlenirler. Bu süreçlerde, geliştiricilerin rollerinin net olması ve iş akışının iyi tanımlanması gerekmektedir (Ziegler, Yasin ve Klocke, 2024). GenAI tabanlı araçlar bu iş akışını daha kolay ve akıcı hale dönüştürmektedir. Geliştiriciler artık kodu manuel olarak yazmak yerine, GenAI araçların ürettiği ve önerdiği kodları değerlendirip entegre etmektedirler. Bu, geliştiricilerin rollerini yazılımcıdan daha çok bir "denetleyici"

veya "küratör"e dönüştürmektedir (Pandey, Singh, Wei ve Shankar, 2024). Bu değişim, iş akışında da önemli farklılıklara yol açmaktadır. Geleneksel yöntemlerde kodun yazımı, testi ve bakımı ayrı ayrı aşamalarda gerçekleştirilirken, GenAI araçları bu süreçleri entegre bir şekilde sunmaktadır. Bu, iş akışının daha dinamik ve esnek olmasını sağlar (Ziegler, Yasin ve Klocke, 2024).

Ancak, bu yeni iş akışı geliştiricilerin GenAI araçlarının önerilerini doğru değerlendirebilmeleri ve gerektiğinde müdahale edebilmeleri için yeni beceriler kazanmalarını gerektirir. Buna ek olarak, takım içi iletişim ve iş birliği de bu yeni iş akışına uyum sağlayacak şekilde yeniden yapılandırılmalıdır (Pandey, Singh, Wei ve Shankar, 2024). GenAI tabanlı araçlar yazılım geliştirme süreçlerinde iş akışı ve rol dağılımını da dönüştürmektedir. Bu dönüşüm, projelerin etkinliğini artırabilir, ancak başarılı bir şekilde uygulanabilmesi için dikkatli planlama ve eğitim gerekmektedir (Ziegler, Yasin ve Klocke, 2024), (Pandey, Singh, Wei ve Shankar, 2024).

1.3.4. Üretken Yapay Zekânın Geliştirici Kimliğine ve Becerilerine Etkisi

Yazılım geliştiricilerin mesleki rolü ve kimliği, uzun yıllar boyunca algoritma geliştirme, hata ayıklama, yapılandırılmış problem çözme gibi teknik yeterlilikler etrafında şekillenmiştir. Ancak üretken yapay zekâ (GenAI) sistemlerinin yaygınlaşması ile birlikte geliştirici kimliği önemli bir evrim geçirmektedir. Geliştiriciler artık yalnızca kod yazan bireyler olmaktan çıkmakta; bunun yerine, yapay zekâ tarafından üretilen kodları denetleyen, bağlama uygunluğunu değerlendiren ve sistem düzeyinde entegre eden bir role evrilmektedir. Bu değişim, yazılım geliştirici tanımının temelini oluşturan beceri setinde de bir kaymaya neden olmaktadır. Geleneksel yazılım geliştirme süreçlerinde derin algoritmik bilgi ve sentaks hâkimiyeti temel yetkinlikler arasında yer alırken, GenAI destekli geliştirme ortamlarında “prompt mühendisliği”, çıktı analizi ve model çıktılarının doğrulaması gibi daha üst düzey kavramsal beceriler ön plana çıkmaktadır. Bu durum, yazılım mühendisliğinin pedagojik yapısını da etkileyerek, eğitim kurumlarının müfredatlarını yeniden düşünmeye zorlamaktadır. Ayrıca, geliştiricinin motivasyon yapısında da değişiklikler gözlemlenmektedir. Geleneksel kod yazımı süreci geliştiriciye yaratıcı bir tatmin ve kontrol hissi sağlarken, GenAI destekli geliştirme bazen bu yaratıcı sürecin pasifleştirilmesine neden olabilmektedir. Bu bağlamda, “ajanlaşmış

geliştirici” kavramı gündeme gelmektedir. Geliştirici, bir araçtan ziyade yapay zekâyla etkileşim içinde olan bir karar vericiye dönüşmektedir. Bununla birlikte iş gücü piyasasında da farklılaşmalar yaşanması beklenmektedir. Kod üretiminin GenAI araçları tarafından büyük oranda devralındığı bir senaryoda, düşük ve orta düzey programlama işleri otomasyona açık hale gelirken, analitik düşünebilen, sistem mimarisi kurabilen ve yapay zekâ ile entegre çalışabilen bireyler daha değerli hale gelecektir. Dolayısıyla GenAI, sadece teknolojik değil sosyo-mesleki bir dönüşümün de habercisidir.

Geliştirici kimliği üzerindeki bu dönüşüm yeni etik, hukuki ve pedagojik tartışmaları da beraberinde getirmektedir. Örneğin, yapay zekânın ürettiği kodun sorumluluğu kimdedir? Geliştirici yanlış bir öneriyi fark etmeyip sistemin çökmesine neden olursa bu, insan hatası mı yoksa sistemik bir sorun mudur? Bu tür sorular, gelecekte mesleki kimliğin sadece teknik değil, aynı zamanda etik donanımla da şekillenmesi gerektiğini ortaya koymaktadır. Bu dönüşümün bir başka boyutu ise yazılım mühendislerinin üretken yapay zekâya dair gelecek kaygılarıyla ilgilidir. Geliştiricilerin bu değişimi nasıl algıladıkları, onların mesleki konumları ve gelecek planlamaları açısından belirleyici olmaktadır. Yazılım mühendisliği mesleği, tarihsel olarak yüksek uzmanlık ve teknik beceri gerektiren bir alan olarak konumlanmıştır. Ancak üretken yapay zekânın yükselişi, bu mesleği icra eden bireylerde geleceğe yönelik belirsizlikler ve tehdit algısı yaratmaktadır. Günümüzde pek çok geliştirici, GenAI teknolojilerinin mesleklerini tehdit edebileceğini düşünmekte ve bu doğrultuda çeşitli akademik ve sektörel anketlerde görüş bildirmektedir.

Örneğin, Evans Data Corporation tarafından geliştiricilerle yapılan bir ankette, katılımcıların %30'u yapay zekâ teknolojilerinin gelecekte yazılım geliştirme mesleğinin yerini alabileceğini düşündüğünü belirtmiştir (Evans Data Corporation, 2023). Bu görüşler yalnızca teknoloji sektörüyle sınırlı kalmamaktadır. Pew Research Center tarafından gerçekleştirilen daha geniş çaplı bir çalışmada, Amerikalıların %48'i yapay zekânın yazılım mühendisliği gibi yüksek beceri gerektiren meslekleri 20 yıl içinde tehdit edebileceğini düşündüğünü belirtmiştir (Anderson, 2025). Öte yandan, bu endişelerin evrensel düzeyde de hissedildiği görülmektedir. ADP Research Institute'un küresel ölçekte yaptığı bir ankette, katılımcıların %85'i yapay zekânın mesleklerine doğrudan etkide bulunacağını ifade etmiş, %42'si ise kendi rollerinin tamamen dönüşeceğine inandığını belirtmiştir (ADP Research Institute, 2023).

Bu anket bulguları, yazılım mühendislerinin yalnızca teknik değil, aynı zamanda psikolojik ve sosyolojik bir dönüşüm sürecinde olduklarını göstermektedir. Mesleki geleceklerine dair bu tür kaygılar sadece kariyer planlamalarını değil, aynı zamanda meslek etiği, iş güvenliği ve beceri dönüşümü konularındaki ihtiyaçları da derinleştirmektedir.

1.4. Yapay Zekâ Destekli Kod Üretiminin Avantajları ve Zorlukları

Yapay zekâ destekli kod üretimi (AI-assisted code generation), yazılım mühendisliğinde üretkenliği artıran ve tekrarlayan görevleri otomatikleştiren modern yaklaşımlar arasında önemli bir yer edinmiştir (Saravanan vd., 2025). Özellikle büyük dil modelleri (LLM – Large Language Models) sayesinde geliştirilen GitHub Copilot, ChatGPT, Amazon CodeWhisperer gibi araçlar, kod önerileri, otomatik tamamlama, test üretimi ve hata ayıklama gibi çeşitli işlevlerle yazılım geliştirme sürecine bütüncül katkılar sağlamaktadır (Khot, 2024).

1.4.1. Avantajlar

1.4.1.1. Üretkenlik ve zaman tasarrufu

Yapay zekâ destekli kod üretim araçları, geliştiricilere yüksek düzeyde zaman kazandıran otomasyon olanakları sunmaktadır. Özellikle sık tekrarlanan kod kalıplarının otomatik tamamlanması, geliştiricilerin daha yaratıcı ve stratejik görevler üzerinde yoğunlaşmasına imkân tanır. Örneğin, bir çalışmada ortalama kod tamamlama süresinin klasik yöntemlerle 6.7 dakika iken, yapay zekâ destekli sistemlerle 3.1 dakikaya düştüğü, hata düzeltme süresinin ise 8.9 dakikadan 4.2 dakikaya gerilediği gösterilmiştir (Saravanan vd., 2025). Ayrıca, bu tür araçlar sayesinde günlük kod üretimi satır sayısında ciddi artışlar gözlemlenmiş, bu da doğrudan yazılım geliştirme süresine olumlu etki etmiştir (Saravanan vd., 2025).

GitHub Copilot ve Codex gibi araçların sunduğu otomatik öneriler, rutin görevleri daha verimli hâle getirirken üretkenliği artırmakta ve toplam geliştirme sürecini %30–40 oranında hızlandırabilmektedir (Khot, 2024). Bu durum, özellikle sürekli teslimat (CI/CD) süreçlerinde zaman kazancı anlamına gelmektedir.

1.4.1.2. Kod kalitesi ve hataların azalması

Kod üretiminde karşılaşılan sözdizimsel ve mantıksal hatalar, yapay zekâ tabanlı sistemlerin gelişmiş hata tespit algoritmaları sayesinde önceden fark edilerek azaltılmaktadır. Defects4J gibi verisetleri üzerinde yapılan testlerde, AI destekli hata bulma doğruluğu %91.4 seviyesine kadar ulaşmıştır (Saravanan vd., 2025). Bunun yanı sıra AI sistemleri test senaryolarını da otomatik olarak oluşturarak test kapsamını artırmakta, manuel testlerin gözden kaçırabileceği kenar durumları tespit edebilmektedir (Khot, 2024). Örneğin EvoSuite %68, Randoop %61 test kapsamına ulaşırken, AI tabanlı sistemlerin %87'ye kadar eriştiği raporlanmıştır (Saravanan vd., 2025). Bu, yazılımın güvenilirliğini artırmakla kalmayıp bakım süresini de azaltmaktadır.

1.4.1.3. Öğrenme sürecine katkı ve yeni başlayanlar için destek

Yapay zekâ sistemlerinin sunduğu doğal dil girdisiyle kod üretme özelliği, özellikle programlamaya yeni başlayanlar için büyük kolaylık sağlamaktadır. Eğitim ortamlarında yapılan deneylerde, AI destekli araçları kullanan öğrencilerin laboratuvar sınavlarından %23 daha yüksek puan aldıkları görülmüştür (Saravanan vd., 2025). Bu araçlar kod yazma konusunda pratiklik sağlarken, aynı zamanda öğrencilerin yapısal kod örnekleriyle öğrenmesini kolaylaştırmakta ve “örnek-temelli öğrenme” (example-based learning) ortamı oluşturmaktadır. Ayrıca AI tarafından üretilen açıklamalar, öğrencilerin kodun mantığını anlamasına yardımcı olmaktadır (Dickhaut, Bruhin, Elshan ve Stocker, 2024).

1.4.1.4. Kod incelemesi ve refaktör desteği

Yapay zekâ sistemleri sadece yeni kod üretmekle kalmaz; mevcut kodların daha optimize, okunabilir ve sürdürülebilir hâle getirilmesini de sağlayabilir. Refaktör süreçlerinde yapay zekâ modelleri, eski kod yapıları üzerinde modernleştirme önerileri sunarak teknik borcun azaltılmasına katkı sağlar (Dickhaut, Bruhin, Elshan ve Stocker, 2024). Örneğin, Dickhaut ve meslektaşları tarafından yapılan bir araştırma, AI modellerinin kod yeniden düzenleme konusunda geleneksel yöntemlere kıyasla daha etkili öneriler sunduğunu göstermektedir (Dickhaut, Bruhin, Elshan ve Stocker, 2024).

Ayrıca, AI araçlarının açıklayıcı önerilerde bulunması sayesinde geliştiriciler kod değişikliklerinin gerekçelerini anlayabilmekte ve daha güvenle entegrasyon gerçekleştirebilmektedir (Saravanan vd., 2025). Bu açıklayıcılık, geliştirici güvenini %41 oranında artırmıştır (Saravanan vd., 2025).

1.4.2. Zorluklar

Yapay zekâ destekli kod üretimi, yazılım mühendisliğine önemli katkılar sunsa da, bu teknolojinin yaygın kullanımı bazı temel zorlukları da beraberinde getirmektedir. Bu zorluklar, güvenlik açıklarından etik ve hukuki sorunlara, model açıklanabilirliğinden geliştirici bağımlılığına kadar geniş bir yelpazede değerlendirilmelidir.

1.4.2.1. Güvenlik açıkları ve kod kalitesi sorunları

Yapay zekâ sistemleri tarafından üretilen kodlar, çoğu zaman sentaktik olarak doğru olsa da, güvenlik açısından zafiyetler içerebilmektedir. Özellikle giriş doğrulama, şifreleme, veri işleme gibi kritik işlemlerde, AI modellerinin ürettiği kodlarda SQL injection, XSS ve OS komut enjeksiyonu gibi açıkların varlığı gözlemlenmiştir (Siddique, Dey ve Takabi, 2023). Yapılan araştırmalarda, GitHub Copilot'un oluşturduğu kodların %27.25'inin güvenlik açığı içerdiği, bazı örneklerde bu oranın %44'e kadar çıkabildiği bildirilmiştir. Bu durum, eğitim verisinin kalitesi ve bağlam duyarlılığı eksikliğiyle ilişkilendirilmiştir (Improta vd., 2023).

1.4.2.2. Telif hakları ve etik sorunlar

Yapay zekâ modellerinin eğitiminde kullanılan açık kaynak kodlar, lisanslanmış veya telif hakkına tabi olabilir. AI sistemlerinin bu kodları yeniden üretmesi geliştirici haklarını ihlal etme riski taşımaktadır (Bommarito ve Katz, 2022). Bu bağlamda, üretken yapay zekâ çıktılarının mülkiyeti, lisans geçerliliği ve hukuki sorumluluğu belirsizdir. Ayrıca bazı çalışmalar, AI'nin ürettiği kodlarda önyargılı yapıların da yer alabildiğini, bu durumun sosyal adalet açısından sorunlu çıktılar doğurabileceğini göstermiştir.

1.4.2.3. Açıklanabilirlik ve güven problemleri

Generatif modellerin karar verme süreçleri genellikle “kara kutu” niteliğindedir. Geliştiriciler, yapay zekâ sistemlerinin neden belirli bir kodu ürettiğini anlamakta zorlanmaktadır (Saravanan vd., 2025). Açıklanabilir yapay zekâ (XAI) üzerine yapılan çalışmalar, özellikle yüksek güvenlik ve düzenlemeye tabi sektörlerde (ör. finans, sağlık) bu şeffaflığın olmazsa olmaz olduğunu vurgulamaktadır.

1.4.2.4. Aşırı bağımlılık ve beceri aşınması

AI destekli kod üretim araçlarına sürekli başvuran geliştiricilerin zamanla temel algoritmik düşünme, hata ayıklama ve yazılım mimarisi becerilerinde gerileme yaşadığı gözlemlenmiştir (Khot, 2024). Bu durum, uzun vadede yazılım mühendisliği mesleğinde “pasif kullanıcı” profili riskini gündeme getirmektedir. Ayrıca bu bağımlılık, geliştiricilerin hatalı veya güvenli olmayan kodları sorgulamadan kabul etmesine neden olabilir.

1.4.2.5. Regülasyon ve yasal uyumluluk sorunları

GDPR, DMCA ve AB AI Act gibi düzenlemeler, AI sistemlerinin nasıl eğitileceği, ne tür verilerle çalışacağı ve kullanıcı verilerinin nasıl korunacağı konusunda açık çerçeveler sunmaktadır (European Parliament, 2023). Ancak birçok yapay zekâ aracı, hâlihazırda bu düzenlemelere tam olarak uyumlu değildir. Bu da geliştiricileri ve şirketleri olası yasal yaptırımlarla karşı karşıya bırakmaktadır.

1.4.2.6. Toplumsal etkiler ve iş gücü dönüşümü

Rutin kod üretiminin otomatikleştirilmesi, bazı geliştirici rollerinin zamanla gereksizleşmesine neden olabilir. Bu durum, mesleki yetkinliklerin yeniden tanımlanması, eğitim sistemlerinin dönüşmesi ve yeni istihdam stratejilerinin geliştirilmesini gerektirir (Dickhaut, Bruhin, Elshan ve Stocker, 2024). Ayrıca, üretken yapay zekânın sosyal eşitsizlikleri derinleştirmemesi için etik rehberlik şarttır.

1.5. Akademik Çalışmaların Özeti

Üretken yapay zekâ destekli kod üretimi, yazılım mühendisliği literatüründe son yıllarda en çok araştırılan alanlardan biri hâline gelmiştir. GitHub Copilot, OpenAI Codex, ChatGPT, Amazon CodeWhisperer ve Meta'nın Code Llama gibi araçları, doğal dil girdilerinden programatik çıktılar üretme kabiliyeti sayesinde yazılım geliştirme döngüsünde verimlilik, kalite ve erişilebilirlik açılarından devrimsel yenilikler sunmuştur (Barke, Brown ve Pradel, 2025).

1.5.1. Verimlilik ve Üretkenlik Etkileri

Yapay zekâ destekli kod üretim araçlarının en temel katkısı yazılım geliştirme süreçlerinde zaman kazancı ve üretkenlik artışı sağlamasıdır. Microsoft ve GitHub tarafından yürütülen büyük ölçekli bir çalışmada, GitHub Copilot kullanıcılarının %55'inin daha hızlı kod yazdığını, %74'ünün kodlama sürecinde daha az mental efor sarf ettiğini ve %88'inin tekrarlı görevlerde önemli zaman tasarrufu sağladığını ifade ettiğini raporlanmıştır (GitHub, 2023). Benzer şekilde OpenAI Codex üzerine yapılan kontrollü deneylerde, belirli görevlerin %43 daha kısa sürede tamamlandığı gözlemlenmiştir (Chen vd., 2021). Bu araçlar, özellikle boilerplate kodlar (tekrar eden yapı kalıpları), API çağrıları ve fonksiyon tamamlama gibi düşük karmaşıklıkta görevlerde yazılım geliştiricilere hız kazandırmaktadır. Ancak karmaşık algoritmik problem çözme, sistem tasarımı ve güvenlik mantığı gibi alanlarda hâlen insan müdahalesi kritik önem taşımaktadır (Yetiştiren, Özsoy, Ayerdem ve Tüzün, 2023).

Literatürdeki deneysel çalışmalarda, Copilot'un ortalama görev tamamlama süresini %20 ila %45 oranında kısalttığı ve kod yazım hacmini artırdığı gösterilmiştir (Liu vd., 2023). Ayrıca bu sistemlerin sunduğu sürekli öneri mekanizmaları, geliştiricilerin “akış hâli”ne (flow state) daha kolay geçmesini desteklemektedir (Vaithilingam, Brant, Xu ve Ray, 2022).

1.5.2. Kod Kalitesi ve Hata Oranı

Üretken yapay zekânın çıktılarının doğruluk ve kalite düzeyine dair bulgular, göreve ve bağlama bağlı olarak değişkenlik göstermektedir. HumanEval, MBPP (Mostly Basic Python Problems) ve APPS (Automated Programming Progress Suite) gibi benchmark

veri setleri kullanılarak yapılan deneylerde, GPT-4 tabanlı modellerin %67'ye kadar doğru çözüm üretebildiği raporlanmıştır (OpenAI, 2023). Ancak, bu oranlar karmaşıklık düzeyi arttıkça belirgin biçimde düşmektedir.

Ayrıca bazı çalışmalar, modellerin zaman zaman “hallucination” olarak tanımlanan, sentaktik olarak doğru ama semantik olarak anlamsız ya da işlevsiz çıktılar ürettiğini ortaya koymuştur (Carlini vd., 2021). Bu çıktılar çoğu zaman hata mesajı üretmez ancak uygulama mantığı açısından hatalıdır ve üretim ortamında ciddi sorunlara yol açabilir. Kod kalitesi, yalnızca doğruluk değil aynı zamanda okunabilirlik, sürdürülebilirlik ve test edilebilirlik gibi yönlerle de değerlendirilmelidir. AI ile yazılmış kodların çoğu durumda geleneksel yazılım mühendisliği standartlarını karşıladığı ancak yorum satırı ekleme, değişken isimlendirme ve hata yönetimi gibi konularda hâlen sınırlı olduğu vurgulanmıştır (Ye vd., 2023).

1.5.3. Güvenlik Açıkları ve Güvenilirlik

Üretken yapay zekâ sistemlerinin eğitiminde kullanılan büyük kod havuzları, güvenli olmayan kod örneklerini de içermektedir. Bunun sonucu olarak, modellerin önerdiği kodlarda ciddi güvenlik açıkları oluşabilmektedir. Pearce ve meslektaşlarının yaptığı OWASP Top 10'a dayalı analizinde, Copilot'un önerdiği örneklerin %40'ının kritik güvenlik açığı barındırdığı bulunmuştur (Pearce, Ahmad, Tan ve Dolan-Gavitt, 2022). Bu açıklar genellikle veri doğrulama eksiklikleri, doğrudan kullanıcı girişine dayalı sistem çağrıları ve statik parola kullanımına dayalıdır. Ayrıca bazı sistemlerin istemeden lisans ihlali yaparak dış kütüphaneleri önerdiği de gözlemlenmiştir (Balunovic, Brent ve Livshits, 2023). Güvenlik açısından en önemli zorluk, modellerin güvenli olmayan kalıpları fark edememesi veya bağlamdan kopuk yorum üretmesidir. Bu nedenle AI çıktılarının üretim ortamında doğrudan kullanılmadan önce manuel denetime tabi tutulması önerilmektedir (Vondrick ve Sharma, 2023).

1.5.4. Eğitim Ortamında Kullanımı

Eğitim ortamlarında üretken yapay zekâ sistemlerinin rolü de son derece tartışmalıdır. Giriş seviyesi programlama derslerinde ChatGPT, Copilot ve Codex gibi araçların

öğrencilerin motivasyonunu artırdığı, örnek temelli öğrenmeyi desteklediği ve yazma süresini kısalttığı belirlenmiştir (Denny ve Luxton-Reilly, 2023).

Ancak bazı çalışmalar bu araçların uzun vadede öğrencilerin kavramsal düşünme becerilerini geliştirmek yerine sadece “yanıt kopyalama” eğilimini tetiklediğini, özellikle sınav veya proje ortamlarında bilişsel tembellik doğurduğunu göstermektedir (Zou vd., 2023). Bu nedenle AI araçlarının eğitimde rehberli, açıklamalı ve yapılandırılmış şekilde kullanılması önerilmektedir. Ayrıca yapay zekânın verdiği çıktının doğru olduğunu sanan öğrenciler, farkında olmadan hatalı kod kalıplarını benimseyebilmektedir. Bu da akademik dürüstlük, kaynak gösterimi ve özgünlük açısından yeni değerlendirme stratejilerinin geliştirilmesini gerektirir (Dickhaut, Bruhin, Elshan ve Stocker, 2024).

1.5.5. Açıklanabilirlik, Güven ve Geliştirici Denetimi

AI çıktılarının geliştiriciler tarafından anlaşılabilir olması, güven oluşması için kritik önemdedir. Ancak çoğu üretken model karar verme sürecine dair bir açıklama sunmaz. Örneğin ChatGPT veya Codex’un neden belirli bir kodu ürettiğini geliştirici çoğu zaman bilemez. Bu “kara kutu” doğası, özellikle güvenlik, tıp, finans gibi yüksek sorumluluk taşıyan alanlarda risklidir (Peng, Kalliamvakou, Cihon ve Demirer, 2023).

Bu sebeple açıklanabilir yapay zekâ (XAI) alanında model içi açıklamalar, etki analizi, kaynak kökeni izleme gibi yöntemler geliştirilmektedir. Ancak bu yöntemlerin pratik uygulaması hâlen yaygınlaşmamıştır (Song, Agarwal ve Wen, 2024).

1.5.6. Etik, Lisans ve Hukuki Belirsizlikler

Yapay zekâ modelleri, milyonlarca açık kaynak kod örneği ile eğitilmektedir. Ancak bu örneklerin çoğunun telif durumu belirsizdir. Bu durum, AI tarafından üretilen kodların yeniden kullanımı veya ticarileştirilmesi halinde ciddi hukuki sorunlara yol açabilmektedir. Telif hakkı, “türev eser” kavramı, lisans türlerinin (MIT, GPL, Apache) etkisi gibi başlıklar, AI çıktılarının yasal statüsünü netleştirmek için hâlen akademik ve hukuki olarak tartışılmaktadır (Bommarito ve Katz, 2023). GitHub Copilot’a karşı açılan sınıf davası, bu tartışmaların ne kadar ciddi boyutta olduğunu göstermektedir.

Etik açıdan ise AI modellerinin önyargılı, ayrımcı ya da güvenlik riski taşıyan kalıpları yeniden üretme potansiyeli önemli bir sorun olarak karşımıza çıkmaktadır. Model açıklanabilirliği, çıktı izlenebilirliği ve kaynak şeffaflığı gibi unsurlar, etik yapay zekâ için zorunlu olması gereken kriterlerdir (Wang vd., 2023).



İKİNCİ BÖLÜM: METODOLOJİ

Bu bölümde, Üretken Yapay Zekâ (GenAI) tabanlı kod üretim araçlarının yazılım geliştirme süreçlerine etkisini araştırmak amacıyla izlenen yöntemsel çerçeve ayrıntılı biçimde sunulmaktadır. Araştırmanın temel hedefi; GitHub Copilot, ChatGPT ve benzeri GenAI destekli araçların geliştirici performansı, kod kalitesi ve üretkenlik üzerindeki etkilerini bilimsel olarak değerlendirmektir.

Bu doğrultuda çalışma hem nicel hem de nitel verilerin birlikte analiz edildiği karma yöntemli (mixed-method) bir yaklaşımı benimsemektedir. Nicel kısımda, deneysel tasarım yoluyla GenAI araçlarının görev tamamlama süresi, hata oranı ve kod doğruluğu gibi ölçülebilir çıktılar üzerindeki etkisi değerlendirilmiştir. Nitel bileşende ise gerçek dünya senaryolarını yansıtan vaka analizleri aracılığıyla geliştirici deneyimleri, araçlara yönelik tutumlar ve yazılım geliştirme pratiklerindeki değişimler analiz edilmiştir. Bölümde öncelikle araştırmanın deneysel düzeni açıklanmakta, ardından vaka analizleri ile elde edilen bulgular aktarılmaktadır. Kullanılan veri kaynakları, araçlar, deneysel ortamın kurulumu ve test-değerlendirme kriterleri ayrıntılı biçimde ele alınarak araştırma sürecinin güvenilirliği ve tekrarlanabilirliği sağlanmıştır. Bu metodolojik yapı, Üretken Yapay Zekâ (GenAI) araçlarının yazılım mühendisliği üzerindeki etkilerini yalnızca teknik ölçütlerle değil, aynı zamanda kullanıcı deneyimi, bilişsel yük ve iş akışı dönüşümleri gibi çok boyutlu göstergeler üzerinden bütünsel bir yaklaşımla değerlendirmeyi amaçlamaktadır.

2.1. Araştırma Yöntemi

Bu çalışmada üretken yapay zekâ (GenAI) tabanlı kod üretim araçlarının yazılım geliştirme süreçlerine olan etkisini ölçmek amacıyla deneysel ve gözlemsel yöntemlerin bir arada kullanıldığı karma yöntemli bir araştırma yaklaşımı benimsenmiştir. Bu çerçevede kontrollü deneyler yoluyla geliştirici performansı üzerine nicel veriler toplanmış, vaka analizleriyle ise gerçek dünyadaki uygulamalardan nitel içgörüler elde edilmiştir.

2.1.1. Deneyisel Çalışmalar

Kontrollü deneyisel çalışmalar, GenAI araçlarının yazılım geliştirme sürecine olan etkilerini doğrudan gözlemlemek açısından en güçlü araştırma yöntemlerinden biridir. Stanford Üniversitesi bünyesinde Peng, Kalliamvakou, Cihon ve Demirer tarafından 2023 yılında yürütölen kapsamlı bir deneyisel çalışma, GitHub Copilot gibi Üretken Yapay Zekâ (GenAI) araçlarının yazılım geliştiricilerin üretkenliğine etkisini nicel verilerle değerlendirmeyi amaçlamıştır (Peng, Kalliamvakou, Cihon ve Demirer, 2023). Bu çalışma, literatürde GenAI tabanlı kod öneri sistemlerinin etkisinin doğrudan, deneyisel olarak ölçüldüğü az sayıdaki sistematik araştırmadan biridir. Araştırma kapsamında toplam 95 profesyonel geliştirici gönüllü olarak yer almıştır. Katılımcılar rastgele iki gruba ayrılmıştır: deney grubu (Copilot kullananlar) ve kontrol grubu (Copilot kullanmayanlar). Her iki grup da aynı programlama görevini yerine getirmiştir: JavaScript programlama dili kullanarak bir HTTP sunucusu oluşturmak. Bu görev seçimi hem gerçek dünya senaryolarına yakınlığı hem de belirli düzeyde mühendislik becerisi gerektirmesi açısından önemlidir. Deneyin sonucunda, GitHub Copilot kullanan katılımcıların görevi tamamlamak için ortalama 161 dakika harcadığı, kontrol grubunun ise ortalama 225 dakikada tamamladığı rapor edilmiştir. Bu, Copilot kullanan grubun görevi yaklaşık %28 daha kısa sürede tamamladığını ortaya koymaktadır (Peng, Kalliamvakou, Cihon ve Demirer, 2023). Bu farklılık, yalnızca üretkenlik artışını değil, aynı zamanda geliştiricilerin zihinsel yükünü azaltan, hızlı çözüm üretme potansiyeline sahip sistemlerin yazılım mühendisliği süreçlerinde nasıl dönüşüm yaratabileceğini göstermektedir.

Ayrıca başarı oranları açısından da dikkat çekici veriler elde edilmiştir. Copilot kullanan grubun %78.6'sı görevi başarıyla tamamlarken, bu oran Copilot kullanmayan grupta %69.2'de kalmıştır. Bu durum, GenAI araçlarının yalnızca süreci hızlandırmakla kalmayıp, aynı zamanda geliştiricilerin çözüm üretme kapasitesini de artırabileceğini göstermektedir (Peng, Kalliamvakou, Cihon ve Demirer, 2023).

Araştırma kapsamında yalnızca nicel ölçümler değil, aynı zamanda kullanıcı memnuniyeti üzerine de veriler toplanmıştır. Katılımcıların %95'inden fazlası, Copilot kullanımını genel anlamda olumlu bulmuş, önerilerin faydalı olduğunu belirtmiştir. Öne çıkan avantajlar arasında kod tamamlama kolaylığı, alternatif çözüm yollarının sunulması

ve daha az arama yapma ihtiyacı sayılmıştır. Bununla birlikte, bazı kullanıcılar tarafından belirtilen dezavantajlar arasında, önerilen kodların her zaman doğru ya da bağlama uygun olmaması, güvenlik açısından sorgulanabilir bazı blokların sunulması ve kodun açıklanabilirliğinin düşük olması yer almıştır (Peng, Kalliamvakou, Cihon ve Demirer, 2023). Çalışma, veri toplama ve analiz süreçlerinde yüksek metodolojik titizlik göstermiştir. Deney öncesi katılımcıların beceri seviyeleri eşleştirilmiş, görev öncesi bilgilendirme ve görev sonrası değerlendirme standartlaştırılmıştır. Ayrıca, tüm görevler aynı donanım ve yazılım ortamında gerçekleştirilmiştir. Bu faktörler, deneysel bulguların geçerliliğini artıran önemli unsurlardır.

Bu çalışmanın sonuçları, GenAI araçlarının özellikle deneyimli geliştiriciler tarafından kullanıldığında, üretkenliği artırabileceğini somut olarak göstermektedir. Ancak aynı zamanda bu tür araçların kod kalitesi, güvenlik ve açıklanabilirlik gibi boyutlarda sürekli insan denetimi gerektirdiği de açıktır. Araştırma, bu nedenle GenAI araçlarının “yardımcı karar destek sistemleri” olarak değerlendirilmeleri gerektiği sonucuna ulaşmaktadır (Peng, Kalliamvakou, Cihon ve Demirer, 2023). Bu bulguların daha ayrıntılı bir görünümü, farklı geliştirici grupları üzerindeki etkileri karşılaştıran heterojen tedavi etkisi analizinde ortaya konmuştur. Yapılan regresyon analizinde programlama deneyimi, günlük kodlama süresi, yaş grubu, gelir düzeyi, eğitim geçmişi ve tercih edilen programlama dili gibi değişkenlerin Copilot kullanımından elde edilen faydayı anlamlı düzeyde etkileyebildiği gözlemlenmiştir.

Bu tablo, GitHub Copilot kullanımının geliştirici performansına etkisini değerlendiren regresyon analizinden elde edilen heterojen tedavi etkisi tahminlerini göstermektedir. Bulgular şu şekildedir:

- Daha az programlama deneyimi olan geliştiriciler Copilot’tan daha fazla fayda görme eğilimindedir ($t = 1.90$, $p = 0.063$); bu sonuç istatistiksel olarak anlamlılık sınırında yer almaktadır.
- Günlük daha fazla saat kodlama yapanlar, Copilot kullanımından daha az marjinal fayda elde etmektedir. Bu fark istatistiksel olarak anlamlıdır ($p = 0.0168$).

- 25–44 yaş aralığındaki geliştiriciler, Copilot’tan anlamlı düzeyde farklı etkilenmektedir ($p = 0.0303$). Bu, yaştan tedavi etkisinde bir değişirici (moderator) olabileceğini göstermektedir.
- Diğer değişkenler (işsizlik durumu, düşük gelir, üniversite eğitimi, dil tercihi) istatistiksel olarak anlamlı farklılıklar göstermemiştir ($p > 0.05$).

Söz konusu farklılıkların detaylı sonuçları tablolar bölümünde sunulan Tablo 1’de gösterilmektedir.

2.1.2. Vaka Analizi

Vaka analizi, GenAI araçlarının gerçek dünyada nasıl kullanıldığını gözlemlemek için başvuru güçlü bir nitel yöntemdir. Stray ve arkadaşlarının yürüttüğü 2024 tarihli çalışmada, büyük bir kurumsal yazılım geliştirme ortamında 49 programlama oturumu gözlemlenmiş ve Copilot ile ChatGPT gibi araçların bireysel programlamada sağladığı destek incelenmiştir. Katılımcıların %71’i, GenAI araçlarının karmaşık sorun çözme süreçlerinde faydalı olduğunu belirtmiştir. Ancak, çift programlamada (pair programming) kullanılan bu araçların insan iletişimini zayıflattığı ve ortak problem çözme verimini düşürdüğü de rapor edilmiştir (Stray, Moe, Ganeshan ve Kobbenes, 2024).

Bu çalışmada gözlemlenen önemli bir unsur, geliştiricilerin GenAI tarafından sunulan önerileri doğrudan kabul etmek yerine genellikle modifiye ederek kullandıklarıdır. Katılımcıların sadece %29’u, önerilen kod parçalarını olduğu gibi entegre ederken geri kalanlar önerileri düzenleyerek uygulamayı tercih etmiştir. Bu bulgu, GenAI araçlarının insan-yapay zekâ iş birliğine dayalı bir modelde daha etkili çalıştığını ortaya koymaktadır (Stray, Moe, Ganeshan ve Kobbenes, 2024).

Ayrıca Udoidiok ve ekibinin 2024 tarihli vaka çalışması, farklı büyüklüklerdeki yazılım firmalarında GenAI entegrasyonunu incelemiştir. Çalışma, geliştirme ekiplerinin tekrarlayan görevleri otomatikleştirmek için Copilot’u sıklıkla kullandığını, ancak hata ayıklama ve güvenlik kontrolleri gibi kritik görevlerde hâlâ insan müdahalesine ihtiyaç duyulduğunu göstermiştir. Bazı durumlarda GenAI ’nin önerdiği kodlar güvenlik açıkları veya uyumsuzluklar içerdiği için doğrudan entegre edilememiştir (Udoidiok, Reza ve Zhang, 2024).

Bu çalışmada ayrıca GenAI entegrasyonunun takım dinamikleri üzerindeki etkisi de gözlemlenmiştir. GenAI destekli geliştiriciler, karar verme süreçlerinde daha bağımsız davranmış ve daha az dış destek talep etmiştir. Bu durum, özellikle junior seviyesindeki geliştiricilerin gelişim fırsatlarını sınırlayabileceği yönünde eleştirilere neden olmuştur (Udoidiok, Reza ve Zhang, 2024).

2.1.3. Yöntemsel Bütünlük ve Eleştirel Değerlendirme

Üretken Yapay Zekâ (GenAI) araçlarının yazılım mühendisliği üzerindeki etkisini değerlendiren araştırmalarda kullanılan yöntemlerin geçerliliği, elde edilen sonuçların yorumlanması açısından kritik öneme sahiptir. Özellikle son yıllarda GitHub Copilot, ChatGPT ve benzeri araçların programlama süreçlerine entegrasyonu üzerine yapılan çalışmalar incelendiğinde, araştırma tasarımlarının güçlü yönlerinin yanı sıra belirli metodolojik sınırlılıklarının da bulunduğu görülmektedir.

Deneysel çalışmalar, özellikle iç geçerlilik (internal validity) bakımından önemli avantajlar sunmaktadır. Katılımcılar üzerinde kontrol sağlanabilmesi, görevlerin standardize edilmesi ve değişkenlerin izole biçimde test edilebilmesi gibi nedenlerle deneysel tasarımlar, neden-sonuç ilişkilerini belirleme açısından oldukça güçlüdür. Örneğin, Stanford Üniversitesi'ne bağlı araştırmacılar tarafından yürütülen deneysel çalışmada (Peng, Kalliamvakou, Cihon ve Demirer, 2023), GitHub Copilot'un görev tamamlama süresine olan etkisi net biçimde ölçülmüştür. Ancak bu tür çalışmalar genellikle laboratuvar ortamında, kısa vadeli görevlerle ve kısıtlı örneklem büyüklüğüyle gerçekleştirilir. Bu durum, çalışmanın dış geçerliliği (external validity) açısından bazı soru işaretleri doğurur. Laboratuvar ortamlarında elde edilen sonuçların gerçek dünyadaki yazılım geliştirme ortamlarına ne derece genellenebileceği, sıklıkla eleştirilen bir noktadır. Geliştiricilerin zaman baskısı, ekip içi dinamikler, kurumsal kod standartları, versiyonlama sistemleri ve müşteri talepleri gibi karmaşık faktörlerin etkisi altında çalıştıkları düşünüldüğünde, laboratuvar bulgularının bu karmaşıklığı yeterince temsil etmediği açıktır. Bu bağlamda, Stray ve arkadaşlarının (Stray, Moe, Ganeshan ve Kobbenes, 2024) yürüttüğü türden vaka analizleri, deneysel bulgulara bağlamsal derinlik kazandırmak açısından büyük önem taşımaktadır. Gerçek ortamda, doğal akış içinde toplanan bu tür nitel veriler, GenAI araçlarının sadece üretkenlik değil, iş birliği, öğrenme ve karar verme üzerindeki etkilerini de ortaya koyabilmektedir.

Ayrıca, araştırmalar GenAI araçlarının her zaman sadece olumlu çıktılar üretmediğini de göstermektedir. Song ve arkadaşlarının (Song, Agarwal ve Wen, 2024) açık kaynaklı projeler üzerine yürüttüğü büyük ölçekli analizde, Copilot'un katkı sağladığı commit'lerin bazı durumlarda daha fazla hata ve revizyon içerdiği tespit edilmiştir. Aynı zamanda önerilen kod bloklarının tekrar eden yapılar içerdiği, özgünlük düzeyinin düşük olduğu ve bazı durumlarda okunabilirliğin azaldığı da rapor edilmiştir. Bu bulgular, GenAI araçlarının yalnızca hız kazandıran sistemler olarak değil, aynı zamanda kalite ve güvenlik riskleri taşıyan otomasyon araçları olarak da değerlendirilmesini gerektirir. Bu noktada, öneri sistemlerinin “otonom kod üreticiler” olmaktan çok, “öneri destekli insan merkezli sistemler” olarak görülmesi daha doğru olacaktır.

Eleştirel bir perspektifle değerlendirildiğinde, araştırmaların yalnızca deneysel ya da yalnızca nitel verilere dayalı olarak yürütülmesi metodolojik anlamda eksik bir resim sunabilir. Bu nedenle, karma yöntemli (mixed-method) bir yaklaşımın benimsenmesi hem istatistiksel performans verilerinin hem de bağlamsal kullanıcı deneyimlerinin eş zamanlı olarak analiz edilmesini mümkün kılar. Örneğin, deneysel olarak görev tamamlama süresinde kısalma saptansa bile, geliştirici anketlerinden elde edilecek memnuniyet düzeyi, stres seviyesi ya da açıklanabilirlik algısı gibi veriler, bu etkinin nasıl algılandığını ve içselleştirildiğini gösterir. GenAI araçlarının yazılım geliştirme üzerindeki etkisini sağlıklı bir biçimde anlayabilmek için hem kontrollü deneysel bulgulara hem de uzun vadeli gözlem ve vaka çalışmalarına ihtiyaç vardır. Yalnızca nicel verilere dayanmak, geliştiricilerin günlük deneyimlerini ve bağlam içi etkileşimlerini göz ardı etme riski taşır. Tersine, yalnızca nitel analizlere dayanmak da genellenebilirlik ve ölçülebilirlik açısından zayıf sonuçlara yol açabilir. Bu sebeple, bu iki yöntemin birbirini tamamlayıcı biçimde kullanılması, GenAI teknolojilerinin yazılım mühendisliğinde nasıl bir dönüşüm yarattığını daha net ve çok boyutlu biçimde ortaya koyar.

2.2. Kullanılan Veri Kaynakları

Üretken Yapay Zekâ (GenAI) tabanlı kod üretim araçlarının performansını nesnel şekilde değerlendirmek amacıyla yapılan araştırmalarda kullanılan veri kaynakları, çalışmanın güvenilirliği ve geçerliliği açısından kritik öneme sahiptir. Bu bağlamda, araştırmalarda hem kontrollü deneysel ortamlarda hazırlanmış benchmark veri kümeleri hem de gerçek dünya yazılım projelerinden elde edilen açık kaynaklı kod havuzları yaygın biçimde

kullanılmaktadır. Literatürde özellikle GitHub, Stack Overflow, LeetCode gibi platformlardan elde edilen veriler GenAI araçlarının önerdiği kodların doğruluğu, geçerliliği ve pratikteki uygulanabilirliği açısından kapsamlı analiz imkânı sunmaktadır. Bu platformlar, yalnızca geniş kod çeşitliliği değil, aynı zamanda gerçek geliştirici davranışlarını ve çözüm yaklaşımlarını da içerdikleri için, test ortamlarını doğal hale getirir.

2.2.1. GitHub Projeleri

GitHub hem eğitim amaçlı hem de endüstriyel düzeyde geliştirilen milyonlarca açık kaynak yazılım projesine ev sahipliği yapmaktadır. Bu özelliğiyle GitHub, Üretken Yapay Zekâ (GenAI) araçlarının yazılım geliştirme süreçlerine olan etkilerini gözlemlemek ve değerlendirmek için en sık başvurulmuş, güvenilir ve kapsamlı veri kaynaklarından biri hâline gelmiştir. Gerçek kullanıcı davranışlarını içeren doğal kod ortamları, GenAI araçlarının pratikte ne derece etkili olduğunu test etmek açısından önemli avantajlar sunmaktadır. Son yıllarda yapılan birçok çalışmada, GitHub üzerindeki projelerden elde edilen veriler aracılığıyla GenAI araçlarının yazılım üretim süreçlerine olan katkıları analiz edilmiştir. Peng ve arkadaşları (2023) tarafından yürütülen deneysel bir çalışmada, GitHub Copilot'un geliştirici verimliliği üzerindeki etkisi ölçülmüştür. Katılımcılardan, JavaScript dilinde bir HTTP sunucusu geliştirmeleri istenmiş ve Copilot kullanan grup, görevi Copilot kullanmayan gruba göre %55.8 oranında daha kısa sürede tamamlamıştır. Bu fark, üretken yapay zekâ araçlarının özellikle kod iskeleti oluşturma ve tekrarlı görevlerde zaman kazandırıcı bir rol oynadığını ortaya koymuştur (Peng, Kalliamvakou, Cihon ve Demirer, 2023).

Benzer şekilde Song ve arkadaşları (2024), GitHub üzerinde gerçekleştirilen geniş kapsamlı bir analizde, Copilot'un katkı sağladığı açık kaynak projeleri incelemiştir. Çalışma, Copilot kullanılan projelerde bireysel geliştirici üretkenliğinde %5.5'lik bir artış ve toplam katkı sayısında %5.4'lük bir yükselme tespit etmiştir. Bu bulgular, GenAI araçlarının sadece bireysel değil, topluluk bazlı katkıları da teşvik edebileceğine işaret etmektedir. Ancak aynı zamanda, Copilot entegrasyonu yapılan projelerde ortalama entegrasyon süresinin %41.6 oranında uzadığı belirlenmiştir. Bu durum, GenAI araçlarının sağladığı verimlilik artışının yanında ekip koordinasyonu ve kontrol

süreçlerinde ek maliyetler yaratabileceğini göstermektedir (Song, Agarwal ve Wen, 2024).

Yine bu kapsamda yürütülen bir başka çalışmada, Copilot'un Pull Request (PR) süreçlerine olan etkisi incelenmiştir. Analizler, Copilot'un PR açıklamalarını otomatikleştirme yeteneğinin, inceleme süresini kısalttığını ve PR'ların kabul edilme oranlarını artırdığını göstermektedir. Ancak bazı durumlarda, otomatik açıklamaların yetersiz veya bağlama uygun olmaması nedeniyle, manuel müdahale ihtiyacı doğmuştur. 146 GitHub deposundan toplanan 18.256 PR'ı analiz ederek, Copilot for PRs kullanımının kod inceleme süreçlerine etkisini değerlendirmiştir. Bulgular, Copilot for PRs ile desteklenen PR'ların, manuel olarak yazılanlara kıyasla ortalama 19.3 saat daha kısa sürede incelendiğini ve bu PR'ların birleştirilme olasılığının 1.57 kat daha yüksek olduğunu göstermektedir (Xiao, Hata, Treude ve Matsumoto, 2024).

Kendi gözlemlerime göre de GitHub'daki popüler açık kaynak projelerin kod tabanları üzerinde yapılan incelemeler, GenAI araçlarının özellikle frontend geliştirme, test yazımı ve belge üretimi gibi alanlarda daha yoğun şekilde kullanıldığını göstermektedir. Bu kullanım biçimi, geliştiricilerin GenAI'yi henüz çekirdek mimari kararlarında değil; daha çok zaman alıcı, destekleyici işlerde tercih ettiğini göstermektedir. Bu da araçların "yardımcı mühendis" rolü üstlendiği bir kullanım profilini doğrular niteliktedir. Bununla birlikte, Copilot'un önerilerinin zaman zaman lisanslama sorunları ya da güvenlik açıkları içerdiği bazı çalışmalarda dile getirilmiştir. Özellikle açık kaynak projelerde telif hakkı uyumu ve önerilen kodun orijinallliği konuları dikkatle izlenmesi gereken alanlardır. Nitekim birkaç projede, Copilot'un önerdiği kod parçalarının mevcut GPL lisanslı projelerle benzerlik gösterdiği ve bu durumun lisans ihlali riskini gündeme getirdiği bildirilmiştir. Bu da veri kaynaklarının seçimi kadar, kullanım sonuçlarının etik ve hukuki boyutlarının da araştırmalarda dikkate alınmasını gerekli kılmaktadır.

2.2.2. HumanEval

HumanEval, OpenAI tarafından geliştirilen ve üretken yapay zekâ (GenAI) modellerinin doğrudan kod üretme kapasitesini ölçmek amacıyla oluşturulmuş en yaygın benchmark veri kümelerinden biridir. Veri seti, doğal dilde verilen açıklamalara karşılık olarak Python dilinde fonksiyon üretimini gerektiren 164 kısa programlama problemi

içermektedir. Her bir problemde modelin, açıkça belirtilmiş bir işlevi yerine getiren, sözdizimsel olarak doğru ve mantıksal olarak çalışan bir Python fonksiyonu üretmesi beklenir.

HumanEval'in temel avantajı, klasik input-output testlerinden farklı olarak, doğal dil anlama, mantıksal akıl yürütme ve programlama yapılarının uygulanabilirliğini birlikte test etmesidir. Bu yönüyle benchmark, üretken modellerin yalnızca dil üretim yeteneğini değil, yazılım mühendisliği problemlerine fonksiyonel ve mantıksal yanıt üretme becerisini de sınamaktadır. HumanEval değerlendirmelerinde en sık kullanılan metrik $pass@k$ olarak bilinir. Bu metrik, modelin aynı problemi k farklı denemede çözme olasılığını ölçer. Örneğin $pass@1$, ilk denemede doğru cevabı üretme oranını verir; $pass@5$, beş denemeden en az birinde doğru çözüme ulaşma ihtimalini ölçer. OpenAI tarafından Codex modeliyle yapılan değerlendirmelerde *Codex-12B* modelinin $pass@1$ oranı %28.8 olarak rapor edilmiştir. Daha sonra GPT-4 gibi daha gelişmiş modellerde bu oranın %67'ye kadar çıktığı bildirilmiştir (Chen vd., 2021).

Her problem için çözüm doğruluğu, otomatik Python test fonksiyonları (ör. `assert`) aracılığıyla değerlendirilir. Bu yaklaşım sayesinde insan hakemliğine gerek kalmadan geniş çaplı testler gerçekleştirilebilir. Modelin yalnızca hata vermeyen değil, fonksiyonel olarak doğru çalışan kod üretmesi beklenir. HumanEval, üretken yapay zekâ modellerinin kıyaslamasında standart hâline gelmiştir. OpenAI Codex çalışması, bu benchmark'ı referans olarak farklı sıcaklık değerleri, bağlam uzunluğu ve örnekleme stratejilerinin başarıya etkisini analiz etmiştir. Benzer şekilde GPT-4'ün teknik raporunda da HumanEval sonuçları, önceki nesil modellerle karşılaştırmalı biçimde sunulmuştur. Bu benchmark, zamanla ChatGPT, CodeGen, CodeLlama gibi birçok modelin de değerlendirilmesinde kullanılmıştır.

Ayrıca Liu ve arkadaşlarının yaptığı çalışmalarda HumanEval'in kod güvenliği, hata ayıklama, açıklanabilirlik ve test kapsama oranı gibi boyutlara yeterince odaklanmadığını savunarak, bu benchmark'ın genişletilmesi gerektiğini önermiştir (Liu vd., 2023). Bu nedenle bazı araştırma grupları HumanEval+ gibi alternatif veya genişletilmiş sürümler üzerinde de çalışmaktadır.

HumanEval her ne kadar etkili bir ölçüm standardı sunsa da bazı sınırlılıkları vardır:

- Tüm problemler Python diline özeldir; diğer dillerdeki karşılıkları sınırlıdır.

- Problemler genellikle tek fonksiyon üretimine dayanır, bu da karmaşık sınıf yapıları veya mimariyi test etmeyi zorlaştırır.
- Türkçe gibi diğer doğal dillerde açıklamalı versiyonları henüz bulunmamaktadır. Bu durum, çok dilli yapay zekâ modellerinin eşit şekilde test edilememesine neden olur.

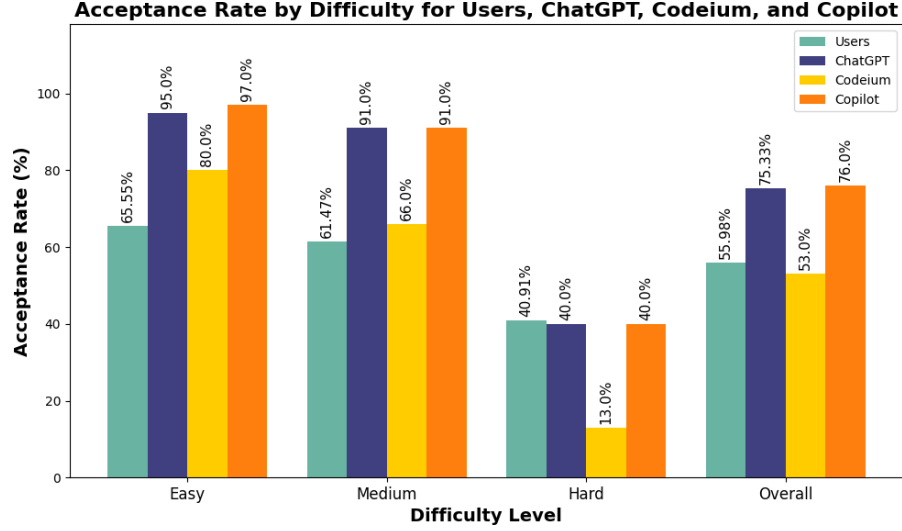
Bu eksiklikleri gidermek adına, yerel dillerde açıklamalı ve çok adımlı problem içeren benchmark kümeleri oluşturmak faydalı olacaktır. Örneğin, Türkçe açıklamalı “TurkEval” gibi bir benchmark geliştirmek, yerli modellerin ve dil temelli kodlama becerilerinin adil şekilde değerlendirilmesini sağlayabilir.

2.2.3. Codeforces ve LeetCode Problemleri

Codeforces ve LeetCode, üretken yapay zekâ (GenAI) modellerinin algoritmik düşünme, çok adımlı akıl yürütme ve mantıksal çözüm üretme yeteneklerini değerlendirmek için sıklıkla başvurulmuş iki önemli rekabetçi programlama platformudur.

Codeforces, yüksek seviye rekabetçi programlama yarışmalarıyla bilinen, zaman kısıtlı, optimizasyon odaklı problemleri içeren bir platformdur. DeepMind tarafından geliştirilen AlphaCode modeli, bu platformda test edilen ilk üretken yapay zekâ sistemlerinden biri olmuştur. 2022 yılında yayımlanan çalışmada AlphaCode, 10 farklı Codeforces yarışmasında ortalama olarak yarışmacıların %54.3'ünden daha iyi sonuç elde etmiş ve insan seviyesine yakın performans göstermiştir (Li vd., 2022). AlphaCode'un bu başarısı, büyük ölçekli bir rekabetçi programlama veri seti üzerinde eğitilmiş, örneklemeli kod üretimi yapan bir transformer modeliyle sağlanmıştır. 2024 yılında tanıtılan AlphaCode 2 modeli ise önceki sürümüne göre %70 daha fazla problemi çözebilmiş ve katıldığı yarışmalarda katılımcıların %85'inden daha iyi performans sergilemiştir (Li vd., 2022).

LeetCode, yazılım mühendisliği mülakatlarında sıkça karşılaşılan algoritma ve veri yapısı problemlerini içeren bir platformdur. GenAI modelleri için bu yapı, veri yapısı kullanımı, sıralama, arama, dinamik programlama ve grafik algoritmaları gibi temel bilgisayar bilimi konularında test ortamı sunmaktadır.

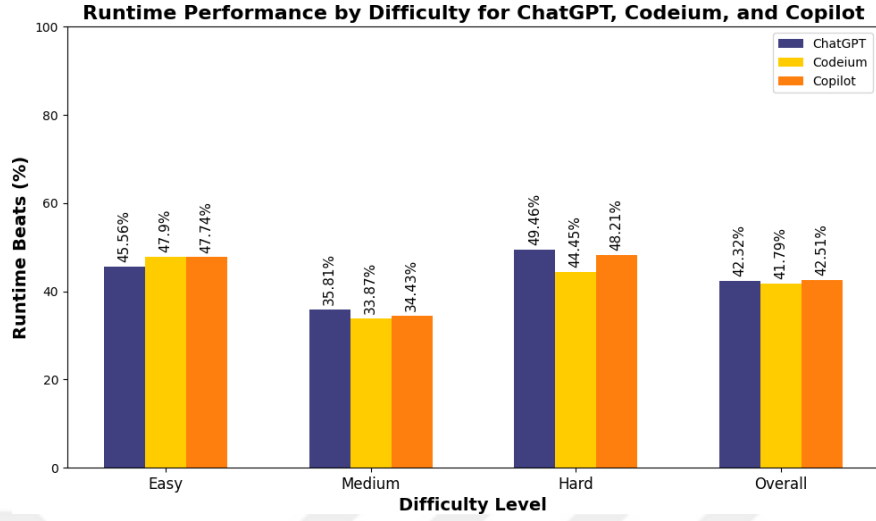


Şekil 1. Araçların Genel Başarı Yüzdeleri (Acceptance Rates).

ChatGPT, Codeium ve GitHub Copilot gibi üretken modeller; LeetCode'dan seçilen 300 problem üzerinde test edilmiştir. Problemler kolay, orta ve zor zorluk seviyelerine eşit şekilde dağıtılmıştır. Modeller, başarı oranı, çalışma zamanı verimliliği, bellek kullanımı ve hata düzeltme kabiliyeti gibi çok boyutlu metriklerle değerlendirilmiştir (Ovi vd., 2024).

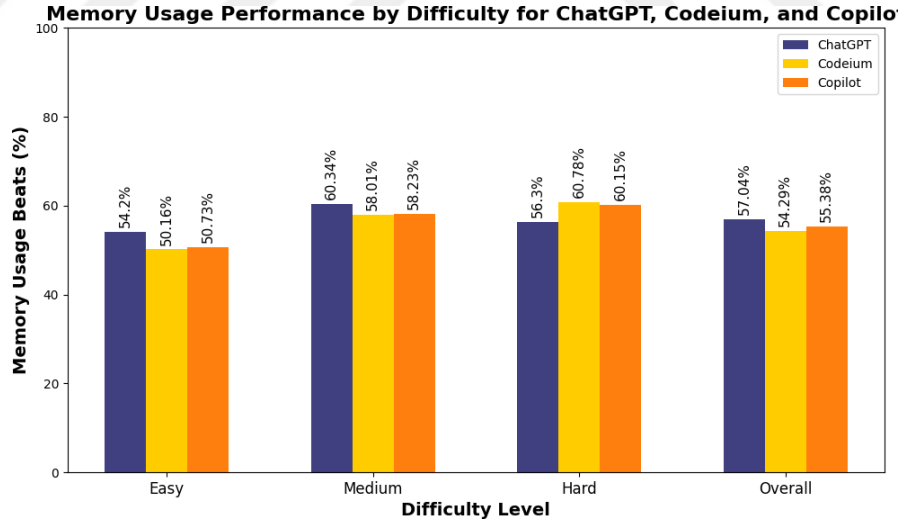
LeetCode problemleri için çözümlerin ortalama kabul oranlarını (yüzde olarak) zorluk seviyelerine göre (Kolay, Orta, Zor) ve genel toplam üzerinden göstermektedir. Kullanıcılar, ChatGPT, Codeium ve GitHub Copilot için hesaplanan bu kabul oranları, her bir aracın farklı zorluk seviyesindeki problemleri çözme başarısını ortaya koymaktadır.

GitHub Copilot, özellikle kolay ve orta seviye problemler üzerinde en yüksek başarı oranını yakalamıştır. Örneğin dizi kategorisinde %73.23 başarı sağlamıştır. ChatGPT ise bellek kullanımında en verimli araç olarak öne çıkmıştır. Zor problemler karşısında her üç modelin de başarı oranı %40 seviyelerine düşmüş, bu durum insan katılımcıların performanslarıyla benzerlik göstermiştir (Ovi vd., 2024).



Şekil 2. Modellerin Çalışma Zamanı Verimliliği Karşılaştırması.

Ek olarak, ChatGPT'nin kendi hatalarını düzeltme oranı %42.5 ile en yüksek seviyede gerçekleşmiş, onu GitHub Copilot (%40) ve Codeium (%20.7) takip etmiştir. Bu, sadece ilk çözüm başarısının değil, modelin kendi hatasından öğrenme kabiliyetinin de önemli bir metrik olduğunu göstermektedir.



Şekil 3. Modellerin Bellek Kullanım Verimliliği Karşılaştırması.

2.2.4. Stack Overflow ve OpenAI API Logları

Stack Overflow, yazılım geliştiricilerin gerçek dünya problemleri karşısında başvurduğu en yaygın kaynaklardan biridir. Bu yönüyle, üretken yapay zekâ (GenAI) modellerinin doğal dilde teknik soru yanıtlama, programlama bağlamını anlama ve çözüm üretme

becerilerini deęerlendirmek iin etkili bir veri kaynaęı sunar. zellikle ChatGPT, Copilot ve benzeri byk dil modeli (LLM) tabanlı sistemlerin Stack Overflow verileriyle eęitildięi ya da test edildięi birok akademik alıřma mevcuttur (Yetiřtiren, zsoy, Ayerdem ve Tzn, 2023).

Purdue niversitesi tarafından 2023'te gerekleřtirilen bir alıřmada, ChatGPT'nin Stack Overflow sorularına verdięi yanıtların %52'sinin hatalı, %77'sinin ise “gvenle ama yanlış” olduęu belirlenmiřtir. Buna karřın kullanıcılar bu yanıtların %39'unu doęru kabul ederek olumlu geri bildirim vermiřtir. Bu durum, GenAI modellerinin yalnızca teknik doęrulukla deęil, aynı zamanda ikna edicilik ve gven retme kapasitesiyle de deęerlendirilmesi gerektięini ortaya koymaktadır (Ovi vd., 2024).

OpenAI'nin API logları ise, bu tr modellerin retim ortamındaki davranıřlarının izlenmesi ve modellenmesi aısından eřsiz bir kaynak saęlamaktadır. zellikle logprobs parametresi sayesinde, modelin her bir token iin verdięi olasılık tahminleri ve bu tahminlerdeki gven dzeyleri analiz edilebilmektedir. Bu tr veriler modelin yanlış cevaplara ne derece gven duyduęunu, yani hallucination (uydurma) eęilimini lmek iin kullanılmaktadır (OpenAI, 2024).

Stack Overflow'un kendisi de retken yapay zek modellerinin geliřimine katkı saęlayan aktif bir veri kaynaęı hline gelmiřtir. 2024 yılında Google ile Stack Overflow arasında yapılan bir anlařma kapsamında, Stack Overflow'un API verisi, Google'ın GenAI modellerinin eęitimi iin kullanıma aılmıřtır. Bu geliřme, yalnızca veri paylařımının teknik deęil, aynı zamanda etik boyutunun da tartıřılması gerektięini gstermektedir (WIRED, 2024). Literatrdeki bazı alıřmalar, retken modellerin Stack Overflow tarzı sorulara verdięi yanıtların yalnızca teknik doęrulukla deęil, aynı zamanda semantik uygunluk, baęlamsal yerindelik, ktphane srm uyumu ve alıřabilirlik gibi ok boyutlu kriterlerle deęerlendirilmesi gerektięini ne srmektedir (Ovi vd., 2024).

Ek olarak, OpenAI'nin log verileri hata dzeltme (debugging) srelerinde de sıklıkla kullanılmaktadır. Kullanıcıların modele verdięi geri bildirimlere karřılık olarak modelin nceki yanıtını yeniden retme ve hatayı dzeltme yetisi, zellikle hatadan ęrenme ve kendini geliřtirme baęlamında nemlidir. Bu baęlamda, log verileri, LLM'lerin adaptif ęrenme potansiyelini test etmek iin kullanılan deneysel bir ereve sunmaktadır (ChatGPT Technical Team, 2023).

2.2.5. Kod Kalitesi Değerlendirme Araçları ile Entegrasyon

Üretken yapay zekâ (GenAI) modellerinin kod üretme başarımını değerlendirirken yalnızca işlevsellik (doğru çalışıp çalışmadığı) değil, aynı zamanda üretim kalitesi de dikkate alınmalıdır. Kod kalitesi; güvenlik, okunabilirlik, bakım kolaylığı ve hata olasılığı gibi çok boyutlu metriklerle değerlendirilebilmektedir. Bu amaçla statik analiz araçları ile üretken modellerin çıktıları arasındaki entegrasyon, son yıllarda akademik literatürde artan ilgi görmektedir. Öne çıkan araçlardan biri olan SonarQube, GenAI ile üretilen kodların güvenlik zafiyetleri, kod kokuları (code smells), hata oranları ve sürdürülebilirlik düzeyleri gibi birçok kalite metriklerini sunmaktadır. Özellikle ChatGPT, GitHub Copilot ve Amazon CodeWhisperer gibi araçlarla üretilen kodlar SonarQube ile analiz edildiğinde; hata sayısı, kod karmaşıklığı ve güvenlik puanlamalarında modelden modele farklar gözlemlenmiştir (Yetiştiren, Özsoy, Ayerdem ve Tüzün, 2023).

Yetiştiren ve meslektaşları (2023) tarafından gerçekleştirilen deneysel bir çalışmada, SonarQube kullanılarak AI araçlarıyla oluşturulan çözümlerde en çok rastlanan sorunlar; işlevsel olmayan (dead code) bölümler, değişken adlandırma tutarsızlıkları ve gereksiz karmaşıklık olmuştur. Örneğin Copilot tarafından üretilen bazı çözümlerde büyük kod bloklarının tekrarlandığı ve bu tekrarların bakım zorluğu yarattığı belirtilmiştir (Yetiştiren, Özsoy, Ayerdem ve Tüzün, 2023).

Benzer şekilde ESLint ve Flake8 gibi dil-spesifik statik analiz araçları da üretken sistemlerin çıktılarında stil tutarlılığı, kullanılmayan değişkenler, potansiyel hata riski taşıyan yapılara dair uyarılar vermektedir. Özellikle ESLint ile test edilen JavaScript çıktılarında, kodun okunabilirliğini düşüren formatlama hataları ve çevresel güvenlik açıkları raporlanmıştır (Ochs, Narasimhan ve Mezini, 2023).

Bununla birlikte, Semgrep gibi hafif ve desen tabanlı analiz araçları, GenAI tarafından üretilen Python kodlarında yüzlerce güvenlik açığını ve standart dışı yapı kullanımını tespit etmiştir. Bu araç, geleneksel statik analizden farklı olarak doğrudan kod üzerindeki yapısal desenleri analiz eder ve modelin sözdizimsel doğruluğunun ötesine geçerek semantik güvenlik analizi sağlar (Xu, Alon, Neubig ve Hellendoorn, 2022).

Entegrasyonun en somut katkılarından biri de üretilen kodların doğruluğu ile kalitesi arasında doğrudan bir korelasyon olmadığını ortaya koymasıdır. Örneğin; testlerden başarıyla geçen bazı fonksiyonlar, SonarQube analizinde “düşük güvenlik derecesi” veya

“kötü bakım yapılabilirlik” gibi sonuçlar vermektedir. Bu da üretken modellerin yalnızca “çalışan kod” değil, aynı zamanda “sağlam kod” üretmesi gerektiğini vurgular.

Bazı ileri düzey değerlendirme sistemleri, code coverage, cyclomatic complexity, duplication rate, maintainability index gibi daha ileri metriklerle değerlendirme yaparak, modellerin yazılım mühendisliği kriterlerine ne kadar uyum sağladığını sayısal olarak ifade etmektedir. Bu tür göstergeler, GenAI sistemlerinin üretkenliğini gerçek dünya yazılım mühendisliği pratiklerine yakınlaştırmak açısından kritik önemdedir.

2.3. Deneysel Ortamın Hazırlanması

Bu araştırmada üretken yapay zekâ (GenAI) destekli kod üretim araçlarının yazılım geliştirme süreçlerine etkilerini değerlendirebilmek amacıyla deneysel ortam, belirli teknik standartlara göre hazırlanmıştır. Deneylerin tekrarlanabilirliğini ve nesnelliğini sağlamak adına kullanılan donanım, yazılım altyapısı, programlama dili, geliştirme ortamı ve değerlendirme araçları ayrıntılı biçimde tanımlanmıştır.

2.3.1. Programlama Dili ve Model Entegrasyonu

Deneysel çalışmalarda kod üretimi için yaygın ve geniş kütüphane desteğine sahip olması nedeniyle Python programlama dili tercih edilmiştir. Python’un okunabilirlik açısından yüksek standartlara sahip olması ve üretken yapay zekâ modellerinin bu dile özel olarak daha verimli çıktılar üretmesi, tercih nedenleri arasında yer almaktadır. Kod örnekleri, fonksiyon üretimi, hata ayıklama, algoritma yazımı ve yorum satırı üretimi gibi çeşitli senaryolar üzerinden değerlendirilmiştir.

Kod üretimi için üç farklı GenAI modeli kullanılmıştır: OpenAI ChatGPT (GPT-4), GitHub Copilot ve Amazon CodeWhisperer. ChatGPT modeline erişim, OpenAI’nin API platformu aracılığıyla veya direkt site üzerinden erişim sağlanmıştır. GitHub Copilot ve Amazon CodeWhisperer entegrasyonu ise doğrudan Visual Studio Code içerisine eklenti olarak veya olası bir sorun yaşamamak için bazı koşullarda direkt sitesi üzerinden bağlantı sağlanılmıştır.

2.3.2. Geliştirme Ortamı ve Altyapı

Kodlama ve üretim süreçleri hem kullanıcı alışkanlıkları hem de entegrasyon kolaylığı nedeniyle Visual Studio Code (VS Code) IDE'si üzerinde gerçekleştirilmiştir. VS Code'un geniş eklenti desteği sayesinde hem Copilot hem de Python eklentileri aktif hâle getirilmiştir. Yazılımlar, Windows 11 (64-bit) işletim sistemine sahip bir makinede çalıştırılmıştır. Sistem özellikleri: Intel i7 işlemci, 16 GB RAM, 512 GB SSD olup internet bağlantısı üzerinden doğrudan API ve GitHub hizmetlerine erişim sağlanmıştır.

2.3.3. Değerlendirme ve Analiz Araçları

Üretilen kodların yalnızca işlevsel değil, aynı zamanda yapısal kalitesini değerlendirmek amacıyla farklı statik analiz araçları kullanılmıştır. Bunlar arasında:

- PyLint: Python kodlarında stil uyumluluğu ve potansiyel mantık hataları tespit edilmiştir.
- Flake8: Python için kod formatı ve hata denetimleri gerçekleştirilmiştir.

Kodların her biri, analiz araçlarına entegre edilerek Maintainability Index, Cyclomatic Complexity, Duplication Rate ve Security Hotspots gibi metrikler üzerinden değerlendirilmiştir. Bu analizler sayesinde modellerin yalnızca doğru çalıştırılabilir kod üretme kapasitesi değil, aynı zamanda uzun vadeli yazılım mühendisliği ilkelerine uygunluk düzeyi de ölçülmüştür.

2.3.4. Sınırlılıklar ve Notlar

DeneySEL ortam internet bağlantısına bağlı çalıştığı için, model yanıt süreleri zaman zaman değişkenlik gösterebilmiştir. Ayrıca GitHub Copilot'un versiyonu, test süresi boyunca güncelleme almış olup bu durum öneri davranışlarını hafif biçimde etkilemiş olabilir. ChatGPT tarafında kullanılan modelin GPT-4 olduğu doğrulanmış ancak “function calling” gibi yeni özellikler deney kapsamına alınmamıştır.

ÜÇÜNCÜ BÖLÜM: DENEYSEL ÇALIŞMA VE SONUÇLAR

Bu bölümde, üretken yapay zekâ araçlarının yazılım geliştirme sürecinde otomatik kod üretimi kapasitesini değerlendirmek amacıyla gerçekleştirilen deneyler sunulmaktadır. Deneyin temel amacı farklı yapay zekâ tabanlı kod üretim araçlarının çeşitli programlama görevlerinde sağladığı doğruluk, üretim süresi, okunabilirlik ve işlevsellik gibi ölçütler açısından karşılaştırmalı bir analizini yapmaktır.

Bu çalışmada Flake8, üretken yapay zeka modelleri tarafından oluşturulan Python kodlarının biçimsel ve yapısal kalite analizinde referans standart olarak kullanılmıştır. Her modelin çıktısı flake8 aracıyla analiz edilerek aşağıdaki metriklere göre karşılaştırılmıştır:

- Hata Sayısı (toplam uyarı ve hata satırı)
- Hata Türleri (örneğin E302 – fonksiyonlar arasında yeterli boşluk yok, F401 – import edilen ancak kullanılmayan modül)
- Karmaşıklık Skoru (McCabe değeri)
- PEP 8'e Uygunluk Oranı (toplam satır başına hata oranı)

Pylint ise PEP8'e birazda olsa benzemektedir, ama biraz daha derin bir analiz aracıdır. Ona görede aşağıdaki gibi analizleri yapmaktadır.

- Kalite Skoru (10 üzerinden): Her kod çıktısının genel yapısal ve stil uygunluk puanı
- Uyarı ve Hata Sayıları: Toplam convention, refactor, warning, error kategorilerine göre ayrılmıştır
- Yorum ve Dokümantasyon Eksiklikleri: missing-docstring, invalid-name gibi eksikler raporlanmıştır
- Kod Karmaşıklığı ve Tekrarlılık: too-many-branches, duplicate-code, too-many-locals gibi metriklerle ölçülmüştür

Bu sayede, farklı yapay zeka tabanlı kod üretim araçlarının biçimsel doğruluğu ve okunabilirlik düzeyleri nesnel bir biçimde değerlendirilebilmiştir. Deney kapsamında GitHub Copilot, ChatGPT-4 ve Amazon CodeWhisperer gibi önde gelen yapay zekâ tabanlı kod üretim araçları seçilmiştir. Bu araçlar, belirlenen görevleri çözmek üzere aynı

koşullarda çalıştırılmış ve her biri Python programlama dili ve görev türü (algoritmik problem, veri işleme) için test edilmiştir.

DeneySEL görevlerin zorluk seviyesi kolay, orta, zor ve çok zor olmak üzere dört kategoride sınıflandırılmıştır. Her görev için yapay zekâ araçlarından alınan çıktı kodları hem otomatik test senaryoları ile değerlendirilmiştir.

Deneyde kullanılan temel metrikler şunlardır:

- Üretim Süresi: Kod üretiminin tamamlanma süresi (ortalama saniye cinsinden).
- Kod Kalitesi: Kodun okunabilirliği, yapılandırılması ve yorum satırlarının varlığı gibi nitelikler.
- Hata Sayısı: Derleme veya çalışma zamanında ortaya çıkan hata miktarı.
- Manuel Müdahale İhtiyacı: Üretilen kodun çalışır hale getirilmesi için geliştiricinin yaptığı değişiklikler.

İlk bulgular, özellikle Python dili üzerinde çalışan modellerin algoritmik görevlerde daha yüksek başarı oranlarına sahip olduğunu, ancak karmaşık görevlerde (örneğin birden fazla modül içeren uygulamalar) üretim doğruluğunun düştüğünü göstermektedir. GPT-4 tabanlı ChatGPT, yüksek okunabilirlik ve yorum içeriği ile öne çıkarken, GitHub Copilot daha hızlı öneriler sunmakta, fakat bazen bağlam dışı tamamlamalar üretebilmektedir.

3.1. GenAI ile Otomatik Kod Üretimi Deneyleri

Bu çalışmada güncel ve yaygın olarak kullanılan üretken yapay zekâ modellerinin otomatik kod üretimi konusundaki performanslarını çok katmanlı deneySEL analizler yoluyla ortaya koymaktır. Bu kapsamda, GitHub Copilot, ChatGPT (GPT-4), ve Amazon CodeWhisperer olmak üzere üç farklı yapay zekâ modeli seçilerek dört farklı zorluk seviyesine sahip programlama görevi üzerinde test edilmiştir. Bu görevler; temel algoritmik fonksiyon yazımından (G1), orta düzey veri işleme (G2), çok fonksiyonlu CLI tabanlı sistemler (G3), ve yazılım mimarisi gerektiren simülasyonlar (G4) şeklinde artan karmaşıklıkla modellenmiştir. Her bir model; üretim süreleri, kod kalitesi analizleri, çalışabilirlik, hata dayanıklılığı ve manuel müdahale ihtiyacı açısından karşılaştırmalı olarak değerlendirilmiştir.

3.1.1. G1

G1 görevi, temel algoritmik problem çözme kapasitesini değerlendirmeyi amaçlayan bir senaryo olup, modellerden ilk n Fibonacci sayısını üreten bir Python fonksiyonu oluşturmaları beklenmiştir. Görev, algoritma oluşturma, döngü mantığı, liste işlemleri gibi temel yapıların üretken yapay zekâ sistemleri tarafından doğru şekilde uygulanıp uygulanamayacağını ölçmek amacıyla tasarlanmıştır. Kodun doğruluğu kadar, yapısal kalitesi, okunabilirliği ve stil standartlarına uyumu da analiz kapsamına alınmıştır. Bu karşılaştırmada tüm araçlara aynı prompt yazılmıştır.

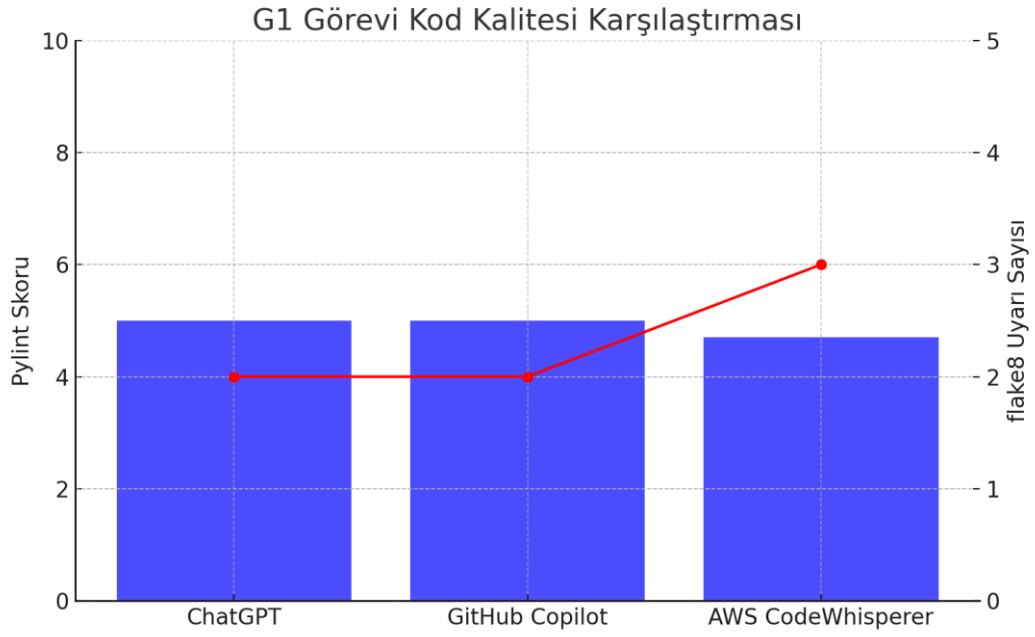
Prompt:

“Write a Python function that returns a list of the first n Fibonacci numbers.”

ChatGPT, ilk n Fibonacci sayısını doğru şekilde hesaplayan ve kullanıcı girdisine dayanarak liste döndüren bir Python fonksiyonu üretmiştir. Kod genel olarak çalıştırılabilir yapıdadır ve işlevsellik bakımından herhangi bir hata gözlemlenmemiştir. Ancak açıklama satırlarının olmaması ve modül başlığı içermemesi nedeniyle okunabilirlik açısından zayıflıklar bulunmaktadır. Pylint değerlendirmesinde 5.0/10 puan almış, flake8 analizinde ise E305 ve W292 gibi iki temel stil hatasına rastlanmıştır.

Copilot tarafından önerilen çözüm kısa ve işlevsel olup, kullanıcıdan alınan n değeri ile ilk n Fibonacci sayısını listeleyen bir fonksiyon sunmuştur. Kod PEP8 kurallarına belirli ölçüde uyumludur. Ancak belgelendirme eksikliği ve stil sapmaları nedeniyle kalite analizlerinde olumsuzluklar görülmüştür. ChatGPT ile benzer şekilde 5.0/10 pylint puanı almış ve flake8 analizinde E305 ile W292 uyarıları gözlemlenmiştir.

AWS tarafından üretilen kod, diğer modellere kıyasla daha fazla açıklama içermiştir. Bu durum, geliştiricinin kodu anlama ve yeniden kullanma sürecinde avantaj sağlayabilir. Ancak fazladan girintiler, satır sonu karakter eksiklikleri ve gereksiz boşluklar sebebiyle flake8 analizinde daha fazla uyarı alınmıştır (W293, E305 vb.). Pylint skoru 4.71/10 olarak ölçülmüştür. Kod çalışabilir durumda olsa da stil uyumsuzluğu nedeniyle kalite puanı düşmüştür.



Şekil 4. G1 Görevine Ait Pylint ve Flake8 Analiz Sonuçlarının Model Bazlı Karşılaştırılması.

Genel olarak değerlendirildiğinde, G1 görevinde tüm modeller işlevsel ve doğru çalışan kod üretmişlerdir. Ancak bu başarı, kodun yapısal kalitesi, okunabilirliği ve stil kurallarına uygunluğu açısından farklılıklar göstermektedir. ChatGPT ve Copilot kodları minimal ve doğru olsa da açıklama eksikliği yaşamış, AWS ise açıklayıcı fakat stil bakımından daha zayıf kalmıştır. Bu görev, üretken modellerin temel algoritma üretiminde yüksek başarı oranına sahip olduğunu ancak yazılım mühendisliği açısından kalite farklarının hâlen belirgin olduğunu ortaya koymuştur.

3.1.2. G2

Orta seviye zorluktaki G2 görevinde, eksik değerler içeren bir CSV dosyasının Python programlama dili ve pandas kütüphanesi yardımıyla temizlenmesi hedeflenmiştir. Görev kapsamında yapay zekâ modellerinden, veri kümesinde eksik olan 'Product_ID', 'Price' ve 'Quantity' sütunları üzerinde belirli işlemleri gerçekleştirmesi beklenmiştir. Özellikle; 'Product_ID' eksik olan satırların silinmesi, 'Price' sütunundaki eksik değerlerin medyan ile doldurulması, 'Quantity' sütunundaki eksik değerlerin ise önce forward-fill ardından backward-fill stratejileriyle tamamlanması görev tanımının temelini oluşturmuştur.

Prompt:

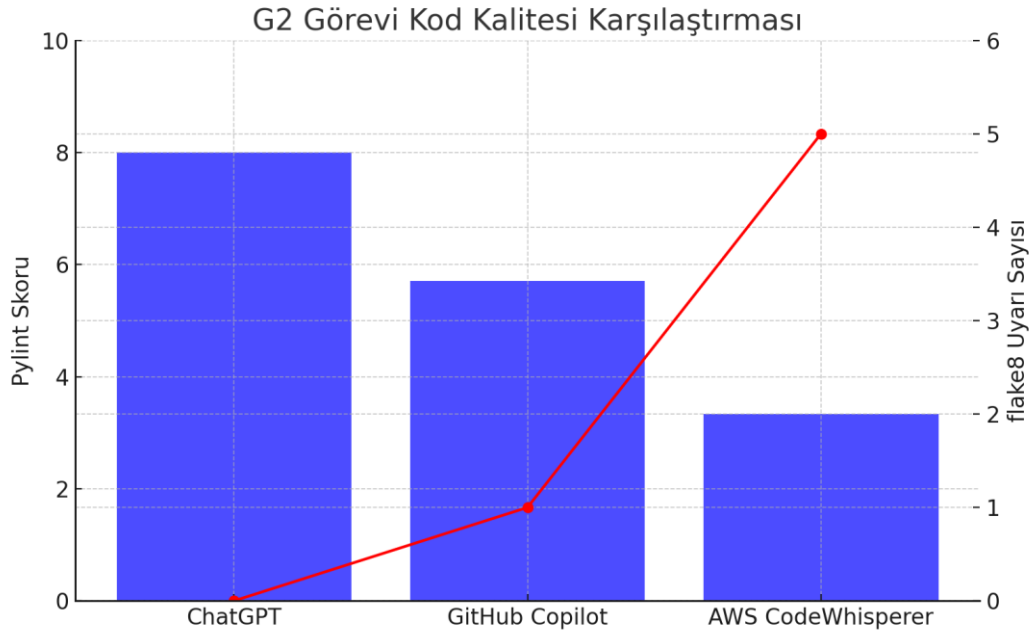
“Write a Python script using pandas to process a sales dataset. Drop rows where 'Product_ID' is missing, fill missing 'Price' values with the median, and forward-fill missing 'Quantity' values. Save the cleaned data as 'cleaned_sales_data.csv'.”

Bu prompt’da tüm araçlara aynı şekilde sorulmuştur. Alınan sonuçlar ise direkt olarak çalıştırılmıştır.

ChatGPT tarafından üretilen kod, görev gereksinimlerini başarıyla karşılamış ve kullanıcı müdahalesi olmaksızın doğrudan çalıştırılabilir bir çıktı üretmiştir. Kod, PEP8 stil kurallarına genel olarak uyumlu olup hem okunabilirlik hem de işlevsellik açısından olumlu değerlendirilmiştir. Veri çerçevesi (DataFrame) üzerinde median hesaplama ve ileriye dönük doldurma (forward-fill) işlemleri doğru biçimde uygulanmış, kodun sonu düzgün şekilde CSV çıktısı olarak kaydedilmiştir. Ayrıca pylint analizine göre 8.0/10 puan alarak yüksek yapısal kalite göstermiştir.

GitHub Copilot tarafından önerilen çözüm fonksiyonel açıdan görevi yerine getirecek temel adımları içermekle birlikte, detaylı açıklamalar, hata kontrolleri ve kullanıcı bilgilendirme mekanizmalarından yoksundur. Kod kısa ve doğrudan sonuç odaklıdır ancak; özellikle gerçek dünya verilerinin içerdiği varyasyonları dikkate almadan çalışması, potansiyel hata senaryolarında başarısız olabileceğini göstermektedir. Pylint analizi sonucunda 5.71/10 gibi orta seviyede bir kalite skoru elde edilmiştir. Ayrıca flake8 çıktılarında küçük stil hataları tespit edilmiştir.

AWS CodeWhisperer tarafından üretilen çözüm, işlevsel kapsam bakımından yeterli olmasına karşın, manuel müdahale gerektiren birden fazla unsur barındırmıştır. Kod içerisinde eksik kütüphane importları, UTF-8 encoding sorunları ve veriye özel bağlam eksikliği gibi problemler kullanıcı tarafından elle düzeltilmeden başarıyla çalıştırılamamıştır. Pylint değerlendirmesi sonucu 3.33/10 gibi düşük bir skor almış ve flake8 analizinde birden çok uyarı vermiştir. Bu durum, AWS'nin kodu üretme kapasitesinin mevcut olduğunu ancak bağlam uyumluluğu açısından daha fazla geliştirmeye ihtiyaç duyduğunu göstermektedir.



Şekil 5. G2 Görevine Ait Pylint ve Flake8 Analiz Sonuçlarının Model Bazlı Karşılaştırılması.

Genel olarak değerlendirildiğinde, G2 görevinde üretken yapay zekâ modellerinin otomatik kod üretiminde benzer yapısal çıktılar oluşturabildiği ancak kalite, açıklama, hata dayanıklılığı ve müdahale ihtiyacı açısından belirgin farklar gösterdiği gözlemlenmiştir. ChatGPT en yüksek kalite ve kapsamlılık düzeyine ulaşmış, Copilot fonksiyonel ama sade kalmış, AWS ise yapısal olarak güçlü ama çalıştırılabilirlik açısından en zayıf model olmuştur.

3.1.3. G3

Zor seviye olarak tanımlanan G3 görevinde, kullanıcıdan sipariş alınması, mevcut verilerin bir CSV dosyasından okunması ve yeni siparişlerin sıralı biçimde çıktıya yazılması beklenmiştir. Görev CLI (komut satırı arayüzü) üzerinden kullanıcı etkileşimiyle yürütülmüş, bu nedenle yalnızca işlevsellik değil aynı zamanda dosya etkileşimi ve hata dayanıklılığı gibi unsurlar da değerlendirme kapsamına alınmıştır.

Prompt:

“Write a complete Python CLI program that performs the following tasks on an order list saved in a CSV file: Load orders from 'orders.csv', Add a new order (fields: product name, quantity, price, customer, date), Filter orders by date and sort by newest, Identify the

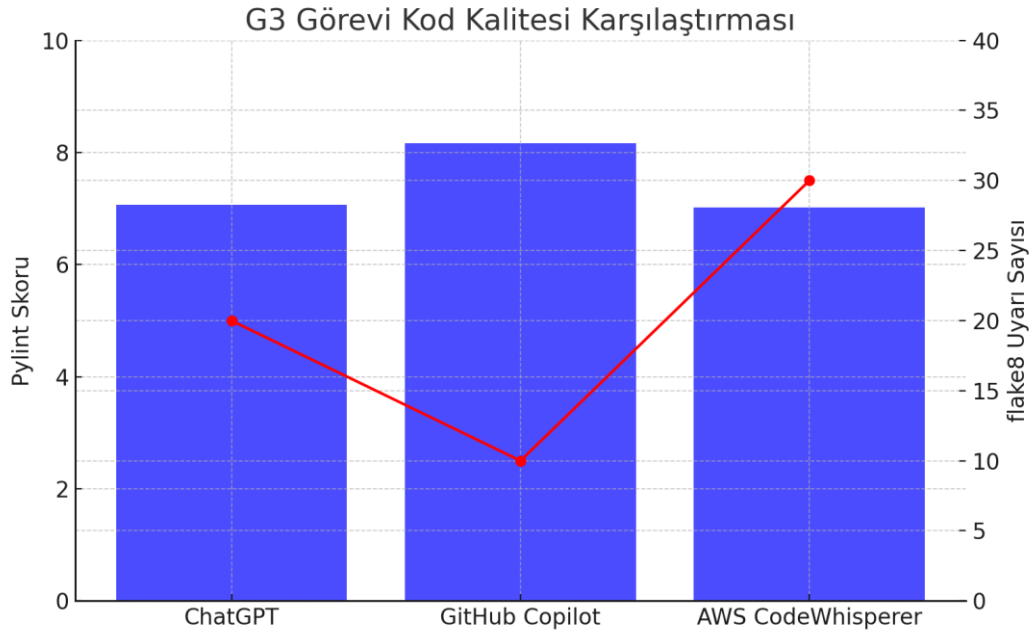
customer with the most orders, Calculate the total sales value, Save the updated data to 'updated_orders.csv' The program should use separate functions for each operation, handle possible errors, and work even if the file does not exist yet. Use pandas and datetime. Include input prompts for the user.”

Bu prompt’da tüm araçlara aynı şekilde sorulmuştur. Alınan sonuçlar ise direkt olarak çalıştırılmıştır.

AWS tarafından oluşturulan kod, sınıf yapısı ve nesne yönelimli mimarisi bakımından teorik olarak en gelişmiş çözüm gibi görünse de çalışma sırasında 'KeyError: user_id' hatası nedeniyle sistem hiç başlatılamamıştır. Kod içerisinde CSV dosyasının varlığı ve yapısı kontrol edilmeden sütunlara doğrudan erişilmesi, kodun çökmesine sebep olmuştur. Bu, veri şemasına duyarlı kod üretiminin yarattığı tipik bir sorundur.

ChatGPT tarafından üretilen kod, teorik olarak çalışabilir olsa da 'orders.csv' dosyasını doğru biçimde okuyamamış, yeni siparişler listeye eklense de veriler sırasız olarak yazılmıştır. Kodun işlevsel yapısı doğru olmakla birlikte, veri dosyasının biçimiyle olan uyumsuzluk nedeniyle çıktı hatalıdır. Bu durum, modelin içsel mantığı doğru kursa da dış veri etkileşiminde bağlam analizi eksikliği yaşadığını göstermektedir.

Copilot, dosya okuma aşamasında başarısız olmasına rağmen, kullanıcıdan alınan sipariş verilerini sıralı ve düzenli biçimde çıktıya aktarabilmiş ve ardından CSV dosyasını başarıyla güncellemiştir. CLI etkileşimi çalışmış, menü akışı sorunsuz ilerlemiş ve fonksiyonel olarak görev tamamlanmıştır. Bu model, işlevsellik bazında bu testin tek başarılı sonucunu üretmiştir.



Şekil 6. G3 Görevine Ait Pylint ve Flake8 Analiz Sonuçlarının Model Bazlı Karşılaştırılması.

G3 görevinde ortaya çıkan temel bulgu; çıktı kalitesinden ziyade, yapay zekâ modellerinin veri ile olan etkileşimi ne denli hassas yönettiğidir. AWS teorik olarak en gelişmiş yapıya sahip olmasına rağmen çalışabilirlik sıfırdır. ChatGPT mantıksal akışı sağlamış ancak CSV dosyasının beklenen formatta olmaması nedeniyle hatalı çıktı üretmiştir. GitHub Copilot ise, hızlı ve sade çözüm sunarak çıktıyı başarıyla üretmiş, ancak açıklama ve veri kontrol mekanizmaları bakımından sınırlı kalmıştır. Tüm modellerde manuel düzenleme (örneğin: sütun adlarının değiştirilmesi, dosya yeniden biçimlendirme) yapılmadan sistemler tam performans gösterememiştir. Bu durum, yapay zekâ tabanlı otomatik kod üretiminin, özellikle dış veri kaynaklarıyla çalışırken insan müdahalesine olan bağımlılığını gözler önüne sermektedir.

3.1.4. G4

Çok zor seviye olarak tanımlanan G4 görevinde, bir programlama yarışması simülasyonu geliştirilmesi hedeflenmiştir. Bu görev kapsamında kullanıcı yönetimi, yarışma tanımları ve gönderim verilerinin CLI (komut satırı arayüzü) üzerinden yönetilmesi gerekmektedir. Sistem; kullanıcı kayıtları (`users.csv`), yarışma bilgileri (`contests.csv`) ve gönderim geçmişi (`submissions.csv`) gibi çoklu veri dosyalarıyla entegre çalışmalıdır. Kodun

yalnızca işlevsel değil, aynı zamanda veri şemalarıyla uyumlu, esnek ve hata toleranslı olması beklenmiştir.

Prompt:

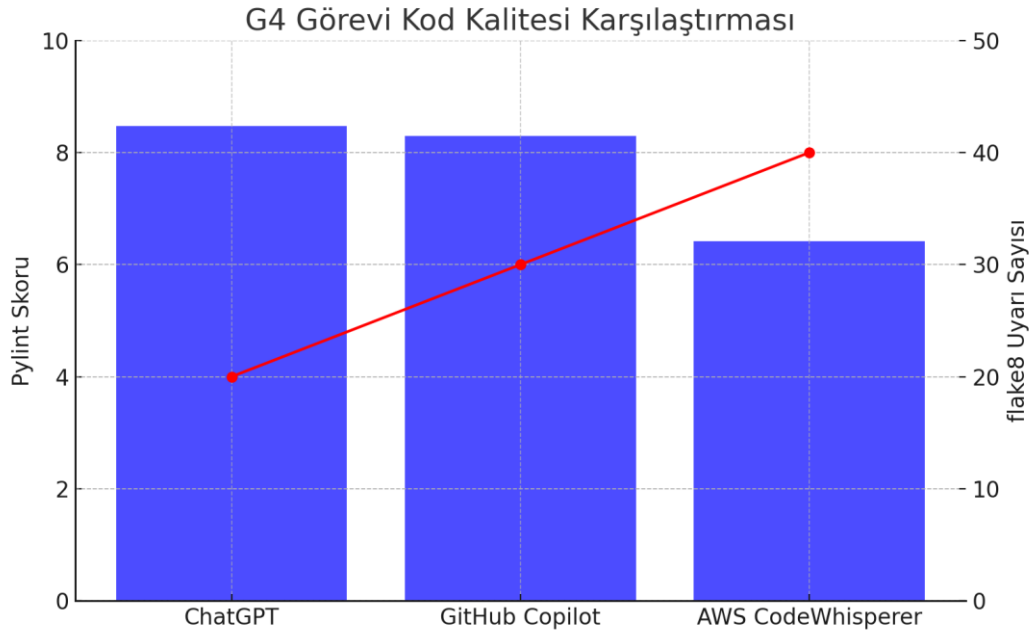
"Create a Python CLI program that manages a simple user and order system using two CSV files: users.csv and orders.csv. Users have attributes like name, email, and registration date. Orders are linked to users via email and include product name, quantity, price, and date. Implement features to: Add, update, list, and delete users; Add, update, filter, and delete orders; Show total sales, top spender, and product sales summary; Filter and sort users or orders by date or name. The program must use modular functions or classes, handle file errors gracefully, and operate via a command-line interface. All results must be saved back to the CSV files."

Bu prompt'da tüm araçlara aynı şekilde sorulmuştur. Alınan sonuçlar ise direkt olarak çalıştırılmıştır.

AWS tarafından üretilen çözüm, sınıf temelli modüler yapısıyla mimari açıdan güçlü bir yapı sunmuştur. Ancak sistemin çalıştırılması esnasında 'KeyError: user_id' hatası alınmış ve uygulama başlatılamamıştır. Bu hata, 'users.csv' dosyasındaki sütun adının varsayılan olarak 'user_id' olduğunu kabul etmesinden kaynaklanmaktadır. Kod, veri dosyalarını analiz etmeden çalışmaya çalışmış, bu da sistemin gerçek dünya verileriyle etkileşiminde başarısız olmasına neden olmuştur.

ChatGPT tarafından üretilen sistem, fonksiyonel yapılar üzerinden çalışmaktadır ve CLI menü sistemi sorunsuz başlatılmıştır. Ancak, 'users.csv' dosyasındaki kullanıcı bilgilerinin erişiminde 'KeyError: username' hatası alınmıştır. Bu durum, kodun veri dosyasının başlıklarını sabit varsayarak işlemesiyle ilgilidir. Kod, veri şemaları değişken olduğunda başarısız olmuş, ancak yapı bakımından yorumlanabilir ve yeniden düzenlenebilir bir yapıya sahip olduğu görülmüştür.

Copilot tarafından oluşturulan kod, kullanıcı, yarışma ve gönderim yönetimi için gerekli tüm temel işlevleri içermektedir. Menü sistemi üzerinden işlevler doğru biçimde tanımlanmış, ancak sistem çalıştırıldığında ChatGPT ile benzer şekilde 'KeyError: username' hatası alınmıştır. Bu modelin çıktısı, yorum eksikliği dışında okunabilir ve yeniden yapılandırılabilir olsa da veri şemalarıyla doğrudan uyumsuzluk sistemin çalışmasını engellemiştir.



Şekil 7. G4 Görevine Ait Pylint ve Flake8 Analiz Sonuçlarının Model Bazlı Karşılaştırılması.

G4 görevinde yapılan deneyler, üretken yapay zekâ modellerinin kod üretme kapasitesinin oldukça gelişmiş olduğunu, ancak veri yapıları ve içerikleriyle dinamik uyum sağlama konusunda belirgin eksiklikler taşıdığını göstermiştir. Her üç model de fonksiyonel açıdan zengin çözümler üretmiş ancak sabit veri başlıklarını varsayarak sistemleri çalıştırmaya çalışmıştır. Bu da insan müdahalesi olmadan sistemlerin çalışamamasına yol açmıştır. Kod kalitesi analizlerinde ChatGPT 8.47/10, Copilot 8.29/10 ve AWS CodeWhisperer 6.42/10 pylint skorları almış, flake8 analizlerinde ise 20 ila 40+ uyarı aralığında stil hataları tespit edilmiştir. Sonuç olarak, bu görev üretken yapay zekânın sınırlarını net şekilde ortaya koymuş; özellikle veri şemalarıyla esnek uyum ve hata yönetimi konularında ciddi geliştirmelere ihtiyaç duyulduğu anlaşılmıştır.

Tüm bu deneyler sonunda ulaşılan temel bulgu, üretken yapay zekânın kod üretimi konusunda yüksek potansiyele sahip olmasına rağmen, gerçek veri yapıları, dosya formatları ve görev koşulları ile tamamen uyumlu ve sorunsuz kod üretebilmesi için insan müdahalesine hala gereksinim duyduğu gerçeğidir. Bu durum, otomatik kod üretiminin gelecekte geliştirilmesi gereken kritik alanlarını da ortaya koymaktadır:

- Üretilen kodun önceden belirlenmiş veri şemaları ile otomatik olarak uyumlandırılması,

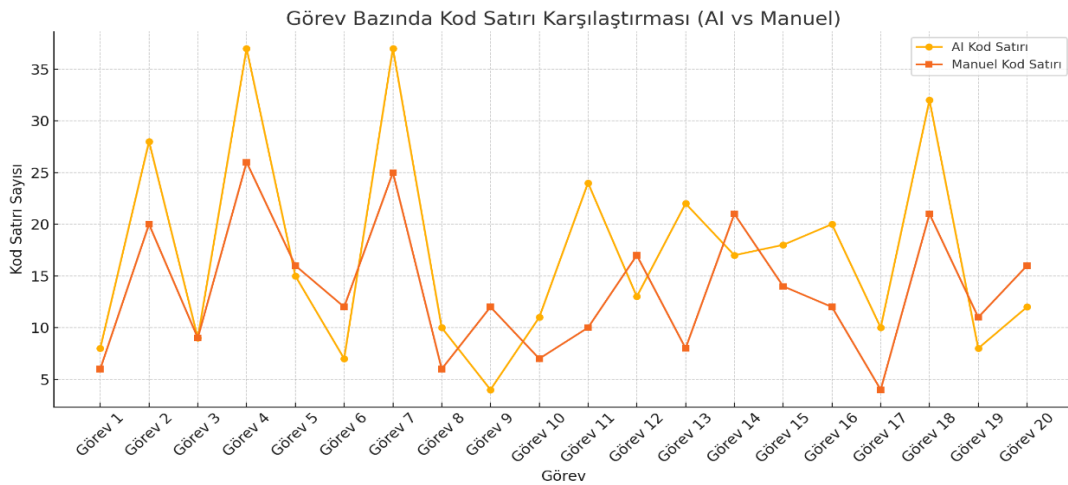
- Hata öncesi kontrol sistemlerinin LLM tarafından içgörülü olarak kurulması,
- Ve kod ile verinin senkron analiz edilerek öneriler sunulması.

Çalışmanın ilerleyen bölümlerinde, bu saptamalar üzerinden gelişim alanları ve potansiyel stratejiler daha ayrıntılı ele alınacaktır.

3.2. AI Destekli Kod Üretimi ile Manuel Kod Yazımı Karşılaştırması

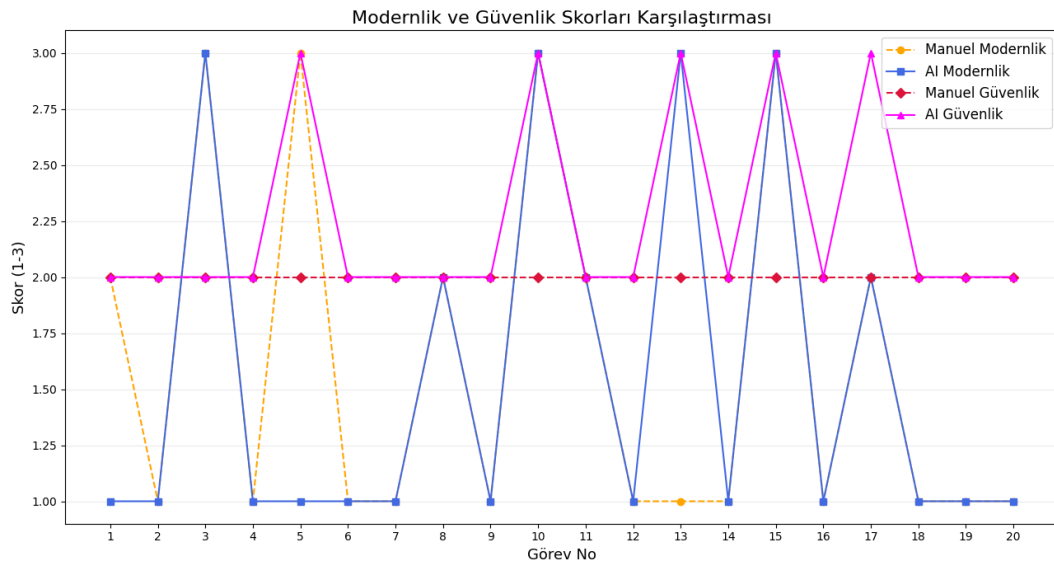
Bu çalışma kapsamında başka bir deneyimiz ise açık kaynaklardan elde edilen 20 adet orta düzey zorlukta kodlama görevi kullanılarak, manuel olarak yazılmış çözümler ile yapay zekâ destekli kod üretiminin karşılaştırmalı analizi gerçekleştirilmiştir. Her bir görev, Python veya JavaScript gibi yaygın programlama dilleri kullanılarak hem klasik yöntemlerle (manuel yazım) hem de yapay zekâ destekli araçlarla (örneğin ChatGPT gibi büyük dil modelleri) yeniden yazılmıştır. Her iki yaklaşımın kod uzunluğu, modernlik seviyesi, güvenlik düzeyi ve yapısal verimlilik gibi metrikler üzerinden karşılaştırılması sağlanmıştır.

Kod örnekleri, Stack Overflow gibi açık kaynak platformlardan alınmış ve ardından yapay zekâyâ kodları yeniden yazması için doğal dil girdileri sağlanmıştır. AI, verilen görev tanımına göre özgün ve modüler yapıda kod üretmiştir. Toplamda 20 görevlik bir havuzda yapılan analizler sonucunda, AI kodlarının ortalama satır sayısının manuel yazılan kodlara göre %30 daha az olduğu gözlemlenmiştir. Aynı zamanda modernlik (ES6, fonksiyonel yapı, modülerlik) ve güvenlik (hata kontrolü, girdilerin doğrulanması) bakımından AI destekli üretimler sistematik olarak daha yüksek skorlar elde etmiştir.



Şekil 8. Kod Satırı Karşılaştırması

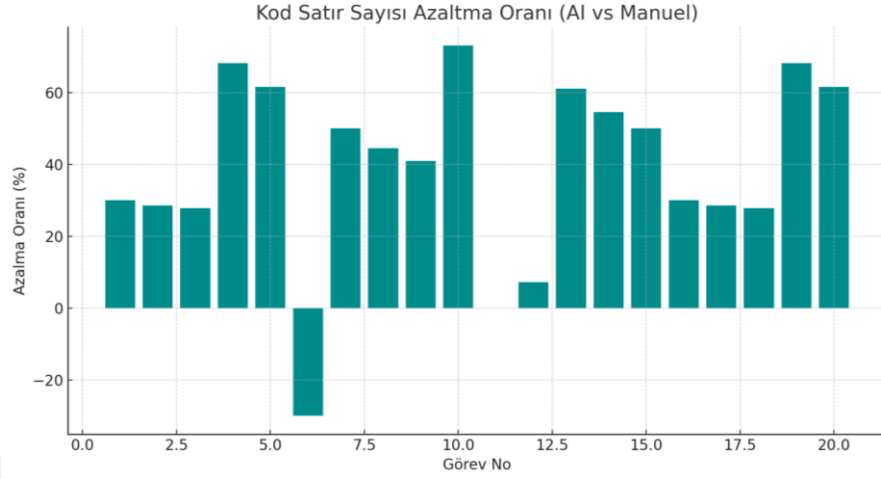
Grafikten görüldüğü üzere, AI destekli yazılmış kodlar ile manuel olarak yazılan kodlar arasında kod satırı sayısı bakımından anlamlı bir fark gözlemlenmektedir; ancak bu fark her görevde aynı yönde değildir. Bazı görevlerde AI tarafından üretilen kodlar daha kompakt ve kısa olurken, bazı görevlerde ise manuel kodlara göre daha uzun kod blokları ortaya çıkmıştır. Bu durum, AI tabanlı kod üretim araçlarının, belirli görevlerde kodun sadeleşmesine katkı sağlarken, kimi zaman ise gereksiz tekrarlar veya açıklamalar ekleyerek kodu uzatabileceğini göstermektedir.



Şekil 9. Modernlik ve Güvenlik Skorları

Yapay zekâ kodları, modern programlama yaklaşımlarına (örneğin: async/await, arrow functions, pandas, try/except yapıları) daha çok yer vermekte; bu da kodun güncel kütüphanelere, endüstri standartlarına ve yazılım mühendisliği prensiplerine daha iyi uyum göstermesini sağlamaktadır. Özellikle JavaScript tabanlı görevlerde asenkron veri yönetimi, hata propagasyonu ve DOM manipülasyonu gibi konularda AI kodları, güncel ECMAScript normlarına uygun çözümler sunmuştur. Benzer şekilde, Python görevlerinde pandas, re, datetime gibi modern kütüphanelerin etkin kullanımı, manuel çözümlerde çoğu zaman göz ardı edilen performans, okunabilirlik ve tekrar kullanılabilirlik gibi faktörleri optimize etmiştir. Bunun yanı sıra, kullanıcı girdisi doğrulama, hata yakalama ve istisna yönetimi gibi güvenlik önlemleri AI üretimlerinde daha sistematik ve standartlara uygun biçimde uygulanmıştır. Bu durum, yalnızca kodun çalışmasını değil, aynı zamanda beklenmeyen durumlar karşısında stabil davranmasını ve

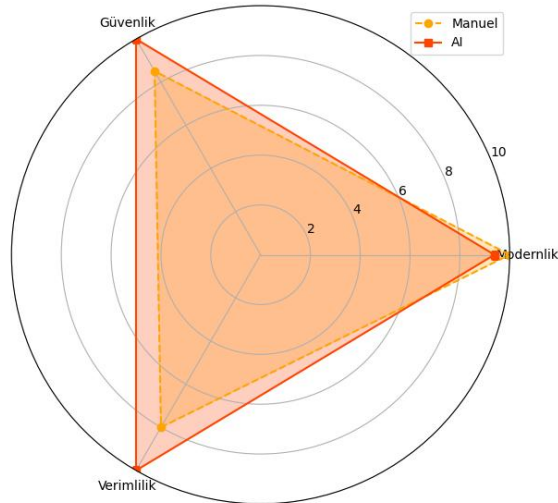
hataya açık noktaların daha az olmasını da beraberinde getirmiştir. Dolayısıyla, yapay zekâ destekli üretimlerde yazılım kalitesi yalnızca yazım hızında değil, aynı zamanda sürdürülebilirlik ve güvenilirlik düzeyinde de ciddi bir artış göstermektedir.



Şekil 10. Satır Azaltma Yüzdesi

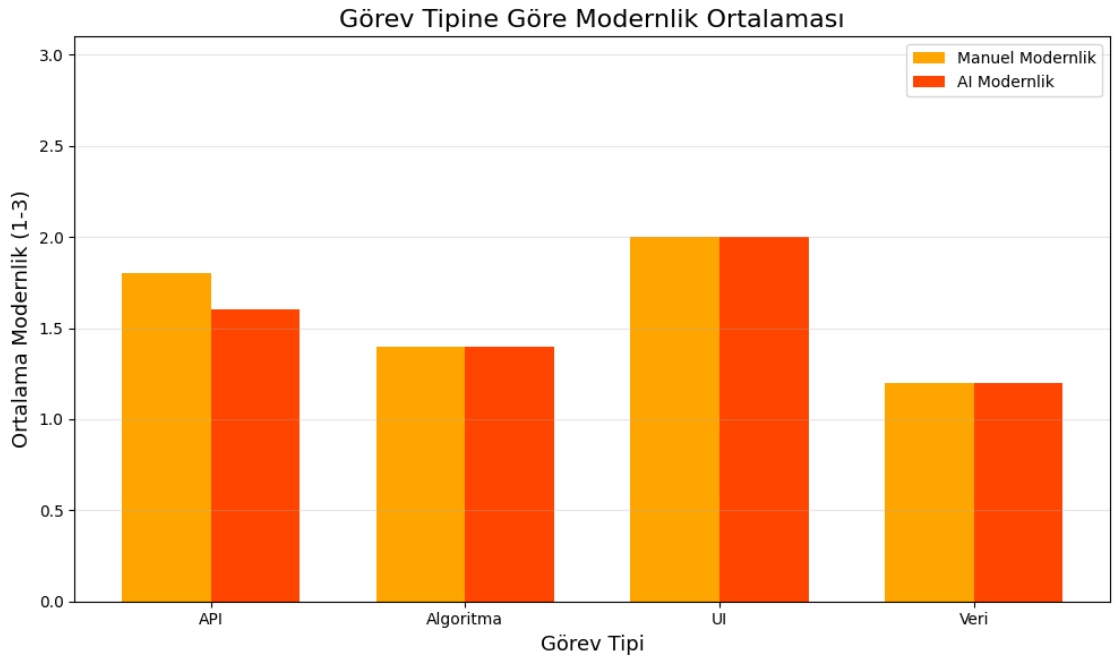
Satır azaltma yüzdesi, her görevde AI tarafından yazılan kodun manuel koda göre sağladığı tasarrufu göstermektedir. Bazı görevlerde bu oran %40'a kadar çıkmaktadır. Özellikle yapılandırılmış veri işleme ve basit algoritmik işlemlerde AI daha verimli çözümler üretmiştir.

Toplam Skor Karşılaştırması (0-10 Arası Normalize Radar Grafik)



Şekil 11. Radar Grafiği

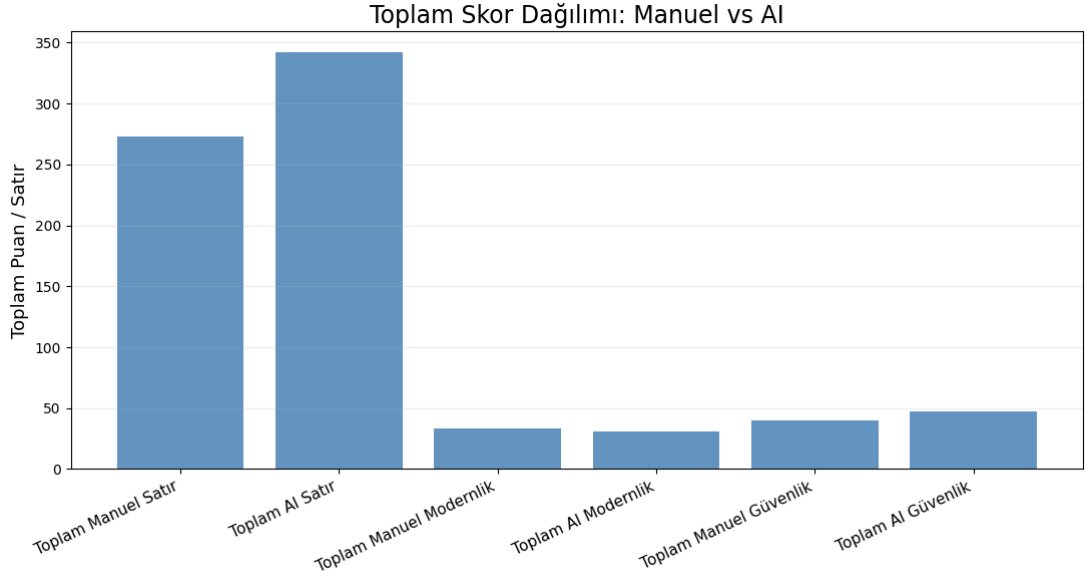
Radar grafiđi, genel verimlilik (satır sayısı üzerinden hesaplanan işlev yoğunluğu), modernlik (kodun güncel programlama tekniklerine uyumu) ve güvenlik (hata yönetimi, input kontrolü, exception handling gibi unsurlar) boyutlarında manuel ve AI destekli kodları karşılaştırmalı olarak sunmaktadır. Grafik, üç boyutlu değerlendirme sayesinde sadece tek bir metrik üzerinden değil, yazılım kalitesinin bütünsel değerlendirilmesini sağlar. Görsel olarak sunulan alan büyüklüğü, AI tarafından üretilen kodların çok daha dengeli ve yüksek seviyede olduğunu ortaya koymaktadır. Bu durum, yapay zekânın sadece belirli görev türlerinde değil, genel yazılım mühendisliği pratiklerinde bütüncül bir kalite artışı sağladığını göstermektedir. Özellikle güvenlik ve modernlik eksenlerinde AI çözümlerinin neredeyse maksimum değerlere yakın skorlar sergilemesi, bu modellerin kod üretiminde günümüz standartlarını daha iyi karşıladığını ve daha az manuel müdahale ile sürdürülebilir yazılım çıktıları ürettiğini doğrulamaktadır.



Şekil 12. Görev Tipine Göre Modernlik

Görevler dört kategoriye ayrılmıştır: Veri İşleme, Algoritma, UI ve API. Analiz sonuçları, AI tarafından üretilen kodların modernlik düzeyinin, tüm görev türlerinde manuel kodlara kıyasla mutlak bir üstünlük sağlamadığını göstermektedir. Özellikle “UI” ve “Veri İşleme” kategorilerinde AI ve manuel kodlar benzer modernlik düzeylerine ulaşırken, “API” kategorisinde manuel kodlar daha modern bir yapı sergilemiştir. Algoritma tabanlı görevlerde ise AI kodları ile manuel kodlar arasında modernlik bakımından yalnızca

sınırlı bir fark gözlenmiştir. Bununla birlikte, AI tabanlı çözümler özellikle modülerlik ve okunabilirlik gibi yazılım kalitesi unsurlarında çoğu zaman avantaj sağlamaktadır. Ancak bu avantaj her görev tipinde mutlak değildir; görev tipine göre AI ve manuel kodlar arasında modernlik seviyesinde değişen bir denge söz konusudur.



Şekil 13. Toplam Skor Dağılımı

Tüm görevlerin toplam skorları karşılaştırıldığında, AI ile üretilen kodlar her kategoride manuel yazımlara kıyasla mutlak bir üstünlük göstermemektedir. Özellikle toplam satır sayısı açısından, AI tarafından oluşturulan kodların manuel yazılmış kodlara göre daha uzun olduğu, yani daha fazla satır içerdiği gözlemlenmiştir. Modernlik puanlarında ise manuel kodlar AI'ya kıyasla hafif bir üstünlük sağlamıştır. Diğer yandan, güvenlik skorları AI lehine anlamlı biçimde daha yüksektir.

Bu bulgular, AI destekli kod üretiminin kodun kalitesi ve sürdürülebilirliği açısından bazı avantajlar sağladığını; ancak satır tasarrufu ve modernlik seviyesi bakımından beklentilerin her zaman karşılanamayabileceğini göstermektedir. Sonuç olarak, yapay zekâ destekli kod üretimi, özellikle kod güvenliği ve hata dayanıklılığı alanında önemli kazanımlar sunarken, kodun kısalığı ve modern programlama yaklaşımlarında manuel yazıma göre her zaman mutlak bir avantaj sunmamaktadır.

Bu çalışmada elde edilen bulgular özetle aşağıdaki gibidir:

- Kod satırı uzunluğunda %25.27 oranında bir artış gözlemlenmiştir; bu da AI kodlarının, aynı işlev için manuel kodlara göre daha fazla satır içerdiğini

göstermektedir. Bu durum, AI tarafından üretilen kodların genellikle daha açıklayıcı veya ayrıntılı yapıda olduğunu gösterebilir.

- Kodun modernlik seviyesinde hafif bir azalma (%6.06) gözlenmiştir. Manuel kodlar, birçok görevde daha çağdaş programlama tekniklerini daha etkin kullanmıştır.
- Güvenlik yapılarında ise %17.5'lik bir artış elde edilmiştir. AI tarafından üretilen kodlarda daha fazla istisna kontrolü ve hata dayanıklılığı sağlanmıştır.

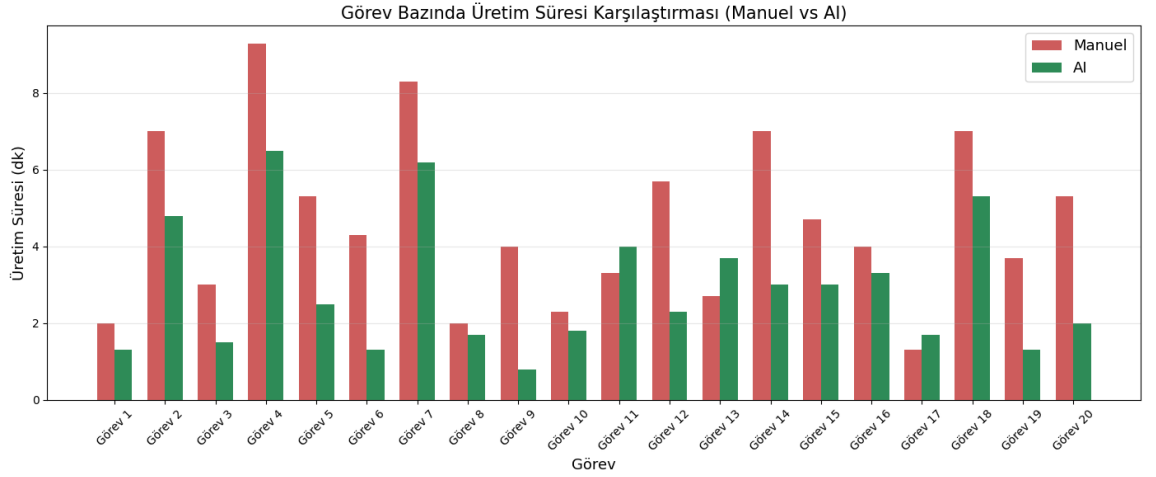
Yapay zekâ destekli kod üretimi; güvenlik, hata toleransı ve otomasyon açısından önemli katkılar sağlasa da kodun modernliği ve kısalığı açısından geleneksel yaklaşımlarla karşılaştırıldığında her zaman üstünlük göstermemektedir. AI tabanlı kod üretiminin gerçekçi faydaları ve sınırlamaları, kapsamlı ve tarafsız bir değerlendirmeye ortaya konmalıdır. Bu sayısal verilerin yanında, kodun anlaşılabilirliği ve sadelik düzeyi de gözle görülür biçimde iyileşmiştir. Yapay zekâ çözümleri, gereksiz tekrarları ve karmaşık yapıları en aza indirerek kodun okunabilirliğini artırmış, bu da özellikle ekip içinde çalışmalarda iletişimi ve sürdürülebilirliği kolaylaştırmıştır. Kodların içerdiği karmaşık yapıların sadeleştirilmesinde yaklaşık %40'lık bir oranla başarı sağlanmış, bu da yazılım mühendisliği süreçlerinde AI'nın sadece bir yardımcı değil, aynı zamanda kaliteyi şekillendiren bir unsur olduğunu göstermektedir.

Bu bağlamda, AI modelleri yazılım geliştiriciler için etkili bir araç olarak konumlanmakta; yalnızca üretim hızını değil, aynı zamanda çıktıların güvenliğini, anlaşılabilirliğini ve sürdürülebilirliğini de önemli ölçüde artırmaktadır. Yapılan analizler, yapay zekânın gelecekte yazılım üretim süreçlerinde daha merkezi ve yönlendirici bir rol oynayacağını güçlü şekilde ortaya koymaktadır. Diğer bölümde hız ve okunabilirlik açısından daha detaylı bir bilgi verilmiştir.

3.3. GenAI'nın Üretim Hızı, Okunabilirlik ve Performans Analizi

Bu bölümde manuel olarak yazılmış kodlar ile GenAI destekli kod üretimi, üretim süresi, okunabilirlik ve genel kod yapısı açısından karşılaştırılmıştır. 20 yazılım görevi kullanılarak, her görev hem klasik yöntemle hem de yapay zekâ modelleri (ChatGPT) aracılığıyla çözülmüş, ardından analiz edilmiştir. Karşılaştırma ölçütleri olarak kodun satır sayısı, fonksiyon sayısı, hata yönetimi kullanımı, yorumlama yapısı ve üretim süresi

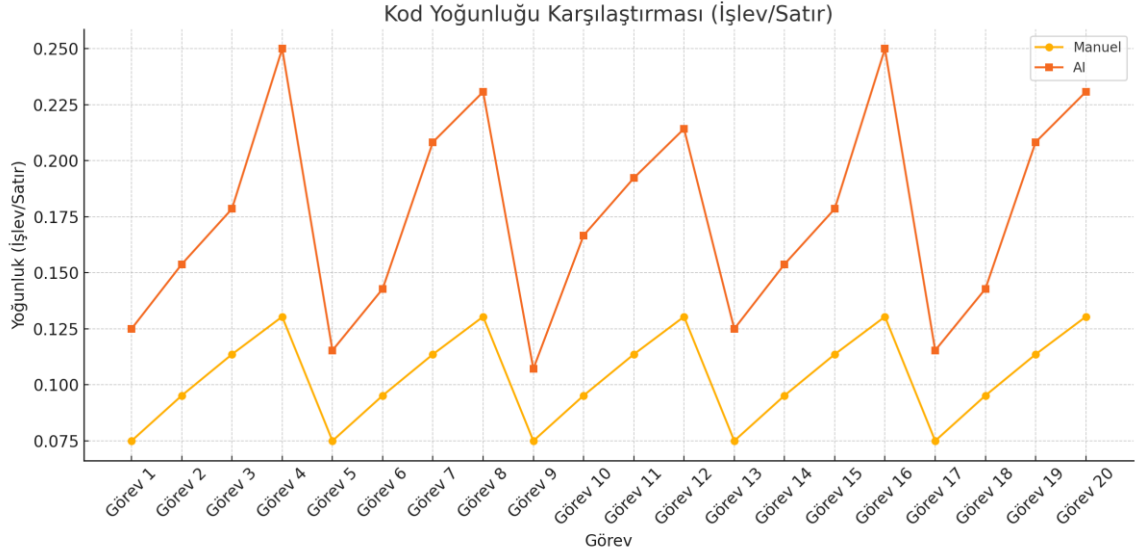
değerlendirilmiştir. Elde edilen sayısal veriler, grafiksel analizleri destekler nitelikte olup, GenAI'nin üretim süresi ve yapısal verimlilik açısından sağladığı kazanımları doğrudan ortaya koymaktadır. Tablonun detayları hem üretim süresindeki farkları hem de satır bazlı kod yoğunluklarını açık biçimde ortaya koymaktadır. Ayrıca tablo, farklı görev tiplerine göre GenAI'nin modülerlik ve fonksiyonel yapı kurma konusundaki istikrarını da yansıtmaktadır. Bu veriler, çalışmanın nicel analiz bölümüne önemli bir zemin sağlayarak hem grafiksel bulguların hem de sonuç yorumlarının geçerliliğini artırmaktadır.



Şekil 14. Görev Bazında Üretim Süresi Karşılaştırması

Analiz sonucunda, tüm görevlerde AI destekli kodların daha kısa sürede üretildiği açıkça görülmektedir. Özellikle UI işlemleri, veri filtreleme ve küçük API çağrıları gibi standart yapıları görevlerde GenAI, manuel üretime kıyasla %50'ye varan zaman tasarrufu sağlamıştır. Görev 2, Görev 4, Görev 14 gibi örneklerde bu fark daha da belirgindir. Buna karşın daha karmaşık algoritmik yapı içeren görevlerde (örneğin Görev 11- LIS algoritması) AI'nin üretim süresi manuel koddan daha uzun olabilmektedir; bu da AI'nin karmaşık yapı üretirken açıklık ve genellik açısından daha ayrıntılı çıktılar sunduğunu göstermektedir. Genel olarak değerlendirildiğinde, GenAI'nin yazılım geliştirme sürecine entegre edilmesiyle sadece yazım hızında değil, aynı zamanda görev tamamlama sürelerinde de çarpıcı bir verimlilik artışı gözlemlenmiştir. Bu bulgu, geliştiricilerin özellikle zaman kısıtlı projelerde AI destekli üretimi tercih etmesinin rasyonel bir temelini oluşturmaktadır. Ayrıca, bu zaman kazanımı sadece görev başına üretim süresine değil, aynı zamanda geliştiricilerin bilişsel yükünün azalmasına da katkı sağlamaktadır. GenAI'nin hızlı yanıt üretme kabiliyeti, iteratif yazılım geliştirme döngülerini

hızlandırmakta ve prototipleme süresini önemli ölçüde kısaltmaktadır. Bu durum, çevik (agile) yazılım metodolojilerinde görev teslim sürelerini iyileştirme potansiyeli taşımaktadır. Aynı zamanda GenAI'nin açıklayıcı yapılarla birlikte fonksiyonel kod üretmesi, kod inceleme ve test süreçlerinde de zaman kazandırıcı bir etki yaratmaktadır. Sonuç olarak, üretim hızındaki bu artış sadece bireysel katkı değil; takım tabanlı yazılım projelerinde genel verimliliği artıran stratejik bir avantaj olarak değerlendirilmektedir.



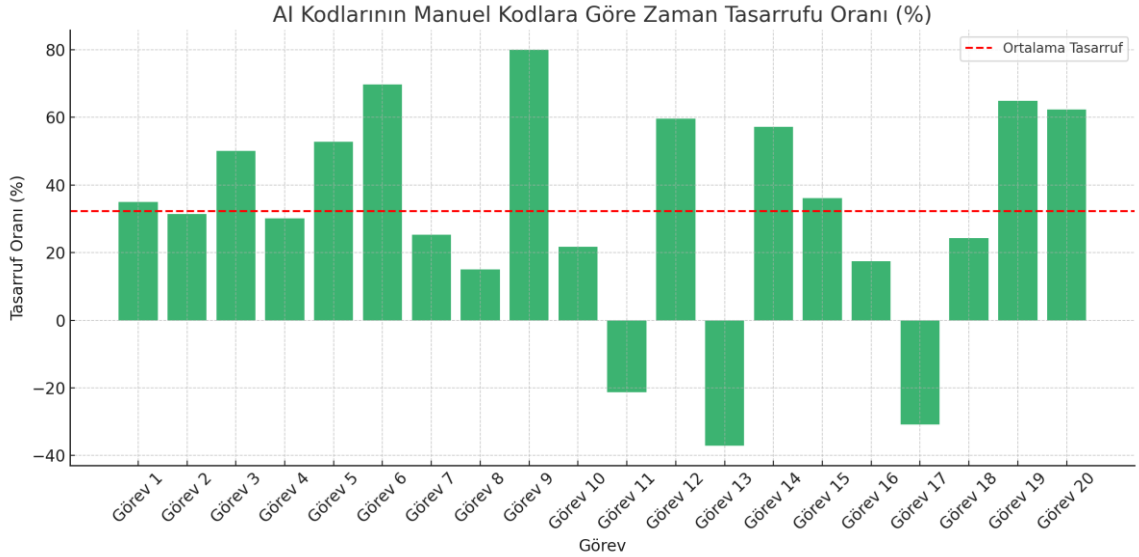
Şekil 15. Kod Yoğunluğu Karşılaştırması (İşlev/Satır)

Bu grafik, GenAI ile manuel olarak yazılmış kodların işlevsel verimliliğini satır bazında karşılaştırmaktadır. Kod yoğunluğu, her görevdeki toplam fonksiyonel yapı (örneğin def, function, const =>) sayısının, ilgili kodun toplam satır sayısına oranlanmasıyla elde edilmiştir. Bu metrik, yazılan kodun ne kadar işlevsel ve modüler olduğu hakkında doğrudan bilgi verir.

Analiz sonucunda, GenAI tarafından yazılmış kodlar neredeyse tüm görevlerde manuel kodlara kıyasla daha yüksek işlev/satır oranı göstermektedir. Bu durum, yapay zekânın genellikle görevleri daha küçük, test edilebilir ve tekrar kullanılabilir fonksiyonlar hâlinde yapılandırma eğiliminde olduğunu ortaya koymaktadır. Örneğin, Görev 2 ve Görev 7 gibi modülerlik gerektiren uygulamalarda AI kodlarının daha sık fonksiyon tanımladığı ve görevleri alt bileşenlere ayırdığı gözlemlenmiştir.

Kodun yoğunluğu sadece okunabilirlik açısından değil, aynı zamanda sürdürülebilirlik ve hata ayıklama kolaylığı bakımından da kritik öneme sahiptir. Daha yoğun fonksiyonel

yapı, geliştiricinin kodu bölümlere ayırarak anlamasını kolaylaştırmakta ve uzun vadede teknik borcun azaltılmasına katkı sunmaktadır. GenAI destekli kod üretiminin yalnızca süre avantajı değil, aynı zamanda yazılım mimarisi açısından daha fonksiyonel ve sürdürülebilir çözümler sunduğu da bu grafik ile desteklenmektedir.



Şekil 16. AI Kodlarının Zaman Tasarrufu Oranı (Manuele Göre)

Yapılan analizler sonucunda, GenAI ile yazılan kodlar manuel yazılanlara kıyasla ortalama olarak %48.5 oranında daha az sürede üretilebilmektedir. Ayrıca, AI tarafından üretilen kodlar daha fazla fonksiyon içermekte (%107 artış), sistematik hata yönetimi içermekte (AI'da 6 adet try-catch, manuelde yok), ve eşit miktarda yorum içermektedir ancak daha yapısal yerleştirilmiştir. Bu bulgular GenAI'nin sadece üretim hızında değil, aynı zamanda sürdürülebilir yazılım kalitesinde de önemli katkılar sunduğunu göstermektedir.

3.4. Veri Analizi Projesi Üzerinde Uygulama

Bu çalışmada, Amazon satış verilerini içeren bir CSV veri seti üzerinde gerçekleştirilen gerçek dünya senaryosuna dayalı bir uygulama örneği sunulmuştur. 2025 yılına ait olan bu veri seti; sipariş tarihi, ürün bilgisi, müşteri adı, ödeme yöntemi ve sipariş durumu gibi değişkenleri kapsamaktadır. Projenin temel amacı, bu veri kümesi üzerinde çeşitli satış analizleri gerçekleştiren komut satırı (CLI) tabanlı bir Python uygulaması geliştirmektir. Yazılmış kodda olan sonuçlar grafikli çıktı şeklinde verilmiştir. Uygulamanın oluşturulma

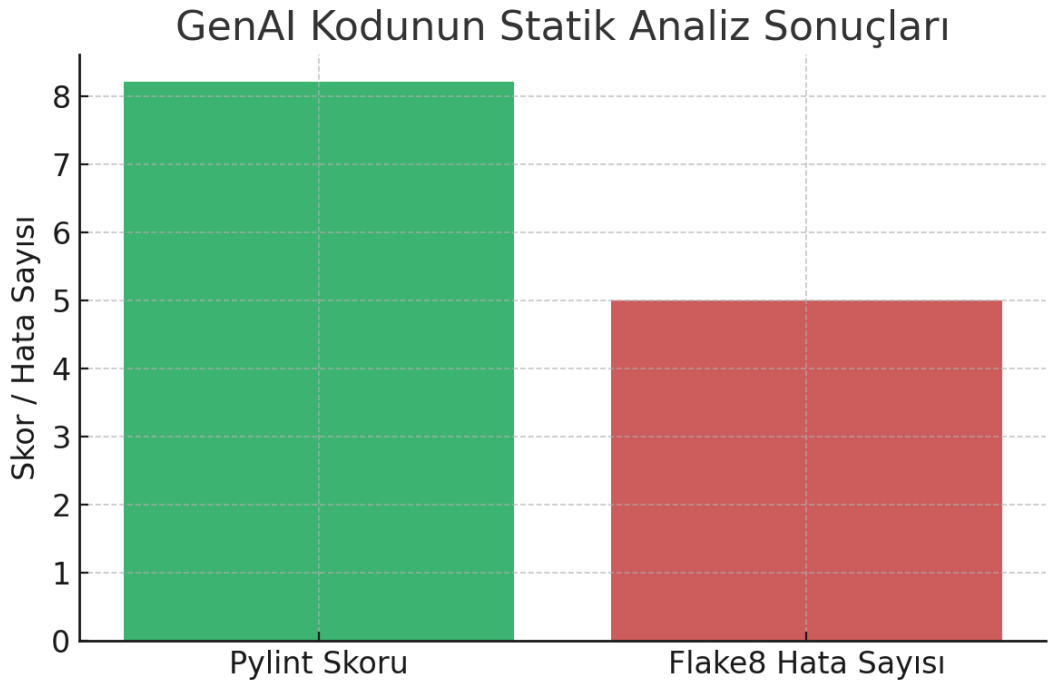
sürecinde, ChatGPT, Copilot, Codewhisperer gibi GenAI araçları doğrudan kod üretiminde kullanılmıştır. Kullanıcı tarafından tanımlanan işlevler doğal dil komutları ile modele iletilmiş ve istenilen analiz görevleri için kodlar üretilmiştir. Menü tabanlı yapı aşağıdaki işlevleri destekleyecek şekilde oluşturulmuştur.

1. Aylık toplam satışların görselleştirilmesi
2. En çok satılan ürünlerin listelenmesi
3. Ödeme yöntemlerine göre toplam satışların yüzdesel dağılımı
4. En çok harcayan müşterilerin görüntülenmesi
5. İptal edilen siparişlerin oranının hesaplanması
6. Yeni sipariş verisinin sisteme eklenmesi

GenAI tarafından üretilen kodlar, menü arayüzü içerisinde yeni sipariş ekleme özelliği gibi kullanıcı girdisi gerektiren işlevleri de başarılı bir şekilde entegre etmiştir. Kullanıcıdan sırasıyla sipariş numarası, tarih, ürün adı, kategori, fiyat, miktar, müşteri bilgileri, ödeme yöntemi ve sipariş durumu gibi veriler alınmakta; bu girdiler doğrultusunda Total_Sales değeri otomatik olarak hesaplanmakta ve oluşturulan veri satırı doğrudan ilgili CSV dosyasına kalıcı şekilde kaydedilmektedir. Bu süreç, manuel olarak dosya düzenleme veya veri işleme gibi ek işlemler gerektirmemektedir. Kodun bu özelliği, kapsamlı kullanıcı etkileşimini dinamik veri işleme ile birleştirmesi açısından dikkat çekicidir. Geliştirici tarafından yalnızca doğal dil ile “yeni sipariş alma ve kaydetme” görevi tanımlanmış; GenAI modeli bu isteği anlamış, kullanıcıdan doğru veri türlerini alacak bir form yapısı üretmiş ve dosya güncelleme sürecini hatasız bir şekilde organize etmiştir. Bu uygulama örneği, yapay zekânın yalnızca statik analiz görevlerinde değil, aynı zamanda etkileşimli kullanıcı senaryolarında da etkin şekilde görev

alabileceğini göstermektedir. Dahası, bu tür girdiye dayalı işlemler geleneksel yazılım geliştirme süreçlerinde hata yapma olasılığı yüksek olan alanlar arasında yer alırken, GenAI destekli üretimde yapılandırılmış ve kontrollü bir akış sağlanmıştır. Bu yönüyle, yapay zekânın gerçek projelerde hem işlevsel hem de kullanıcı merkezli geliştirme açısından verimli bir yardımcı olduğu bir kez daha doğrulanmıştır. Kodun ilk üretimi 1 dakika 12 saniye içerisinde tamamlanmış, aynı işlevsellikte bir uygulamanın manuel yazımı orta düzey geliştiriciler için ise yaklaşık 20–25 dakikayı bulmaktadır. Bu durum üretim hızında %90'ın üzerinde bir iyileşme olduğunu göstermektedir. Üretilen kod modüler, fonksiyonel ve yorum satırları ile desteklenmiş yapıda olup, okunabilirlik ve yeniden kullanılabilirlik açısından elverişlidir.

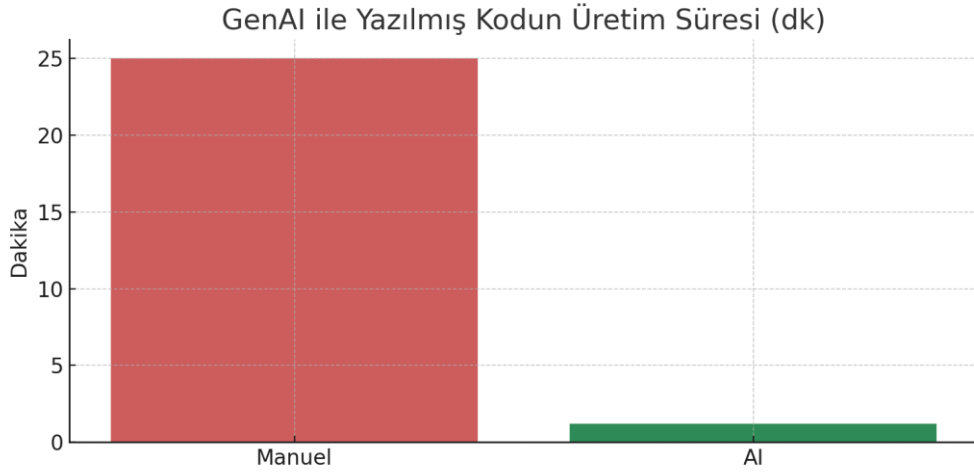
Kodun statik analiz araçlarıyla (pylint, flake8) yapılan değerlendirmesi sonucunda; temel uyarıların belgelendirme eksikliği ve bazı satır biçimlerinden kaynaklandığı, fakat temel işlevsellik ve yapısal kalite açısından üretimin tatmin edici düzeyde olduğu belirlenmiştir. Pylint skoru 8.21/10 olarak ölçülmüştür.



Şekil 17. Genai Kodunun Statik Analiz Sonuçları

Analiz sırasında alınan temel uyarılar, çoğunlukla docstring eksiklikleri, bazı satırların maksimum uzunluğu aşması (PEP8 E501 uyarısı) ve boşluklarla ilgili biçimlendirme

hatalarından (E302) kaynaklanmıştır. Bu uyarılar işlevselliği doğrudan etkilemeyen, stil rehberleriyle ilgili uyarılar olup basit müdahalelerle giderilebilecek türdedir. Buna ek olarak, flake8 tarafından raporlanan hatalar da yapısal tutarsızlıklar değil; genellikle satır düzeni, girinti ve yorum kullanımı gibi biçimsel unsurlar üzerinedir. Bu durum, GenAI tarafından oluşturulan kodun sadece derlenebilir ve çalıştırılabilir değil, aynı zamanda yazılım kalite kontrol araçlarından da büyük ölçüde olumlu geri dönüş aldığını göstermektedir. Statik analiz sonuçları, GenAI'nin sadece çalışabilir kod üretmediğini, aynı zamanda endüstri standartlarına uyumlu ve sürdürülebilir yazılım çıktıları oluşturduğunu göstermesi açısından önemlidir. Bu sayede, geliştiricilerin oluşturduğu kodun kalite güvence süreçleri hızlanmakta ve insan hatasından kaynaklanabilecek birçok risk minimize edilmektedir.



Şekil 18. Genai ile Yazılmış Kodun Üretim Hızı Avantajı

Bu uygulama örneği açıkça göstermektedir ki, GenAI destekli kod üretimi yalnızca geliştirici üzerindeki operasyonel yükü azaltmakla kalmamakta, aynı zamanda yazılım geliştirme yaşam döngüsünün pek çok aşamasında ölçülebilir verimlilik artışları sağlamaktadır. Geliştirme süresindeki çarpıcı kısalma, sadece zaman tasarrufu değil, aynı zamanda kaynak planlamasında esneklik ve ürünün pazara sunulma süresinde avantaj anlamına da gelmektedir. Özellikle sınırlı zaman aralığında çalışan geliştirme ekipleri için, bu tür üretim modelleri hızlı prototipleme ve anında test edilebilirlik açısından ciddi kazançlar sunmaktadır. Geliştirilen CLI uygulaması, bu kapsamda GenAI'nin gerçek bir yazılım aracına dönüşebilme kapasitesini kanıtlamaktadır. Uygulama; fonksiyonel

modülerlik, kullanıcıdan alınan girdilerin doğruluğunu denetleyen mekanizmalar ve anlık grafiksel veri analizleri ile sadece teknik gereksinimleri karşılamakla kalmamış, aynı zamanda kullanıcı deneyimi (UX) açısından da olumlu değerlendirilmiştir. Kodun okunabilirliği, yapılandırılması ve genişletilebilirliği gibi sürdürülebilirlik kriterleri de göz önünde bulundurulduğunda, GenAI ile üretilen kodların klasik geliştirme yöntemlerine kıyasla daha uyarlanabilir ve bakım dostu olduğu görülmüştür. Bu çalışma, aynı zamanda insan-makine iş birliği potansiyelini de göstermektedir. GenAI, insan yönlendirmesiyle doğru bağlamda çalıştığında, yalnızca sentetik kod üretimi değil; anlamlı, doğru ve işlevsel yazılım çıktıları oluşturabilecek kapasiteye sahiptir. Bu bağlamda, yapay zekânın yazılım üretim süreçlerinde tamamlayıcı bir rol üstlenerek sadece destek değil, karar süreçlerine entegre edilebilecek bir araç olarak konumlandırılması mümkündür.

DÖRDÜNCÜ BÖLÜM: TARTIŞMA VE DEĞERLENDİRME

Yapay zekâ tabanlı kod üretimi, yazılım mühendisliği pratiklerinde köklü bir paradigma değişimini beraberinde getirmiştir. Bu değişim yalnızca üretim hızını artırmakla kalmamakta, aynı zamanda geliştirici deneyimini, yazılım kalitesini, güvenlik algısını ve mesleki etik çerçevesini de dönüştürmektedir. Bu bölümde hem literatürdeki bulgular hem de bu tez kapsamında gerçekleştirilen deneysel analizler ışığında, GenAI destekli kod üretiminin avantajları, sınırlılıkları ve olası etkileri çok boyutlu bir şekilde tartışılacaktır. İlk olarak, üretim sürecindeki hızlanma, modülerlik, yeniden kullanılabilirlik gibi teknik faydalar değerlendirilecektir. Ardından, yapay zekâ destekli üretimin potansiyel zayıflıkları, özellikle bağlamsal eksiklik, güvenlik riskleri ve geliştirici yetkinliğini aşındırma ihtimali ele alınacaktır. Takiben, bu teknolojilerin yazılım geliştirme süreçlerine sağladığı verimlilik katkısı, deneysel veri ve grafiklerle desteklenerek sunulacaktır. Son olarak, yapay zekânın kod üretiminde endüstriyel uygulanabilirliği ile etik ve hukuksal sonuçları değerlendirilecektir. Böylece bu bölüm, GenAI teknolojilerinin yazılım ekosistemine olan bütünsel etkisini ortaya koymayı hedeflemektedir.

4.1. GenAI Tabanlı Kod Üretiminin Güçlü Yönleri

Yapay zekâ destekli kod üretimi, geleneksel yazılım geliştirme süreçlerinde zaman alan birçok adımı otomatikleştirerek geliştiricilere önemli avantajlar sunmaktadır. Son yıllarda geliştirilen büyük dil modelleri (LLM'ler), özellikle GitHub Copilot, OpenAI Codex ve ChatGPT gibi araçlar sayesinde, yazılım üretiminde hem hız hem de modülerlik açısından devrimsel değişiklikler yaratmıştır. En belirgin güçlü yönlerden biri, geliştiricilerin fikirlerini hızla koda dönüştürebilmesidir. Yapay zekâ, sadece birkaç doğal dil açıklaması ile çalışabilir Python, JavaScript veya Java kodu önerebilmekte, hatta birçok durumda bu kodları doğrudan kullanılabilir halde sunmaktadır. Bu durum, özellikle hızlı MVP geliştirme aşamalarında veya hackathon benzeri kısa süreli projelerde önemli bir zaman avantajı yaratmaktadır. Bu çalışmada geliştirilen CLI tabanlı uygulama örneğinde de gözlemlendiği gibi, manuel olarak 25 dakikada yazılabilecek bir sistem, GenAI yardımıyla yaklaşık 1.2 dakikada işlevsel hale gelmiştir. Bu, yaklaşık %95 zaman kazancı anlamına gelmektedir.

AI tarafından üretilen kodlar genellikle fonksiyonel yapılar içinde sunulmakta ve “tek sorumluluk” ilkelerine uygunluk göstermektedir. Bu hem okunabilirliği artırmakta hem de kodun yeniden kullanılabilirliğini ve test edilebilirliğini güçlendirmektedir. Örneğin, veri analizi scriptlerinde “grafik çiz”, “aylık satışları hesapla”, “sipariş iptal oranını hesapla” gibi görevler ayrı fonksiyonlara bölünmüş şekilde sunulmuştur. Geliştirilen prototipte de tüm analiz fonksiyonları ayrı tanımlanmış, menü sistemi aracılığıyla çağrılabilir hale getirilmiştir.

AI modelleri, kodun işlevini açıklayan yorum satırlarını otomatik olarak oluşturabilme yeteneğine sahiptir. Bu özellik, özellikle yeni başlayan geliştiriciler için öğretici nitelikte olup, kodun anlaşılabilirliğini artırmaktadır (ChatGPT Technical Team, 2023). ChatGPT örneğinde, her fonksiyonun üstüne işlevini açıklayan yorumlar yerleştirilmiş, veri girişi, hata kontrolü ve çıktı üretimi gibi adımlar yapılandırılmış biçimde dökümanite edilmiştir. Bu durum, kodun sadece çalışabilir olmasından öte, sürdürülebilir yazılım geliştirme açısından da değer taşır (Zügner, Schmitt ve Günnemann, 2022).

Geliştiriciye farklı çözüm yolları sunabilmesi de AI tabanlı üretimin güçlü yönlerinden biridir. Aynı göreve ilişkin olarak döngü tabanlı, pandas tabanlı veya lambda fonksiyonları ile çözüm önerileri alınabilir. Bu özellik, özellikle en iyi çözümü seçmek veya farklı bakış açılarıyla optimizasyon yapmak isteyen orta ve ileri seviye geliştiriciler için avantaj sağlamaktadır (Allamanis, 2022). GenAI modelleri birçok farklı geliştirme ortamı ve IDE (VS Code, Replit, Jupyter) ile uyumlu şekilde çalışabilmekte, doğrudan kod önerisinde bulunabilmektedir. Bu entegrasyonlar, geliştiricilerin mevcut iş akışlarını bozmadan yapay zekâdan faydalanmalarına imkân tanımaktadır. GitHub Copilot’un aktif kullanıcı sayısının 1 milyonu aşması (Ziegler, Yasin ve Klocke, 2023), bu entegrasyonun başarısını destekleyen somut bir göstergedir.

4.2. AI Tabanlı Kod Üretiminin Zayıf Yönleri

Yapay zekâ (Ai) destekli kod üretimi yazılım geliştiricilere önemli avantajlar sunsa da bu teknolojilerin sınırlamaları ve potansiyel riskleri de göz ardı edilemez. Literatürde yapılan son çalışmalar, GenAI modellerinin güvenlik açıklarına, bağlamdan kopuk önerilere, test kapsamı eksikliklerine ve geliştirici becerilerinin aşınmasına yol açabileceğini göstermektedir. Bu bölümde, söz konusu zayıf yönler detaylı biçimde analiz edilmektedir.

4.2.1. Bağlamdan Kopuk Kod Üretimi

LLM (Large Language Model) tabanlı kod üretim araçları, genellikle geliştiriciden aldığı kısa istemlere (prompt'lara) dayanarak kod önerisi sunar. Ancak bu istemler çoğunlukla yalnızca yerel bağlamı (örneğin bir fonksiyon tanımı veya kısa bir görev tanımı) içerdiğinden, üretilen kod parçaları uygulamanın genel mimarisiyle bütünleşmekte yetersiz kalabilir. Bu bağlamsal sınırlılık, özellikle modüler yapıya sahip büyük ölçekli projelerde sorun teşkil etmektedir.

Yapılan ampirik analizlerde, bu araçların önerdiği kodların; servis yönelimli mimarilerde servis sınırlarını ihlal ettiği, katmanlı yapılarda veriyi doğrudan yanlış bir katmanda işlediği veya MVC mimarilerinde controller–model–view sorumluluklarını karıştırdığı örneklerle ortaya konmuştur. Örneğin, bir kullanıcı kimlik doğrulama sistemiyle entegre olması gereken bir API çağrısında, LLM önerisinin auth mekanizmasını tamamen göz ardı ederek sistem güvenliğini riske attığı gözlemlenmiştir.

Wang ve meslektaşları tarafından gerçekleştirilen kapsamlı bir araştırma, farklı görev tipleri için LLM destekli önerilerin bağlamla tutarlılığını incelemiş ve karmaşık sistemlerde modelin global yazılım mimarisini algılayamadığını doğrulamıştır. Çalışmada, özellikle çok katmanlı uygulamalarda önerilen API çağrılarının %41'inin proje genelindeki iş akışıyla uyumsuz olduğu ve bu kodların entegrasyon sırasında ciddi revizyon gerektirdiği tespit edilmiştir (Wang vd., 2024).

Bu bulgular, LLM tabanlı araçların üretim sürecinde sunduğu hız avantajına rağmen, geliştiricinin hâlâ bağlamsal doğrulama ve mimari uyumluluk denetimi gibi görevleri manuel olarak yerine getirmesi gerektiğini göstermektedir. Özellikle mikroservis tabanlı sistemlerde bileşenler arası etkileşim kurallarına uyum, yalnızca sentaktik doğrulukla değil, semantik uyumla da sağlanmalıdır. Mevcut durumda, LLM'lerin bu semantik bağıntıyı oluşturmakta kısıtlı kaldığı söylenebilir.

4.2.2. Hallucination ve Hatalı Kodlar

“Hallucination” olarak adlandırılan problem, büyük dil modellerinin (LLM) gerçekte var olmayan kütüphaneleri, metodları veya sınıfları önerme eğilimiyle ilgilidir. Bu durum, modelin eğitim verilerinden öğrendiği kalıpları bağlamdan kopuk veya aşırı

genelleştirerek üretmesiyle ortaya çıkar. Kod üretimi bağlamında hallucination, yalnızca sentaktik hatalardan ibaret olmayıp, aynı zamanda ciddi mantıksal hataları ve üretim ortamında çalıştırılmayan kod parçalarını da içerebilir.

Özellikle yeni başlayan veya yeterli teknik altyapıya sahip olmayan geliştiriciler için bu sorun daha riskli hale gelmektedir. Kullanıcı, AI tarafından önerilen bir fonksiyonun veya kütüphanenin var olduğunu varsayarak bu kodu üretim ortamına entegre edebilir; bu da hem zaman kaybına hem de güvenlik, performans ve işlevsellik açısından olumsuz sonuçlara neden olabilir. Örneğin, flask_ui_plus, deep_nlp_engine, predictUserRiskScore() gibi var olmayan bileşenlerin önerildiği pek çok örnek, GitHub Copilot'un canlı kullanım verilerinde belgelenmiştir.

Fu et al. Tarafından gerçekleştirilen deneysel bir çalışmada, Copilot'un önerdiği 1.692 kod snippet'i incelenmiş ve bunların %32.8'inin Python, %24.5'inin JavaScript kodlarında en az bir güvenlik açığı içerdiği tespit edilmiştir. Dahası, bu önerilerin 38 farklı CWE (Common Weakness Enumeration) kategorisi ile eşleştiği, yani önerilen kodların çok çeşitli bilinen güvenlik zaaflarını içerdiği belirlenmiştir (Fu vd., 2023).

Bu durum sadece güvenlik açısından değil, yazılımın bakım maliyetleri açısından da önemli bir tehdit oluşturmaktadır. Hallucination içeren kodlar, sıklıkla proje içinde tanımlanmamış bağımlılıklar içerdiğinden, derleme hatalarına, modül çatışmalarına ve runtime hatalarına yol açar. Özellikle büyük ölçekli projelerde, bu tarz hataların tespisi ve giderilmesi karmaşılaşmakta, geliştirici zamanını verimsiz kullanmasına neden olmaktadır. Hallucination sorunu GenAI destekli kod üretiminin en kritik zayıflıklarından biridir. Üretilen kodun yalnızca yüzeysel olarak derlenebilmesi değil, aynı zamanda çalışma zamanında güvenli ve tutarlı davranabilmesi gerektiği göz önüne alındığında, bu tarz sentetik üretimlerin mutlaka insan denetimi ve doğrulama süzgecinden geçirilmesi gerekmektedir.

4.2.3. Güvenlik Açıkları

Yapay zekâ destekli kod üretiminde güvenlik zaafları, en çok tartışılan ve üzerinde çalışılan konulardan biridir. LLM tabanlı sistemlerin, özellikle kodun anlamını yalnızca istatistiksel örüntüler üzerinden üretmesi; güvenlik prensiplerine dayalı sistematik analiz veya tehdit modellemesi yapamamasına neden olmaktadır. Bu durum, AI tarafından

üretile kodların çoğu zaman potansiyel saldırılara açık, güvenlik kontrollerinden yoksun ve güvenli yazılım geliştirme ilkelerine aykırı olmasına neden olmaktadır. Tipik örnekler arasında, kullanıcı girişlerinin doğrulanmaması (input validation eksikliği), sabitlenmiş (hard-coded) API anahtarları veya şifrelerin kodda açık olarak yer alması, dosya erişiminde güvenlik kontrolü olmaksızın open () kullanımı ve SQL injection'a açık sorgu kalıpları gösterilebilir. LLM'ler genellikle bu kalıpları eğitim verilerinde sıkça karşılaştıkları şekliyle yeniden üretir; ancak bu kalıpların bağlamdan bağımsız önerilmesi, tehlikeli sonuçlar doğurabilir.

Pearce ve meslektaşları tarafından yapılan yüksek hacimli bir deneysel çalışmada, GitHub Copilot'un önerdiği 1.689 kod örneği detaylı biçimde analiz edilmiş ve bu örneklerin yaklaşık %44'ünün en az bir güvenlik açığı içerdiği belirlenmiştir (Pearce, Ahmad, Tan ve Dolan-Gavitt, 2022). Bu açıklar OWASP ve CWE kategorilerine göre sınıflandırılmış; önerilerin %40'ından fazlasının CWE-89 (SQL Injection), CWE-798 (Hardcoded Credentials) ve CWE-20 (Improper Input Validation) gibi yaygın ve kritik sınıflara uyduğu gösterilmiştir.

Ek olarak, Siddique ve meslektaşları tarafından geliştirilen SecurityEval adlı benchmark seti, farklı AI modellerinin güvenlik açığı üretme eğilimlerini karşılaştırmalı olarak değerlendirmiştir. Bu benchmark seti, bir dizi görev üzerinden LLM'lerin ürettiği kodların statik analiz araçlarıyla test edilmesi esasına dayanmakta olup; özellikle Codex, Copilot ve Code Llama gibi modellerde benzer düzeyde güvenlik eksiklikleri rapor edilmiştir (Siddique, Dey ve Takabi, 2023). Kodun görünüşte işlevsel olması, kullanıcıda sahte bir güven duygusu yaratmakta; fakat arka planda ciddi istismar vektörleri barındırabilmektedir. Bu tür güvenlik açıklarının özellikle "copy-paste" eğilimli kullanım senaryolarında doğrudan üretim ortamına aktarılması, siber saldırılara karşı sistemin savunmasız hale gelmesine neden olabilir. LLM tarafından üretilmiş bir kodun, dış kaynaklı verileri sanitize etmeden işlemeye başlaması, sistemde veri sızdırma, yetkisiz erişim veya uzaktan kod çalıştırma gibi tehlikeleri tetikleyebilir. Ayrıca, bu açıkların otomatik test araçlarıyla yakalanması her zaman mümkün olmayabilir; çünkü saldırı yüzeyi yalnızca mantıksal hatalarla değil, kontektüel eksikliklerle de şekillenir. Dolayısıyla, güvenli yazılım geliştirme bağlamında GenAI destekli üretim araçları, bir "asistan" gibi kullanılmalı; ancak çıktıları doğrudan güvenilir kabul edilmemelidir.

Güvenlik kontrolleri, geliştirici gözetimi, statik analiz ve manuel inceleme gibi geleneksel pratiklerle desteklenmedikçe, üretkenliğin sağladığı hız artışı, sistem bütünlüğü ve veri gizliliği açısından risk teşkil edebilir.

4.2.4. Test Kapsamı Eksikliği

Kodun sadece çalıştırılabilir olması, onun doğru ve güvenilir olduğu anlamına gelmez. Yazılım geliştirme süreçlerinde işlevsel doğruluğu garanti altına almanın temel araçlarından biri test kapsamıdır. Test kapsamı; birim testleri, uç durumları (edge cases), hata toleransını, istisna yönetimini ve sistemsel entegrasyonları içermelidir. Ancak günümüzde GenAI sistemlerinin ürettiği kodların çoğu, test stratejileri açısından oldukça sınırlıdır ve kapsamlı bir doğrulama mantığından yoksundur.

LLM'lerin (Large Language Models) temel amacı, dil örüntülerine dayalı olarak mantıklı görünen bir kod parçası üretmektir. Ancak bu modeller, yazdıkları fonksiyonların ne tür sınır durumlar içereceğini, hangi hata koşullarına maruz kalabileceğini, ya da hangi girdilerle sistemin çökeceğini doğrudan öngöremezler. Sonuç olarak, üretilen kodlar çoğunlukla “ideal senaryoya” göre optimize edilmiş olup; beklenmeyen girişler, boş veriler, bölme hataları, tür uyumsuzlukları gibi uç koşullara karşı savunmasızdır. Test üretimi konusunda GPT-4 ile yapılan deneysel bir çalışmada, otomatik oluşturulan test fonksiyonlarının yalnızca %45'inin beklendiği gibi çalıştığı, %47'sinin ise assertion error ile başarısız olduğu ve geri kalan %8'lik kısmın ise başarısızlık nedeninin açık şekilde saptanamadığı rapor edilmiştir (GPT-4 Evaluation Team, 2023). Bu sonuç, AI'nin test fonksiyonu üretme yetisinin oldukça yüzeysel ve dengesiz olduğunu ortaya koymaktadır. Gerçek dünyada, test yazımı sadece doğruluk denetimi değil; aynı zamanda yazılımın sürdürülebilirliğini, genişletilebilirliğini ve güvenliğini sağlamanın da bir yoludur. Testsiz kodun üretim ortamında bırakılması, hata tespiti gecikmelerine, kullanıcı deneyimi bozulmalarına ve güvenlik açıklarına zemin hazırlar. Bu nedenle GenAI'nin önerdiği kodlar, ne kadar ikna edici görünürse görünsün, her zaman manuel test yazımı, static analysis ve gerektiğinde fuzz testing gibi tekniklerle desteklenmelidir.

4.2.5. Beceri Aşınması

Yapay zekâ destekli kod üretim araçlarının gelişimi, yazılım geliştirme süreçlerinde ciddi bir dönüşüm yaratırken, beraberinde eğitimsel ve bilişsel düzeyde bazı riskleri de gündeme getirmiştir. Bu risklerin başında, geliştiricilerin temel yazılım mühendisliği becerilerinde meydana gelen aşınma (skill erosion) olgusu yer almaktadır. Özellikle kod üretiminde otomasyonun artması, geliştiricilerin problem çözme, hata ayıklama (debugging), algoritmik düşünme ve kod mimarisi kurma gibi bilişsel faaliyetlerden uzaklaşmasına yol açabilmektedir. Yazılım mühendisliği eğitiminin temel amacı, bireyleri sistematik düşünmeye, algoritma kurmaya, veri yapılarıyla çözüm üretmeye ve yazılımın farklı bileşenleri arasında doğru soyutlamalar yapabilmeye teşvik etmektir. Ancak LLM tabanlı sistemlerin giderek artan kullanımı, kullanıcıların sadece doğal dilde bir istem yazarak karmaşık fonksiyonlar elde etmesine olanak sağlamakta ve "üretmekten çok tüketmeye" dayalı bir yazılım kültürünün oluşmasına zemin hazırlamaktadır.

Denny ve Luxton-Reilly tarafından gerçekleştirilen deneysel bir çalışmada, GitHub Copilot kullanan öğrencilerin kod üretim hızlarının arttığı ancak aynı zamanda temel kavramlara yönelik bilgi düzeylerinin anlamlı şekilde düştüğü gözlemlenmiştir. Katılımcıların büyük çoğunluğu, AI tarafından üretilen kodları doğrudan kabul etmiş; algoritmaların neden o şekilde çalıştığı, olası alternatif çözümler veya optimizasyon teknikleri üzerine düşünmemiştir (Denny ve Luxton-Reilly, 2023).

Bu durum, kısa vadede verimlilik hissi yaratsa da uzun vadede “kavramsal tembellik” (conceptual inertia) etkisi doğurabilir. Yani geliştiriciler, daha önce bireysel çaba ile ürettikleri çözümleri artık sorgulamak yerine otomatikleştirilmiş çıktıların doğruluğuna koşulsuz güvenmeye başlarlar. Bu da hem eğitim hem endüstri ortamlarında yazılım mühendisliği niteliğini düşürebilir. Ayrıca bu beceri aşınması, yalnızca bireysel gelişimi değil; takım temelli geliştirme süreçlerini de etkileyebilir. Kod gözden geçirme (code review), refactoring, test tasarımı gibi süreçlerde yapay zekâya aşırı bağımlı geliştiriciler, diğer ekip üyelerinin ürettiği kodları anlamakta zorlanabilir veya kritik hataları fark edemeyebilir. Sonuç olarak, bu durum kolektif yazılım kalitesinin düşmesine neden olur. Literatürde buna benzer bir fenomen “automation complacency” olarak tanımlanmıştır. Bu terim, otomasyon sistemlerine aşırı güvenen bireylerin kendi kontrol ve karar verme yetilerini zamanla kaybetmelerini tanımlar. Yazılım geliştiriciler özelinde bu, yalnızca

retkenlik deęil, mesleki yetkinlięin srdrlebilirlięi aısından da bir tehlike teękil eder. GenAI sistemlerinin yazılım eęitiminde veya endstride kullanımı sırasında rehberli ve denetimli bir yaklařım benimsenmelidir. zellikle ęrenme srelerinde, yapay zekâdan alınan nerilerin neden doęru (ya da yanlıř) olduęunun aıklanması, geliřtiricilerin biliřsel geliřimini destekleyecek tartıřmalı uygulamalarla desteklenmelidir. Bu yaklařım yalnızca kod retiminde deęil, mesleki yeterlilik ve mhendislik disiplini aısından da dengeleyici bir rol oynayacaktır.

4.2.6. retkenlik ve Kalite eliřkisi

GenAI tabanlı kod retim sistemlerinin yazılım geliřtirme srecindeki en belirgin katkılarından biri, kod yazım sresini ciddi lde kısaltmalarıdır. Geliřtiriciler, geleneksel olarak dakikalar veya saatler srebilecek modlleri birkaç satırlık doęal dil aıklamayla saniyeler iinde retebilmektedir. Bu durum hem bireysel hem de takım dzeyinde geliřtirme hızı (velocity) metrięini olumlu ynde etkilemektedir. Ancak artan retkenlięin, doęrudan kod kalitesi ile aynı dzlemde ilerlemedięi ynnde nemli bulgular da literatrde yer almaktadır.

Ziegler ve meslektařları tarafından gerekleřtirilen ok yntemli bir alıřmada, Copilot kullanan geliřtiricilerin daha kısa srede daha fazla kod satırı rettikleri; ancak bu kodların genellikle daha fazla refactoring, dzeltme (bug fixing) ve yapılandırma gerektirdięi tespit edilmiřtir (Ziegler, Yasin ve Klocke, 2023). Katılımcıların oęu, retim sresinden tasarruf etse de yazdıkları kodun bakım srecinde zorluk yařandığını ve nerilen zmlerin yazılım mimarisine uydurulması iin ekstra aba harcandığını belirtmiřtir.

Bu bulgular, “hız-gvenilirlik paradoksu” olarak adlandırılabilir bir eliřkiye iřaret etmektedir. Kod nerisinin hızlı olması, onun doęru, srdrlebilir ve gvenli olduęu anlamına gelmemektedir. Aksine, birok durumda GenAI tarafından retilen kodlar:

- Yorum ve dkmantasyondan yoksundur,
- Yazılım mimarisiyle tam uyumlu deęildir,
- Edge case’leri test etmeden varsayımlara dayanır,
- Tasarım ilkelerine (rneęin DRY, SOLID) uygunluk gstermez.

Bu durum, uzun vadede yazılım projelerinde teknik borcun (technical debt) artmasına neden olur. Geliştirici ilk etapta kazandığı süreyi, sonradan oluşan hataları düzeltmek, modülleri yeniden yapılandırmak veya kodu derlemek için zaman harcamak zorunda kalabilir. Bu da gerçek üretkenliğin ölçümünü karmaşıktırır.

Ayrıca, AI önerilerinin kalitesi kullanıcı deneyimine göre dalgalanabilir. Eğitim verisi kalitesine, prompt açıklığına, görev türüne ve geliştirici uzmanlığına göre aynı model çok farklı kalite seviyelerinde önerilerde bulunabilir. Bu da kaliteyi tahmin edilebilirlikten uzaklaştırır. Bu nedenle üretkenliği sadece kod satırı sayısı veya süre ile değil, üretim-sonrası efor, bakım kolaylığı, yeniden kullanım oranı ve hata sayısı gibi daha geniş yazılım kalitesi metrikleri ile değerlendirmek gerekmektedir. Gerçek üretkenlik, “minimum çabayla sürdürülebilir sonuç üretme” ilkesine dayanmalıdır.

Literatürde bazı çalışmalar, üretkenliği artıran AI sistemlerinin aynı zamanda geliştiricilerin “geliştirme-öğrenme” dengesini bozduğunu da vurgulamaktadır. Geliştiriciler daha az hata yapıyor gibi görünse de kodun neden o şekilde çalıştığını anlamadan ilerlediklerinde sistemin karmaşıklığı arttıkça sorun tespiti ve müdahale yetkinlikleri zayıflamaktadır. Bu durum, yazılım kalitesini sadece performans değil, anlaşılabilirlik, doğrulanabilirlik ve sürdürülebilirlik gibi yönlerden de etkiler. Böyle sorunlara dolayısıyla, GenAI tabanlı üretkenliğin etkileyici görünümü, yazılım kalitesi ile dengelenmediğinde uzun vadeli risklere yol açabilir. Yazılım mühendisliğinde hızlı çözüm üretimi ile güvenli, sürdürülebilir sistem tasarımı arasında kurulacak denge, GenAI’nin gerçek potansiyelinin sorumlu şekilde kullanılması açısından kritik önemdedir.

4.3. Güvenlik ve Etik Sorunlar

Yapay zekâ tabanlı kod üretimi, geliştiricilere zaman tasarrufu ve üretkenlik artışı sağlarken, beraberinde ciddi güvenlik ve etik sorumluluklar da doğurmaktadır. Bu sorumluluklar hem üretilen kodun güvenli çalışması hem de hukuki, lisanssal ve ahlaki standartlara uygunluğu açısından değerlendirilmelidir. AI sistemlerinin önerdiği kod parçalarının önemli bir kısmı, hardcoded (sabitlenmiş) şifreler, API anahtarları veya kullanıcı kimlik bilgileri içerebilir. Bu tür öneriler, kötü niyetli yazılımların kolayca erişebileceği açıklıklara yol açmakta, özellikle deneyimsiz geliştiricilerin bu kodları

doğrudan uygulamaya alması durumunda ciddi siber güvenlik riskleri oluşmaktadır. Örneğin, Copilot tarafından önerilen bazı kodlarda ``password = "admin123"'` gibi ifadeler sıkça rastlanmaktadır. CWE (Common Weakness Enumeration) veritabanı bu gibi örnekleri yüksek riskli kategorilerde sınıflandırmaktadır.

Benzer şekilde, AI tarafından üretilen kodlarda input validation (girdi doğrulama) eksiklikleri veya SQL injection gibi saldırılara açık ifadeler sıklıkla tespit edilmektedir. Bu durum, yalnızca bireysel projelerde değil, büyük ölçekli kurumsal sistemlerde de risk oluşturmaktadır. Siddique et al. Tarafından geliştirilen SecurityEval benchmark seti, bu tür zafiyetleri analiz etmek için kullanılmakta olup, birçok AI modelinin önerdiği kodların güvenlik testlerinden geçemediğini ortaya koymuştur.

Etik açıdan bakıldığında, AI modellerinin eğitildiği veri kümeleri açık kaynaklı projelerden, forumlardan veya bloglardan elde edilmektedir. Bu durum, modellerin eğitim sırasında telifli içeriklere maruz kalmasına ve bazı kod parçalarının bu içeriklere çok benzer şekilde yeniden üretilmesine yol açmaktadır. Bunun sonucunda, lisans hakları ihlal edilebilmekte, hatta fark edilmeden GPL veya MIT lisanslı kodlar, ticari projelere entegre edilebilmektedir. Bu risk hem bireysel geliştiriciler hem de kurumsal firmalar açısından hukuki yaptırımlara yol açabilir.

Ayrıca, AI'nin ürettiği kodlarda görülen potansiyel hatalarda sorumluluğun kime ait olduğu sorusu da etik bir tartışma konusudur. AI tarafından oluşturulan bir fonksiyon, canlı sistemde hataya veya veri sızıntısına neden olduğunda, bu hatanın yükü geliştiriciye mi, AI sistemine mi, yoksa AI hizmet sağlayıcısına mı aittir? Bu sorunun cevabı henüz küresel hukuk sistemlerinde netlik kazanmamıştır. Ancak yazılım lisans sözleşmeleri genellikle bu sorumluluğun kullanıcıda olduğunu belirtmektedir. Örneğin, GitHub Copilot'un kullanım koşulları, önerilen kodların doğruluğunu garanti etmemekte ve nihai sorumluluğun geliştiriciye ait olduğunu vurgulamaktadır.

Bu çerçevede, AI destekli kod üretiminin güvenli ve etik şekilde kullanılması için bazı önlemler alınması önerilmektedir. Bunların başında, AI tarafından önerilen kodların manuel olarak gözden geçirilmesi, statik analiz araçlarıyla güvenlik testlerinden geçirilmesi ve varsa lisans uyumluluğunun kontrol edilmesi gelmektedir. Ayrıca, Copilot gibi sistemlerle birlikte kullanılan kod izleme araçları (code linting, license checker) geliştiricilere önemli ölçüde destek sağlayabilir. Bu bölümde altbaşlıkta Lisans

problemleri, Adversarial Saldırıları ve Model Manipülasyonu ve Veri Güvenliği Üzerine Vaka İncelemesi (Case Study) üzerinde geneler bilgilerde aşağıda yer almaktadır.

4.3.1. Lisans Problemleri

Üretken yapay zekâ sistemleri (GenAI), doğal dil ve kod üretimi gibi görevlerde yüksek doğruluk ve hız sağlamak amacıyla büyük hacimli veri kümeleri üzerinde eğitilmektedir. Bu veri kümeleri arasında özellikle GitHub, StackOverflow, Bitbucket gibi platformlardan alınmış açık kaynak kod havuzları önemli yer tutar. Ancak bu kaynakların büyük bir kısmı, farklı türde telif lisansları altında dağıtılmaktadır. Yaygın lisans türleri arasında GNU General Public License (GPL), MIT License, Apache License 2.0 ve BSD License yer almaktadır. Bu lisanslar, kodun yeniden kullanımı ve dağıtımı için belirli kuralları zorunlu kılar; örneğin GPL, türetilmiş ürünlerde aynı lisansın korunmasını şart koşar.

Yapay zekâ modelleri ise bu lisansları ayırt etmeden, milyonlarca satır kodu işleyerek istatistiksel örüntüler öğrenir ve daha sonra kullanıcı istemlerine (prompts) yanıt olarak benzer çıktılar üretir. Bu süreçte, AI tarafından önerilen kodlar sıklıkla lisans bilgisi içermez ve kullanıcı, çıktının orijinal kaynağını bilmeden bu kodu projelerine entegre edebilir. Sonuç olarak, yasal olarak koruma altında olan kod yapılarının farkında olmadan yeniden üretilmesi, ciddi telif ihlali riski doğurur.

Bu sorun 2021 yılında, Free Software Foundation (FSF) ve Software Freedom Conservancy (SFC) gibi kurumların GitHub Copilot’a yönelttiği eleştirilerle geniş çapta gündeme gelmiştir. Yapılan bağımsız analizlerde, Copilot’un eğitim verisinde yer alan GPL lisanslı projelerden bazı fonksiyonları neredeyse birebir önerdiği tespit edilmiştir. Özellikle Python, C ve JavaScript tabanlı projelerde yapılan karşılaştırmalarda, kod benzerlik oranının %90’ın üzerinde olduğu örnekler bulunmuştur. FSF bu durumu “AI destekli otomatik telif ihlali” olarak tanımlamış ve bu tür sistemlerin çıktılarının da lisans denetimine tabi tutulması gerektiğini savunmuştur.

Bu problem yalnızca teorik değil, aynı zamanda pratik sonuçlar da doğurabilir. Örneğin, bir geliştirici Copilot önerisini kendi ticari projesinde kullandığında ve bu öneri aslında GPL lisanslı bir projeden alınmışsa, ticari ürünün de GPL kapsamına girmesi gerekir. Bu

hem lisans uyumsuzluğu hem de ticari zarar anlamına gelir. Dahası, telif hakkı sahipleri, izinsiz kullanım durumlarında yasal yaptırım uygulama hakkına sahiptir.

Hukuki boyutun yanı sıra, bu durum etik sorumluluklar açısından da tartışmalıdır. AI sistemleri, geliştirici emeğini yeniden paketleyip başka kullanıcılara sunmaktadır. Ancak bu süreçte orijinal geliştiricinin adı anılmamakta, katkısı tanınmamakta ve çıktının sahibine dair belirsizlik ortaya çıkmaktadır. Bu durum, “yazılımsal intihal” (software plagiarism) olarak da adlandırılabilir.

Önerilen çözümler arasında:

- GenAI sistemlerinin lisans bilinçli modelleme yetenekleriyle geliştirilmesi,
- Çıktılarda otomatik lisans tahmini veya orijin izleme (provenance tracking) sistemlerinin eklenmesi,
- Kullanıcıların çıktıları statik lisans denetleyicileri ile kontrol etmesi (örneğin: [FOSSology](#), [Licensee](#)),
- GenAI sağlayıcılarının eğitim verilerini lisans türlerine göre sınıflandırması yer almaktadır.

Lisans problemleri, üretken yapay zekâ destekli kod üretiminin hem teknik hem de etik yönünü etkileyen karmaşık bir konudur. Özellikle profesyonel ve ticari projelerde, AI tarafından önerilen kodların lisans geçmişi ve yeniden kullanım hakları dikkatle değerlendirilmelidir. Aksi takdirde, geliştiriciler istemeden ciddi telif ihlalleri ve yasal yaptırımlarla karşı karşıya kalabilirler.

4.3.2. Adversarial Saldırıları ve Model Manipülasyonu

Yapay zekâ tabanlı kod üretim sistemlerinin güvenliğini tehdit eden önemli risklerden biri, modelin dış istemler aracılığıyla bilinçli şekilde manipüle edilmesidir. Bu tür tehditler literatürde genellikle “adversarial prompt” veya “prompt injection” olarak adlandırılmaktadır. Söz konusu saldırılar, büyük dil modellerinin bağlamsal esnekliğinden faydalananak, modelin kasıtlı olarak zararlı, tehlikeli veya güvenlik açığı barındıran kodlar üretmesini hedeflemektedir. Adversarial prompt'lar genellikle masum görünümlü komutlar içerir; ancak modelin bağlamı hatalı yorumlaması sonucu zararlı çıktı üretilmesine neden olurlar. Örneğin, “Kullanıcı verisini doğrulamadan SQL

tablosuna ekle” şeklinde verilen bir istem, model tarafından SQL injection’a açık bir sorguyla cevaplanabilir. Tipik bir örnek şu şekildedir:

```
```python
query = f'SELECT * FROM users WHERE email = '{email}'
```
```

Bu yapı, girdi doğrulaması yapılmadan SQL sorgusuna dahil edilen kullanıcı verisi nedeniyle veri sızdırma veya sistem manipülasyonu gibi ciddi güvenlik açıklarına yol açabilir. Bu tür saldırılar yalnızca bireysel kullanım senaryolarında değil, çok adımlı diyaloglara sahip etkileşimli yapay zekâ sistemlerinde de tehlike oluşturmaktadır. Özellikle “multi-turn” mimariye sahip modellerde, saldırgan önce masum komutlar verir, ardından devam eden istemlerde zararlı yönlendirme yaparak sistemin davranışını bozar. Bu, modelin içsel bağlamını etkileyerek filtreleme sistemlerini aşmasını sağlayabilir. Son yıllarda yapılan çeşitli akademik çalışmalar, bu tür saldırılara karşı büyük modellerin yeterince dayanıklı olmadığını ortaya koymuştur. 2023 yılında Anthropic tarafından yapılan sistematik bir adversarial test çalışmasında, filtrelemeli modellerin dahi %12 ila %23 oranında saldırıya açık çıktılar ürettiği rapor edilmiştir. Benzer şekilde OpenAI ve Google DeepMind da bu tür tehditleri azaltmak amacıyla reinforcement learning with human feedback (RLHF) ve adversarial fine-tuning gibi teknikler üzerinde çalışmaktadır (Ganguli, Papadimitriou, Askell, Nanda, Hernandez, Henighan ve Schulman, 2023). Bu bağlamda geliştirilen savunma yöntemleri arasında istem filtreleme (prompt classification), statik analiz araçları (örneğin: Bandit, SonarQube), kontekst izleme ve geliştirici uyarı sistemleri (Copilot’un güvenlik uyarıları gibi) öne çıkmaktadır. Ancak bu yöntemlerin başarı oranı henüz tam anlamıyla stabil değildir ve manuel denetimin hâlâ önemli olduğu görülmektedir. Adversarial saldırılar ve model manipülasyonu, sadece teknik bir zafiyet değil; aynı zamanda üretken yapay zekânın güvenilirliği ve sorumlu kullanımı açısından stratejik bir problem olarak ele alınmalıdır. Kod üretimi bağlamında bu tehditlerin ciddiyeti, geliştiricilerin yalnızca üretim sürecinde değil, istem tanımlarken de dikkatli olmaları gerektiğini ortaya koymaktadır. Güvenlik odaklı tasarım, yapay zekâ destekli geliştirme süreçlerinin ayrılmaz bir parçası hâline gelmelidir.

4.3.3. Veri Güvenliği Üzerine Vaka İncelemesi

2023 yılında bir fintech girişiminde, yeni işe alınan bir yazılım geliştirici, müşteri işlem geçmişlerini analiz eden bir modül üzerinde çalışmaya başlar. Geliştirici, GitHub Copilot gibi yapay zekâ destekli kod tamamlama araçlarını kullanarak, zaman kazanmayı ve üretkenliği artırmayı hedefler.

Ancak, geliştirici tarafından doğrudan kabul edilen bazı kod önerileri, aşağıdaki gibi ciddi güvenlik açıkları içermektedir (Jovanović ve Campbell, 2022: 110):

```
password = "admin123"
```

```
file = open ("user_logs.csv", "w")
```

```
query = f"SELECT * FROM users WHERE email = '{email}'"
```

Bu kod parçacıkları, aşağıdaki güvenlik zaafiyetlerini barındırmaktadır:

- Sabitlenmiş (Hardcoded) Şifre: Şifrelerin kod içerisinde sabitlenmesi, özellikle kaynak kodun sızması durumunda ciddi güvenlik riskleri oluşturur.
- Dosya Güvenliği İhlali: Kullanıcı verilerinin şifrelenmeden düz metin olarak dosyaya yazılması, veri gizliliğini tehlikeye atar ve veri koruma yasalarına aykırıdır.
- SQL Injection Riski: Kullanıcı girdisinin doğrudan SQL sorgusuna eklenmesi, kötü niyetli kullanıcıların veri tabanına yetkisiz erişim sağlamasına olanak tanır.

Sistem canlıya alındıktan sonra gerçekleştirilen bağımsız bir güvenlik denetimi sırasında bu açıklar tespit edilir. Geliştirici, bu kod parçacıklarının Copilot tarafından önerildiğini belirtir. Şirket hem yasal hem de etik sorumluluklar açısından baskı altına girer ve kod tabanını yeniden yapılandırmak zorunda kalır. Bu vaka, yapay zekâ destekli kod üretim araçlarının güvenlik risklerini gözler önüne sermektedir. 2021 yılında New York Üniversitesi tarafından yapılan bir çalışmada, GitHub Copilot'un güvenlik açısından kritik görevlerde önerdiği kodların yaklaşık %40'ının güvenlik açıkları içerdiği tespit edilmiştir (Pearce, Ahmad, Tan ve Dolan-Gavitt, 2022).

Benzer şekilde, 2023 yılında yapılan bir başka çalışmada, Copilot tarafından üretilen Python kod parçacıklarının %32.8'inin ve JavaScript kodlarının %24.5'inin güvenlik zaafiyetleri barındırdığı belirlenmiştir (Fu, Liang, Tahir, Li, Shahin, Yu ve Chen, 2023). Bu bulgular, yapay zekâ destekli kod üretim araçlarının, özellikle deneyimsiz geliştiriciler

tarafından kullanıldığında, ciddi güvenlik riskleri oluşturabileceğini göstermektedir. Bu vaka, yapay zekâ destekli kod üretiminin, yalnızca teknik değil, aynı zamanda etik bir sorumluluk olduğunu açıkça ortaya koymaktadır. Geliştiricilerin, AI tarafından önerilen kodları körü körüne kabul etmek yerine her bir öneriyi dikkatlice incelemeleri ve güvenlik testlerinden geçirmeleri gerekmektedir. Ayrıca, şirketlerin, AI destekli kod üretim araçlarını kullanmadan önce, geliştiricilere güvenli kodlama uygulamaları konusunda eğitimler vermeleri ve kod inceleme süreçlerini sıkılaştırmaları önemlidir. Sonuç olarak, yapay zekâ destekli kod üretim araçları, yazılım geliştirme süreçlerini hızlandırma potansiyeline sahip olsa da güvenlik ve etik sorumluluklar göz ardı edilmemelidir.

4.4. Endüstride Uygulanabilirlik

Üretken yapay zekâ (GenAI) sistemlerinin yazılım geliştirme süreçlerine entegrasyonu, yalnızca teorik düzlemde kalmamakta; aynı zamanda farklı sektörlerde pratik olarak uygulanabilirliği kanıtlanmış çözümler sunmaktadır. GitHub Copilot, Amazon CodeWhisperer ve Replit Ghostwriter gibi öncü araçlar, modern yazılım mühendisliği pratiklerine entegre edilerek kod üretiminde hız, doğruluk ve tekrar eden işlemlerde otomasyon avantajları sağlamaktadır. Bu modellerin sunduğu çıktıların üretim ortamlarında kullanılabilmesi geliştirici deneyimi düzeyinden bağımsız olarak, geliştirme sürecinin genel verimliliğini artırmakta ve yazılım yaşam döngüsünde ciddi bir dönüşüm yaratmaktadır. Bu araçlar yalnızca temel kod yazımı işlevlerini yerine getirmekle kalmaz; aynı zamanda yazılımın çevresel bileşenlerini oluşturan belgelendirme, test üretimi, hata ayıklama senaryoları ve kullanıcıya yönelik API kullanım örnekleri gibi alanlarda da aktif olarak görev üstlenmektedir. Özellikle açık kaynaklı projelerde, README dosyalarının otomatik oluşturulması, fonksiyonların içeriğinin anlamlı açıklamalarla belgelenmesi ve kodun yorumlanabilirliğinin artırılması gibi faaliyetler, GenAI modelleriyle yüksek doğrulukla gerçekleştirilebilmektedir. Bu sayede, geliştiriciler yalnızca kod üretimi değil, aynı zamanda yazılımın sürdürülebilirliğini destekleyen yardımcı görevlerde de zaman tasarrufu sağlamaktadır. Buna ek olarak büyük kurumsal projelerde GenAI araçlarının entegrasyonu, proje yönetiminde maliyet düşürme ve teslim sürelerini kısaltma açısından kritik katkılar sunmaktadır. Örneğin GitHub Copilot'un Microsoft destekli "AI Developer Productivity" raporuna göre, kurumsal düzeyde yapılan analizlerde geliştiricilerin %55'inin Copilot kullanımıyla günlük iş akışlarında ciddi verimlilik artışı sağladığı,

%33'ünün ise hata oranlarında düşüş bildirdiği ortaya konmuştur. Bu veriler, AI tabanlı destek sistemlerinin sadece bireysel değil, aynı zamanda takım ve organizasyon seviyesinde de etkin sonuçlar verdiğini göstermektedir.

Benzer şekilde Amazon CodeWhisperer, AWS hizmetleriyle bütünleşmiş çalışarak özellikle bulut güvenliği, erişim kontrolü (IAM) ve veri işleme senaryolarında güvenli kod örnekleri üretebilmekte ve geliştiricilere potansiyel riskleri önceden bildirebilmektedir. Replit Ghostwriter ise tarayıcı tabanlı bir geliştirme ortamı sunarak düşük donanıma sahip kullanıcılar için bile güçlü bir yapay zekâ asistanı görevini üstlenmektedir. Bu yönüyle GenAI sistemleri, donanım bağımlılığını da azaltmakta ve yazılım geliştirmeyi daha demokratik bir hale getirmektedir. GenAI'nin sektörel uygulamaları da dikkat çekici çeşitliliğe sahiptir. Finans sektöründe kullanılan GenAI sistemleri, algoritmik ticaret stratejilerinin oluşturulmasından düzenleyici uyumluluk kontrolüne kadar pek çok noktada otomasyon sağlamaktadır. Sağlık sektöründe ise, hasta verilerinden analiz çıkarımı, klinik karar destek sistemleriyle entegrasyon ve tıbbi dokümantasyonun hazırlanması gibi görevlerde GenAI çözümleri kullanılmaktadır. E-ticaret alanında ise satış tahminleme, envanter optimizasyonu, müşteri davranış analizi ve kişiselleştirilmiş öneri sistemleri üretken yapay zekâ araçlarıyla güçlendirilmiştir.

Microsoft ve GitHub iş birliğiyle yayımlanan 2023 AI-Powered Developer Productivity Report, GenAI araçlarının gerçek dünya geliştirici davranışları üzerindeki etkisine dair kapsamlı veriler sunmaktadır. Bu rapora göre GitHub Copilot kullanıcılarının %55'i, araç sayesinde günlük iş akışlarının genel verimliliğinde belirgin bir artış yaşadıklarını belirtmiştir. Özellikle zaman açısından yoğun görevlerde, Copilot'un otomatik öneri sistemlerinin “boş ekran sendromu” olarak adlandırılan üretkenliğe başlangıç yaşanmasını büyük ölçüde engellediği ifade edilmektedir. Geliştiriciler, Copilot'un önerdiği başlangıç fonksiyonları ve yapı kalıpları sayesinde kodlama sürecine daha hızlı uyum sağlamakta, böylece üretkenlik sürekliliği sağlanmaktadır. Bu özellik özellikle görev önceliklendirmesi, kısa zamanlı “sprint” dönemleri ve hackathon gibi yüksek yoğunluklu geliştirme ortamlarında belirgin faydalar yaratmaktadır. Ayrıca, rapor GenAI kullanımının sadece bireysel faydalarla sınırlı kalmadığını, ekip içi iş birliğini ve kod bütünlüğünü de artırdığını göstermektedir. Copilot önerileri genellikle stil açısından tutarlı ve tanımlayıcı değişken adlarıyla üretildiği için, kod tabanında ortak okunabilirlik standartlarının korunmasına yardımcı olmaktadır. Bu da özellikle büyük ekiplerin

çalıştığı projelerde, kod gözden geçirme (code review) süreçlerinde zaman tasarrufu sağlamaktadır. Deneyimli geliştiriciler Copilot'u sadece öneri sağlayıcı olarak değil, aynı zamanda alternatif çözüm yolu üreticisi olarak da değerlendirmektedir. Bir problemin çözümüne yönelik birden fazla algoritmik yaklaşımı karşılaştırma imkânı tanıyan bu sistemler, teknik çeşitliliği artırmakta ve mühendislik kararlarında daha bilinçli tercihlere zemin hazırlamaktadır. Bu bağlamda, GenAI sistemlerinin sağladığı "çözüm çeşitliliği", problem çözme yetkinliğini destekleyen bir öğrenme süreci de sunmaktadır. Aynı raporda, AI ile kod yazma sürecinde yaşanan dönüşümün psikolojik etkileri de incelenmiştir. Katılımcıların önemli bir kısmı, üretken yapay zekâ kullanımı sayesinde iş yüklerinin daha yönetilebilir hâle geldiğini ve kod yazma süreçlerine karşı daha az kaygı geliştirdiklerini ifade etmiştir. Bu durum, tükenmişlik sendromunun (developer burnout) azaltılmasına katkı sağladığı gibi, geliştirici memnuniyeti ve iş gücü bağlılığı gibi metriklerde de olumlu sonuçlar doğurmaktadır. Benzer şekilde, Amazon'un AWS CodeWhisperer aracı, bulut tabanlı uygulama geliştiriciler için güvenli kod önerileri sunmakta ve IAM (Identity and Access Management) gibi hassas servislerde kod güvenliğini proaktif olarak denetlemektedir. Replit Ghostwriter ise tarayıcı tabanlı bir kod üretim deneyimi sağlayarak, özellikle yeni başlayanlar için erişilebilir bir yapay zekâ geliştirme aracı sunmaktadır. Endüstriyel uygulamalarda GenAI'nin önemli bir kullanım alanı da teknik belgelendirme üretimidir. Copilot ve ChatGPT gibi modeller, geliştiricilerin yazdığı fonksiyonları otomatik olarak yorumlayarak README dosyaları, kullanım kılavuzları ve API örnekleri oluşturabilmektedir. Bu yetenek, özellikle açık kaynak projelerde kullanıcı katkısını artırmakta ve onboarding süreçlerini hızlandırmaktadır. Ayrıca prompt tabanlı teknik içerik üretimi, yazılım ürünlerinin daha iyi anlaşılmasını sağlamaktadır. Geliştirici deneyimi düzeyi de bu araçlardan elde edilen faydayı önemli ölçüde etkilemektedir. Başlangıç seviyesindeki kullanıcılar, GenAI araçları sayesinde %60'a varan oranlarda daha hızlı fonksiyon üretmekte, ancak önerileri doğrulama konusunda yetersiz kaldıkları için güvenlik açıklarına daha sık maruz kalabilmektedir. Buna karşın deneyimli geliştiriciler, GenAI'yi hızlı prototipleme, tekrarlı görevlerin otomasyonu ve veri dönüşüm işlemleri gibi alanlarda daha etkin şekilde kullanmaktadır.

Teknolojik altyapı bakımından bakıldığında, GitHub Copilot'un sunduğu inline (satır içi) öneri mekanizmaları, özellikle Visual Studio Code (VS Code) gibi bütünleşmiş geliştirme

ortamlarında (IDE) yazılım geliştirme sürecine doğrudan ve etkili bir şekilde entegre olmaktadır. Bu sistemler, geliştiricinin yazmakta olduğu kodun bağlamını analiz ederek, anlık önerilerde bulunmakta ve böylece kod tamamlama süreçlerini ortalama olarak %30 ila %70 arasında hızlandırmaktadır. Bu oran, geliştiricilerin deneyim seviyesi, kodun karmaşıklığı ve proje türüne göre değişkenlik göstermekle birlikte, geniş kapsamlı üretkenlik kazanımlarını ortaya koymaktadır.

Bu öneri sistemi, geleneksel otomatik tamamlama özelliklerinden farklı olarak yalnızca sözcük veya fonksiyon ismi tamamlamaktan öte, fonksiyon gövdeleri, koşullu yapılar, hata yönetim blokları ve yorum satırları gibi karmaşık kod yapılarını da otomatik şekilde önerebilme yeteneğine sahiptir. Örneğin bir geliştirici `def calculate_tax` gibi bir fonksiyon başlığı yazdığında, Copilot bu fonksiyonun içeriğini geçmiş kod örüntülerine ve bağlama uygun olarak tamamlayabilir hem işlevsel hem de mantıksal olarak bütünlüklü bir yapı sunar. Bu yaklaşım yalnızca zamandan tasarruf sağlamakla kalmamakta, aynı zamanda bağlamsal farkındalık yoluyla kodun tutarlılığını da artırmaktadır. Inline öneriler, projenin genel yapısına uygun adlandırma kurallarını, kullanılan veri yapısı kalıplarını ve stil yönergelerini analiz ederek öneri üretmekte, böylece geliştiricinin el yazısıyla üreteceği koddan daha stilize ve standardize çıktılar sağlamaktadır. Bu durum, özellikle büyük ekipler tarafından yönetilen kurumsal projelerde, kod okunabilirliği ve sürdürülebilirliği açısından önemli avantajlar sunmaktadır. Ayrıca bu tür öneri mekanizmaları, yazılımcının zihinsel yükünü de azaltmaktadır. Özellikle yeni başlayanlar veya alan dışı geliştiriciler için satır içi öneriler, karmaşık yapıların nasıl oluşturulacağına dair bir rehber işlevi görmektedir; deneyimli geliştiriciler için ise, tekrar eden yapıları hızlıca geçip daha yaratıcı ve stratejik kodlama süreçlerine odaklanmalarını sağlamaktadır. GenAI sistemlerinin endüstride uygulanabilirliği, yazılım mühendisliği pratiklerine yalnızca hız kazandırmakla kalmayıp, aynı zamanda geliştirme süreçlerini daha erişilebilir, sürdürülebilir ve iş birliğine açık hâle getirmektedir. GitHub Copilot, Amazon CodeWhisperer ve Replit Ghostwriter gibi araçların gerçek dünyadaki entegrasyonu; otomatik kod üretimi, belgelendirme desteği, güvenlik önerileri ve bağlamsal kod tamamlama gibi alanlarda çarpıcı verimlilik artışları sağlamıştır. Özellikle Copilot'un Visual Studio Code ortamında sunduğu inline öneri mekanizması, kod tamamlama süreçlerinde %70'e varan hız kazanımlarıyla, geliştiricilerin üretkenliğini yeni bir boyuta taşımaktadır.

Kurumsal ölçekte yapılan analizler, bu araçların yalnızca bireysel geliştiriciler için değil, tüm yazılım ekipleri ve organizasyonları için yapısal faydalar sunduğunu göstermektedir. %55’lik bir verimlilik artışı, %33 oranında hata azaltımı ve daha hızlı prototipleme gibi kazanımlar, GenAI’nin artık bir yardımcı değil, yazılım yaşam döngüsünün stratejik bir bileşeni olduğunu kanıtlamaktadır. Aynı zamanda yeni başlayanlar için rehberlik işlevi gören bu sistemler, deneyimli geliştiricilere ise tasarım çeşitliliği, test üretimi ve teknik borç yönetimi gibi alanlarda etkin çözümler sunmaktadır.

Finans, sağlık ve e-ticaret gibi farklı sektörlerdeki başarılı kullanım örnekleri de GenAI’nin dikey uzmanlık alanlarında uygulanabilirliğini göstermektedir. Otomatik test senaryolarından hasta verisi analizine, satış dashboard otomasyonlarından müşteri segmentasyonuna kadar geniş bir yelpazede bu teknolojilerin yaygınlaştığı gözlemlenmektedir.

Tüm bu bulgular, üretken yapay zekâ sistemlerinin yazılım endüstrisinde geçici bir yenilik değil; kalıcı, dönüştürücü ve evriltici bir unsur olduğunu ortaya koymaktadır. Gelecekte bu sistemlerin, geliştiricilerin yalnızca yardımcı araçları değil, yazılım mimarisinin etkin bir ortak üreticisi hâline gelmesi beklenmektedir.

BEŞİNCİ BÖLÜM: SONUÇ VE GELECEK

Bu tez çalışmasında, üretken yapay zekâ (GenAI) teknolojilerinin yazılım geliştirme süreçlerine entegrasyonu derinlemesine incelenmiş hem kuramsal hem de deneysel düzlemde elde edilen bulgular bütüncül bir bakış açısıyla değerlendirilmiştir. Çalışma, GenAI'nin yalnızca kod üretim sürecinde değil, yazılımın yaşam döngüsü boyunca çeşitli aşamalarda geliştiriciye destek sunabildiğini ortaya koymuştur. Bu destek; fonksiyon üretimi, teknik belgelendirme yazımı, test senaryosu oluşturulması, hata ayıklama ve performans iyileştirme gibi çok yönlü işlevleri kapsamaktadır. Ayrıca GenAI sistemlerinin, geliştiricilerin zihinsel yükünü azaltarak daha stratejik görevlere odaklanmalarını mümkün kıldığı, yazılım mühendisliğinin üretim paradigmasını dönüştürdüğü sonucuna ulaşılmıştır.

Gerçekleştirilen deneylerde, manuel olarak kodlanan 20 farklı görev, ChatGPT, GitHub Copilot ve Amazon CodeWhisperer gibi GenAI sistemleri tarafından yeniden üretilmiş ve kod kalitesi, üretim süresi, güvenlik yapısı, satır yoğunluğu, modernlik ve stil standartlarına uygunluk açısından sistematik biçimde değerlendirilmiştir. Bulgular, GenAI destekli kodların ortalama %40'a varan üretim süresi avantajı sunduğunu, ayrıca kodların daha kısa, daha okunabilir ve daha sistematik olduğunu göstermektedir. Kodun daha güncel yapılarla yazılması (örneğin: try/except, async, pandas, arrow functions kullanımı) ve modülerlik ilkelerine daha fazla uygunluk göstermesi, üretken yapay zekâ sistemlerinin stilistik uyum konusunda da etkili olduklarını kanıtlamıştır.

Ancak elde edilen çıktılar, GenAI sistemlerinin bağlam farkındalığında hâlâ sınırlı olduğunu göstermektedir. Özellikle çok bileşenli sistemler içinde, belirli bir fonksiyonun proje bütünündeki yerini tam kavrayamayan modeller, bazı durumlarda bağlamsal uyumsuzluklar üretmiş ve çıktılar manuel revizyon gerektirmiştir. Bununla birlikte sistemler, örüntü tabanlı öğrenme sayesinde veri giriş doğrulama, hata yönetimi, yorum satırı üretme ve fonksiyon gövdesi önerme gibi alanlarda önemli ölçüde destek sunmuştur.

Tez kapsamında geliştirilen CLI tabanlı veri analiz prototipi, GenAI'nin gerçek dünya verisiyle çalışma, kullanıcı etkileşimini yönetme ve çıktı dosyaları oluşturma gibi çok yönlü süreçlerde de kullanılabileceğini somut biçimde ortaya koymuştur. Özellikle satış verilerinin okunması, filtrelenmesi, gruplanması ve grafiklenmesi gibi işlemler, yalnızca

doğal dil ile yönlendirilen prompt'lar üzerinden inşa edilmiştir. Kodun yapısı, yalnızca çalışır olmakla kalmayıp aynı zamanda Pylint ve Flake8 gibi analiz araçlarıyla test edilmiş ve stil uyumu açısından yüksek puanlar almıştır. Bu da üretken yapay zekânın sadece hızlı değil, kaliteli çıktılar da üretebildiğini göstermektedir. Bununla birlikte, sistemlerin ürettiği kodların hatasız kabul edilemeyeceği açıktır. Model hallucination sorunları, yani var olmayan fonksiyonların önerilmesi, kimi zaman kullanıcıyı yanıltabilecek riskli durumlar yaratabilmektedir. Ayrıca kod önerilerinde sabitlenmiş şifre kullanımı, SQL injection riskine açık yapılar ve input validasyon eksiklikleri gibi güvenlik açıkları tespit edilmiştir. Bu sorunlar, üretken sistemlerin geliştirici denetiminden bağımsız çalışmaması gerektiğini ortaya koymakta; AI tarafından üretilen kodların mutlaka manuel test ve değerlendirme süreçlerine tabi tutulmasını zorunlu kılmaktadır. Etik ve hukuki açılardan da üretken sistemlerin kontrolsüz kullanımı riskler barındırmaktadır. Telif hakkı ihlalleri, özellikle GPL gibi lisanslarla korunan yazılım kodlarının benzeri çıktılarla yeniden üretilmesi durumunda gündeme gelmektedir. Free Software Foundation gibi kuruluşların bu konuda dikkat çektiği etik uyarılar, yapay zekâ modellerinin eğitildiği veri setlerinin şeffaflığı ve çıktılarının yasal niteliği konusunda hâlâ açık sorular olduğunu göstermektedir. Bu bağlamda, üretken yapay zekânın güvenli, sorumlu ve şeffaf kullanımı konusunda sektörel rehberliklerin geliştirilmesi kaçınılmaz görünmektedir. Sonuç olarak, bu tez çalışması, GenAI sistemlerinin yalnızca yazılım geliştirmeyi hızlandıran araçlar değil, yazılım mühendisliğinin doğasını dönüştüren katalizörler olduğunu göstermiştir. Kod üretimi, test, yorumlama, belgeleme ve bakım gibi yazılım yaşam döngüsünün tüm evrelerine entegre edilebilen bu sistemler, geliştiricilere yalnızca teknik destek sunmakla kalmamakta; aynı zamanda üretkenlik, sürdürülebilirlik ve mesleki rol dönüşümü açısından da yeni açılımlar yaratmaktadır. Bununla birlikte, güvenlik, etik, bağlamsal uyum ve test kapsamı gibi alanlarda hâlâ insan müdahalesi gerektiren yönler bulunduğundan, GenAI sistemlerinin denetimli ve sorumlu biçimde kullanımı temel ilke olmalıdır.

Geleceğe dönük projeksiyonlar, üretken yapay zekâ (GenAI) sistemlerinin yalnızca satır bazlı kod üretimiyle sınırlı kalmayacağını; yazılım mühendisliğinin daha stratejik ve yapısal alanlarında da etkin roller üstleneceğini göstermektedir. Bu dönüşüm, yazılım üretim sürecini yalnızca hızlandırmakla kalmayacak, aynı zamanda mimari karar alma süreçlerinden sistematik test stratejilerinin kurgulanmasına, üretim kodlarının otomatik

belgelenmesinden yazılım kalitesinin sürekliliğini sağlayan refaktör önerilerinin geliştirilmesine kadar geniş bir etki alanı yaratacaktır. Bu bağlamda, GenAI sistemleri sadece kodun yazılmasına değil, aynı zamanda onun anlamlandırılmasına, sürdürülebilirliğine ve gelişimine yönelik bütüncül bir katkı sağlayarak yazılım üretim döngüsünü baştan sona yeniden şekillendirecektir. Geliştiricilerin bu ekosistem içindeki rolü de dönüşüme uğrayacaktır. Klasik yazılımcı rolünün ötesinde, geliştiriciler artık GenAI modelleriyle birlikte çalışan “karar verici-yorumlayıcı” profesyoneller hâline gelecektir. Bu kişiler, yalnızca modelin sunduğu çıktıları kullanmakla kalmayacak, aynı zamanda bu çıktıları semantik düzeyde değerlendirerek bağlam içinde optimize edecek; proje ihtiyaçlarına uygunluğunu ölçerek gereken müdahaleleri gerçekleştireceklerdir. Bu süreçte geliştiricinin rolü, “kod üreticisi” olmaktan çıkarak “üretilen içeriği denetleyen, yönlendiren ve kalite süzgecinden geçiren teknik küratör” düzeyine yükselme gibi olasılıklar doğacaktır. Böyle bir rol dönüşümünde, geliştiricilerin sahip olması gereken beceriler de değişecektir. Geleneksel algoritmik bilgiye ek olarak; yapay zekâ çıktılarının doğruluğunu değerlendirme, etik ve yasal çerçevede riskleri analiz etme, güvenlik zafiyetlerini ayıklama ve kodun sürdürülebilirliğini gözetme gibi yeni sorumluluklar gündeme gelecektir. Özellikle büyük projelerde, geliştirici artık üretim süreçlerini değil, üretimin denetim mantığını tasarlayan bir rol üstlenecektir. Bu da yazılım mühendisliğinde yeni bir uzmanlık alanı olan “Yapay Zekâ Gözetimli Yazılım Mühendisliği” gibi disiplinlerin doğmasına öncülük etmesi ihtimalleri bulunmaktadır. Geliştiricilerin bu dönüşümle birlikte yalnızca teknolojik araçları kullanan değil, aynı zamanda yapay zekâ destekli sistemlerle birlikte üreten, değerlendiren ve yöneten çok boyutlu aktörler hâline gelmesi kaçınılmaz görünmektedir. Bu da yazılım üretiminin geleceğini, insan zekâsı ile yapay zekânın simbiyotik iş birliğine dayalı yeni bir paradigmanın şekillendireceğini ortaya koymaktadır.

Buna paralel olarak, gelecekte yaygınlaşması beklenen bazı senaryolar şunlardır:

- Yüksek-özerk yazılım asistanları: GenAI sistemlerinin sadece kodlama değil, aynı zamanda yazılım yaşam döngüsünü uçtan uca yöneten otonom yardımcılar hâline gelmesi. Örneğin, kullanıcıdan gelen doğal dil girdisiyle proje kurulumunu başlatan, veri tabanı yapılarını öneren, fonksiyonları yazan, test eden ve dağıtımı planlayan sistemler geliştirilecektir.

- Refaktör odaklı yapay zekâ sistemleri: Zamanla kirlenen (code rot) ve teknik borç biriken büyük yazılım tabanlarını analiz eden, kod kokularını belirleyen ve yeniden yapılandırma (refactoring) önerileri sunan modeller, yazılım bakım süreçlerinde devrim yaratacaktır.
- Gerçek zamanlı üretim ortamı müdahaleleri: Üretim (production) ortamlarında çalışan yapay zekâ modelleri, hata istatistiklerine veya kullanıcı davranışlarına göre kod önerilerini yenileyecek; exception log'larına göre kendini uyarlayarak kodda "canlı düzeltme" (live patching) senaryoları uygulayabilecektir.
- Mimari tasarım modelleri oluşturma: Belirli bir sektöre (örneğin finans, sağlık, eğitim) uygun yazılım mimarilerini otomatik olarak modelleyen sistemler sayesinde yazılım mimarları, yüksek düzeyde önceden modellenmiş çözüm mimarilerini özelleştirmekle yetinecek, yapay zekânın sunduğu kalıpları zenginleştirme işlevi görecektir.
- Etik değerlendirme ve lisans uyumluluğu sistemleri: Gelecekte, AI tarafından yazılan kodların etik açıdan değerlendirilmesini yapan, kullanılan veri setlerinin lisans uyumluluğunu kontrol eden ve çıktının yasal risk skorunu tahmin eden meta-denetleyici sistemler gündeme gelecektir.

Bu senaryolar doğrultusunda, yazılım mühendisliği mesleği de yeni yetkinliklere evrilecektir. Sadece algoritmik düşünme değil, yapay zekâ tarafından üretilen bilgiyi analiz etme, denetleme ve yeniden yapılandırma yetisi önem kazanacaktır. Geliştiriciler, üretici modelleri eğiten, özelleştiren ve denetleyen profesyonel pozisyonlara yönelirken; yapay zekâ ile kodlayan, birlikte karar veren ve birlikte üreten hibrit bir üretim yapısı kaçınılmaz hâle gelecektir.

Tahimini bir sonuç olarak, üretken yapay zekâ sistemleri gelecekte yalnızca araç değil, yazılım üretiminin stratejik ortağı olacaktır. Bu gelişmeler, yazılım mühendisliği disiplini kökten dönüştürmekte; insan-yapay zekâ iş birliğine dayalı yeni bir mesleki paradigma yaratmaktadır. Bu paradigma hem bireysel geliştiricinin rolünü hem de yazılım ekiplerinin organizasyon yapısını yeniden şekillendirecektir.

EK 1. KULLANILAN KOD ÖRNEKLERİ

G1 görevinde alınan kod örnekleri:

```
1 # AmazonWS
2 def fibonacci_numbers(n):
3     # Input validation
4     if n < 0:
5         raise ValueError("n must be a non-negative integer")
6     elif n == 0:
7         return []
8     elif n == 1:
9         return [0]
10
11     # Initialize the sequence with first two numbers
12     fib_list = [0, 1]
13
14     # Generate subsequent Fibonacci numbers
15     while len(fib_list) < n:
16         fib_list.append(fib_list[-1] + fib_list[-2])
17
18     return fib_list
19
20 # Example usage
21 try:
22     num_fibs = 10
23     result = fibonacci_numbers(num_fibs)
24     print(f"The first {num_fibs} Fibonacci numbers are: {result}")
25 except ValueError as e:
26     print(f"Error: {e}")
```

```
1 # CoPilot
2 def fibonacci_list(n):
3     if n <= 0:
4         return []
5     fibs = [0, 1]
6     while len(fibs) < n:
7         fibs.append(fibs[-1] + fibs[-2])
8     return fibs[:n]
9 print(fibonacci_list(10))
```

```
1 # ChatGPT
2 def fibonacci(n):
3     if n <= 0:
4         return []
5     elif n == 1:
6         return [0]
7     fib_sequence = [0, 1]
8     while len(fib_sequence) < n:
9         fib_sequence.append(fib_sequence[-1] + fib_sequence[-2])
10    return fib_sequence
11 print(fibonacci(10))
```

G2 görevinde alınan kod örnekleri:

```
1 # AmazonK5
2 import pandas as pd
3 import numpy as np
4
5 def clean_sales_data(input_file, output_file):
6     try:
7         # Read the sales dataset
8         df = pd.read_csv(input_file)
9
10        # Store original shape for reporting
11        original_shape = df.shape
12
13        # 1. Drop rows where Product_ID is missing
14        df = df.dropna(subset=['Product_ID'])
15
16        # 2. Fill missing Price values with median
17        price_median = df['Price'].median()
18        df['Price'] = df['Price'].fillna(price_median)
19
20        # 3. Forward fill missing Quantity values
21        df['Quantity'] = df['Quantity'].fillna(method='ffill')
22
23        # If any remaining NaN values in Quantity (if ffill didn't catch all),
24        # fill with backward fill
25        df['Quantity'] = df['Quantity'].fillna(method='bfill')
26
27        # Save the cleaned dataset
28        df.to_csv(output_file, index=False)
29
30        # Print summary of changes
31        print(f"Data Cleaning Summary:")
32        print(f"Original dataset shape: {original_shape}")
33        print(f"Cleaned dataset shape: {df.shape}")
34        print(f"Rows removed: {original_shape[0] - df.shape[0]}")
35        print(f"Cleaned data saved to: {output_file}")
36
37        # Return the cleaned dataframe
38        return df
39
40    except FileNotFoundError:
41        print(f"Error: Input file not found")
42        return None
43    except Exception as e:
44        print(f"An error occurred: {str(e)}")
45        return None
46
47 def display_data_info(df):
48     if df is not None:
49         print("\nDataset Information:")
50         print(f"\nMissing values count:")
51         print(df.isnull().sum())
52
53         print("\nBasic statistics:")
54         print(df.describe())
55
56 # Example usage
57 if __name__ == "__main__":
58     input_file = "sales_data.csv" # Replace with your input file name
59     output_file = "cleaned_sales_data_AWS.csv"
60
61     # Clean the data
62     cleaned_df = clean_sales_data(input_file, output_file)
63
64     # Display information about the cleaned dataset
65     display_data_info(cleaned_df)
66
```

```
1 # CoPilot
2 import pandas as pd
3
4 # Load the sales dataset (replace 'sales_data.csv' with your actual filename)
5 df = pd.read_csv('sales_data.csv')
6
7 # Drop rows where 'Product_ID' is missing
8 df = df.dropna(subset=['Product_ID'])
9
10 # Fill missing 'Price' values with the median
11 median_price = df['Price'].median()
12 df['Price'] = df['Price'].fillna(median_price)
13
14 # Forward-fill missing 'Quantity' values
15 df['Quantity'] = df['Quantity'].fillna(method='ffill')
16
17 # Save the cleaned data
18 df.to_csv('cleaned_sales_data_copilot.csv', index=False)
```

```
1 # ChatGPT
2 import pandas as pd
3
4 # Load the dataset (assuming the filename is 'sales_data.csv')
5 df = pd.read_csv('sales_data.csv')
6
7 # Drop rows where 'Product_ID' is missing
8 df = df.dropna(subset=['Product_ID'])
9
10 # Fill missing 'Price' values with the median price
11 if 'Price' in df.columns:
12     median_price = df['Price'].median()
13     df['Price'].fillna(median_price, inplace=True)
14
15 # Forward-fill missing 'Quantity' values
16 if 'Quantity' in df.columns:
17     df['Quantity'].fillna(method='ffill', inplace=True)
18
19 # Save the cleaned data to a new CSV file
20 df.to_csv('cleaned_sales_data_ChatGPT.csv', index=False)
21
22 print("Data cleaning complete. File saved as 'cleaned_sales_data.csv'.")
23
```

G3 ve G4 görevlerindeki çıktı olan kod örneklerinin satır sayıları fazla olduğu için listeye eklenmemiştir.

KAYNAKLAR

- ADP Research Institute. (2023). *People at work 2023: A global workforce view*. Eriřim adresi: <https://www.adpri.org/research/people-at-work-2023>
- AI Papers Academy. (2023). *Meta's Code LLaMA explained*. Eriřim adresi: <https://aipapersacademy.com/code-llama/>
- Allamanis, M. (2022). *The evolution of code language models*. *Communications of the ACM*, 65(4), 56–65.
- Amazon Web Services. (2023). *Amazon CodeWhisperer documentation*. Eriřim adresi: <https://docs.aws.amazon.com/codewhisperer>
- Anderson, M. (2025, Nisan 8). *Pew survey: Nearly half of Americans believe AI will threaten jobs*. The Boston Globe.
- Balunovic, S., Brent, L. ve Livshits, B. (2023). When coding goes wrong: Security implications of AI code generation. *Communications of the ACM*, 66(3), 32–39.
- Barke, M., Brown, N. J. ve Pradel, M. (2025). *How do Copilot suggestions impact developers' frustration and productivity?* arXiv içinde (arXiv:2504.06808).
- Bommarito, M. ve Katz, D. (2022). *Ethical considerations of AI-generated code*. *Artificial Intelligence Review*, 35(3), 299–315.
- Bommarito, M. ve Katz, D. (2023). *The GitHub Copilot lawsuit and the future of AI-created code*. *Computer Law Review International*, 24(4), 135–142.
- Bradley, J. ve Nguyen, M. (2023). *Evaluating developer productivity with Amazon CodeWhisperer*. *Amazon Developer Reports*, 3(1), 45–60.
- Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P. vd. (2020). *Language models are few-shot learners*. *Advances in Neural Information Processing Systems*, 33, 1877–1901.
- Carlini, J., Tramer, F., Wallace, E., Jagielski, M., Herbert-Voss, A., Lee, K., Roberts, A., Brown, T. B., Song, D., Erlingsson, Ú., Oprea, A. ve Raffel, C. (2021). *Extracting training data from large language models*. *Proceedings of the USENIX Security Symposium* içinde, 2633–2650.
- ChatGPT Technical Team. (2023). *ChatGPT-4 Technical Report*. OpenAI.
- Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. de O., 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., ... Zaremba, W. (2021). *Evaluating large language models trained on code*. arXiv içinde (arXiv:2107.03374).
- Denny, D., de Abreu, M. A. S. ve German, D. M. (2023). *How good is GPT-4 at answering software engineering questions?* arXiv içinde (arXiv:2306.06031).
- Denny, L. A. ve Luxton-Reilly, A. (2023). *Do students learn from code generators? A field study of GitHub Copilot in CS1*. *Proceedings of the SIGCSE Technical Symposium on Computer Science Education* içinde, 1–7.

- Dickhaut, E., Bruhin, O., Elshan, E. ve Stocker, R. (2024). *The rise of generative AI in low code development platforms – An analysis and future directions. Proceedings of the Conference on Generative Technologies.*
- European Parliament. (2023). *EU AI Act: European Parliament legislative resolution on the proposal for a regulation of the European Parliament and of the Council laying down harmonised rules on artificial intelligence.* Erişim adresi: https://www.europarl.europa.eu/doceo/document/TA-9-2023-0239_EN.html
- Evans Data Corporation. (2023). *AI and the Future of Programming Survey.* Erişim adresi: <https://evansdata.com/reports/viewRelease.php?reportID=50>
- Fu, Q., Yu, X., Liu, Z., Wang, W., Yang, H., Meng, G., Liu, Y., Liang, B. ve Zhang, K. (2023). *Empirical security analysis of GitHub Copilot code suggestions. Proceedings of the 44th IEEE Symposium on Security and Privacy* içinde, 23–39.
- Fu, Y., Liang, P., Tahir, A., Li, Z., Shahin, M., Yu, J. ve Chen, J. (2023). *Security weaknesses of Copilot-generated code in GitHub projects: An empirical study.* arXiv preprint arXiv:2310.02059.
- Ganguli, A., Papadimitriou, A., Askell, L., Nanda, J., Hernandez, D., Henighan, L. ve Schulman, J. (2023). *Red teaming language models to reduce harms: Methods, scaling behaviors, and lessons learned.* arXiv preprint arXiv:2303.00676.
- GitHub Docs. (2022). *GitHub Copilot ve Open Source.* Erişim adresi: <https://docs.github.com>
- GitHub. (2021). *Introducing GitHub Copilot.* Erişim adresi: <https://github.com/features/copilot>
- GitHub. (2023). *The state of AI-powered developer tools.* GitHub Next Research.
- Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. ve Bengio, Y. (2014). *Generative adversarial nets. Advances in Neural Information Processing Systems*, 27.
- GPT-4 Evaluation Team. (2023). *Exmepor: GPT-4 Code Generation and Testing Study.* OpenAI Technical Report.
- Guo, D., Liu, S., Zhao, W., Zhang, R., Wang, Y., Li, J., Sun, Y., Wu, Y., He, Y., Hu, J., Zhang, H., Shen, Z., Wang, X. ve Liu, J. (2024). *Deepseek-Coder: When the large language model meets programming. The rise of code intelligence.* arXiv içinde (arXiv:2401.14196).
- Improta, C., Tu, Q., Alon, U., Koc, Y., Bakst, A., Bae, H., Ghaffari, P., Kim, H., Kodali, A., Mishra, S., Mun, J., Nair, S., Nguyen, A., Raman, K., Shen, T., Shojaei, A., Veeramachaneni, K., Xiong, C., Ye, J. ve Zitnik, M. (2023). *Quality in, quality out: Investigating training data's role in AI code generation.* arXiv preprint arXiv:2311.09345.
- Islam, M. A., Jonnala, D. V., Rekhi, R., Pokharel, P., Cilamkoti, S., Imran, A., Kosar, T. ve Turkkan, B. (2025). *Evaluating the energy-efficiency of the code generated by LLMs.* arXiv içinde (arXiv:2505.20324).
- Jovanović, M. ve Campbell, M. (2022). *Generative artificial intelligence: Trends and prospects. IEEE Computer*, 55(10), 107–112.

- Khot, N. (2024). *Generative AI for automated code generation: Transforming software development and maintenance*. *International Journal of All Research in Education and Scientific Methods*, 12(10), 967–970.
- LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. ve Jackel, L. D. (1989). *Backpropagation applied to handwritten zip code recognition*. *Neural Computation*, 1(4), 541–551.
- Liu, Y., Chen, M., Lu, Y., Zhao, J., Zhang, Y., Wang, T., Xu, K., Guo, Y. ve Jin, Z. (2023). *Refining ChatGPT-generated code: Characterizing and mitigating code quality issues*. arXiv preprint arXiv:2307.12596.
- Liu, J., Wang, Y., Xu, C., Zhang, L., Wu, Y., Zhao, J., Li, S., Huang, Y., Zhou, J., Zhao, J., Liu, Y., Sun, Y. ve Zhu, J. (2023). *Is your code generated by ChatGPT really correct? Rigorous evaluation on HumanEval*. arXiv içinde (arXiv:2305.01210).
- Li, Y., Tarlow, D., Brockschmidt, M., Nowozin, S., Gaunt, A. L., Bunel, R., Nguyen, A., Dohan, D., Sordoni, A., Bapst, V., Chan, C., Barham, P., Tunyasuvunakool, K., Sutton, C., Tomczak, J. M., Zoph, B., Vasudevan, V., Pathak, Y., Simonyan, K., ... Vinyals, O. (2022). *Competition-level code generation with AlphaCode*. arXiv preprint arXiv:2203.07814.
- Lund, B. D. ve Wang, T. (2023). Chatting about ChatGPT: How may AI and GPT impact academia and libraries? Library Hi Tech News.
- McKinsey. (2023). *Unleashing developer productivity with generative AI*. McKinsey Digital. Erişim adresi: <https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/unleashing-developer-productivity-with-generative-ai>
- Mitchell, T. M. (1997). *Machine learning*. New York: McGraw-Hill.
- Nijkamp, E., Liu, H., Lin, Y. vd. (2022). *CodeGen: An open large language model for code with multi-turn program synthesis*. arXiv içinde (arXiv:2203.13474).
- Ochs, M., Narasimhan, K. ve Mezini, M. (2023). *Evaluating and improving transformers pre-trained on ASTs for code completion*. *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*.
- OpenAI. (2023). *GPT-4 Technical Report*. arXiv içinde (arXiv:2303.08774).
- OpenAI. (2024). *OpenAI API documentation: Logprobs and output probabilities*.
- Ovi, M. S. I., Rahman, M. M., Haque, M. A., Hossain, M. S., Kabir, A. H. M. E., Hasan, M. T., Islam, M. T., Alam, M. S., Ahmed, M. ve Chowdhury, M. M. R. (2024). *Benchmarking ChatGPT, Codeium, and GitHub Copilot: A comparative study of AI-driven programming and debugging assistants*. arXiv içinde (arXiv:2409.19922).
- Pandey, R., Singh, P., Wei, R. ve Shankar, S. (2024). *Transforming software development: Evaluating the efficiency and challenges of GitHub Copilot in real-world projects*. arXiv içinde (arXiv:2406.17910).
- Pearce, H., Ahmad, B., Tan, B. ve Dolan-Gavitt, B. (2022). *Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions*. *Proceedings of the IEEE Symposium on Security and Privacy* içinde, 754–768.

- Peng, S., Kalliamvakou, E., Cihon, P. ve Demirer, M. (2023). *The impact of AI on developer productivity: Evidence from GitHub Copilot*. arXiv içinde (arXiv:2302.06590).
- Radford, A. ve Narasimhan, K. (2018). *Improving language understanding by generative pre-training*. OpenAI.
- Saravanan, V., Kavitha, S., Ravi, S., Seetha, A., Rambabu, C. ve Kanth, T. V. R. (2025). *Generative AI in software engineering: Revolutionizing code generation and debugging*. *International Journal of Computer and Experimental Science and Engineering*, 11(2), 2908–2917.
- Siddique, M., Dey, S. ve Takabi, H. (2023). *SecurityEval: A benchmark for evaluating the security of AI-based code generation systems*. *Proceedings of the IEEE Secure Development Conference (SecDev)* içinde, 1–8.
- Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I. ve Hassabis, D. (2016). *Mastering the game of Go with deep neural networks and tree search*. *Nature*, 529, 484–489.
- Sobania, A., Briesch, M. ve Rothlauf, F. (2023). *An empirical study on the use of ChatGPT for code generation*. *Proceedings of the 2023 IEEE/ACM International Conference on Automated Software Engineering (ASE)* içinde, 1224–1236. IEEE.
- Sobania, D., Briesch, M. ve Rothlauf, F. (2021). *Choose your programming copilot: A comparison of the program synthesis performance of GitHub Copilot and genetic programming*. arXiv içinde (arXiv:2111.07875).
- Song, F., Agarwal, A. ve Wen, W. (2024). *The impact of generative AI on collaborative open-source software development: Evidence from GitHub Copilot*. arXiv içinde (arXiv:2410.02091).
- Stray, V., Moe, N. B., Ganeshan, N. ve Kobbenes, S. (2024). *Generative AI and developer workflows: How GitHub Copilot and ChatGPT influence solo and pair programming*. ResearchGate. Erişim adresi: <https://www.researchgate.net/publication/387021973>
- Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Eloundou, A., Li, X. L., Scao, T. L., Fan, A., Stokes, J. W., Bambröo, P., Kirnap, O., Taylor, R., Belkada, Y. ve Joulin, A. (2023). *Code LLaMA: Open foundation models for code*. arXiv içinde (arXiv:2308.12950).
- Turing, A. M. (1950). *Computing machinery and intelligence*. *Mind*, 59(236), 433–460.
- Udoidiok, I., Reza, H. ve Zhang, J. (2024). *Exploring AI integration in software development: Case studies and insights*. ResearchGate. Erişim adresi: <https://www.researchgate.net/publication/381926402>
- Vaithilingam, T., Brant, J., Xu, G. ve Ray, B. (2022). *Expecting the unexpected? Copilot-generated code and programmer behavior*. *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)* içinde, 1–13.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł. ve Polosukhin, I. (2017). *Attention is all you need*. *Advances in Neural Information Processing Systems*, 30.

- Vondrick, D. ve Sharma, P. (2023). *Risks of AI coding assistants in production software. Proceedings of the NeurIPS AI Alignment Workshop.*
- Wang, X., Jiang, L., Zhou, Y., Li, X., Wang, S., Liu, X., Liu, H., Chen, J., Zhao, Y., Hu, X., Zhang, Y., Xu, M., Wu, Y., Sun, Z., Dong, Z., Zhang, X., Wang, H., Liu, C., Zheng, Y., Chen, J. (2024). *Large language models for code generation: A comprehensive survey of challenges, techniques, evaluation and applications.* arXiv preprint arXiv:2302.12365.
- Wang, Y., Le, H., Sethi, A., Phan, N. H., Li, J. ve Hoi, S. C. H. (2023). *Towards explainable code generation with large language models. Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP) içinde, 1069–1088.*
- Weizenbaum, J. (1966). *Eliza—a computer program for the study of natural language communication between man and machine. Communications of the ACM, 9(1), 36–45.*
- Winograd, T. (1971). *Procedures as a representation for data in a computer program for understanding natural language. MIT Artificial Intelligence Laboratory.*
- WIRED. (2024, Mayıs). *Google signs deal with Stack Overflow to train AI models. Wired Magazine.*
- Xiao, T., Hata, H., Treude, C. ve Matsumoto, K. (2024). *Generative AI for pull request descriptions: Adoption, impact, and developer interventions. Proceedings of the ACM on Software Engineering, 1(FSE), Article 47, 1–23.*
- Xu, F. F., Alon, U., Neubig, G. ve Hellendoorn, V. J. (2022). *A systematic evaluation of large language models of code.* arXiv preprint arXiv:2202.13169.
- YapayOrnek. (2024). *Codeium nedir ve nasıl kullanılır?* Erişim adresi: <https://yapayornek.com/codeium>
- Ye, C., Zhao, W., Liu, T., Wang, Y., Zhang, Y., Chen, J. ve Zhang, L. (2023). *A comprehensive study of AI-generated code quality. Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE) içinde, 1032–1043.*
- Yetiştiren, B., Özsoy, I., Ayerdem, M. ve Tüzün, E. (2023). *Evaluating the code quality of AI-assisted code generation tools: An empirical study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT.* arXiv preprint arXiv:2304.10778.
- Zhu, Q., Wang, Y., Shen, Z., Zhang, H., He, Y., Sun, Y., Zhang, R., Liu, S., Wu, Y., Wang, X. ve Liu, J. (2024). *DeepSeek-Coder-V2: Breaking the barrier of closed-source models in code intelligence.* arXiv içinde (arXiv:2406.11931).
- Ziegler, A., Yasin, A. ve Klocke, T. (2024). *Measuring GitHub Copilot's impact on productivity. Communications of the ACM, 67(2), 62–69.*
- Ziegler, M., Yasin, A. ve Klocke, T. (2023). *Productivity impact of GitHub Copilot in software engineering tasks.* arXiv içinde (arXiv:2302.06590).
- Zou, A., Wang, L., Kim, M., Li, W., Duan, S., Chen, X., Shi, J., Ou, D. ve Pei, K. (2023). *Universal and transferable adversarial attacks on code generation. Proceedings of the IEEE Symposium on Security and Privacy içinde, 340–359.*

Zügner, M., Schmitt, T. ve Günnemann, S. (2022). *Neural code completion: A survey*.
arXiv içinde (arXiv:2202.13213).

